Advanced Java

Daniel Hinojosa

Table of Contents

JShell

Introducing JShell

- Java Shell tool (JShell) is an interactive tool for learning the Java programming language and prototyping Java code
- Read-Evaluate-Print Loop (REPL)
- Evaluates Expressions as they are entered
- JShell accepts Java
  - statements
  - variables
  - methods
  - `class` definitions
  - `import` definitions
  - expressions

Reasons for JShell

- Trial of code before implementation
- Establish and communicate ideas with code
- Bring production code and dissect problems
- Take ideas or fixes back to the IDE

JShell Prerequisites

- JDK 9 or higher

Starting JShell

```
% jshell
```

```
|  Welcome to JShell -- Version 10
|  For an introduction type: /help intro

jshell>
```

JShell in verbose mode

- `-v` will enter you in verbose mode
- Full description of actions performed within JShell

```
% jshell -v
```

Trying out JShell

- Notice there are no semicolons in using JShell when it deals with *some* assignments
- Later versions of Java can make using of `var` same in JShell

```
jshell> var list = List.of(3,4,5,6)
list ==> [3, 4, 5, 6]

jshell> var mapped = list.stream().map(x -> x +
1).collect(Collectors.toList())
mapped ==> [4, 5, 6, 7]

jshell>
```

Replacing Variables in JShell

In this mode you have the availability to reassign a variable in JShell

```
jshell> var x = List.of(1,2,3,4)
x ==> [1, 2, 3, 4]
|  created variable x : List<Integer>

jshell> var x = 30
x ==> 30
|  replaced variable x : int
|    update overwrote variable x : List<Integer>
```

`final` is ignored

- The keyword `final` is ignored as a top level assignment within JShell
- If the verbose is turned on in JShell using `-v` you can review the message

```
jshell> final var d = 30
|  Warning:
|  Modifier 'final'  not permitted in top-level declarations, ignored
|  final var d = 30;
|  ^---^
d ==> 30
|  created variable d : int
```

Scratch Variable

- If you do not create a variable name, one will be created for you
- This is called a *scratch variable*
- It is assigned with a variable `$n` where n is a monotonically increasing integer

```
jshell> 2 + 2
$3 ==> 4
|  created scratch variable $3 : int
```

It can then be subsequently called by that variable

```
jshell> $3 + 2
$4 ==> 6
|  created scratch variable $4 : int
```

Creating methods

- Methods can also be created *without* a surrounding `class`
- They can then be invoked afterwards by name with a surrounding `class`

```
jshell> public String times(int n, String s) {
   ...>    StringBuilder sb = new StringBuilder();
   ...>    for (int i = 0; i < n; i++) {
   ...>       sb.append(s);
   ...>    }
   ...>    return sb.toString();
   ...> }
```

Invocation can then be invoked using standard Java:

```
jshell> times(3, "Foo")
$6 ==> "FooFooFoo"
|  created scratch variable $6 : String
```

Changing method definitions

- If you want to rewrite a method, you can do so by just creating a different implementation
- If you have `-v` verbose mode on, this will show that you are replacing the definition

```
jshell> public String times(int n, String s) {
   ...>   return IntStream.range(0, n).boxed()
   ...>           .map(x -> s).collect(Collectors.joining());
   ...> }
|  modified method times(int,String)
|    update overwrote method times(int,String)
```

Creating a `class`

- Much like a method, a `class` can be creating in JShell
- Merely type in the declaration and use the class at will

```
jshell> public class Country {
   ...>    private String name;
   ...>    private String capital;
   ...>    public Country (String name, String capital) {
   ...>      this.name = name;
   ...>      this.capital = capital;
   ...>    }
   ...>    public String toString() {
   ...>      return "Country {" + name + ", " + capital + "}";
   ...>    }
   ...> }
|  created class Country
```

Subsequently, you can then instantiate the class

```
jshell> var china = new Country("China", "Beijing");
china ==> Country {China, Beijing}
|  created variable china : Country

jshell> var poland = new Country("Poland", "Warsaw");
poland ==> Country {Poland, Warsaw}
|  created variable poland : Country
```

Making a mistake

If you make a mistake, you can hit <**UP-arrow**> and edit the previous unrunnable code

```
jshell> public String times(int n, String s) {
   ...>    StringBuilder sb = new StringBuilder()
   ...>    for (int i = 0; i < n; i++) {
   ...>       sb.append(s);
   ...>    }
   ...>    return sb.toString()
   ...> }
|  Error:
|  ';' expected
|     StringBuilder sb = new StringBuilder()
|                                           ^
|  Error:
|  ';' expected
|     return sb.toString()
|
```

- Hitting the <**UP-arrow**> key, it gives you the ability to edit again
- Typing <**Return**> anywhere in the code edit section will execute the method

```
jshell> public String times(int n, String s) {
   ...>    StringBuilder sb = new StringBuilder();
   ...>    for (int i = 0; i < n; i++) {
   ...>       sb.append(s);
   ...>    }
   ...>    return sb.toString()
   ...> }
| created method times(int,String)
```

Analyzing an `Exception`

- You can trace the root of the `Exception` by:
  - Reading the stack trace
  - Tracing the stack trace

In an exception back-trace, a location within REPL entered code is displayed as the `#id/linenumber`, where snippet id is the number displayed in `/list` and line-number is the line-number within the snippet.

```
jshell> int divide(int a, int b) {
...> return a / b;
}

jshell> divide(5, 0);
| java.lang.ArithmeticException thrown: / by zero
| at divide (#1:2)
| at (#2:1)
```

Running an Editor

- You can edit any code using `/edit` command along with a declaration.
- Declarations can either be `class`, method, or variable

Edit `Country`, in this case it is a `class`

```
jshell> /edit Country
```

This will in turn open an editor so that you can edit fully with full cursor support

Figure 1. Standard Editor in JShell

Running your own editor

```
jshell> /set editor vi
```

Resetting your editor

```
jshell> /set editor -default
```

Typical Commands

- `/vars` will show all variables bound
- `/methods` will show all methods bound
- `/types` will show all types, e.g. classes
- `/list` shows a list of entered "snippets"

Viewing Commands

- We can view the possible list of commands by typing `/` and then <**TAB**>

```
jshell> /
/!          /?          /drop       /edit
/env        /exit       /help       /history
/imports    /list       /methods    /open
/reload     /reset      /save       /set
/types      /vars

<press tab again to see synopsis>

jshell> /
```

Typical Commands

- `/vars` will show all variables bound
- `/methods` will show all methods bound
- `/types` will show all types, e.g. classes
- `/list` shows a list of entered "snippets"

Tab Completion

Press <**TAB**> after some code to see some auto-complete alternatives

```
jshell> System.out.
append(         checkError()    close()         equals(
flush()         format(         getClass()      hashCode()
notify()        notifyAll()     print(          printf(
println(        toString()      wait(           write(
```

Type some more of the signature and get documentation

```
jshell> System.out.format(
Signatures:
PrintStream PrintStream.format(String format, Object... args)
PrintStream PrintStream.format(Locale l, String format, Object... args)

<press tab again to see documentation>
```

Pressing <**TAB**> again will show the full documentation

Snippet Transformation

- When invoking the keys combination of <**SHIFT**>+<**TAB**>, you can then use the following to transform your line
  - **m** will create a method
  - **v** will create a variable
  - **i** will provide a selection of `import` for a `package` of a class

Variable Snippet Transformation

- Given that we have already typed the following
- And our cursor is at the end of the line
- We can type <**SHIFT**>+<**TAB**> and then **v** to create a variable
- The cursor will be in a position to create the variable

```
jshell> new BigInteger("302021")
```

- After <**SHIFT**>+<**TAB**> and then <**v**>
  - The cursor will be positioned before the `=`
  - You now have the opportunity to enter a variable name

```
jshell> BigInteger  = new BigInteger("302021")
```

If we named it `i`, this would result in...

```
jshell> BigInteger i = new BigInteger("302021")
```

Method Snippet Transformation

- After creating snippet and our cursor is at the end of the line
- We can type <**SHIFT+TAB**> and then <**m**> to create a method
- The cursor will be in a position to create the method
- This will only work if all variables can be resolve

Establishing a variable

```
jshell> BigInteger i = new BigInteger("302021")
i ==> 302021
|  created variable i : BigInteger
```

We reference `i` *and we realize we want as a method*

```
jshell> i.add(new BigInteger("4")
```

- The following will give you the opportunity to name the method
- It will place the cursor before the `()`

After <**SHIFT+TAB**> *and* <**m**>

```
jshell> BigInteger () { return i.add(new BigInteger("4")); }
```

Import Snippet Transformation

- <**SHIFT+TAB**> and <**i**> will provide choices for `import` if it is not in the *java.base* module
- Type the `class` you need and at the end <**SHIFT+TAB**> and <**i**>.
- Select the package you wish to import

<**SHIFT+TAB**> *and* <**i**> *after typing* `DriverManager`

```
jshell> DriverManager
0: Do nothing
1: import: java.sql.DriverManager
Choice:
```

Input Line Navigation

Editing is supported for editing:

- The current line

- Accessing the history through previous sessions of JShell.
- <**CTRL**> and <**META**> key are used in key combinations.

Meta key is a key like the Windows key or Apple key, if not available, then use the <**ALT**> key

Editing Navigation

- When navigating forwards and backwards within a line:
  - Use the <**RIGHT-Arrow**> and <**LEFT-Arrow**>
  - or, <**CTRL+B**> or <**CTRL+F**>
- When bringing up previous snippets and commands use <**UP-Arrow**>
- If the previous commands or snippets contains multiple lines you can edit that line with <**UP-Arrow**> or <**DOWN-Arrow**>

Additional Keys for Editing Navigation

Keys	Action
<**Return**>	Enters the current line
<**LEFT-arrow**>	Moves backward one character
<**RIGHT-arrow**>	Moves forward one character
<**UP-arrow**>	Moves up one line, backward through history
<**DOWN-arrow**>	Moves down one line, forward through history
<**CTRL+A**>	Moves to the beginning of the line
<**CTRL+E**>	Moves to the end of the line
<**META+B**>	Moves backward one word
<**META+F**>	Moves forward one word

Setting the Classpath

- You can use external code that is accessible through the class path in your JShell session.
- Use `--class-path` to load external directories and jar files

```
% jshell --class-path myOwnClassPath
```

While in an already loaded session you can use `/env --class-path`

```
jshell> /env --class-path myOwnClassPath
| Setting new options and restoring state.
```

To view the current classpath:

```
jshell> /env
|     --class-path guava-27.0.1-jre.jar
```

Scripting

- A JShell script is a sequence of snippets and JShell commands in a file, one snippet or command per line.
- Scripts can be a local file, or one of the following predefined scripts:

Script Name	Script Contents
DEFAULT	Includes commonly needed import declarations. This script is used if no other startup script is provided.
PRINTING	Defines JShell methods that redirect to the `print`, `println`, and `printf` methods in `PrintStream`.
JAVASE	Imports the core Java SE API defined by the `java.se` module, which causes a noticeable delay in starting JShell due to the number of packages.

Startup Scripts

- The default startup script has common imports, e.g. `java.lang`
- You can create custom imports as needed
- Startup scripts are loaded everytime:
  - JShell is started
  - When `/reset`, `/reload`, or `/env` commands are invoked
  - The `DEFAULT` script is used by default

Setting up the Startup Script

- You can set the startup script using the following script
- This will only be active in the current session

```
jshell> /set start mystartup.jsh
jshell> /reset
| Resetting state.
jshell
```

You can retain the setup using `-retain` for subsequent sessions

```
jshell> /set start -retain DEFAULT PRINTING
```

Showing what has been retained

Use `/set start` with no arguments to show startup scripts

```
jshell> /set start
|  /set start -retain DEFAULT PRINTING
|  ---- DEFAULT ----
|  import java.io.<strong>;
|  import java.math.</strong>;
|  import java.net.<strong>;
|  import java.nio.file.</strong>;
```

Start Multiple Scripts from Command Line

- If starting from your shell command line, you can use the `--startup` flag
- This will establish startup scripts when first running `jshell`

```
% jshell --startup DEFAULT --startup PRINTING
```

Creating Scripts

- Scripts can be created in an editor or generated in JShell
- To save the JShell session, use either of the following:

```
jshell> /save mysnippets.jsh
```

Saves the history of all of the snippets and commands, both valid and invalid

```
jshell> /save -history myhistory.jsh
```

Saves the contents of the current startup script setting to *mystartup.jsh*.

```
jshell> /save -start mystartup.jsh
```

Loading Scripts

Load a script when first starting JShell

```
% jshell mysnippets.jsh
```

Within JShell a script can be loaded with `/open`

```
jshell> /open mysnippets.jsh
```

Exiting JShell

```
jshell> /exit
```

Where to find more information

- We covered some of the major functionality of JShell
- More fine-grained information can be found in the Official JShell Documentation

Lab: JShell

Step 1: Launch JShell in verbose mode

Step 2: Create a method or `class` that given a `List<String>` of people's name it would return a random winner. Use `java.util.Random` for the randomization

Step 3: Create variables `thirdPlace`, `secondPlace` and `firstPlace` that will return the winners from the `List` of `String`

Step 4: Exit the JShell

Setup

Project Check

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

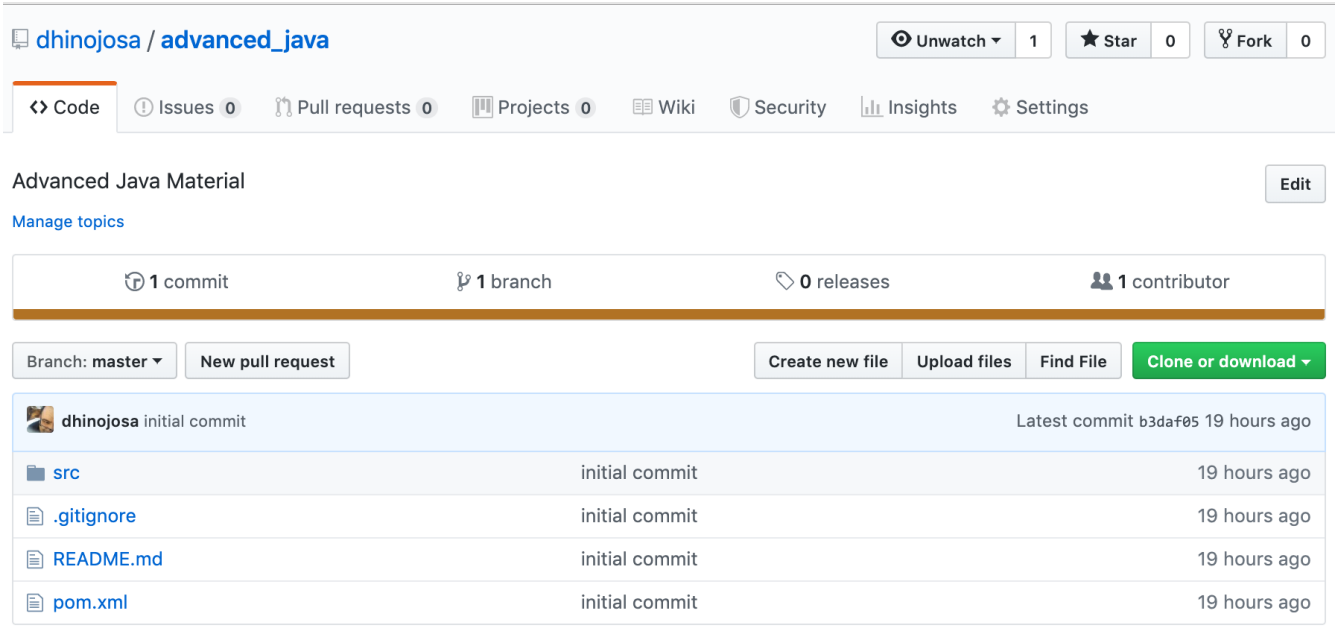
- JDK 12 or 13 (latest java is 13)
- Maven 3.6.2

To verify that all your tools work as expected

```
% javac -version
javac 13

% java -version
java version "13"
Java(TM) SE Runtime Environment (build 1.8.0_13-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.6.2 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11
-10T09:41:47-07:00)
Maven home: /usr/lib/mvn/apache-maven-3.6.2
Java version: 13, vendor: Oracle Corporation
Java home: /usr/lib/jvm/jdk13/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-34-generic", arch: "amd64", family:
"unix"
```

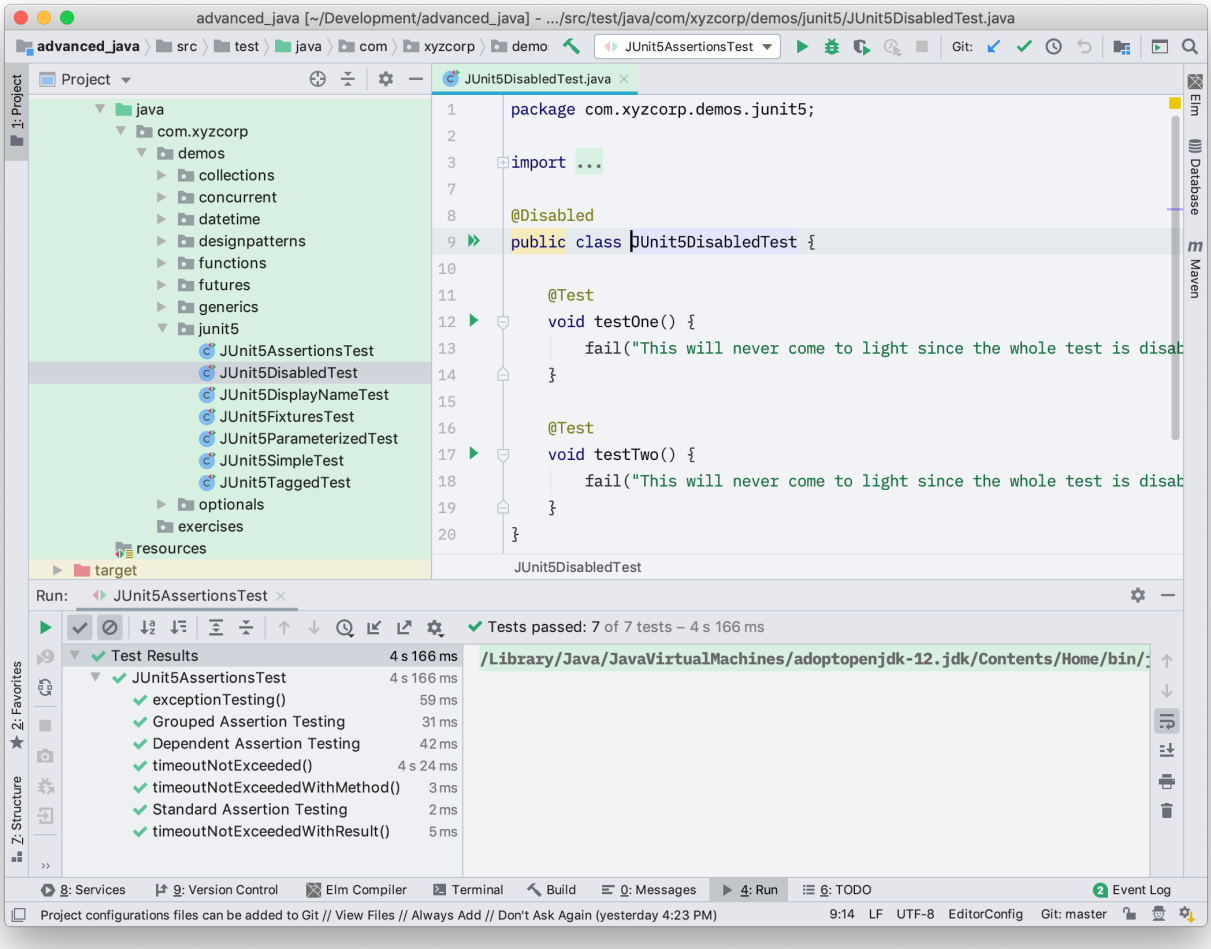
Download the Project

From https://github.com/dhinojosa/advanced_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.



Opening Project in IntelliJ

Once *advanced_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.

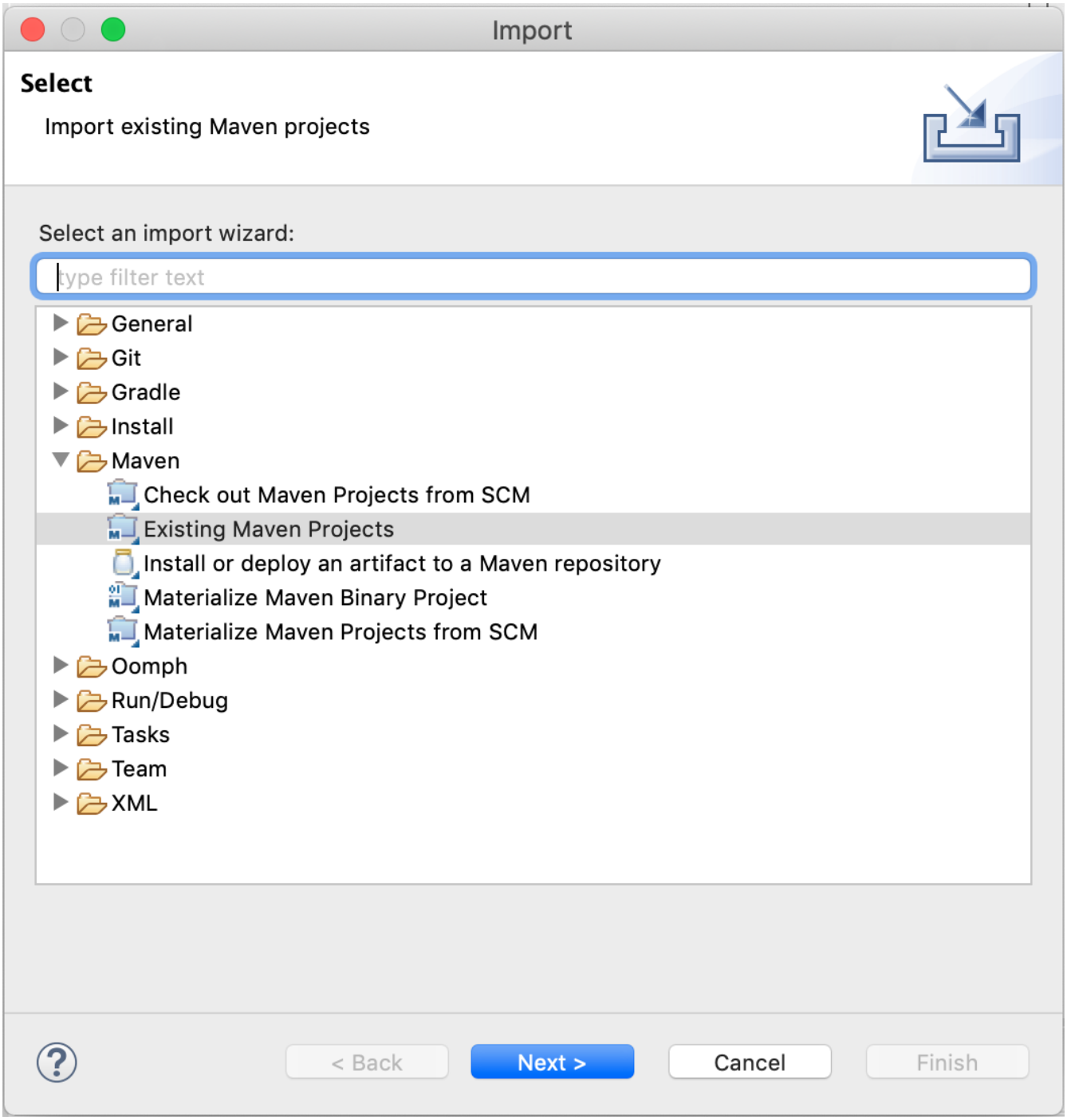


Opening the project in Eclipse

Once downloaded and extracted:

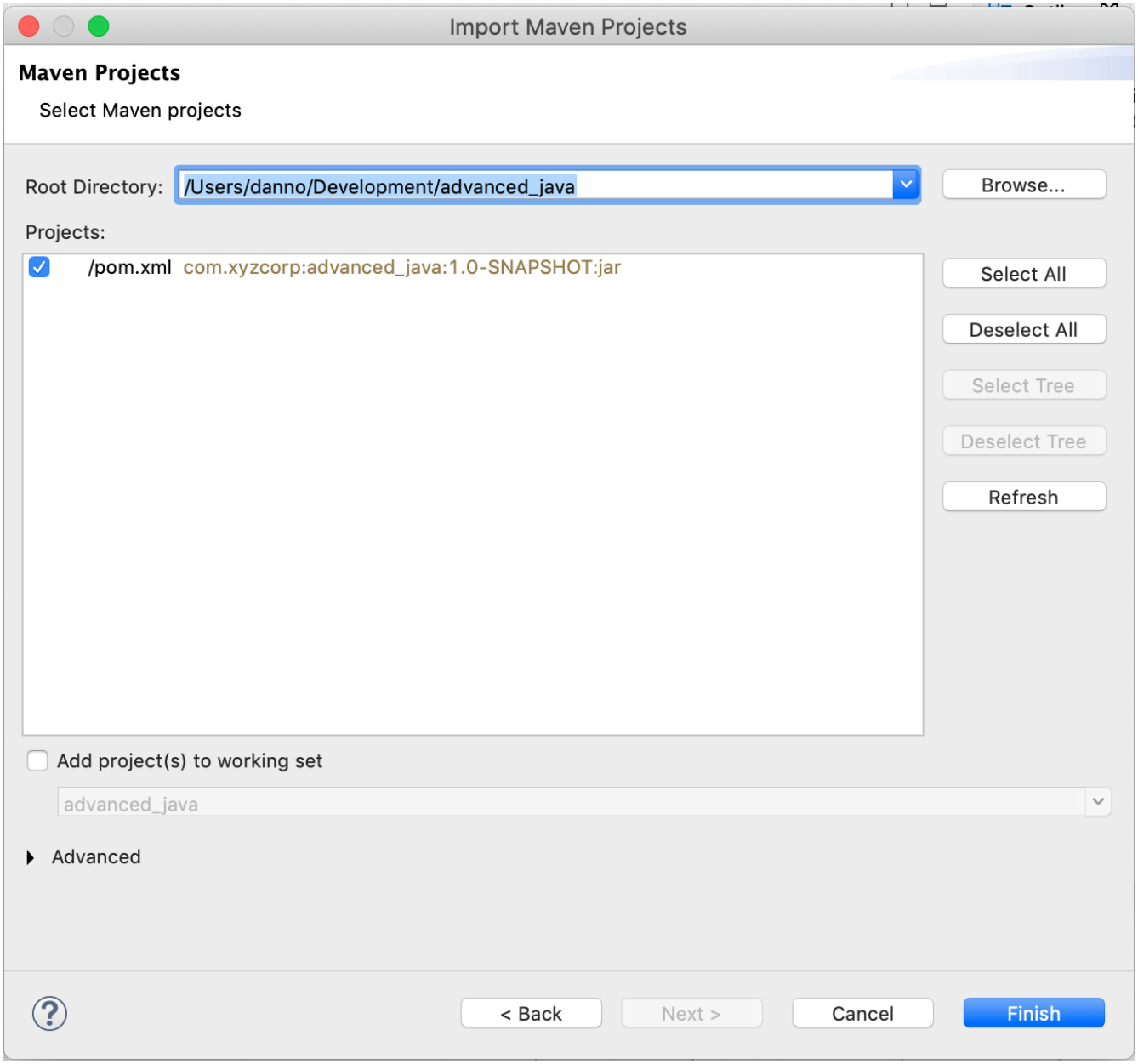
Step 1: Select *File* > *Import Project* in the menu.

Step 2: In the following dialog box:



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

Step 3:



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

Step 4: Click *Finish*

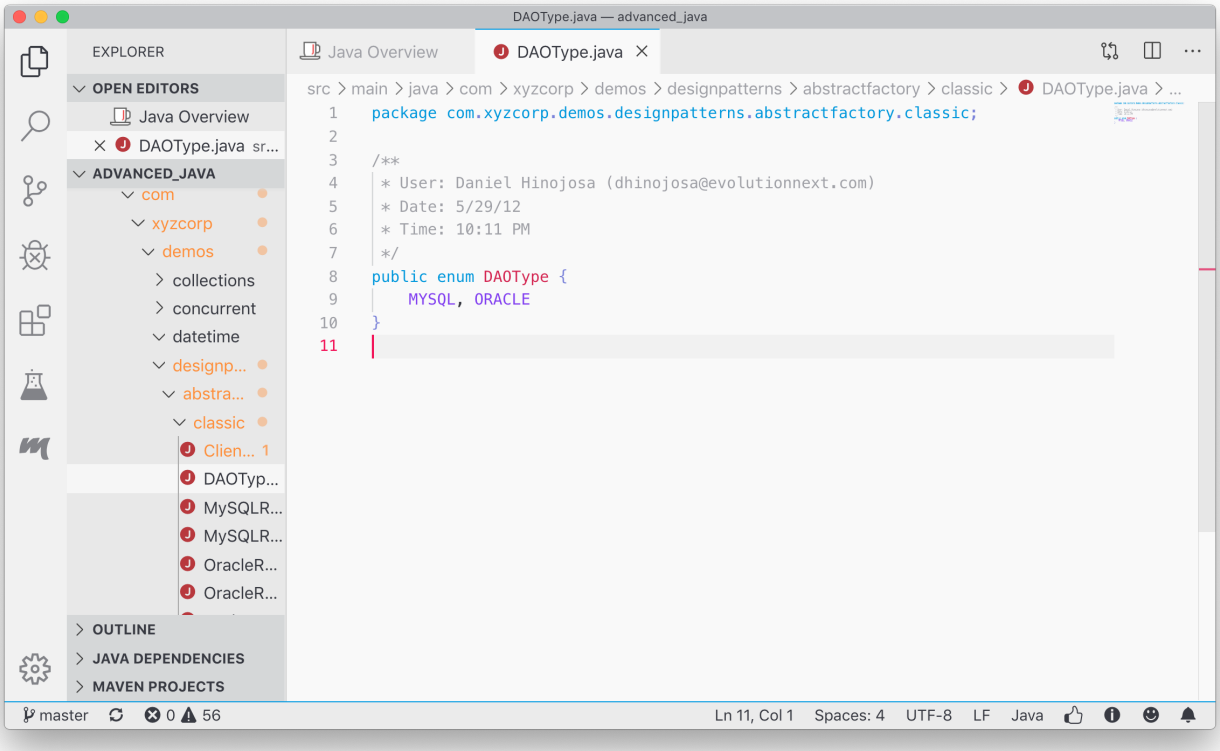
Opening the project in VSCode

Once downloaded and extracted:

Step 1: In a terminal, go to the *advanced_java* directory

Step 2: Run `code .` in the directory

```
% code .
```


DAOType.java — advanced_java
EXPLORER
OPEN EDITORS
Java Overview
DAOType.java sr...
ADVANCED_JAVA
com
xyzcorp
demos
collections
concurrent
datetime
designp...
abstra...
classic
Clien... 1
DAOTyp...
MySQLR...
MySQLR...
OracleR...
OracleR...
OUTLINE
JAVA DEPENDENCIES
MAVEN PROJECTS
Java Overview
DAOType.java
src > main > java > com > xyzcorp > demos > designpatterns > abstractfactory > classic > DAOType.java > ...
1 package com.xyzcorp.demos.designpatterns.abstractfactory.classic;
2
3 /**
4 * User: Daniel Hinojosa (dhinojosa@evolutionnext.com)
5 * Date: 5/29/12
6 * Time: 10:11 PM
7 */
8 public enum DAOType {
9 MYSQL, ORACLE
10 }
11
master
0 56
Ln 11, Col 1
Spaces: 4
UTF-8
LF
Java

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());
   }
}
```

Defining `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.

src/main/java/com.xyzcorp.demos.functions.MyPredicate

```
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

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

Using `MyPredicate`

src/main/java/com.xyzcorp.functions.Functions#myFilter

```
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`

Testing `MyPredicate`

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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`.

Convert `MyPredicate` into a Lambda

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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);
```

Defining `MyFunction`

src/main/java/com.xyzcorp.demos.functions.MyFunction

```
public interface MyFunction<T, R> {
    public R apply(T t);
}
```

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]`

Using `MyFunction`

src/main/java/com.xyzcorp.demos.functions.Functions#myMap

```
public static <T, R> List<R> myMap(List<T> list, MyFunction<T, R>
myFunction) {
    ArrayList<R> result = new ArrayList<>();
    for (T t : list) {
        result.add(myFunction.apply(t));
    }
    return result;
}
```

Testing `MyFunction`

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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);
```

Convert `MyFunction` into a Lambda

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyMap

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
List<Integer> mapped = Functions.myMap(numbers, t -> t + 2);
```

Functional `flatMap`

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

Functional `flatMap` by example

1. Given List of `list`: `[1,2,3,4]`
2. Given a function `f`: `x → [x - 1, x, x + 1]`
3. When calling `flatMap` on a `list` with `f`: `[1,2,3,4].flatMap(f)`
4. Then a copy of the `list` should return: `[0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5]`

Using `MyFunction` with `flatMap`

src/main/java/com.xyzcorp.demos.functions.Functions#myFlatMap

```
public static <T, R> List<R> myFlatMap(List<T> list, MyFunction<T, List
<R>> myFunction) {
    ArrayList<R> result = new ArrayList<>();
    for (T t : list) {
        List<R> application = myFunction.apply(t);
        result.addAll(application);
    }
    return result;
}
```

Testing `MyFunction` with `flatMap`

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyMap

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
List<Integer> mapped = Functions.myFlatMap(numbers,
    t -> List.of(t - 1, t, t + 1));
```

Defining `MyConsumer`

src/main/java/com.xyzcorp.demos.functions.MyConsumer.java

```
package com.xyzcorp;

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

Notice that it does not return anything, a `void`

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.

Using `MyConsumer`

src/main/java/com.xyzcorp.demos.functions.Functions#myForEach

```
public static <T> void myForEach(List<T> list, MyConsumer<T> myConsumer)
{
    for (T t : list) {
        myConsumer.accept(t);
    }
}
```

Testing `MyConsumer`

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyForEach

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
Functions.myForEach(numbers, new MyConsumer<Integer>() {
    @Override
    public void accept(Integer x) {
        System.out.println(x);
    }
});
```

Convert `MyConsumer` into a Lambda

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyForEach

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
Functions.myForEach(numbers, x -> System.out.println(x));
```

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`

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

`forEach` with a method reference

Before:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyForEach

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
Functions.myForEach(numbers, new MyConsumer<Integer>() {
    @Override
    public void accept(Integer x) {
        System.out.println(x);
    }
});
```

With a lambda:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyForEach

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
Functions.myForEach(numbers, x -> System.out.println(x));
```

After:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMyForEach

```
List<Integer> numbers = Arrays.asList(4, 5, 7, 8);
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.

Static method with a method reference

Before:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAStaticMethod

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

With a Lambda:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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)));
```

With a Method Reference:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAStaticMethod

```
List<Integer> numbers =
   Arrays.asList(2, 4, 5, 1, 9, 15, 19,
                 21, 33, 78, 93, 10);

System.out.println(Functions.myMap(numbers, Math::abs));
```

Containing Type as a Method Reference

Before:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAContainingType

```
List<String> words = Arrays.asList("One", "Two", "Three", "Four");
List<Integer> result = Functions.myMap(words, new MyFunction<String,
    Integer>() {
    @Override
    public Integer apply(String s) {
        return s.length();
    }
});
System.out.println(result);
```

With a Lambda:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAContainingType

```
List<String> words = Arrays.asList("One", "Two", "Three", "Four");
List<Integer> result = Functions.myMap(words, s -> s.length());
System.out.println(result);
```

With a Method Reference:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAContainingType

```
List<String> words = Arrays.asList("One", "Two", "Three", "Four");
List<Integer> result = Functions.myMap(words, String::length);
System.out.println(result);
```

Owner of the Method Reference

- The owner of the method might be any super type
- Not always the type that you are using

Before:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAContainingTypeTrickQuestion

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

With a Lambda:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAContainingType

```
List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
        21, 33, 78, 93, 10);
System.out.println(Functions.myMap(numbers, integer -> integer.
toString()));
```

With a Method Reference:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#testMethodReferenceAContainingType

```
List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
        21, 33, 78, 93, 10);
System.out.println(Functions.myMap(numbers, Object::toString));
```

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

Method Reference with an Instance

Given a custom TaxRate class:

src/main/java/com.xyzcorp.demos.functions.TaxRate

```
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;
    }

    //Getters, toString, equals elided
}
```

Before:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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,
  new MyFunction<Integer, Double>() {
    @Override
    public Double apply(Integer subtotal) {
        return taxRate2016.apply(subtotal);
    }
}));
```

With a Lambda:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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)));
```

After:

src/test/java/com.xyzcorp.demos.functions.FunctionsTest#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, taxRate2016::apply));
```

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

Method Reference with an New Type

Before:

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

With A Lambda:

```
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)));
```

After:

```
List<Integer> numbers = Arrays.asList(2, 4, 5, 1, 9, 15, 19,
            21, 33, 78, 93, 10);
System.out.println(Functions.myMap(numbers, Double::new));
```

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

`new Double` is deprecated in favor of `Double.valueOf(..)`. This was used for demonstration.

Create `MySupplier`

```
package com.xyzcorp;

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

Compare the difference to `MyConsumer`

Create a `myGenerate` in *Functions.java*

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

Use `myGenerate` in *FunctionsTest.java*

```
package com.xyzcorp;

import org.junit.Test;

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

public class FunctionsTest {

    ...

    @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.

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

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

Multi-line Lambdas

```
package com.xyzcorp;

import org.junit.Test;

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

public class FunctionsTest {

    ...

    @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.

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));
}
```

Create Duplicated Code

An application for a closure is to avoid repetition.

```
package com.xyzcorp;

import org.junit.Test;

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

public class FunctionsTest {

    ...

    @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));
    }
}
```

Refactor Duplicated Code with a Closure

An application for a closure is to avoid repetition.

```
package com.xyzcorp;

import org.junit.Test;

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

public class FunctionsTest {

    ...

    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)));
    }
}
```

Lab: Functions

Step 1: Open *src/test/java/com.xycorp.exercises.function.FunctionExercises*

Step 2: Understand how to write a test with methods, `testSampleWithStandardJUnit` and `testSampleWithStandardJUnitAndAssertJAssertion` in case you are unfamiliar with testing

Step 3: Preferably using Test Driven Development, create a class called `MyTimer` with a static method called `measureTime`. `measureTime` should take a `Supplier` that returns an `Integer` representing how long the lambda will take when executed. `measureTime` should return a custom class called `TimeResult` that will have two methods:

- `getResult` - This will return the result of the lambda
- `getTime` - This will return the result of how long `measureTime` took

Here is an example of what it *may* look like:

```
var result = MyTimer.measureTime(() -> {
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 40;
});
```

Step 4: Extra Credit. Make it generic! What if the lambda is not returning an `Integer` can you make this so it uses any type?

`Optional`

Sir Richard Antony Hoare on the `null` reference

> 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.

`Optional` Empty

- The following represents an absence of value
- Fully Generic

```
Optional<Integer> middleName = Optional.empty();
```

`Optional` Non Empty

- The following represents an answer, a full representation of value

```
Optional<String> middleName = Optional.of("Hello");
```

Trapping Something Possibly `null`

- A commonly used technique to trap something that can possibly be `null`
- Returns an `Optional` describing the given value, if non-`null`, otherwise returns an empty `Optional`

src/test/java/com.xyzcorp.demos.optionals.OptionalsTest#testOptionalFromSomethingPossiblyNull

```
String possibleNull = getSomethingRandomlyNull();
Optional<String> optional = Optional.ofNullable(possibleNull);
if (possibleNull == null) assertThat(optional.isPresent()).isFalse();
else assertThat(optional.isPresent()).isTrue();
```

`Optional.get`

- Used to obtain the value of the `Optional`
- Typically the *unsafe way* to do so, although can be useful when known that it is present
  - e.g. after `filter(Optional::isPresent)`

The following returns `40L`

src/test/java/com.xyzcorp.demos.optionals.OptionalsTest#testGettingTheValueAndLucky

```
Optional<Long> optionalLong = Optional.of(40L);
optionalLong.get();
```

The following will throw a `NoSuchElementException`

```
Optional<Long> optionalLong = Optional.empty();
optionalLong.get();
```

Responsible retrieval using `orElse`

- A safe way to retrieve is `orElse` which will retrieve the value
- The following will return `-1`

src/test/java/com.xyzcorp.demos.optionals.OptionalsTest#testGettingTheValueTheGoodWayUsingGetOrElse

```
Optional<Long> optionalLong = Optional.empty();
Long result = optionalLong.orElse(-1L);
```

Lazy Retrieval with a `Supplier`

- Nearly the same as `orElse`

- Lazy alternative with `Supplier`

src/test/java/com.xyzcorp.demos.optionals.OptionalsTest#testGettingTheValueTheGoodWayUsingOrElseGet

```
Optional<Long> optionalLong = Optional.of(40L);
Long result = optionalLong.orElseGet(this::getDefaultAverage);
```

`ifPresentOrElse`

- Functional conditional handling whether present or empty
- First argument is a Consumer of the contained type of the `Optional`
- Second is a `Runnable` of a side effect for the otherwise case
- Perfect for test assertions

src/test/java/com.xyzcorp.demos.optionals.OptionalsTest#testGettingTheValueWithIfPresentOrElse

```
Optional<Long> optionalLong = Optional.of(40L);
optionalLong
    .ifPresentOrElse(
        x -> assertEquals(40L, x),
        () -> System.out.println("Not good"));
```

`Optional` and `map`

- `Optional` has `map` functionality
- Applies function to the value inside of a present `Optional`

```
Optional<Integer> i = Optional.of(40);
Optional<Integer> result = i.map(a -> a * 20);
```

`Optional` and `flatMap`

- `Optional` has `flatMap` functionality
- Applies a function to the value inside that also returns another `Optional` of a present `Optional`

```
Optional<Integer> i = Optional.of(40);
Optional<Integer> j = Optional.of(90);
Optional<Integer> result = i.flatMap(a -> j.map(b -> a * b));
```

`Optional` is not `Serializable`

> This answer is in response to the question in the title, "Shouldn't Optional be Serializable?" The short answer is that the Java Lambda (JSR-335) expert group considered and rejected it. That note, and this one and this one indicate that the primary design goal for Optional is to be used as the return value of functions when a return value might be absent. The intent is that the caller immediately check the Optional and extract the actual value if it's present. If the value is absent, the caller can substitute a default value, throw an exception, or apply some other policy. This is typically done by chaining fluent method calls off the end of a stream pipeline (or other methods) that return Optional values.
>
> It was never intended for Optional to be used other ways, such as for optional method arguments or to be stored as a field in an object. And by extension, making Optional serializable would enable it to be stored persistently or transmitted across a network, both of which encourage uses far beyond its original design goal.

Stack Overflow on `Optional` Serialization

Lab: `Optional`

Step 1: Open *src/test/java/com.xycorp.exercises.optionals.OptionalExercises*

Step 2: Read and understand the two `Map` instances: `europeanCountriesCapitals` and `europeanCapitalPopulation`

Step 3: In the test `testGettingGreeceCapital`, use `Optional` to safely read from `europeanCountriesCapitals` using `Optional`. Assert that the capital of `Greece` is `Athens`. *Do not use plain* `if` *statements*. This is an exercise for functional programming so use either `map`, `flatMap`, `filter`, `forEach`, or any of the `Optional` methods

Step 4: In the test `testGettingHungaryCapital`, use `Optional` to safely read from `europeanCountriesCapitals` using Optional. Assert that the capital of `Hungary` is not available. *Do not use plain* `if` *statements*. This is an exercise for functional programming so use either `map`, `flatMap`, `filter`, `forEach`, or any of the `Optional` methods.

Step 5: In the test `testGettingFromNorwayTheCapitalAndPopulation`, use `Optional` to safely read from `europeanCountriesCapitals` given a `String` of `Norway`. Whatever the result from the capital, use that capital to query from `europeanCapitalPopulation` and get the population. Return the pairing of the capital and the population. *Do not use plain* `if` *statements*. This is an exercise for functional programming so use either `map`, `flatMap`, `filter`, `forEach`, or any of the `Optional` methods. Consider creating a custom `Pair` to do so.

Step 6: In the test `testGettingFromGreeceTheCapitalAndPopulation`, use `Optional` to safely read from `europeanCountriesCapitals` given a `String` of `Greece`. Whatever the result from the capital, use that capital to query from `europeanCapitalPopulation` and get the population. Assert that the population of `Greece` is not available for population. *Do not use plain* `if` *statements*. This is an exercise for functional programming so use either `map`, `flatMap`, `filter`, `forEach`, or any of the `Optional` methods. Consider creating a custom `Pair` class to do so.

Step 7: In the test `testGettingFromNorwayTheCountryAndCapital`, use `Optional` to safely read from `europeanCountriesCapitals` given a `String` of `Norway`. Whatever the result from the capital, use that capital to query from `europeanCapitalPopulation` and get the population. Return a *triplet* of the capital and the population. *Do not use plain* `if` *statements*. This is an exercise for functional programming so use either `map`, `flatMap`, `filter`, `forEach`, or any of the `Optional` methods. Consider creating a custom `Triple` to do so.

Step 8: Refactor aggressively, make the methods look nice.

Step 9: Identify the closures.

Hint. In IntelliJ the closured variables are in purple

JUnit 5

- Latest project for JUnit 5
- Built atop JUnit Classic
- Applies Functional Programming
- Extension set called Jupiter Provides
  - Fixtures
  - Parameterized Tests
  - Tagging
  - more...

Running Tests on the Command Line

```
% mvn test
```

To run a specific test

```
% mvn -Dtest=MyTest test
```

Simple Test

Typical `import`

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
public class JUnit5SimpleTest {
    @Test
    public void myFirstTest() {
        assertEquals(2, 1 + 1);
    }
}
```

Standard Assertions

src/test/java/ com.xyzcorp.demos.junit5.JUnit5AssertionsTest#standardAssertions

```
@Test
@DisplayName("Standard Assertion Testing")
void standardAssertions() {
    assertEquals(2, 2);
    assertEquals(4, 4,
       "The optional assertion message is now the last parameter.");
    assertTrue(2 == 2, () ->
       "Assertion messages can be lazily evaluated -- "
       + "to avoid constructing complex messages unnecessarily.");
}
```

Grouped Assertions

src/test/java/ com.xyzcorp.demos.junit5.JUnit5AssertionsTest#groupedAssertions

```
@Test
@DisplayName("Grouped Assertion Testing")
void groupedAssertions() {
    // In a grouped assertion all assertions are executed, and any
    // failures will be reported together.

    Employee e = new Employee("Douglas", "Adams");
    assertAll("person",
            () -> assertEquals("Douglas", e.getFirstName()),
            () -> assertEquals("Adams", e.getLastName())
    );
}
```

Dependent Assertions

```
@Test
@DisplayName("Dependent Assertion Testing")
void dependentAssertions() {
    // Within a code block, if an assertion fails the
    // subsequent code in the same block will be skipped.
    Employee employee = new Employee("Douglas", "Adams");
    assertAll("Employee Properties",
            () -> {
                String firstName = employee.getFirstName();
                assertNotNull(firstName);

                // Executed only if the previous assertion is valid.
                assertAll("first name",
                        () -> assertTrue(firstName.startsWith("D")),
                        () -> assertTrue(firstName.endsWith("s"))
                );
            },
            () -> {
                // Grouped assertion, so processed independently
                // of results of first name assertions.
                String lastName = employee.getLastName();
                assertNotNull(lastName);

                // Executed only if the previous assertion is valid.
                assertAll("last name",
                        () -> assertTrue(lastName.startsWith("A")),
                        () -> assertTrue(lastName.endsWith("s"))
                );
            }
    );
}
```

Exception Testing

```
@Test
void exceptionTesting() {
    Throwable exception = assertThrows(IllegalArgumentException.class,
        () -> {
           throw new IllegalArgumentException("a message");
        }
    );

    assertEquals("a message", exception.getMessage());
}
```

Timeout Testing

```
@Test
void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofSeconds(5), () -> {
        Thread.sleep(4000);
    });
}
```

```
@Test
void timeoutNotExceededWithResult() {
    // The following assertion succeeds, and
    // returns the supplied object.
    String actualResult = assertTimeout(
        ofSeconds(2), () -> "a result");
    assertEquals("a result", actualResult);
}
```

```
@Test
void timeoutNotExceededWithMethod() {
    Employee employee = new Employee("Carl", "Sagan");
    // The following assertion invokes
    // a method reference and returns an object.
    String lastName = assertTimeout(ofSeconds(2),
       employee::getLastName);
    assertEquals("Sagan", lastName);
}
```

Disabling Tests

Disabling the Test on the class level

```
@Disabled
public class JUnit5DisabledTest {

    @Test
    void testOne() {
        fail("This will never come to light since the whole test is
disabled");
    }

    @Test
    void testTwo() {
        fail("This will never come to light since the whole test is
disabled");
    }
}
```

Disabling the Test on the method level

```
@Test
@Disabled("because I don't want the test to fail")
void failingTest() {
    fail("a failing test");
}
```

Lab: JUnit 5

Step 1: Open *src/test/java/com.xycorp.exercises.junit5.FindSmallestIntegerToZeroTest*

Test Driven Development is defined using the following rules:

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

Test Driven Development is meant to be done in an incremented fashion, starting small and working your way up in complexity. The goal of Test Driven Development is seek out red bar tests and not necessarily green bar tests

Step 2: Use Test Driven Development and JUnit 5 Parameterized Testing and create a class called `FindSmallestIntegerToZero` with the following specification:

https://cyber-dojo.org

> Given a list of integers find the closest to zero. If there is a tie, choose the positive value.

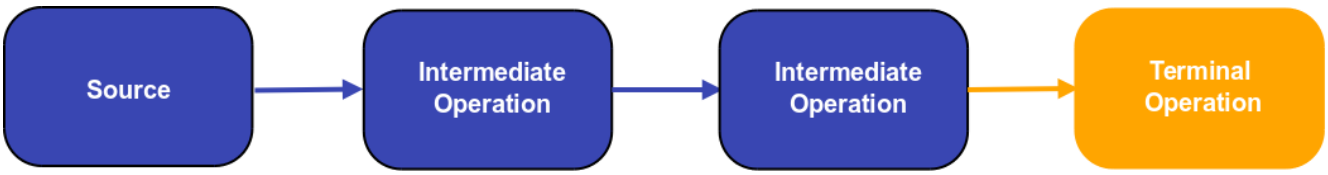
Step 3: Create a test that ensure that the list provided is not `null` using JUnit 5's Exception Handling Assertions

Streams

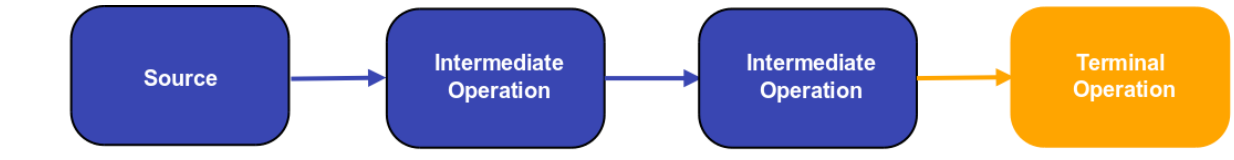
`Streams` differ from Collections in the following ways:

- 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



Creation

Creating a Stream using `of`

- `of` is the fastest way to create a `Stream`
- Does not require an initial collection

- Returns a sequential ordered stream whose elements are the specified values

src/test/java/com.xyzcorp.demos.streams.StreamsTest#createAStreamFromElements

```
Stream<Integer> integerStream = Stream.of(3, 4, 5);
List<Integer> result =
    integerStream
        .map(x -> x + 2)
        .collect(Collectors.toList());
```

Results in:

```
[5, 6, 7]
```

Creating a Stream from a `Collection`

- The `stream()` call converts nearly any `Collection` 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
- ***The stream is lazily evaluated until you call a terminal operation***

Creating a Basic Stream from `List`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testBasicStreamFromList

```
List<Integer> integers = Arrays.asList(1, 4, 5);
List<Integer> result = integers.stream()
     .map(x -> x + 1).collect(Collectors.toList());
```

Results In:

```
[2, 5, 6]
```

Creating a Basic Stream from `Set`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testBasicStreamFromSet

```
Set<Integer> integers = Set.of(30, 10, 12, 4);
Set<Integer> result =
    integers.stream()
            .map(x -> x + 1)
            .collect(Collectors.toSet());
```

Results In:

```
Set.of(31, 11, 13, 5)
```

`Collectors` Don't Need to Match to the `Stream`

- Once you have `Stream` you can collect it to whatever type you'd like
- Regardless of how you created the `Stream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testBasicStreamFromSetToList

```
Set<Integer> integers = Set.of(30, 10, 12, 4);
List<Integer> result =
    integers.stream()
            .map(x -> x + 1)
            .collect(Collectors.toList());
```

Creating a Basic Stream from `Map`

- Creating a Stream from `Map`, call `entrySet`
- This will return a set of `Map.Entry` elements

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testBasicStreamFromMap

```
Map<String, String> countriesAndCapitals =
    Map.of("Denmark", "Copenhagen", "China", "Beijing");

List<String> result = countriesAndCapitals
    .entrySet()
    .stream()
    .map(e -> e.getKey() + "~" + e.getValue())
    .collect(Collectors.toList());
```

Results In:

```
[China~Beijing, Denmark~Copenhagen]
```

Creating a Basic Stream from an Array of Objects

- Creating a Basic Stream from an Array of Objects will yield a `Stream<T>`
- If they were primitive, they'd return a *Specialized Stream* for that type (see later)

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testBasicStreamFromArray

```
String[] stringArray = new String[]{"Foo", "Bar", "Baz", "Bam"};
Stream<String> stream = Arrays.stream(stringArray);
List<String> result =
    stream
          .map(x -> x + "!")
          .collect(Collectors.toList());
```

The return type ,*in this case*, from `Arrays.stream` is a `Stream<String>`

Specialized Streams

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

Specialized `IntStream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedIntStream

```
int result = IntStream.of(30, 12, 50).map(x -> x * 3).sum();
```

Specialized `LongStream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedIntStream

```
long result = LongStream.of(30, 12, 50).map(x -> x * 3).sum();
```

Specialized `DoubleStream`

/src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedDoubleStream

```
double result = DoubleStream.of(30, 12, 50).map(x -> x * 3).sum();
```

Creating a `Stream` from an Array of Primitives

Recall previously, the following returns a `Stream<String>`

```
String[] stringArray = new String[]{"Foo", "Bar", "Baz", "Bam"};
Stream<String> stream = Arrays.stream(stringArray);
```

But when dealing with primitives, the following returns an `IntStream`

```
int[] primitiveIntArray = new int[]{1, 3, 4, 5};
IntStream stream = Arrays.stream(primitiveIntArray);
```

Recall `IntStream` is specialized for integers

Creating Specialized Streams with `Builder`

- We can use `add` continually to add elements to builder
- Add as many elements
- Call `build` when done
- No calls for `add` are allowed after `build`

```
LongStream longStream =
    LongStream.builder().add(40L).add(10L).add(20L).build();
longStream
    .average()
    .ifPresentOrElse(avg -> System.out.printf("Average: %2.2f", avg),
        () -> System.out.println("No Average"));
```

Creating Specialized Streams with `Builder` and `accept`

- `add` offers the ability to chain calls
- `accept` returns `void` and operate one line at a time
- Use `build` when done
- No calls for `add` and `accept` are allowed after `build`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedLongStreamWithBuilderAndAccept

```
LongStream.Builder builder =
    LongStream.builder().add(40L).add(10L).add(20L);
builder.accept(31L);
builder.accept(41L);
builder.accept(51L);
builder.accept(61L);
LongStream longStream = builder.build();
String result =
    longStream
        .boxed()
        .map(String::valueOf)
        .collect(Collectors.joining(", "));
```

Specialized Streams have `range`

`range` is available in all Specialized Streams

src/main/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedIntStreamRange

```
int result = IntStream.range(0, 5).sum(); //10
```

`rangeClosed` is also available

src/main/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedLongStreamRangeClosed

```
long result = LongStream.rangeClosed(0, 5).sum(); //15
```

Converting

Converting from Specialized Streams to General Streams

- Use the method `mapToObj`
- Takes a primitive element and converts it to an `Object`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFromSpecializedStreamToGeneralStream

```
int[] primitiveIntArray = new int[]{1, 3, 4, 5};
IntStream stream = Arrays.stream(primitiveIntArray);
List<Integer> result =
    stream
          .map(x -> x + 1)
          .mapToObj(Integer::valueOf)
          .collect(Collectors.toList());
```

Converting from Specialized Streams to General Streams with `boxed`

- Offers the same functionality as `mapToObj`
- Cleaner alternative

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFromSpecializedStreamToGeneralStreamWithBoxed

```
int[] primitiveIntArray = new int[]{1, 3, 4, 5};
List<Integer> result =
    Arrays.stream(primitiveIntArray)
          .map(x -> x + 1)
          .boxed()
          .collect(Collectors.toList());
```

Converting from General Streams to Specialized Streams

Java offers helpful methods to convert to Specialized Streams

- `mapToInt`
- `mapToLong`
- `mapToDouble`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFromGeneralStreamToSpecializedIntStream

```
Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
IntStream intStream = integerStream.mapToInt(x -> x);
int result = intStream.sum();
```

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFromGeneralStreamToSpecializedLongStream

```
Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
LongStream longStream = integerStream.mapToLong(x -> x);
long result = longStream.sum();
```

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFromGeneralStreamToSpecializedDoubleStream

```
Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
DoubleStream doubleStream = integerStream.mapToDouble(x -> x);
double result = doubleStream.sum();
```

Intermediate Operators

Mapping a `Stream`

`map` applies a function to everything within a `Stream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testMap

```
List<Integer> result = Stream
   .of(1, 2, 3, 4)
   .map(x -> x * 2)
   .collect(Collectors.toList()); //List(2,4,6,8)
```

Filtering a `Stream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFilter

```
List<Integer> result = Stream
  .of(1, 2, 3, 4)
  .filter(x -> x % 2 == 0)
  .collect(Collectors.toList());
```

FlatMapping a `Stream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFlatMap

```
Stream<Integer> stream =
    Stream.of(1, 2, 3, 4)
          .flatMap(x -> Stream.of(-x, x, x + 2));
List<Integer> result =
    stream.collect(Collectors.toList());
```

Results in:

```
List.of(-1, 1, 3, -2, 2, 4, -3, 3, 5, -4, 4, 6);
```

`flatMap` as a Cleaner

- We will create a helper method, `safeCharAt`
  - If we can retrieve the `char` at index it will return `Optional.present`
  - Otherwise it will be `empty`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#safeCharAt

```
private static Optional<Character> safeCharAt(String s, int index) {
    try {
        return Optional.of(s.charAt(index));
    } catch (java.lang.StringIndexOutOfBoundsException e) {
        return Optional.empty();
    }
}
```

- `Optional.stream`
  - Extremely useful
  - Is short for `o -> o.stream()` where `o` is an `Optional<T>`
    - `o.stream` will return Stream of the element inside the `Optional` if present
    - `empty` if the `Optional` is empty

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFlatMapAsCleaner

```
var result =
  Stream.of("Apple", "Orange", "", "Banana", "Tomato", "Grapes", "")
        .map(s -> safeCharAt(s, 0))
        .flatMap(Optional::stream)
        .collect(Collectors.toList());
```

This returns:

```
List.of('A', 'O', 'B', 'T', 'G')
```

`peek` into your `Stream`

- `peek`
  - Accepts a `Consumer`
  - Provides an opportunity to perform a side effect
  - Typically, a log, or `System.out.println`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#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());
```

Limiting the Number of Elements

- Returns a stream consisting of the elements of this stream
- Truncated to be no longer than maxSize in length.
- Particularly useful for *infinite* streams
- Optimized to only work on those number of elements

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testLimit

```
Stream<Integer> integerStream = Stream.iterate(0, x -> x + 1);
List<Integer> result =
    integerStream
        .map(x -> x + 4)
        .limit(10)
        .collect(Collectors.toList());
```

Skipping Elements

`skip` returns a stream consisting of the remaining elements of this stream after discarding the first `n` elements of the stream

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSkip

```
List<Integer> result =
IntStream.range(0, 20).boxed().skip(10).collect(Collectors.toList());
```

Results In:

```
List.of(10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

Filtering Distinct Elements

`distinct` returns a stream consisting of the distinct elements (according to `Object.equals(Object)`) of this stream

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testDistinct

```
List<String> distinct = Stream
   .of("Right", "Left", "Right", "Right")
   .distinct()
   .collect(Collectors.toList());
```

Results In:

```
["Left", "Right"]
```

Sorting a Stream

- `sorted` will use a `Comparator<T>` to determine how to sort a `Stream`
- The following example will sort by the length of the `String`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSortedWithComparator

```
List<String> list =
stream
    .sorted(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return Integer.compare(o1.length(), o2.length());
        }
    })
    .collect(Collectors.toList());
```

The result is:

```
List.of("Kiwi", "Apple", "Orange", "Banana", "Tomato", "Grapes");
```

`Comparator` methods

- `Comparator` has methods that aid in common tasks
  - Comparing common types: `int`, `double`
  - Multiple levels of comparision: first name then last name
  - Extracting certain fields from objects

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSortedWithComparatorUtility

```
Stream<String> stream =
    Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes", "Kiwi");

List<String> list = stream
    .sorted(Comparator.comparingInt(String::length))
    .collect(Collectors.toList());
```

Advanced editors often help with suggestions

Identity Function Defined

$$f(x) = x$$

Identity Function Definition

> 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;
}
```

Combination Comparators

`thenComparing` offers possibility to chained comparisons

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSortedWithNestedComparators

```
Stream<String> stream =
    Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes", "Plum"
        , "Kiwi");
System.out.println(stream
    .sorted(Comparator
        .comparing(String::length)
        .thenComparing(x -> x))
    .collect(Collectors.toList()));
```

Using the Identity Function:

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSortedWithNestedComparators

```
Stream<String> stream =
    Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes", "Plum"
        , "Kiwi");
System.out.println(stream
    .sorted(Comparator
        .comparing(String::length)
        .thenComparing(Function.identity()))
    .collect(Collectors.toList()));
```

Terminal Operators

Counting Elements

- Terminal operation
- Counts the number of elements in `Stream`
- Supported in both general and specialized streams

/src/test/java/com.xyzcorp.demos.streams.StreamsTest#testGeneralStreamCount

```
long count = Stream.of(12, 4, 10).count();
```

/src/test/java/com.xyzcorp.demos.streams.StreamsTest#testSpecializedStreamCount

```
long count = Stream.of(12, 4, 10).count();
```

Reducing without a `seed`

- Terminal Operation
- Reduction iteratively calls a BiFunction to calculate the next iteration
- Continues until exhausted
- Returns an `Optional` in case it received an empty `Stream`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testReduceWithoutASeed

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

The `System.out.println` renders:

```
total: 1, next: 2
total: 3, next: 3
total: 6, next: 4
total: 10, next: 5
total: 15, next: 6
```

The result is:

```
Optional.of(21);
```

Reduce with a seed

- Reduction has a seed
- This means that it always has an answer

src/test/java/com.xyzcorp.demos.streams.StreamsTest#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;
});
```

The `System.out.println` renders:

```
total: 1, next: 2
total: 3, next: 3
total: 6, next: 4
total: 10, next: 5
total: 15, next: 6
```

The result is:

```
21
```

The above answer is not an `Optional`

Finding the First Element

- `findFirst` returns the first element of a `Stream`
- Returns an `Optional<E>`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFindFirstSuccessful

```
Optional<Integer> first = Stream.of(1, 2, 4, 5).findFirst();
```

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testFindFirstUnSuccessful

```
Optional<Integer> first = Stream.<Integer>empty().findFirst();
```

Specialized Streams Terminal Operations

The major benefit for a specialized stream, like `IntStream` are the terminal operations

- `average`
- `sum`
- `summaryStatistics`
- `max`
- `min`

Sample of `summaryStatistics`

src/test/com.xyzcorp.demos.streams.StreamsTest#testSummaryStatistics

```
IntStream intStream = IntStream.of(12, 19, 40, 60, 100, 200, 15, 0, 3);
System.out.println(intStream.summaryStatistics());
```

Results in:

```
IntSummaryStatistics{count=9, sum=449, min=0, average=49.888889, max=
200}
```

Infinite Streams

`iterate`

- Returns an infinite sequential ordered Stream produced by iterative application of a function `f`
- First applies to an initial element seed, producing a Stream consisting of `seed`, `f(seed)`, `f(f(seed))`
- The first element (position 0) in the `Stream` will be the provided `seed`
- For $n > 0$, the element at position `n`, will be the result of applying the function `f` to the element at position `n - 1`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testIterate

```
Stream<Integer> iterate = Stream.iterate(0,
            integer -> integer + 3);
List<Integer> result = iterate.limit(5).collect(Collectors.toList());
```

Results In:

```
[0, 3, 6, 9, 12]
```

`iterate` with a `Predicate`

- Iterate can also have a predicate as a condition for the next element
- The resulting sequence may be empty if the `hasNext` predicate does not hold on the seed value
- The stream will continue until the `Predicate` dictates that it should terminate
- Use Case: Custom ranges

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testIterateWithPredicate

```
List<Integer> result =
   Stream
      .iterate(0, x -> x <= 5, integer -> integer + 1)
      .collect(Collectors.toList());
```

Results In:

```
[0, 1, 2, 3, 4, 5]
```

`generate`

- `generate` lazily invokes a function when it requires the next element
- Suitable for constant streams

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testGenerate

```
List<LocalDateTime> result =
            Stream.generate(LocalDateTime::now)
                  .flatMap(x -> Stream.of(x, x.plusYears(10)))
                  .limit(5)
                  .collect(Collectors.toList());
```

Results In:

```
[2017-11-06T12:52:04.499934, 2027-11-06T12:52:04.499934,
 2017-11-06T12:52:04.508439, 2027-11-06T12:52:04.508439,
 2017-11-06T12:52:04.508490]
```

Common Collectors

Grouping

- Returns a `Collector` implementing a "group by" operation on input elements of type T
- Grouping elements according to a classification function, and returning the results in a `Map`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testGroupingBy

```
IntStream stream = IntStream.range(0, 10);
Map<Boolean, List<Integer>> groups =
    stream.boxed()
          .collect(Collectors.groupingBy(i -> i % 2 == 0));
```

The result is:

```
{false=[1, 3, 5, 7, 9], true=[0, 2, 4, 6, 8]}
```

Partitioning

- Returns a `Collector` which partitions the input elements according to a `Predicate`
- Organizes them into a `Map<Boolean, List<T>>`
- The returned Map always contains mappings for both `false` and `true` keys.

src/test/java/com.xyzcorp.demos.streams.StreamsTest#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));
```

The result is:

```
{false=[Banana, Tomato, Grapes], true=[Apple, Orange]}
```

Joining

- Returns a Collector that concatenates the input elements
- Separated by the specified delimiter
- With the specified prefix and suffix, in encounter order
- Elements must be a subtype of `CharSequence`

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testJoining

```
Stream<String> stream =
    Stream.of("Apple", "Orange", "Banana", "Tomato", "Grapes");
String result = stream.collect(Collectors.joining(", ", "{", "}"));
```

The result is:

```
{Apple, Orange, Banana, Tomato, Grapes}
```

`toMap`

- Just like you can create a `List` or `Set` as collector.
- You can also create a `Map` using `toMap`
- Keys and values are the result of applying the provided mapping functions to the input elements

Given a setup of objects like an `Order` and `OrderItem` relationship:

/src/main/java/com.xyzcorp.demos.streams.Order

```
public class Order {
    private final String firstName;
    private final String lastName;
    private final String city;
    private final String state;
    private final List<OrderItem> orderItems;
}
```

/src/main/java/com.xyzcorp.demos.streams.OrderItem

```
public class OrderItem {
    private final int quantity;
    private final Product product;
    private final Order order;
}
```

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testToMap

```
Map<String, Integer> result = orders
   .stream()
   .collect(Collectors
       .toMap(Order::getFirstName,
           o -> o.getOrderItems()
                 .stream()
                 .mapToInt(OrderItem::getQuantity).sum()));
```

There is a distinct difference between `toMap` and `groupingBy`. `toMap` is intended to hold the same number of entries as the `Stream`. `groupingBy` is intended as an aggregation.

Custom Collectors

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);
```

Create your own `collect`

- The complete custom `collect`:
  - `Supplier` for the initial element
  - `BiConsumer` for how to bring in individual elements
  - `BiConsumer` for how to bring in different collections

```
@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) {
           integers.add(integer);
       }
   }, new BiConsumer<List<Integer>, List<Integer>>() {
       @Override
       public void accept(List<Integer> left, List<Integer> right) {
          left.addAll(right);
       }
   });
   System.out.println("Ending with the result = " + result);
}
```

Converting `collect` with Lambdas

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testBasicCustomCollector

```
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(ArrayList::new, ArrayList::add, ArrayList::addAll);
System.out.println("Ending with the result = " + result);
```

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

Using `parallel` in Streams

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testParallelCollections

```
List<Integer> collect =
    IntStream.range(0, 10)
             .boxed()
             .parallel()
             .map(x -> x + 1)
             .peek(x-> System.out.println("parallel:" + Thread
.currentThread().getName()))
             .collect(Collectors.toList());
System.out.println(collect);
```

Grouping Concurrently

- Returns a concurrent Collector implementing a "group by" operation on input elements of type `T`
- Groups elements according to a classification function.
- This is a concurrent and unordered Collector.
- The classification function maps elements to some key type `K`
- The collector produces a `ConcurrentMap<K, List<T>>` whose keys are the values resulting from applying the classification function to the input elements, and whose corresponding values are Lists containing the input elements which map to the associated key under the classification function.
- There are no guarantees on the type, mutability, or serializability of the ConcurrentMap or List objects returned, or of the thread-safety of the List objects returned.

src/test/java/com.xyzcorp.demos.streams.StreamsTest#testGroupingByConcurrent

```
IntStream stream = IntStream.range(0, 10);
Map<Boolean, List<Integer>> groups =
    stream.parallel()
          .boxed()
          .collect(Collectors.groupingByConcurrent(i -> i % 3 == 0));
System.out.println(groups);
```

Results In:

```
{false=[1, 2, 8, 7, 4, 5], true=[0, 9, 3, 6]}
```

The above result is a `java.util.concurrent.ConcurrentHashMap`

Lab: Streams

Step 1: Open *src/test/java/com.xycorp.exercises.streams.StreamsExercises*

Step 2: In the test `testConvertIntegerListToHexidecimal`, initialize `Arrays.asList(1,50,100,200,921)` and convert that list to a `List` of hexadecimals. Research how to convert an `Integer` to hexadecimal. *Do not use* `while` *or* `for` *loops or any old style iteration.*

Step 3: In the test `testConvertIntegerListToDouble`, initialize `Arrays.asList(1,50,100,200,921)` and convert that list to a `List` of `Double`. *Do not use* `while` *or* `for` *loops or any old style iteration.*

Step 4: In the test `testFactorialUsingStreams`, test a class that you create that will calculate the factorial of a number. *Do not use* `while` *or* `for` *loops or any old style iteration.*

Step 5: In the test `testFindTopFiveEmployees`, concatenate two `Streams` that read from `jkRowlingEmployees` and `georgeLucasEmployees`, sort the combined streams by `salary` in descending order, find the top five, collect them as a string with a new line. *Do not use* `while` *or* `for` *loops or any old style iteration.*

Step 6: In the test `testAllEmployeesSalaryWithManager`, give the reference `managers` only, find the sum of all the employees and the managers. **You can only use the `managers` reference**. *Do not use* `while` *or* `for` *loops or any old style iteration.*

Step 7: Which managers employees are paid more? J.K. Rowling or George Lucas? In `testWhichManagersEmployeesArePaidMore` find out. *Do not use* `while` *or* `for` *loops or any old style iteration.*

Step 8: In `testCreateAMapOfEmployeeKeyWithManagerValue` create a Map where the keys are all the employees (non-managers) and the values are their managers *Do not use* `while` *or* `for` *loops or any old style iteration.*

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

`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

Terrible Readability of `java.util.Calendar`

```
> 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.S
impleTimeZone[id=America/New_York,offset=-18000000,dstSaving=3600000
,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startD
ayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDa
y=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayOfWeek=1,mini
malDaysInFirstWeek=1,ERA=1,YEAR=2014,MONTH=2,WEEK_OF_YEAR=10,WEEK_OF_MON
TH=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=82,ZONE_OFF
SET=-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

`Instant` Resolution!

An `Instant` can be resolved as 1.844674407x10[19] seconds or 584542046090 years!

`Instant` Exemplified

/src/test/java/com.xyzcorp.demos.datetime.DatesTest#testInstant

```
Instant instant = Instant.now();
System.out.println("instant = " + instant);
System.out.println(instant.getEpochSecond());
System.out.println(instant.getNano());
```

`Enum`

`Month` and `DayOfWeek`

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

`Month` and `DayOfWeek`

```
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
```

Adding `ChronoUnit` to `Instant`

src/test/java/com.xyzcorp.demos.datetime.DatesTest#testChronoUnitAddToInstant

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

`ChronoUnit` to Determine Time Between

src/test/java/com.xyzcorp.demos.datetime.DatesTest#testChronoUnitBetween

```
long between = ChronoUnit.DAYS.between
    (Instant.parse("2019-01-05T12:00:00.0Z"),
        Instant.parse("2019-01-01T12:00:00.0Z"));
System.out.println("between = " + between);
```

The above uses `parse` to parse an ISO-8601 formatted `String`

`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
```

src/test/java/com.xyzcorp.demos.datetime.DatesTest#testChronoField

```
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

Creating a `LocalDate`

```
LocalDate february20th = LocalDate.of(2014, Month.FEBRUARY, 20);
System.out.println(february20th);
```

Creating a `LocalTime`

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

Creating a `LocalDateTime`

```
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 using Java Date Time API

```
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 using the Java Date Time API

- 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 using the Java Date Time API

```
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

`Duration` and `Period`

- 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`

`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));
```

`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
```

Lambda Capable `TemporalAdjuster`

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);
}
```

A Festive Example

```
@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
}
```

Simple Parsing Example

```
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();
```

Lab: Date Time

Step 1: Open *src/test/java/com.xycorp.exercises.datetime.DateTimeExercises*

Step 2: In the test `testIterateWithThreeDays`, initialize the current date time and and return the next three days using functional programming. For example, if today is 2019-10-08T16:05:11 then return a collection with:

2019-10-08T16:05:11
2019-10-09T16:05:11
2019-10-10T16:05:11

Step 3: In the test `testWhatDateTimeIsItInBuenosAiresArgentina`, display the current time in Buenos Aires Argentina.

Step 4: In the test `testFindAllAmericasTimeZones`, find all the time zones in the "America", remove the `America` from the Time Zone identifier and sort by the name of the time zone.

Step 5: In the test `testFindAllTimeZonesInAntarctica`, find all the Antarctica Time Zones, sort by their distance from UTC in ascending order, and display nicely the offset, the zone, and whether it adheres to Daylight Saving Time or not.

Generics

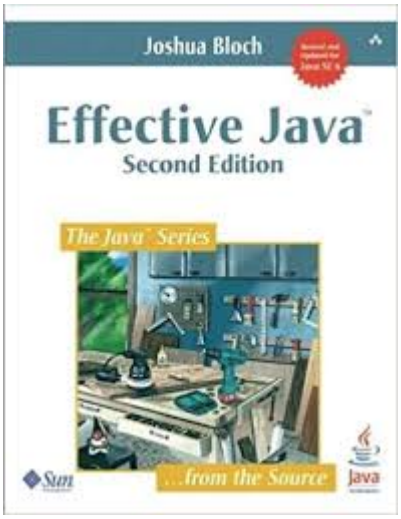
- Generics
- Get Put Principles
- Wildcards

Static vs. Dynamic

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.

Effective Java Item 26



Item 26: Favor generic types

It is generally not too difficult to parameterize your collection declarations and make use of the generic types and methods provided by the JDK. Writing your own generic types is a bit more difficult, but it's worth the effort to learn how.

Generic Terms

Term	Example
Parameterized Type	`List<String>`
Actual Type Parameter	`String`
Generic Type	`List<E>`
Formal Type Parameter	`E`
Unbounded wildcard type	`List<?>`
Raw Type	`List`
Bounded Type Parameter	`<E extends Number>`
Recursive Type Bound	`<T extends Comparable<T>>`
Bounded Wildcard Type	`List<? extends Number>`
Generic Method	`static <E> List<E> asList(E[] a)`
Type Token	`Integer.class`

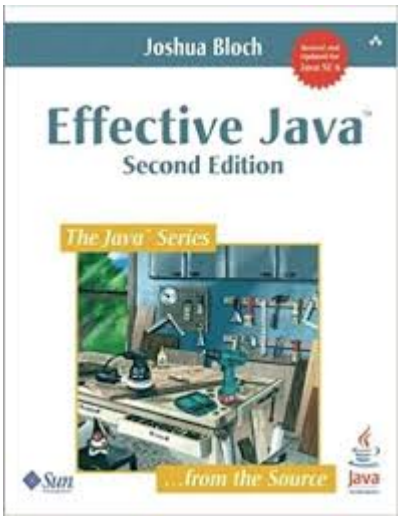
Before Generics

- Terrible verbosity
- Extensive Casting
- Raw Type

src/test/java/com.xyzcorp.demos.generics.GenericsBasicsTest#testRawTypes

```
List list = new ArrayList();
list.add("Foo");
list.add(4);
list.add(10.0);
Object object = list.get(0); //Object
if (object instanceof String) {
    String string = (String) object;
    string.substring(0, 1); //Substring belongs to String
}
```

Effective Java Item 23



Item 23: Don’t use raw types in new code

First, a few terms. A class or interface whose declaration has one or more *type parameters* is a *generic* class or interface [JLS, 8.1.2, 9.1.2]. For example, as of release 1.5, the `List` interface has a single type parameter, `E`, representing the element type of the list. Technically the name of the interface is now `List<E>` (read “list of E”), but people often call it `List` for short. Generic classes and interfaces are collectively known as *generic types*.

Before Generics Another Example

Another example of extensive generics

src/test/java/com.xyzcorp.demos.generics.GenericsBasicsTest#testAnotherWithCastingWithRawTypes

```
List words = new ArrayList(); //raw type
words.add("Hello ");
words.add("world!");
String s = ((String) words.get(0)).substring(1, 2) +
        ((String) words.get(1)).substring(2, 4);
System.out.println(s);
```

Eliminating Casting

The previous examples looks like this with generics

src/test/java/com.xyzcorp.demos.generics.GenericsBasicsTest#testEliminationOfCasting

```
List<String> words = new ArrayList<>(); //JDK 5- 10
words.add("Hello ");
words.add("world!");
String s = words.get(0).substring(1, 2) +
        words.get(1).substring(2, 4);
System.out.println(s);
```

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
```

Boxing Before Generics

- Before Generics there was explicit changing between primitives and `Object`
- An Object was required before putting it into a `Collection`

src/test/java/com.xyzcorp.demos.generics.GenericsBasicsTest#testAutomaticBoxingOfPrimitivesBefore

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

Boxing After Generics

- Cleaner alternative
- No casting

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
int n = ints.get(0);
System.out.println(n);
```

Container of a Container

- Generics keep the type, and the type system that the right type is returned
- This allow the type system to aid us in programming

```
List<List<Integer>> ints = new ArrayList<>();
ints.add(Arrays.asList(4, 10, 11, 12));
ints.add(Arrays.asList(5, 9, 3, 42));
Integer actual = ints.get(0).get(1);
```

Using Generics with static methods

- When needed apply the generic before the method
- This adds clarity to what specific type you need
- Overrides the type inferencer

src/test/java/com.xyzcorp.demos.generics.GenericsBasicsTest#testOverrideStaticGenericTypes

```
Number n = Arrays.<Number>asList(5, 1, 3, 3, 6, 10).get(0);
```

Type Erasure

- Types are erased at runtime
- `List<String>` is just `List`
- `List<Integer>` is just `List`
- Generic types are non-reifiable, meaning they are not fully available at runtime

src/test/java/com.xyzcorp.demos.generics.GenericsBasicsTest#testErasure

```
System.out.format("Runtime type of ArrayList<String>: %s\n",
        new ArrayList<String>().getClass());
System.out.format("Runtime type of ArrayList<Long>  : %s\n",
        new ArrayList<Long>().getClass());
```

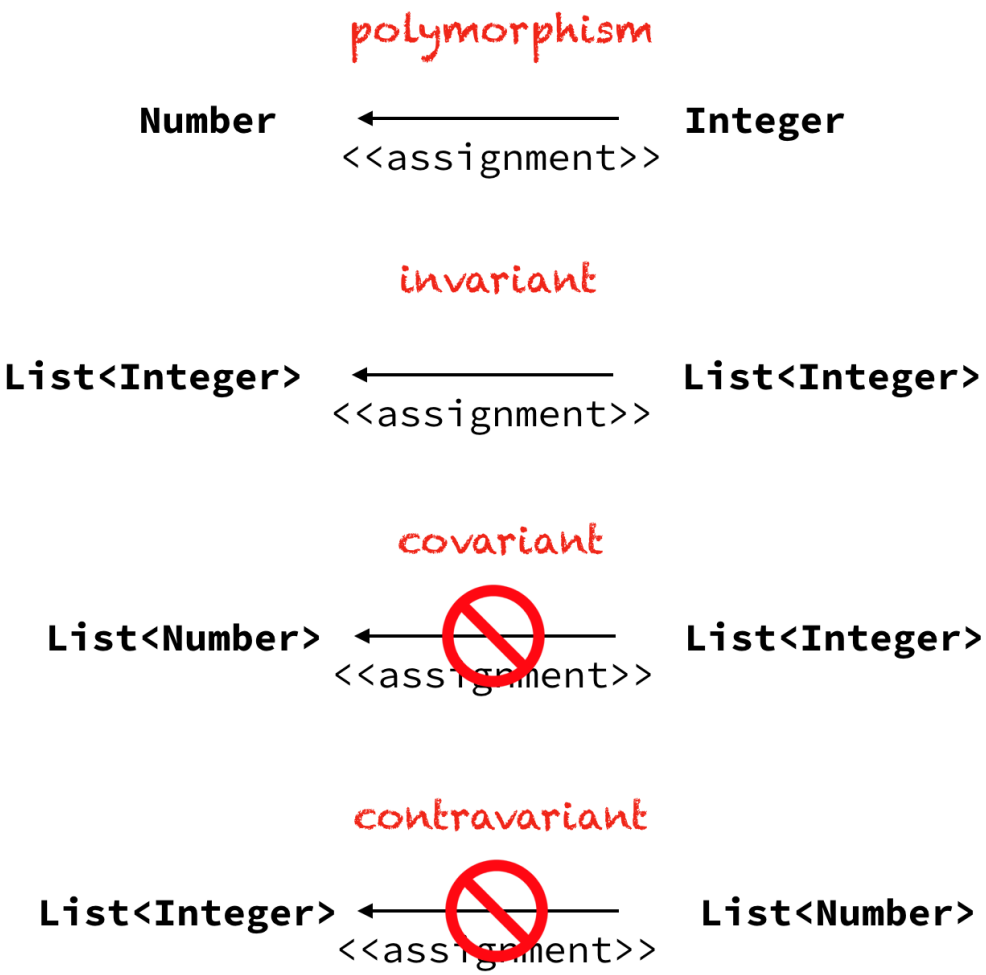
Results In:

```
Runtime type of ArrayList<String>: class java.util.ArrayList
Runtime type of ArrayList<Long>  : class java.util.ArrayList
```

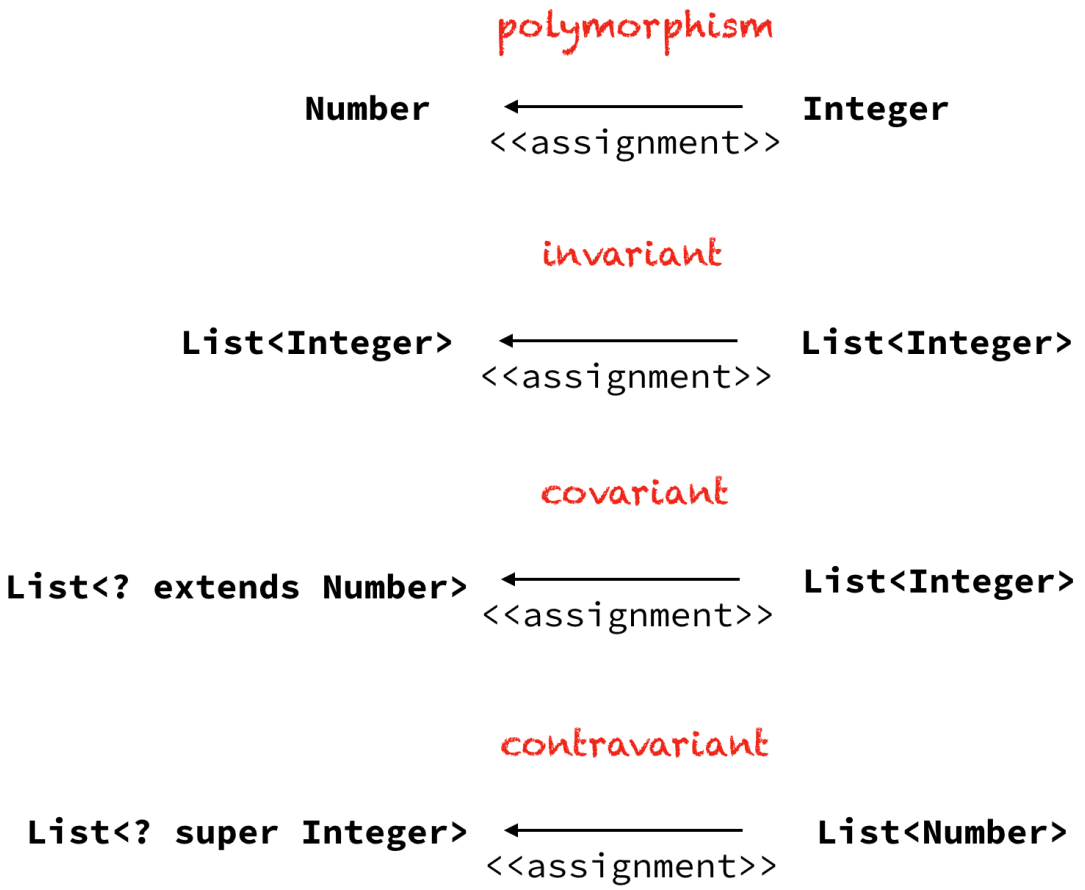
Assignment of Generic Types

- As default, `List<Integer>` can only be assigned to a `List<Integer>`
- Special Consideration for assignments where `List<Integer>` gets assigned a `List<Object>`
- ...or the other way around: `List<Object>` gets assigned a `List<Integer>`

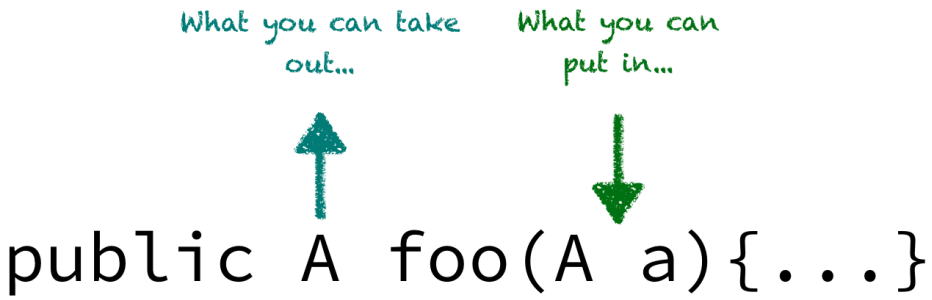
Variant Relationships



Variant Relationships



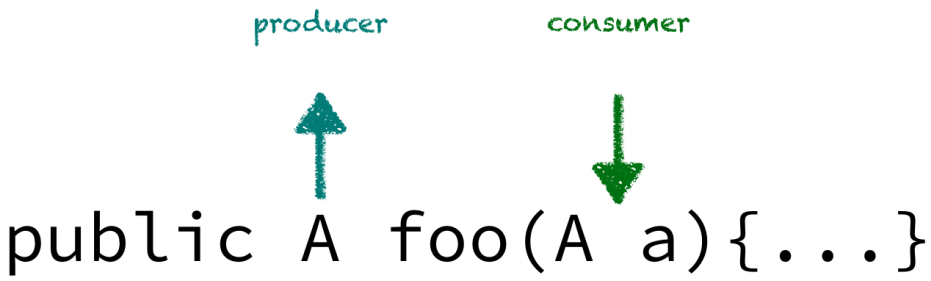
Method Return Types Vs. Parameters



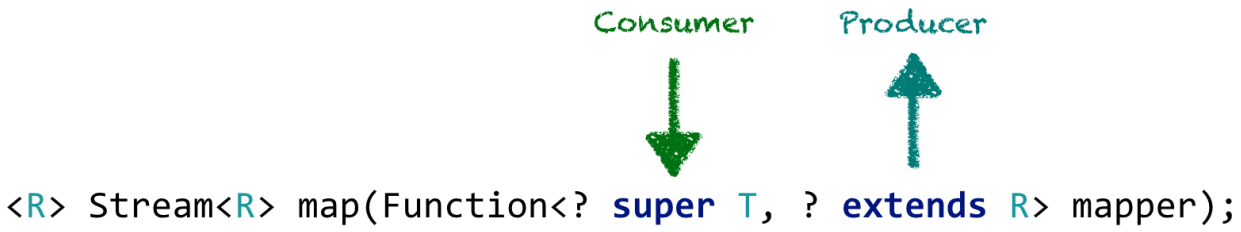
PECS

Producer-`extends`
Consumer-`super`

Consumer and Producer

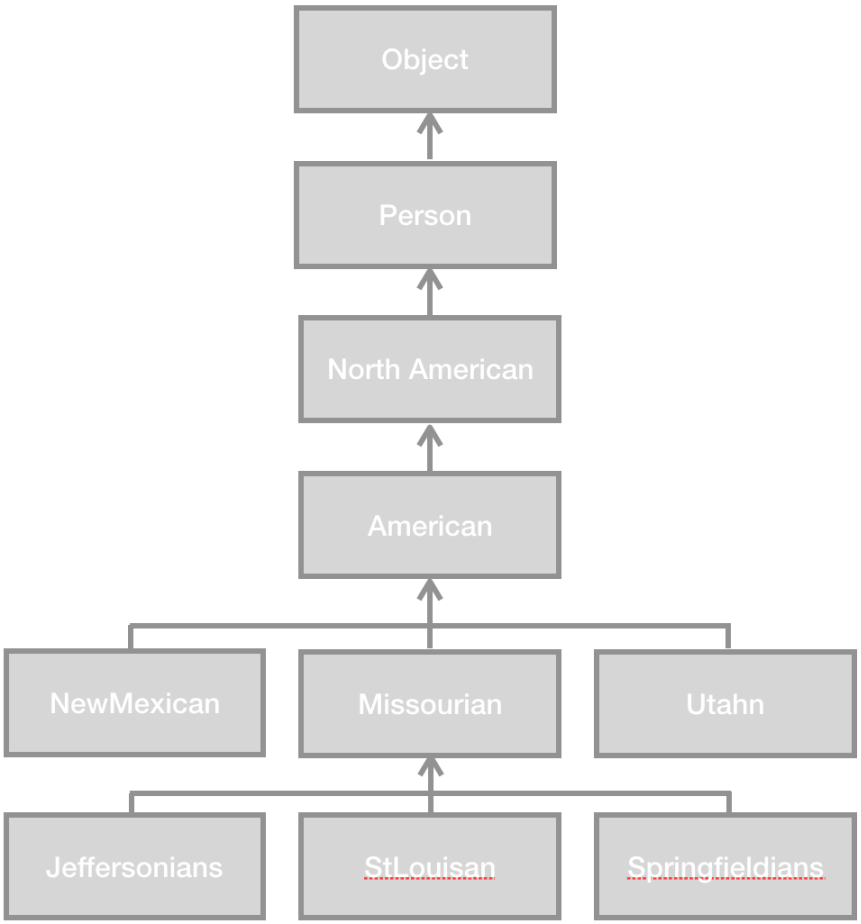


Consumer and Producer

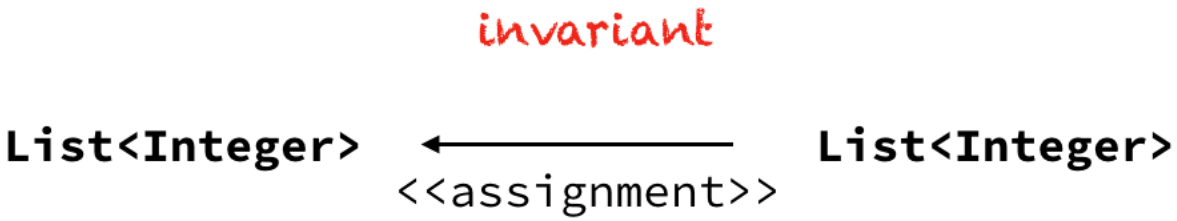


Naftalin and Wadler call it the *Get and Put Principle*

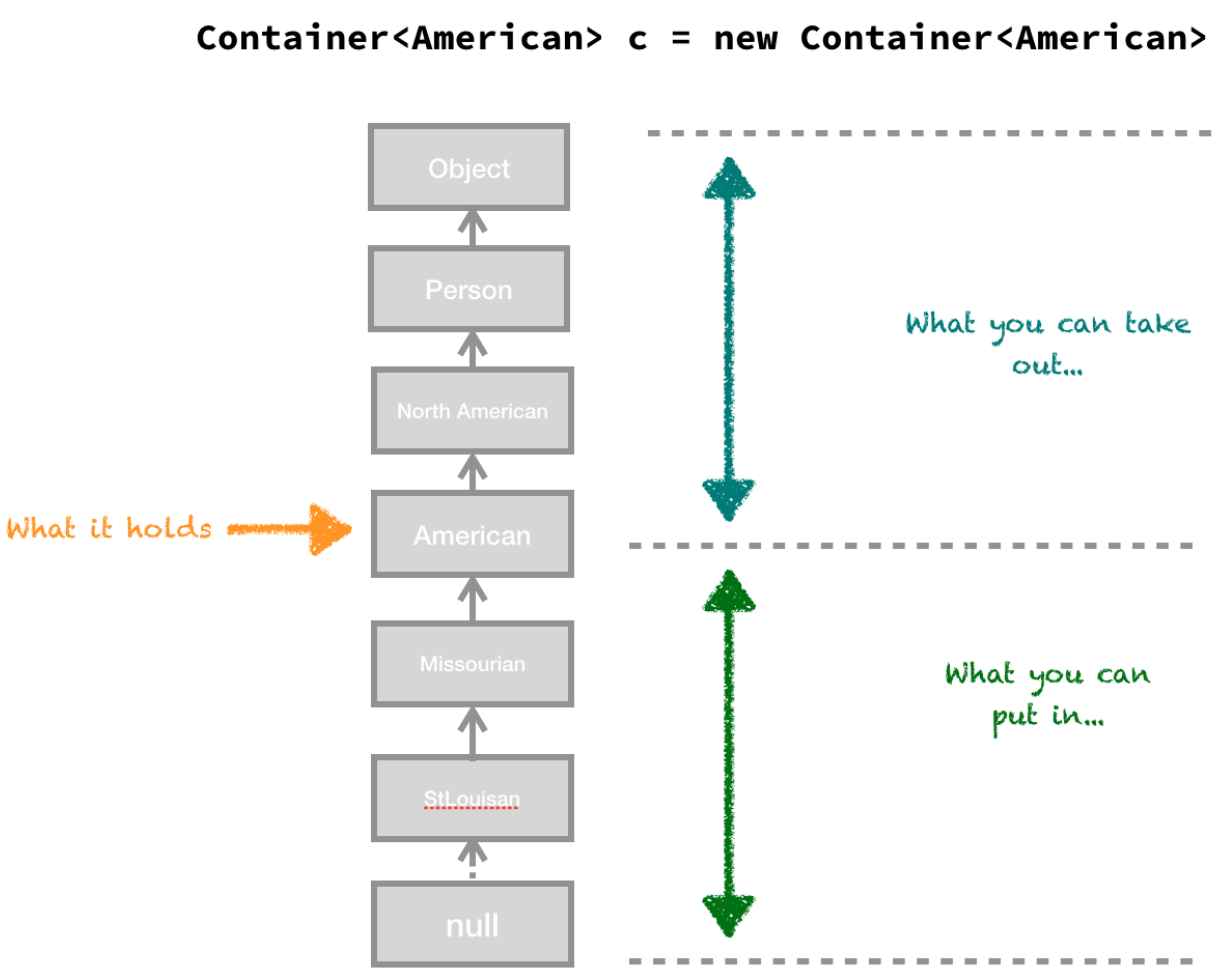
Graph Relationship



Invariant



Invariant Chart



Invariant `American` Method Parameter

src/test/java/com.xyzcorp.demos.generics.InvariantTest#processInvariantList

```
public void processInvariantList(List<American> americans) {
    Object object = americans.get(1);
    Person person = americans.get(1);
    NorthAmerican northAmerican = americans.get(1);
    American american = americans.get(1);
    americans.add(new Bostonian());
    americans.add(new Minnesotan());
    americans.add(null);
}
```

The following will not work:

```
americans.add(new Object());
americans.add(new NorthAmerican());
Massachusettsan massachusettsan = americans.get(1);
Bostonian bostonian = americans.get(1);
Madisonian madisonian = americans.get(0);
```

Invariant `American` Assignment

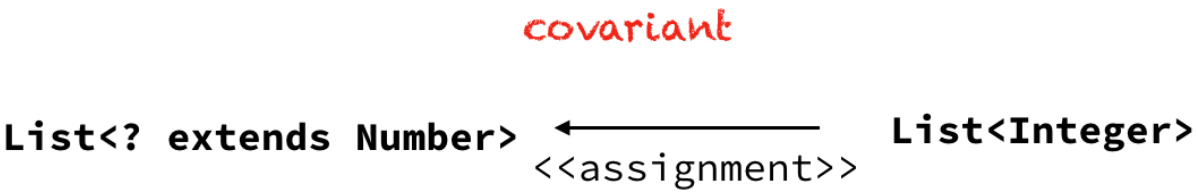
src/test/java/com.xyzcorp.demos.generics.InvariantTest#testInvariantAssignment

```
List<American> americans = new ArrayList<American>();
americans.add(new American());
americans.add(new Massachusettsan());
americans.add(new Bostonian());
americans.add(new Milwaukeean());

Object object = americans.get(1);
Person person = americans.get(1);
NorthAmerican northAmerican = americans.get(1);
American american = americans.get(1);
```

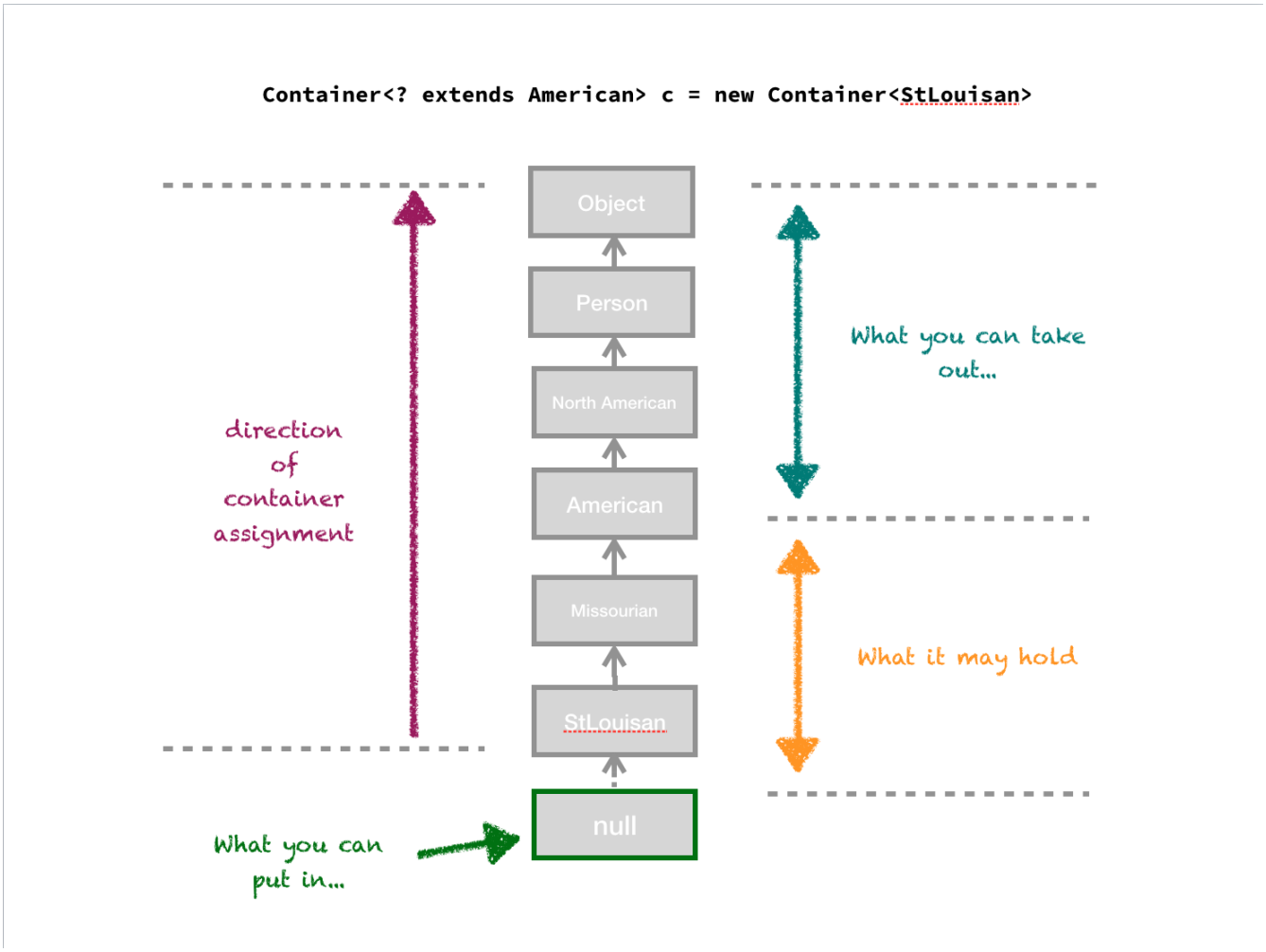
The following will not work:

```
americans.add(new Object());
americans.add(new NorthAmerican());
Massachusettsan massachusettsan = americans.get(1);
Bostonian bostonian = americans.get(1);
Madisonian madisonian = americans.get(0);
```

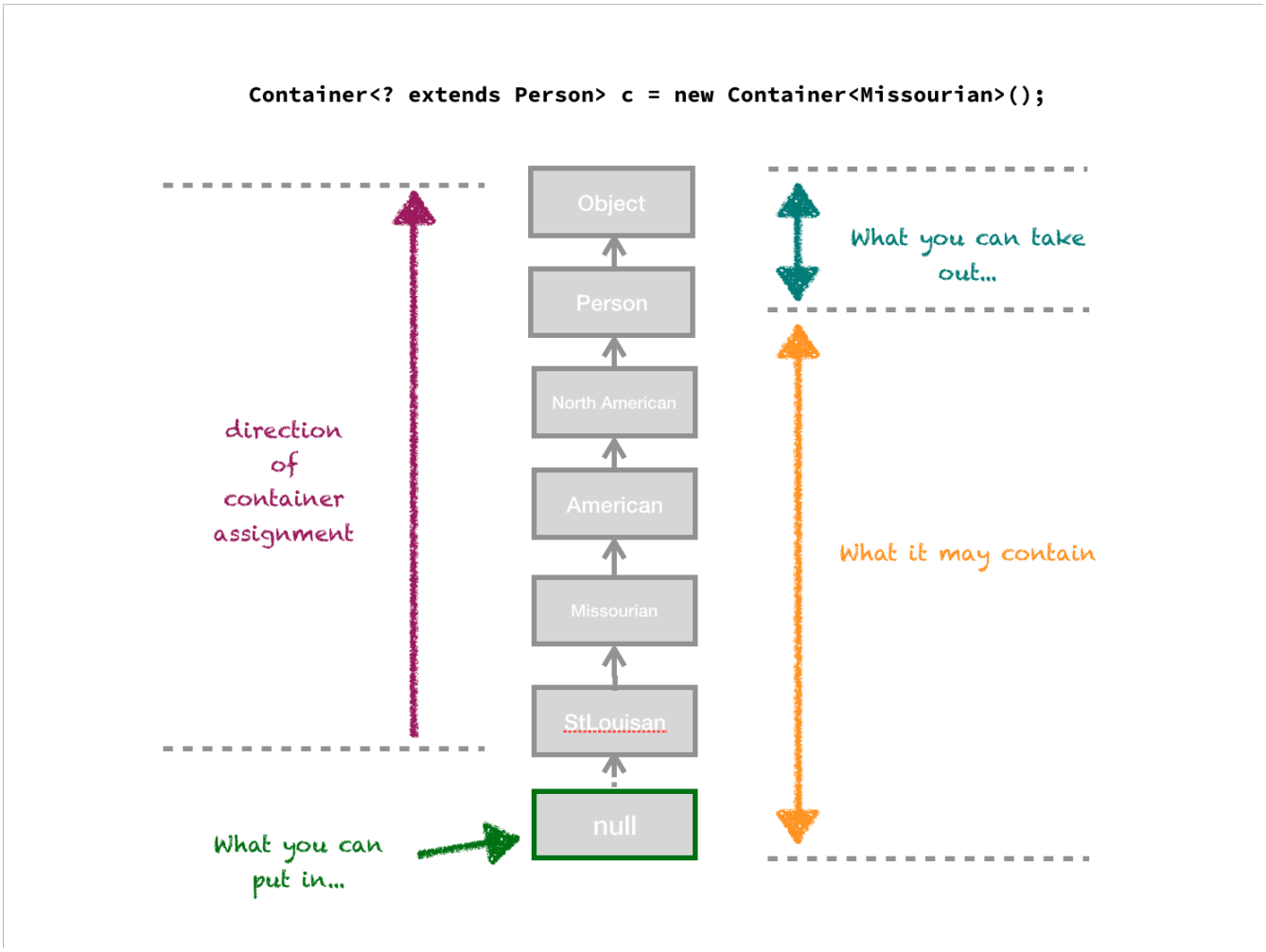
Covariant



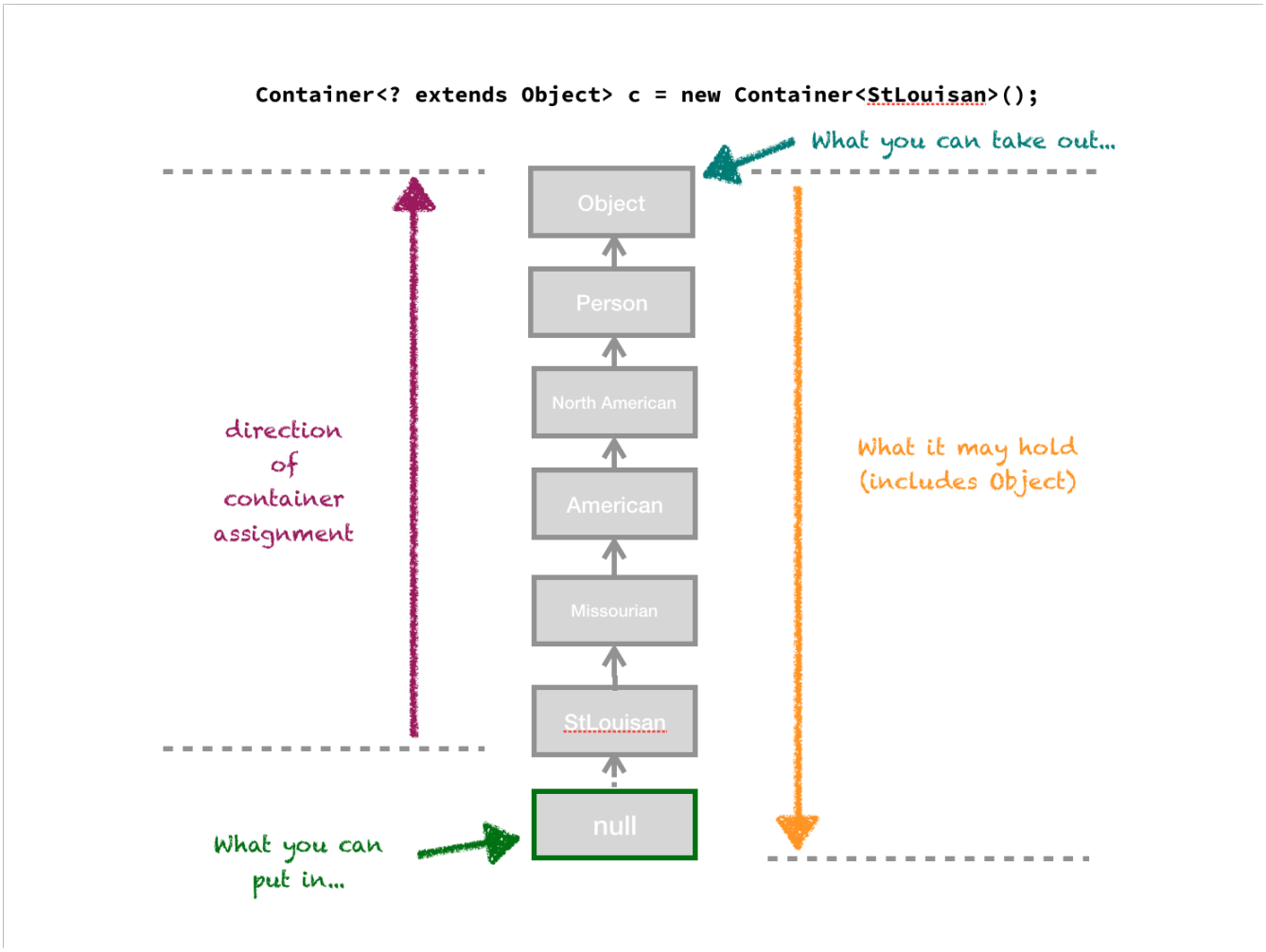
Covariant Chart `<? extends American>` with `StLouisan` Container



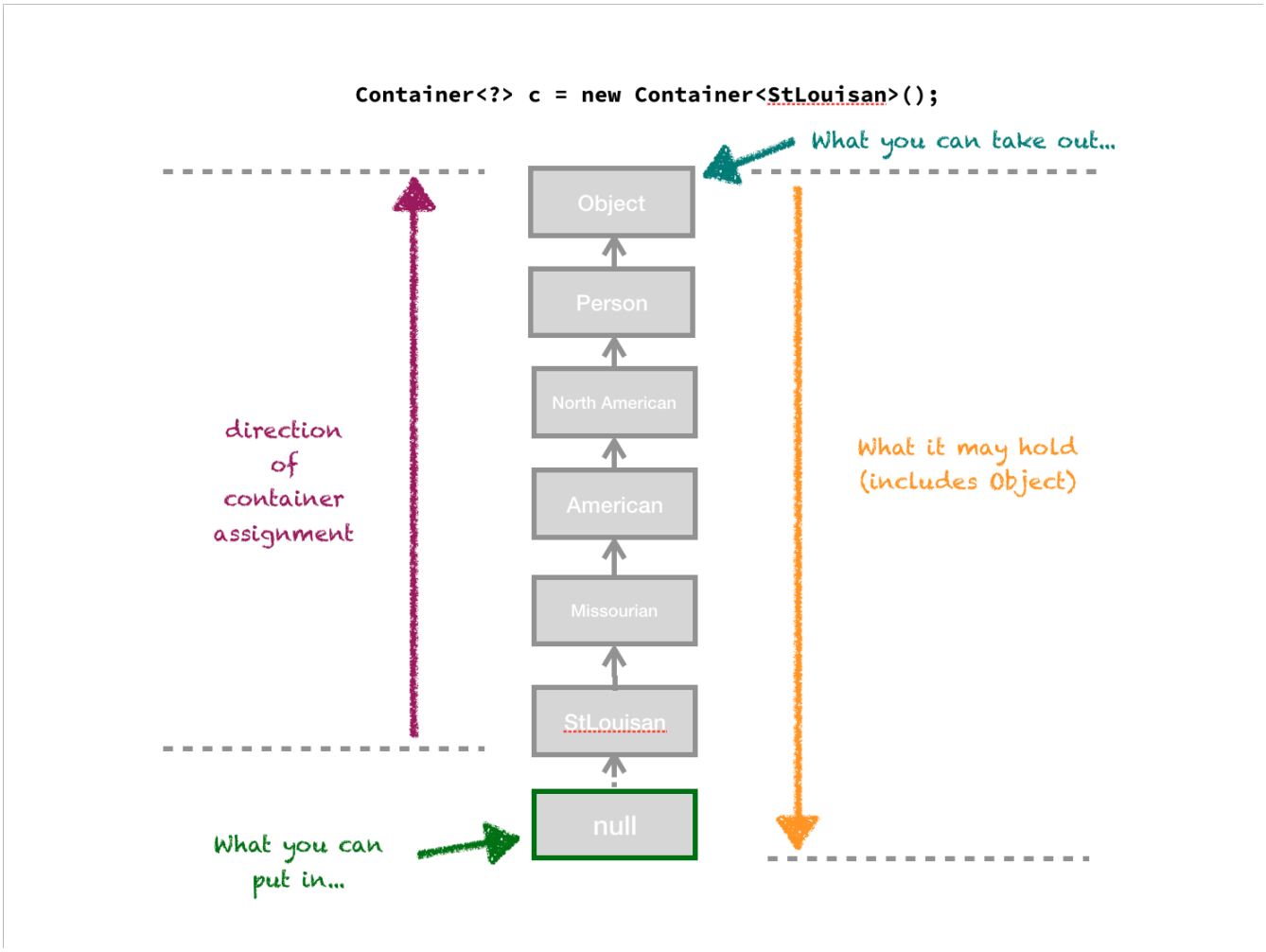
Covariant Chart `<? extends Person>` with `Missourian` Container



Covariant Chart `<? extends Object>` with `StLouisan` Container



Covariant Chart <?> with StLouisan Container



Covariant `NorthAmerican` Method Parameter

/src/test/java/com.xyzcorp.demos.generics.CovariantTest#processCovariantList

```
public void processCovariantList(List<? extends NorthAmerican>
northAmericansOrLower) {
   Object object = northAmericansOrLower.get(0);
   Person person = northAmericansOrLower.get(0);
   NorthAmerican northAmerican = northAmericansOrLower.get(0);
   northAmericansOrLower.add(null);
}
```

The following will not work:

```
northAmericansOrLower.add(new Object());
northAmericansOrLower.add(new NorthAmerican());
northAmericansOrLower.add(new American());
northAmericansOrLower.add(new Minneapolitan());
northAmericansOrLower.add(new NewMexican());
northAmericansOrLower.add(new Bostonian());
northAmericansOrLower.add(new NorthAmerican());
```

Covariant `North American` Assignment

/src/test/java/com.xyzcorp.demos.generics.CovariantTest#testCovariantAssignment

```
List<? extends American> americansOrLower = bostonians;
Object object = americansOrLower.get(0);
Person person = americansOrLower.get(0);
NorthAmerican northAmerican = americansOrLower.get(0);
American american = americansOrLower.get(0);
```

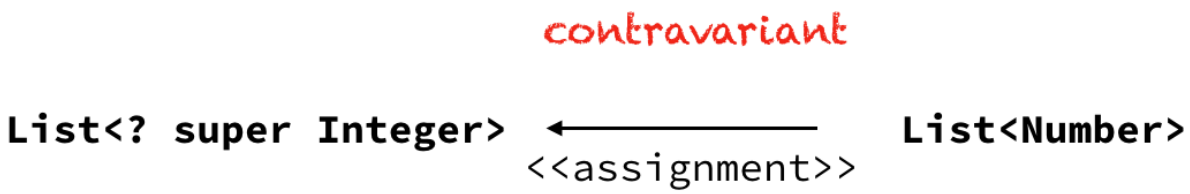
The following will not work:

```
Massachusettsan massachusettsan= americansOrLower.get(0);
Bostonian bostonian = americansOrLower.get(0);
americansOrLower.add(new Object());
americansOrLower.add(new Person());
americansOrLower.add(new NorthAmerican());
americansOrLower.add(new American());
americansOrLower.add(new Massachusettsan());
americansOrLower.add(new Bostonian());
```

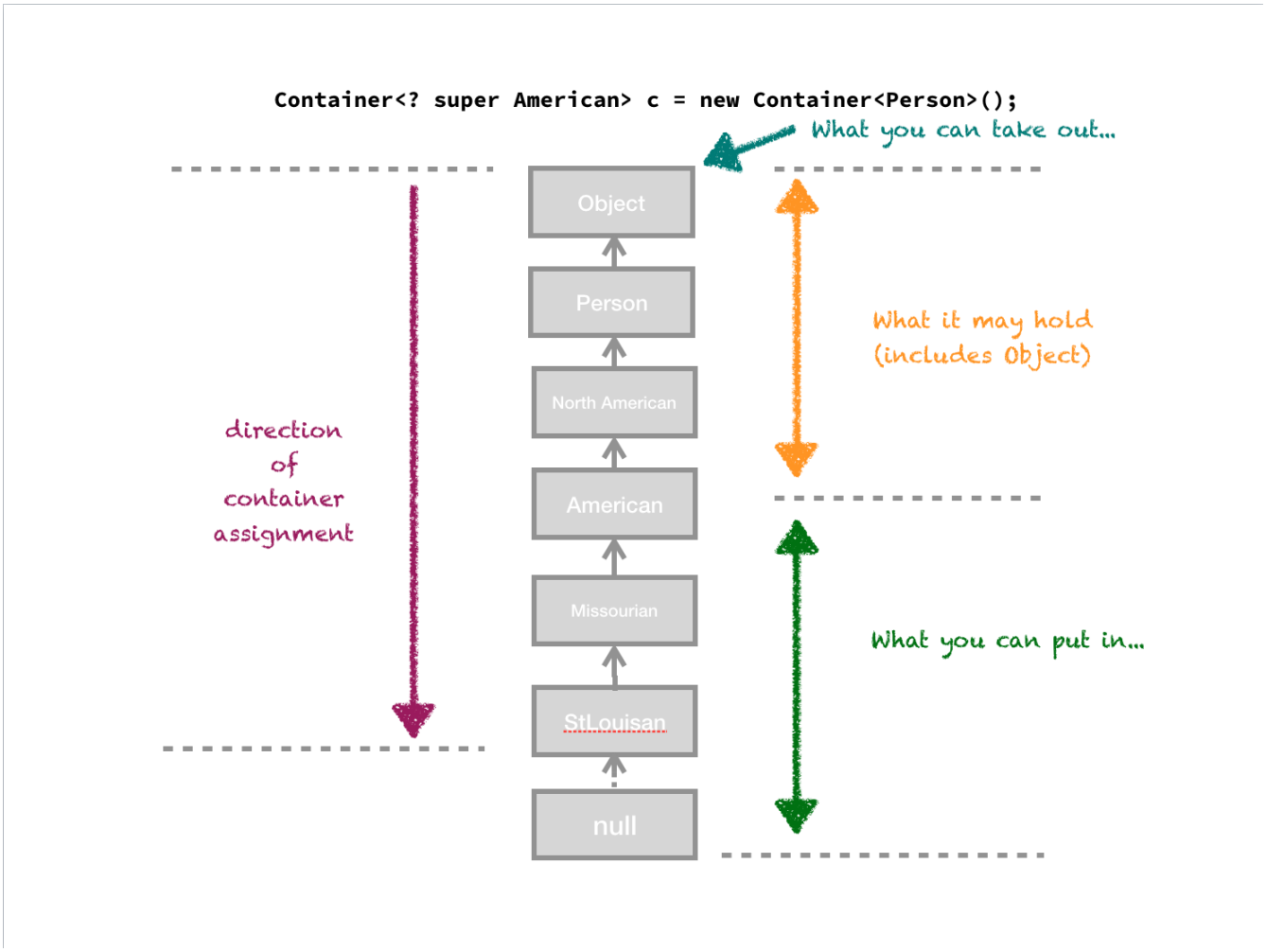
Covariant `Object` Method Parameter

src/test/java/com.xyzcorp.demos.generics.CovariantTest#processUnbounded

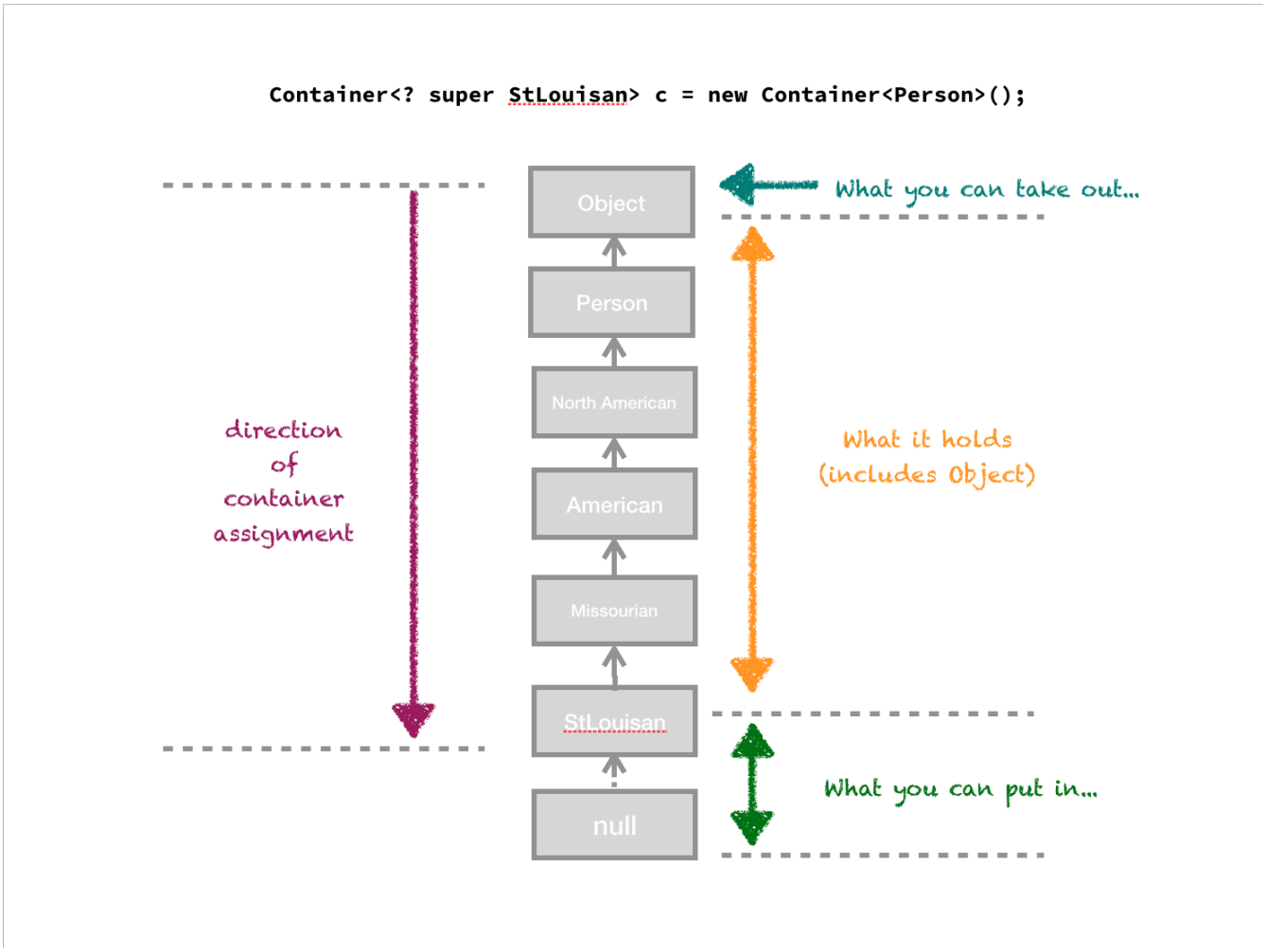
```
public void processUnbounded(List<?> objects) {
  Object o = objects.get(0);
  objects.set(4, null);
  objects.add(null);
}
```



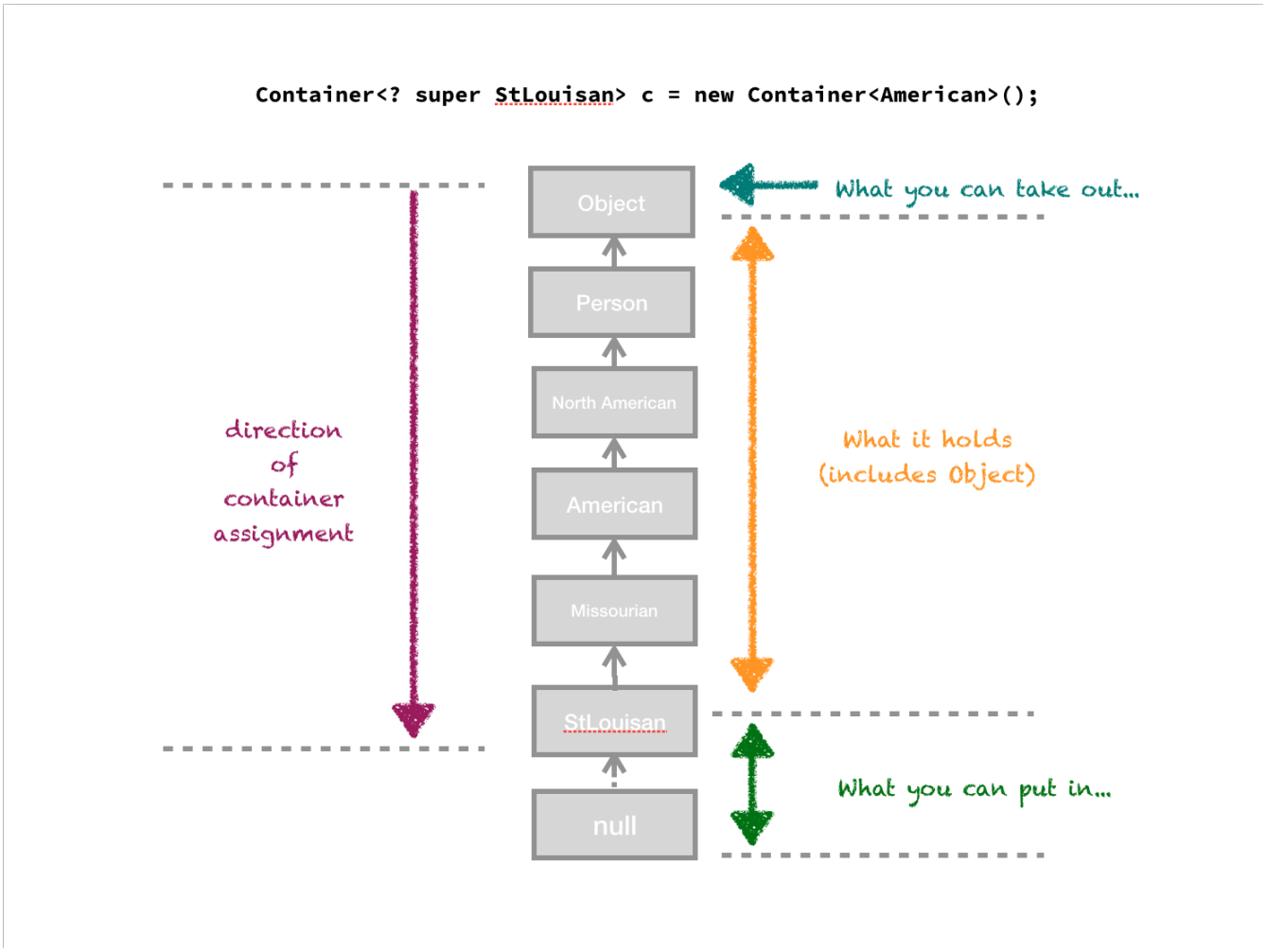
Contravariant `<? super American>` with `Person` Container



Contravariant Chart `<? super StLouisan>` with `Person` Container



Contravariant Chart `<? super StLouisan>` with `American` Container



Contravariant `American` Method

src/test/java/com.xyzcorp.demos.generics.ContravariantTest#processContravariantList

```
public void processContravariantList(List<? super American> americans) {
    americans.add(new American());
    americans.add(new Massachusettsan());
    americans.add(new Bostonian());
    americans.add(new Wisconsinite());
    americans.add(new Madisonian());
    americans.add(new NorthernCalifornian());
    americans.add(new Raleighite());
    americans.add(new Denverite());
    americans.add(new Coloradan());
    americans.add(new Missourian());
    americans.add(new StLouisan());
    americans.add(null);

    Object o = americans.get(0);
}
```

The following does not work:

```
americans.add(new European()); //Nein!
americans.add(new Person());
```

Contravariant `American` Assignment

```
List<? super American> americans = people;

Object object = americans.get(0); //special case

americans.add(new American());
americans.add(new Massachusettsan());
americans.add(new Bostonian());
americans.set(1, new NewMexican());
americans.add(null);
```

The following does not work:

```
Person person = americans.get(0);
NorthAmerican northAmerican = americans.get(0);
American american = americans.get(0);
Massachusettsan Massachusettsan = americans.get(0);
Bostonian bostonian = americans.get(0);

americans.add(new Object());
americans.add(new Person());
americans.add(new NorthAmerican());
```

Array Covariance

- Arrays are covariant
- `Array<Integer>` can be assigned to `Array<Number>` reference.
- Any attempt to do put in something wrong will be met with an `ArrayStoreException`
- An array of `List<String>` at runtime is purely an array of `List`

Array Covariance Example

src/test/java/com.xyzcorp.demos.generics.ArraysTest#testArraysAreNaturallyCovariant

```
American[] americans = new Massachusettsan[3];
americans[0] = new Massachusettsan();
```

This is allowed *at compile time* but not at runtime, as we will get an `ArrayStoreException`

```
americans[1] = new Raleighite();
americans[2] = new Wisconsinite();
```

The following works:

```
American american = americans[0];
Person person = americans[0];
Object object = americans[0];
```

The following will fail at compile time:

```
Massachusettsan massachusettsan = americans[0];
Raleighite raleighite = americans[1];
Wisconsinite wisconsinite = americans[2];
```

Safe VarArgs Heap Pollution

Heap Pollution

- A situation where a variable of a parameterized type refers to an object that is not of that parameterized type.
- This leads likely to a ClassCastException
- Occurs by
  - Mixing Raw Types and Parameterized Types
  - Performing Unchecked Casts
  - Separate Compilation of Transaction Units

Heap Pollution Example

```
List ln = Lists.<Number>newArrayList(5,1,3,5,6,10); //Converting to
RawType!
List<String> ls = ln;  // unchecked warning
```

The following will throw a `ClassCastException`

```
String s = ls.get(0);
```

Safe Varargs

- Annotation that marks that the developer is not performing an unsafe operation using varargs

- Particularly committing heap pollution
- Reminder! Arrays are covariant

Vararg Non-Safety

```
private final void process(List<String>... args) {
   //Masking the List of String, it is after all, a List<String>
   Object[] array = args;
   List<Integer> tmpList = Collections.singletonList(42);

   // Semantically invalid, but compiles without warnings
   array[0] = tmpList;
}
```

The following will become a `ClassCastException`

```
String s = args[0].get(0);
```

The `@Varargs` Annotation

- A programmer assertion that the body of the annotated method or constructor does not perform potentially unsafe operations on its varargs parameter.
- Applying this annotation to a method or constructor suppresses unchecked warnings about
  - A non-reifiable variable arity (vararg) type
  - Suppresses unchecked warnings about parameterized array creation at call sites

Subclassing Generics & Bridge Method

Subclassing Generics

- A class that is generic MyClass<T> can be extended
- The Generic Type can either be
  - Fulfilled by the subclass
  - Kept Generic

Subclassing

- `Node<Integer>` is the assigned variable
- `MySafeNode` is the already a generic that extends `Node<Integer>`

```
public class Node<T> {

    public T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
```

```
public class MySafeNode<T> extends Node<T> {...}
```

Bridge Method

- At times, the compiler may create a synthetic bridge method to add support for the super classes method type.
- This is used to avoid unsafe method calls and ClassCastExceptions
- In the following example, notice that `Node` uses `T` but `MyNode` uses `Integer`

The super class:

src/main/java/com.xyzcorp.demos.generics.bridge.Node

```
public class Node<T> {

    public T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }

    public T getData() {
        System.out.println("Node.setData");
        return data;
    }
}
```

The subclass:

src/main/java/com.xyzcorp.demos.generics.bridge.MyNode

```
public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }

    @Override
    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

Running `javap` on `MyNode` renders the following

```
% javap -cp target/classes com.xyzcorp.demos.generics.bridge.MyNode
public class com.xyzcorp.demos.generics.bridge.MyNode
    extends com.xyzcorp.demos.generics.bridge.Node<java.lang.Integer> {
  public com.xyzcorp.demos.generics.bridge.MyNode(java.lang.Integer);
  public void setData(java.lang.Integer);
  public void setData(java.lang.Object);
}
```

src/test/java/com.xyzcorp.demos.generics.BridgeTest#testBridgeMethod

If we were to use the raw form this would work

```
Node base = new MyNode(5); //Assigning to the raw form
base.setData(40); //Not preferred but should still work
```

Multiple Type Bounds

- A type parameter can have multiple bounds: `<T extends B1 & B2 & B3>`
- If one of the bounds is a class, then it should be first:

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

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

Multiple Type Bound Example

The following is requiring a Product Type of `Appendable`, `Flushable` and `Closeable`

```
private <T extends Appendable & Flushable & Closeable> void foo(T t)
throws
    IOException {
    t.append('c');
    t.append('d');
    t.flush();
    t.close();
}
```

One such class, is the `CharArrayWriter`

src/test/java/com.xyzcorp.demos.generics.MultipleBoundsTest#testMultipleInheritance

```
CharArrayWriter writer = new CharArrayWriter(40);
foo(writer);
System.out.println(writer.toCharArray());
```

Recursive Type Bound

- Bound by an expression involving the type parameter itself
- Usually involves `Comparable<T>` or `Class<T>`
- For example: `String extends Comparable<String>`

Recursive Type Bound Example

Here is the correct way to for a `Comparable` with the type of the class

src/test/java/com.xyzcorp.demos.generics.RecursiveTypeBoundsTest.Foo

```
public class Foo implements Comparable<Foo> {
    private int i = 0;

    Foo(int i) { this.i = i; }

    @Override
    public int compareTo(Foo o) {
        Objects.requireNonNull(o, "cannot compare to null");
        if (o == this) return 0;
        return Integer.compare(i, o.i);
    }
}
```

If this were not recursive, it would look like the following, notice the casting and the `Object` parameter

```
public class Foo implements Comparable {
    private int i = 0;

    Foo(int i) { this.i = i; }

    @Override
    public int compareTo(Object o) {
        Objects.requireNonNull(o, "cannot compare to null");
        if (o == this) return 0;
        return Integer.compare(i, ((Foo)o).i);
    }
}
```

Lab: Generics

Step 1: Open *src/test/java/com.xycorp.exercises.generics.GenericsExercises*

Step 2: Create a test called `testStaticFactory` with the following content

```
@Test
public void testStaticFactory() {
    Box<Integer> box = Box.of(40);
    assertThat(box.getContent()).isEqualTo(40);
}
```

Step 3: Ensure that the test works and you are able to create a `Box`

Step 4: Create a test called `testMapBox` with following content

```
@Test
public void testMapBox() {
    Box<String> bs = new Box<>("Hello"); //Box<String>
    Box<Integer> transformed = bs.map(String::length);
    System.out.println(transformed.getContent() == 5);
}
```

Step 5: Ensure that the test works and you are able to map the contents of the `Box`

Step 6: Create a test called `testMapBoxWithAlternateTypes`

```
@Test
public void testMapBoxWithAlternateTypes() {
    Box<String> bs = new Box<>("Hello"); //Box<String>

    Function <CharSequence, Person> fun = new Function<CharSequence,
Person>(){
        @Override
        public American apply(CharSequence charSequence) {
            return new American();
        }
    };

    //The above function will fit, that's the flexibility that we gain!
    Box<Person> map = bs.map(fun);

    Person person = map.getContent();
    System.out.println(person);
}
```

Step 7: Ensure that the test works and you are able to map the contents of the `Box`

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:

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

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

    public String getFirstName() {..}

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

PriorityQueue

Given:

```
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`:

```
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`

src/main/java/com.xyzcorp.demos.concurrent.threads.MyThread

```
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`

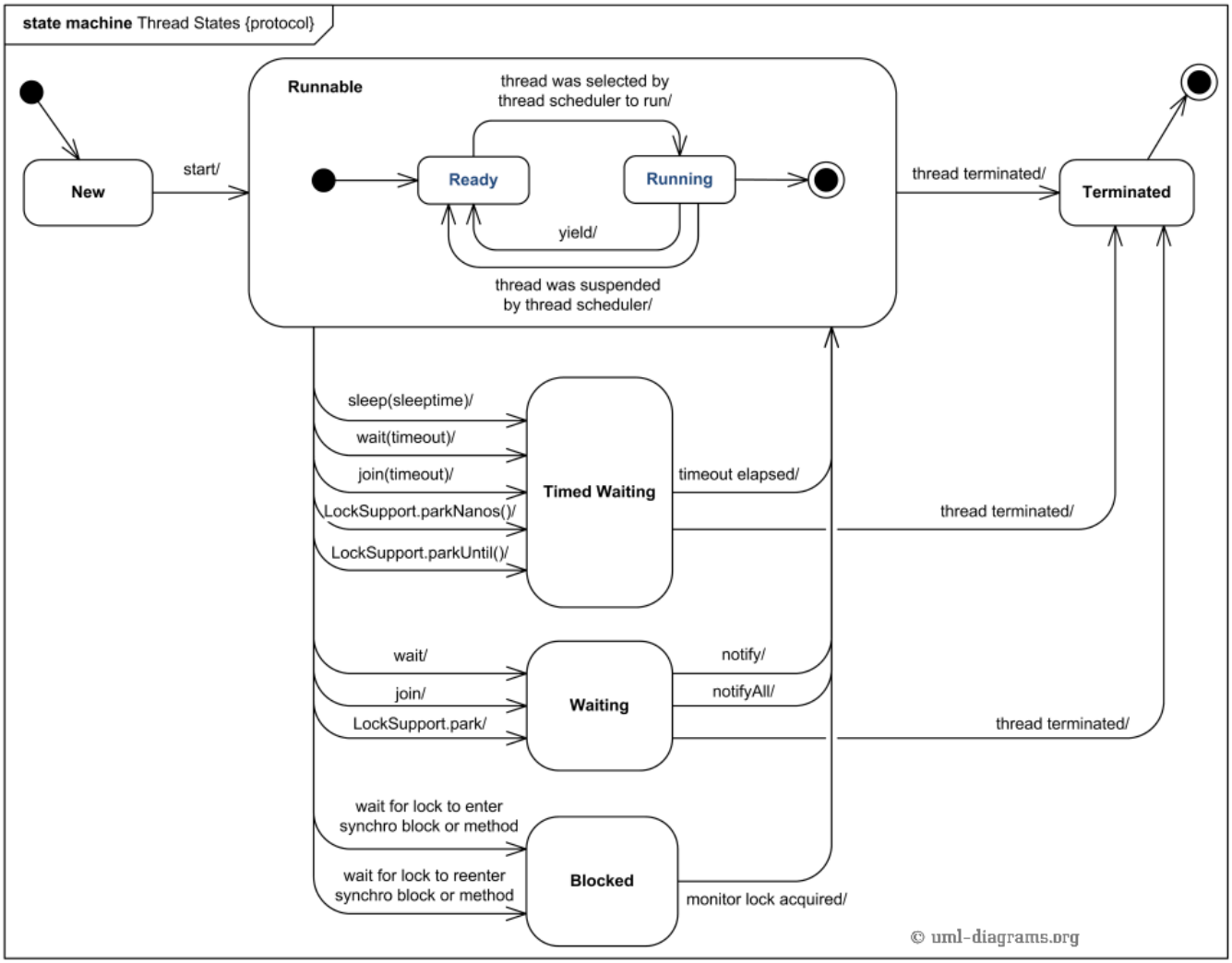
Create a `Thread` with `Runnable`

```
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
```

`join` Threads

```
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");
```

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`.

Daemon Thread Example

src/main/java/com.xyzcorp.demos.threads.DaemonRunner

```
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();
    }
}
```

Immutability

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

```
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

```
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.

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

```
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

src/main/java/com.xyzcorp.demos.concurrent.deadlock.Friend

```
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());
  }

  public static void main(String[] args) {
       final Friend alphonse = new Friend("Alphonse");
       final Friend gaston = new Friend("Gaston");
       new Thread(() -> alphonse.bow(gaston)).start();
       new Thread(() -> 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`

src/main/java/com.xyzcorp.demos.concurrent.livelock.Criminal

```
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`

src/main/java/com.xyzcorp.demos.concurrent.livelock.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

src/main/java/com.xyzcorp.demos.concurrent.livelock.LivelockRunner

```
public class LivelockRunner {
    static final Police police = new Police();
    static final Criminal criminal = new Criminal();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> police.giveRansom(criminal));
        t1.start();

        Thread t2 = new Thread(() -> 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

src/main/java/com.xyzcorp.demos.concurrent.starvation.Worker

```
public class Worker {

    public synchronized void work() {
        String name = Thread.currentThread().getName();

        while (true) {
            System.out.println(name + " is working");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

The Runner looks like the following:

src/main/java/com.xyzcorp.demos.concurrent.starvation.StarvationRunner

```
public static void main(String[] args) {
    Worker worker = new Worker();

    //We are creating ten threads
    for (int i = 0; i < 10; i++) {
        new Thread(worker::work).start();
    }
}
```

Remedy for Starvation

Include a `wait` to allow other threads to have a chance

src/main/java/com.xyzcorp.demos.concurrent.starvation.Worker

```
public class Worker {

    public synchronized void work() {
        String name = Thread.currentThread().getName();

        while (true) {
            System.out.println(name + " is working");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //wait will give other chances to other threads
            try {
                wait(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

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

```
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`.

Cyclic Barrier

- It enables you to define a synchronization object that suspends until the specified number of threads has reached the barrier point.
- Call to `await` must be made within the `Thread`
- When the amount of the barrier has been reached execution will resume

src/main/java/com.xyzcorp.demos.concurrent.synchronizers.cyclicbarrier.CyclicBarrierRunner

```
CyclicBarrier cyclicBarrier = new CyclicBarrier(4, () -> {
   System.out.println("Done");
});
```

The `Runnable`

src/main/java/com.xyzcorp.demos.concurrent.synchronizers.cyclicbarrier.MyRunnable

```
public class MyRunnable implements Runnable {

    private final CyclicBarrier cyclicBarrier;
    private final String name;

    public MyRunnable(CyclicBarrier cyclicBarrier, String name) {
        this.cyclicBarrier = cyclicBarrier;
        this.name = name;
    }

    @Override
    public void run() {
        System.out.format("Processing %s in Thread: %s%n", name,
                Thread.currentThread().getName());
        try {
            cyclicBarrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }

        System.out.format("Finally finishing %s in Thread: %s%n", name,
                Thread.currentThread().getName());
    }
}
```

Create all threads that we will be using:

```
Thread t1 = new Thread(new MyRunnable(cyclicBarrier, "A"));
Thread t2 = new Thread(new MyRunnable(cyclicBarrier, "B"));
Thread t3 = new Thread(new MyRunnable(cyclicBarrier, "C"));
Thread t4 = new Thread(new MyRunnable(cyclicBarrier, "D"));
Thread t5 = new Thread(new MyRunnable(cyclicBarrier, "E"));
Thread t6 = new Thread(new MyRunnable(cyclicBarrier, "F"));
Thread t7 = new Thread(new MyRunnable(cyclicBarrier, "G"));
Thread t8 = new Thread(new MyRunnable(cyclicBarrier, "H"));
Thread t9 = new Thread(new MyRunnable(cyclicBarrier, "I"));
Thread t10 = new Thread(new MyRunnable(cyclicBarrier, "J"));
Thread t11 = new Thread(new MyRunnable(cyclicBarrier, "K"));
Thread t12 = new Thread(new MyRunnable(cyclicBarrier, "L"));

t1.start();
t2.start();
t3.start();
t4.start();
t5.start();
t6.start();
t7.start();
t8.start();
t9.start();
t10.start();
t11.start();
t12.start();
```

Results In:

```
Processing A in Thread: Thread-0
Processing D in Thread: Thread-3
Processing C in Thread: Thread-2
Processing B in Thread: Thread-1
Done
Finally finishing B in Thread: Thread-1
Finally finishing D in Thread: Thread-3
Finally finishing C in Thread: Thread-2
Finally finishing A in Thread: Thread-0
```

Phaser

- Its primary purpose is to enable the synchronization of threads that represent one or more phases of activity.
- For example an order-processing application
  - Phase 1
    - Validate customer information,
    - Check inventory

        - Confirm pricing
    - Phase 2
        - Compute shipping costs and all applicable tax.
    - Phase 3
        - Determines estimated shipping time.

src/main/java/com.xyzcorp.demos.concurrent.synchronizers.phaser.MyRunnable

```
import java.util.concurrent.Phaser;

public class MyRunnable implements Runnable {

    private final Phaser phaser;
    private final String name;

    public MyRunnable(Phaser phaser, String name) {
        this.phaser = phaser;
        this.name = name;
    }

    @Override
    public void run() {
        phaser.register();
        System.out.format("Processing Phase 1 of %s in Thread: %s%n",
name,
                Thread.currentThread().getName());
        phaser.arriveAndAwaitAdvance();
        System.out.format("Processing Phase 2 of %s in Thread: %s%n",
name,
                Thread.currentThread().getName());
        phaser.arriveAndAwaitAdvance();
        System.out.format("Processing Phase 3 %s in Thread: %s%n", name,
                Thread.currentThread().getName());
        phaser.arriveAndAwaitAdvance();
        System.out.format("Processing Phase 4 %s in Thread: %s%n", name,
                Thread.currentThread().getName());
        phaser.arriveAndAwaitAdvance();
        System.out.format("Finally finishing %s in Thread: %s%n", name,
                Thread.currentThread().getName());
    }
}
```

```
Phaser phaser = new Phaser();
Thread t = new Thread(new MyRunnable(phaser, "A"));
Thread t2 = new Thread(new MyRunnable(phaser, "B"));
Thread t3 = new Thread(new MyRunnable(phaser, "C"));
Thread t4 = new Thread(new MyRunnable(phaser, "D"));

t.start();
t2.start();
t3.start();
t4.start();
```

Results In:

```
Processing Phase 1 of C in Thread: Thread-2
Processing Phase 1 of D in Thread: Thread-3
Processing Phase 1 of B in Thread: Thread-1
Processing Phase 1 of A in Thread: Thread-0
Processing Phase 2 of A in Thread: Thread-0
Processing Phase 2 of B in Thread: Thread-1
Processing Phase 2 of D in Thread: Thread-3
Processing Phase 2 of C in Thread: Thread-2
...
```

Countdown Latch

- Sometimes you will want a thread to wait until one or more events have occurred.
- To handle such a situation, the concurrent API supplies CountDownLatch.
- A CountDownLatch is initially created with a count of the number of events that must occur before the latch is released.
- Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.
- To wait on the latch, a thread calls `await( )`

Countdown Latch Listener

- The Listener will trigger when the latch is counted down
- This `Runnable` will block in the `countDownLatch.await` call

src/main/java/com.xyzcorp.demos.concurrent.synchronizers.countdownlatch.RunnableListener

```
public class RunnableListener implements Runnable {

    private CountDownLatch countDownLatch;
    private String name;

    public RunnableListener(CountDownLatch countDownLatch, String name)
{
        this.countDownLatch = countDownLatch;
        this.name = name;
    }

    @Override
    public void run() {
        System.out.format("Processing %s in Thread: %s%n", name,
                Thread.currentThread().getName());
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.format("Finally finishing %s in Thread: %s%n", name,
                Thread.currentThread().getName());
    }
}
```

Countdown Latch Command

- This will issue the command to count down from another thread
- They all share the same `CountDownLatch`

src/main/java/com.xyzcorp.demos.concurrent.synchronizers.countdownlatch.RunnableCommand

```
public class RunnableCommand implements Runnable {

    private final CountDownLatch countDownLatch;
    private final String name;

    public RunnableCommand(CountDownLatch countDownLatch, String name) {
        this.countDownLatch = countDownLatch;
        this.name = name;
    }

    @Override
    public void run() {
        Random random = new Random(304L);
        try {
            Thread.sleep(random.nextInt(3) * 1000);
            System.out.printf("Counting down in %s on Thread %s%n",
name,
                    Thread.currentThread().getName());
            countDownLatch.countDown();
            System.out.printf("Counted down in %s on Thread %s%n", name,
                    Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
CountDownLatch countDownLatch = new CountDownLatch(2);
new Thread(new RunnableListener(countDownLatch, "A")).start();
new Thread(new RunnableListener(countDownLatch, "B")).start();
Thread.sleep(2000);
System.out.println(countDownLatch.getCount());
new Thread(new RunnableCommand(countDownLatch, "A")).start();
Thread.sleep(2000);
System.out.println(countDownLatch.getCount());
new Thread(new RunnableCommand(countDownLatch, "B")).start();
Thread.sleep(2000);
System.out.println(countDownLatch.getCount());
```

Results In:

```
Processing A in Thread: Thread-0
Processing B in Thread: Thread-1
2
Counting down in A on Thread Thread-2
Counted down in A on Thread Thread-2
1
Counting down in B on Thread Thread-3
Counted down in B on Thread Thread-3
Finally finishing A in Thread: Thread-0
Finally finishing B in Thread: Thread-1
0
```

Lab: Threads

Step 1: Run the class `com.xyzcorp.demos.concurrent.threads.MyThread` in either Maven or your IDE **Step 2:** This class will print the PID **Step 3:** Then run `jcmd` which is a thread analysis tool for Java 8

```
jcmd  $PID  Thread.print
```

Step 4: If you have `jconsole` installed, run it against the PID provided by the thread

You may want to change the amount of time `MyThread` will run

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)

- `get` will always block
- Would be best to ensure that there is an result ready before querying

src/test/java/com.xyzcorp.demos.futures.FutureBasicsTest#testBasicFuture

```
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)

- From JDK 5 until JDK 8 this one the standard approach
- There have been libraries like Guava which made this far easier

src/test/java/com.xyzcorp.demos.futures.FutureBasicsTest#testBasicFutureAsync

```
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
- Use a `final` variable for the future

src/test/java/com.xyzcorp.demos.futures.FutureBasicsTest#downloadingContentFromURL

```
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(" ")));
        }
    });
}
```

Retrieving the `Future`

src/test/java/com.xyzcorp.demos.futures.FutureBasicsTest#testGettingUrl

```
Future<Stream<String>> future = downloadingContentFromURL
("https://openjdk.java.net/");
while (!future.isDone()) {
    Thread.sleep(1000);
    System.out.println("Doing Something Else");
}
Stream<String> allStrings = future.get();
allStrings
        .filter(x -> x.contains("Java"))
        .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

Setting up the Completable Future

- We will run in the example 3 futures of a different quality
- We will also include an `ExecutorService` for a Thread Pool

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest

```
private CompletableFuture<Integer> integerFuture1;
private CompletableFuture<Integer> integerFuture2;
private CompletableFuture<String> stringFuture1;
private ExecutorService executorService;
executorService = Executors.newCachedThreadPool();
```

Implementation for the Futures

- Note that when some are created, they may include an `executorService`
- The `Future` that are not using an `executorService` will be using the default thread pool (`ForkJoin`)

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest

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

Simply Accepting the Answer

- `thenAccept`
  - Accepts the answer
  - Argument is a `Consumer`

```
integerFuture1.thenAccept(System.out::println);
```

Functional `map` with `thenApply`

- `thenApply`
  - Analogous to `map`
  - Accepts a `Function`

```
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);
});
```

Asynchronous Functional `map` with `thenApplyAsync`

- `thenApplyAsync` will apply a map but will do so on another `Thread`
  - If an `ExecutorService` is provided it will use a `Thread` from that pool
  - Otherwise it will pull a `Thread` from the `ForkJoin` pool

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#completableFutureWithThenApplyAsync

```
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);
```

Trigger a final action with `thenRun`

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

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#completableFutureWithThenRun

```
@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);
}
```

Trapping Errors with `exceptionally`

- Exceptionally takes an error exception if anywhere on the chain there is an `Exception` thrown
- Provides an alternative answer in case of Exception

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#completableFutureExceptionally

```
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);
```

Trapping Errors with `handle`

- If you wish to handle the error based on both a successful output or an exception, use `handle`
- Gives you complete control of how to handle the response
- Provides both the value and the `Throwable`

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#completableFutureHandle

```
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);
```

`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

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#getTemperatureInFahrenheit

```
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;
    });
}
```

Using `compose` to compose two `CompletableFuture`

- `compose` is `flatMap` for `CompletableFuture` and allows you to build off one another
- Use case, use a Future to look for a value, and based on the value create another `Future` for another task

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#completableCompose

```
CompletableFuture<Integer> composition =
        stringFuture1.thenCompose(s -> getTemperatureInFahrenheit(s));
composition.thenAccept(System.out::println);
```

Using `combine` to combine two unrelated `CompleteableFuture`

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

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#completableCombine

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

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

Creating a Promise using `CompletableFuture`

src/test/java/com.xyzcorp.demos.futures.CompletableFutureTest#testCompletableFuturePromise

```
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);
```

Flow API

- JEP 266 (http://openjdk.java.net/jeps/266)
- Minimal set of interfaces that capture the heart of asynchronous publication and subscription
- Interfaces contain that of what is in reactive-streams.org (Though they are not the same)

Components

- `Publisher`
- `Subscriber`
- `Processor`
- `Subscription`

`Publisher`

- The publisher publishes the stream of data items to the registered subscribers
- It publishes items to the subscriber asynchronously, normally using an Executor
- Publishers ensure that `Subscriber` method invocations for each subscription are strictly ordered

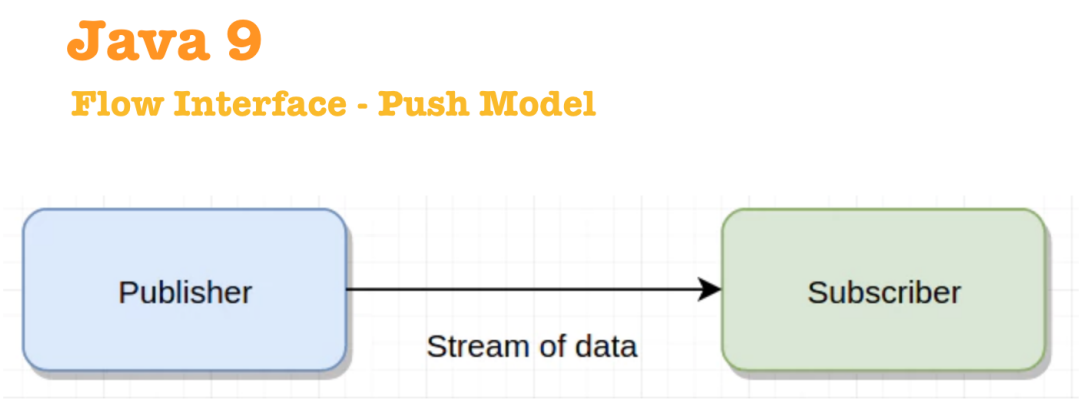
`Subscriber`

- The `Subscriber` subscribes to the `Publisher` for the callbacks.
- Data items are not pushed to the `Subscriber` unless requested, but multiple items may be requested.
- `Subscriber` method invocations for a given `Subscription` are strictly ordered.
- The application can react to the following callbacks, which are available on the subscriber

`Subscription`

- Links a `Flow.Publisher` and `Flow.Subscriber`
- Subscribers receive items only when requested, and may cancel at any time, via the `Subscription`

Push Model



`Processor`

- A component that acts as both a `Subscriber` and `Publisher`
- The processor sits between the `Publisher` and `Subscriber`, and transforms one stream to another.
- There could be one or more processor chained together, and the result of the final processor in the chain, is processed by the `Subscriber`
- The JDK does not provide any concrete Processors so it is left up to the individual to write whatever processor one requires.

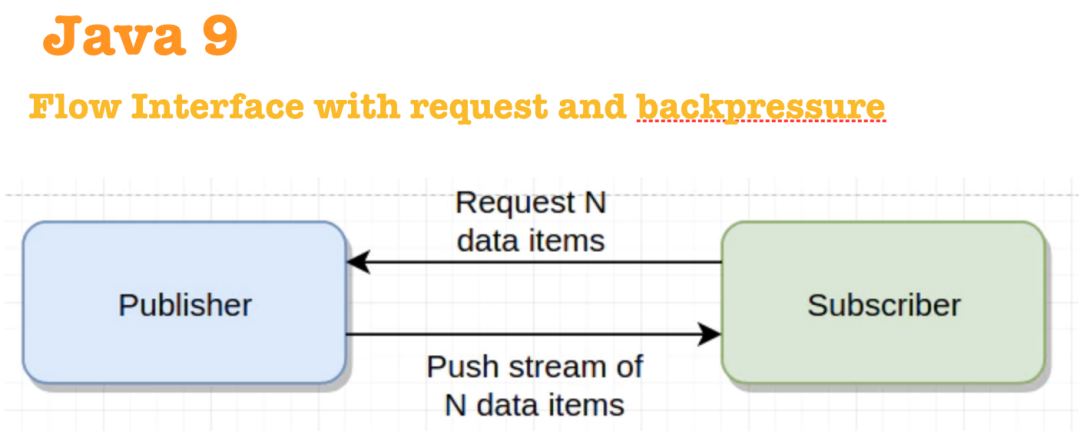
Physical Backpressure

> Back pressure refers to pressure opposed to the desired flow of a fluid in a confined place such as a pipe. It is often caused by obstructions or tight bends in the confinement vessel along which it is moving, such as piping or air vents.

Data Backpressure

> Back pressure is when an asynchronous source is emitting items more rapidly than an operator or subscriber can consume them. This presents the problem of what to do with such a growing backlog of unconsumed items.

Flow API Request and Backpressure



Flow API Backpressure

> The Flow API does not provide any APIs to signal or deal with back pressure as such, but there could be various strategies one could implement by oneself to deal with back pressure

`Publisher`

Example of a `Publisher` if we were to do our own with an `ExecutorService`

/src/main/java/com.xyzcorp.demos.reactive.MyPublisher

```
public class MyPublisher implements Flow.Publisher<Long> {
    private final ExecutorService executorService;

    public MyPublisher(ExecutorService executorService) {
        this.executorService = executorService;
    }

    @Override
    public void subscribe(Flow.Subscriber<? super Long> subscriber) {
         subscriber.onSubscribe(new Flow.Subscription() {
             AtomicLong count = new AtomicLong(0);
             AtomicBoolean done = new AtomicBoolean(false);

             @Override
             public void request(long n) {
                 executorService.submit(() -> {
                     while (!done.get() && ((count.incrementAndGet()) <
n)) {
                         subscriber.onNext(count.get());
                     }
                 });
             }

             @Override
             public void cancel() {
                done.set(true);
             }
         });
    }
}
```

Subscriber and Subscription

```
ExecutorService executorService =
        Executors.newFixedThreadPool(100);


MyPublisher myPublisher = new MyPublisher(executorService);
myPublisher.subscribe(new Flow.Subscriber<Long>() {
    private Flow.Subscription subscription;

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        this.subscription.request(1000);
    }

    @Override
    public void onNext(Long item) {
        if (item < 0) throw new RuntimeException("Nope");
        System.out.println("S1: (" +
                Thread.currentThread().getName() + ")" + item);
        if (item == 350) {
            this.subscription.cancel();
        }
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("S1: Done!");
    }
}
```

A Second Subscriber

Every Subscriber is different and can do different things with the data

```
myPublisher.subscribe(new Flow.Subscriber<Long>() {
  private Flow.Subscription subscription;

  @Override
  public void onSubscribe(Flow.Subscription subscription) {
      this.subscription = subscription;
      this.subscription.request(1000);
  }

  @Override
  public void onNext(Long item) {
      System.out.println("S2: (" +
              Thread.currentThread().getName() + ")" + item);
  }

  @Override
  public void onError(Throwable throwable) {
      throwable.printStackTrace();
  }

  @Override
  public void onComplete() {
      System.out.println("S2: Done!");
  }
});
```

Reactive Streams

- https://www.reactive-streams.org/
- Reactive Streams Standards
- Abides by the Flow API
  - RXJava
  - Project Reactor
  - Akka Streams

Interconnecting Reactive Projects

```
Flux<Integer> reactorFlux = Flux.fromCompletionStage(CompletableFuture
.<Integer>completedFuture(1));

Observable<Integer> observable = Observable.fromPublisher(reactorFlux);

observable.subscribe(
    item -> System.out.println(item),
    error -> error.printStackTrace(),
    () -> System.out.println("Done"));
```

Lab: Reactive

Step 1: Open *src/test/java/com.xycorp.exercises.reactive.ReactiveExercises*

Step 2: Open in your browser the following url

Step 3: Create an RXJava `Observable` that reads from reads from the above url line by line

Step 4: On one branch, find the latest population for the country you live in, output on `System.out.println`

Step 5: On another branch, find the change in population for the country you live in, output on `System.out.println`

Step 6: Extra Credit! Read from the URL using an IO Scheduler and write each of the branches to a file instead of `System.out.println`

Design Patterns

Creational Patterns

Factory Method Pattern

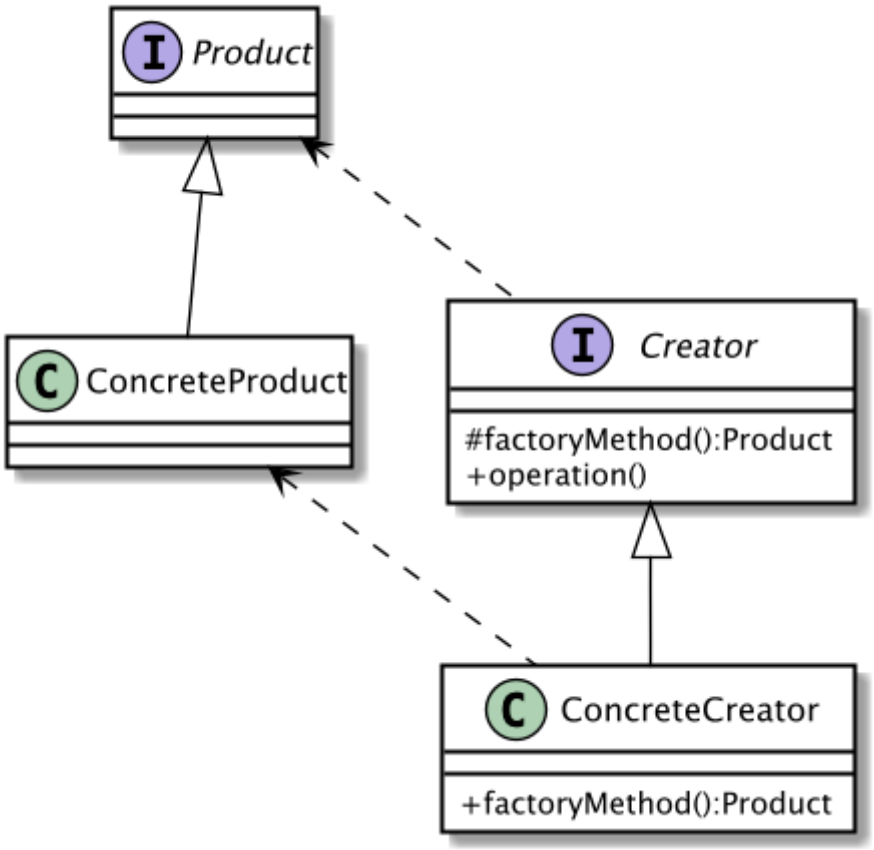
Factory Method Pattern Properties

- **Type:** Creational
- **Level:** Class

Factory Method Pattern Purpose

To define a standard method to create an object, apart from a constructor, but the decision of what kind of an object to create is left to subclasses.

Factory Method Canonical Diagram



Factory Method Ingredients

Product – The `interface` of objects created by the factory

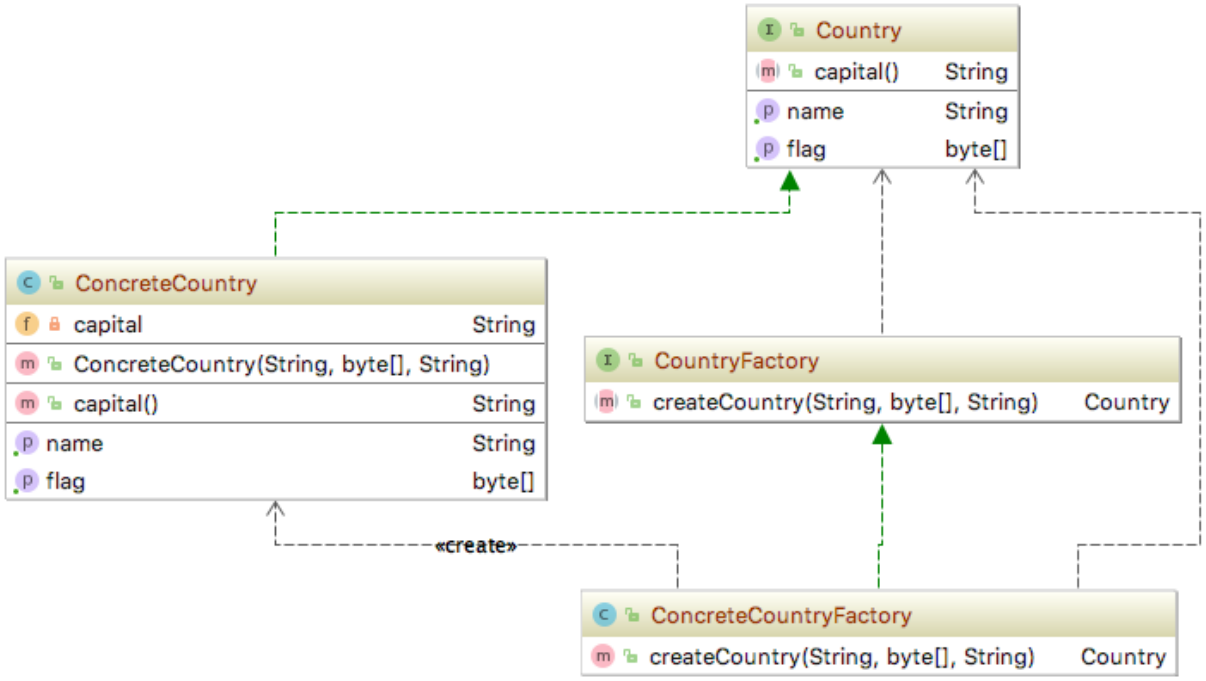
ConcreteProduct – The implementing class of **Product**. Objects of this class are created by the **ConcreteCreator**.

Creator – The `interface` that defines the factory methods

ConcreteCreator – The class that extends **Creator** and that provides an implementation for the factoryMethod. This can return any object that implements the **Product** interface.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method Demo Diagram



Factory Method Advantages

- Extensible
- Leave decision of specificity until later
- Subclass, not superclass, determines the kind of object to create
- You know when to create an object, but not what kind of an object.
- You need several overloaded constructors with the same parameter list, which is not allowed in Java. Instead, use several Factory Methods with different names.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method Disadvantages

- To create a new type you must create a separate subclass

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method: Product

```
public interface Country {
    public String getName();

    public byte[] getFlag();

    public String capital();
}
```

Factory Method: Concrete Product

```
public class ConcreteCountry implements Country {

    private String name;
    private String capital;
    private byte[] flagBytes;

    public ConcreteCountry(String name, byte[] flagBytes, String
capital) {
        this.name = name;
        this.capital = capital;
        this.flagBytes = flagBytes;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public byte[] getFlag() {
        return flagBytes;
    }

    @Override
    public String capital() {
        return capital;
    }
}
```

Factory Method: Creator

```
public interface CountryFactory {
    public Country createCountry(String name, byte[] flag, String
capital);
}
```

Factory Method: Concrete Creator

```
public class ConcreteCountryFactory implements CountryFactory {
    @Override
    public Country createCountry(String name, byte[] flag, String
capital) {
        return new ConcreteCountry(name, flag, capital);
    }
}
```

Builder Pattern

Builder Pattern Properties

- **Type:** Creational
- **Level:** Component

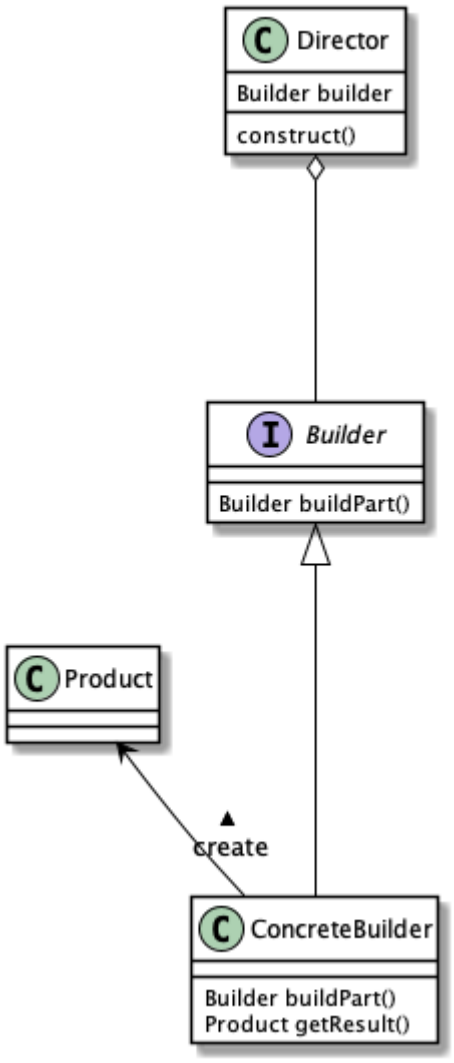
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Builder Pattern Purpose

To simplify complex object creation by defining a class whose purpose is to build instances of another class. The Builder produces one main product, such that there might be more than one class in the product, but there is always one main class.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Builder Pattern Canonical Diagram



Builder Pattern Ingredients

Director – Has a reference to an **AbstractBuilder** instance. The **Director** calls the creational methods on its builder instance to have the different parts and the Builder build.

AbstractBuilder – The interface that defines the available methods to create the separate parts of the product.

ConcreteBuilder – Implements the **AbstractBuilder** interface. The **ConcreteBuilder** implements all the methods required to create a real Product. The implementation of the methods knows how to process information from the **Director** and build the respective parts of a Product. The **ConcreteBuilder** also has either a `getProduct` method or a creational method to return the **Product** instance.

Product – The resulting object. You can define the product as either an interface (preferable) or class.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

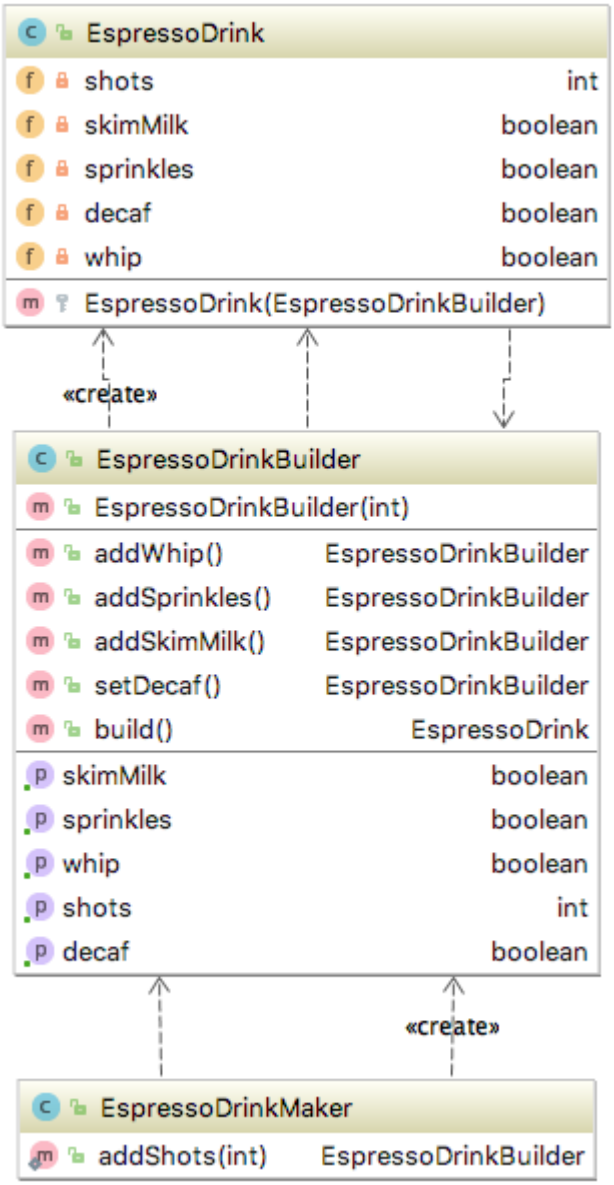
Builder Pattern Advantages

- Works well if you have complex state in an object
- Avoids complicated constructors
- Avoids complicated object graph initialization
- Works particularly well for dependencies that are difficult to setup

Builder Pattern Disadvantages

- Tight coupling in the builder and its product
- Any changes in the product would affect the builder

Builder Pattern Demo Diagram



Builder: Director

```
public class EspressoDrinkMaker {

    public static EspressoDrinkBuilder addShots(int i) {
        return new EspressoDrinkBuilder(i);
    }
}
```

Builder: Concrete Builder

```
public class EspressoDrinkBuilder {
    private boolean whip;
    private boolean sprinkles;
    private int shots;

    public EspressoDrinkBuilder(int shots) {
        this.shots = shots;
    }

    public EspressoDrinkBuilder addWhip() {
        this.whip = true;
        return this;
    }

    public EspressoDrinkBuilder addSprinkles() {
        this.sprinkles = true;
        return this;
    }

    public EspressoDrink build() {
        return new EspressoDrink(this);
    }
}
```

Builder: Product

```
public class EspressoDrink {
    private int shots;
    private boolean sprinkles;
    private boolean whip;

    protected EspressoDrink(EspressoDrinkBuilder espressoDrinkBuilder) {
        this.shots = espressoDrinkBuilder.getShots();
        this.sprinkles = espressoDrinkBuilder.isSprinkles();
        this.whip = espressoDrinkBuilder.isWhip();
    }
}
```

Singleton Pattern

Singleton Pattern Properties

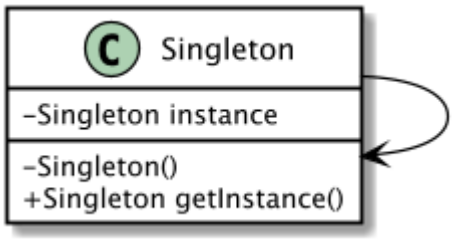
- **Type:** Creational
- **Level:** Object

Singleton Pattern Purpose

To have only one instance of this class in the system, while allowing other classes to get access to this instance.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton Canonical Diagram

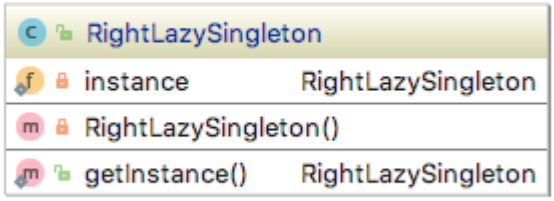


Singleton Ingredients

Singleton – Provides a private constructor, maintains a private static reference to the single instance of this class, and provides a static accessor method to return a reference to the single instance.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton Demo Diagram



Singleton Pattern Advantages

- If done right, can delay use of an object
- Ensures a single object at all times

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton Pattern Disadvantages

- Abuse especially among beginning programmers
- Difficulty in unit testing
- Often unnecessary, particularly in dependency injection frameworks
- No control over who accesses the object
- Once you go singleton, it's tough to change
- Can expose threading issues, where duplicates can be created

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton: An Eager Implementation

```
public class EagerSingleton {
    private static EagerSingleton instance = new EagerSingleton();

    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

Singleton: A Lazy Non-Thread-Safe Implementation

```
public class WrongLazySingleton {
    private static WrongLazySingleton instance = null;

    private WrongLazySingleton() { }

    public static WrongLazySingleton getInstance() {
        if (instance ==== null) {
            instance = new WrongLazySingleton();
        }
        return instance;
    }
}
```

Singleton: A Lazy Thread-Safe Implementation

```
public class RightLazySingleton {
    private static RightLazySingleton instance;

    private RightLazySingleton() {}

    public static synchronized RightLazySingleton getInstance() {
        if (instance ==== null) {
            instance = new RightLazySingleton();
        }
        return instance;
    }
}
```

`synchronized` would obtain a lock on the class since the method is `static`

Singleton: An `enum` of one

Recommended way to create a Singleton in Java:

```
public enum EnumSingleton {
     INSTANCE;
}
```

To use this...

```
var a = EnumSingleton.INSTANCE;
var b = EnumSingleton.INSTANCE;

a.equals(b);
a.hashCode() == b.hashCode();
a.toString().equals(b.toString());
```

The above uses Java 10's new `var`

Factory Method Pattern

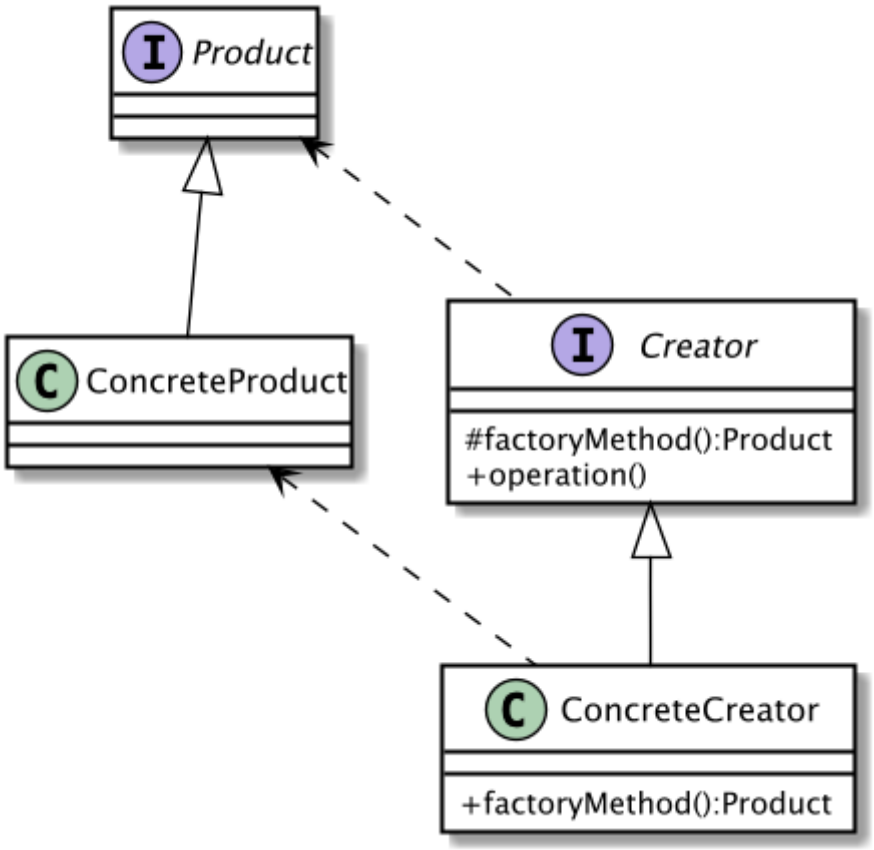
Factory Method Pattern Properties

- **Type:** Creational
- **Level:** Class

Factory Method Pattern Purpose

To define a standard method to create an object, apart from a constructor, but the decision of what kind of an object to create is left to subclasses.

Factory Method Canonical Diagram



Factory Method Ingredients

Product – The `interface` of objects created by the factory

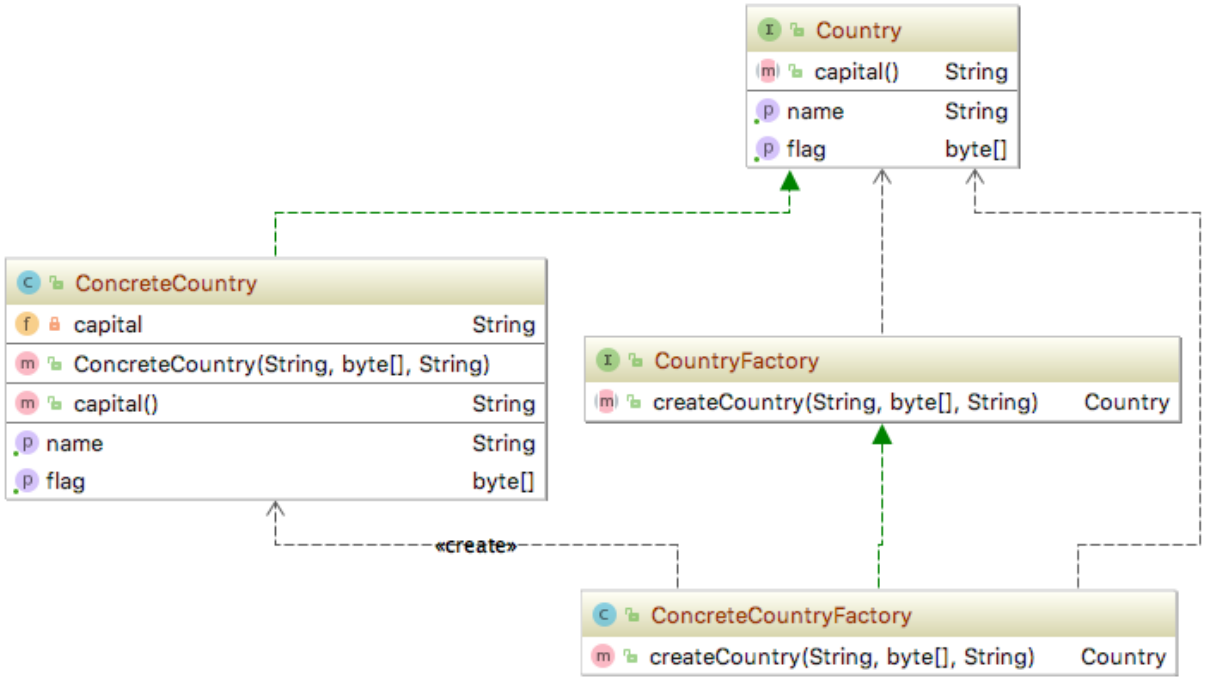
ConcreteProduct – The implementing class of **Product**. Objects of this class are created by the **ConcreteCreator**.

Creator – The `interface` that defines the factory methods

ConcreteCreator – The class that extends **Creator** and that provides an implementation for the factoryMethod. This can return any object that implements the **Product** interface.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method Demo Diagram



Factory Method Advantages

- Extensible
- Leave decision of specificity until later
- Subclass, not superclass, determines the kind of object to create
- You know when to create an object, but not what kind of an object.
- You need several overloaded constructors with the same parameter list, which is not allowed in Java. Instead, use several Factory Methods with different names.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method Disadvantages

- To create a new type you must create a separate subclass

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method: Product

```
public interface Country {
    public String getName();

    public byte[] getFlag();

    public String capital();
}
```

Factory Method: Concrete Product

```
public class ConcreteCountry implements Country {

    private String name;
    private String capital;
    private byte[] flagBytes;

    public ConcreteCountry(String name, byte[] flagBytes, String
capital) {
        this.name = name;
        this.capital = capital;
        this.flagBytes = flagBytes;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public byte[] getFlag() {
        return flagBytes;
    }

    @Override
    public String capital() {
        return capital;
    }
}
```

Factory Method: Creator

```
public interface CountryFactory {
    public Country createCountry(String name, byte[] flag, String
capital);
}
```

Factory Method: Concrete Creator

```
public class ConcreteCountryFactory implements CountryFactory {
    @Override
    public Country createCountry(String name, byte[] flag, String
capital) {
        return new ConcreteCountry(name, flag, capital);
    }
}
```

Abstract Factory Pattern

Abstract Factory Pattern Properties

Type: Creational, Object
Level: Component

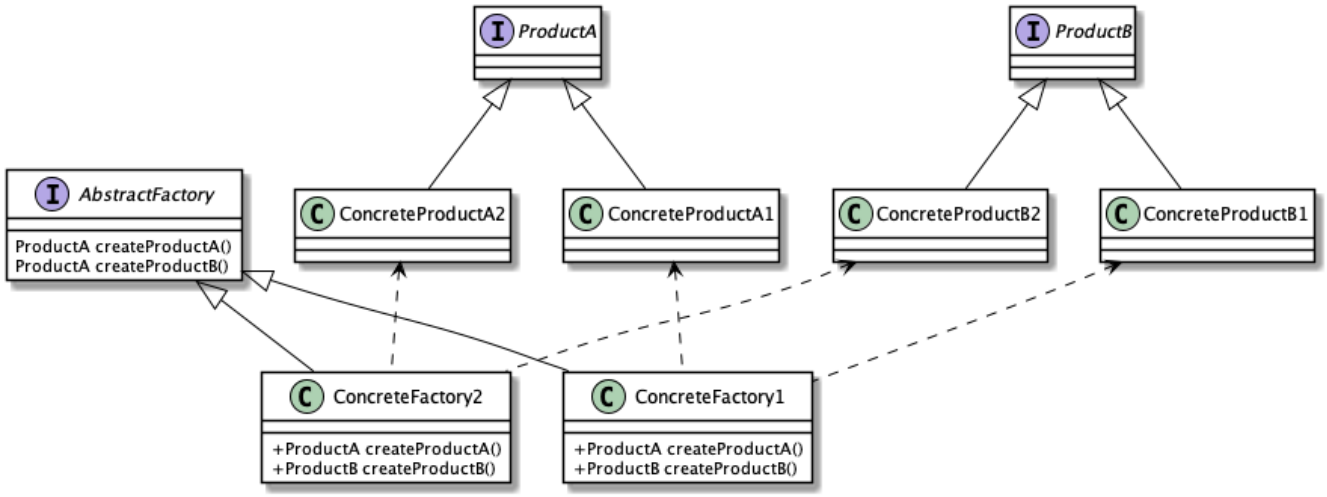
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Pattern Purpose

To provide a contract for creating families of related or dependent objects without having to specify their concrete classes.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Canonical Diagram



Abstract Factory Ingredients

AbstractFactory – An abstract class or interface that defines the create methods for abstract products.

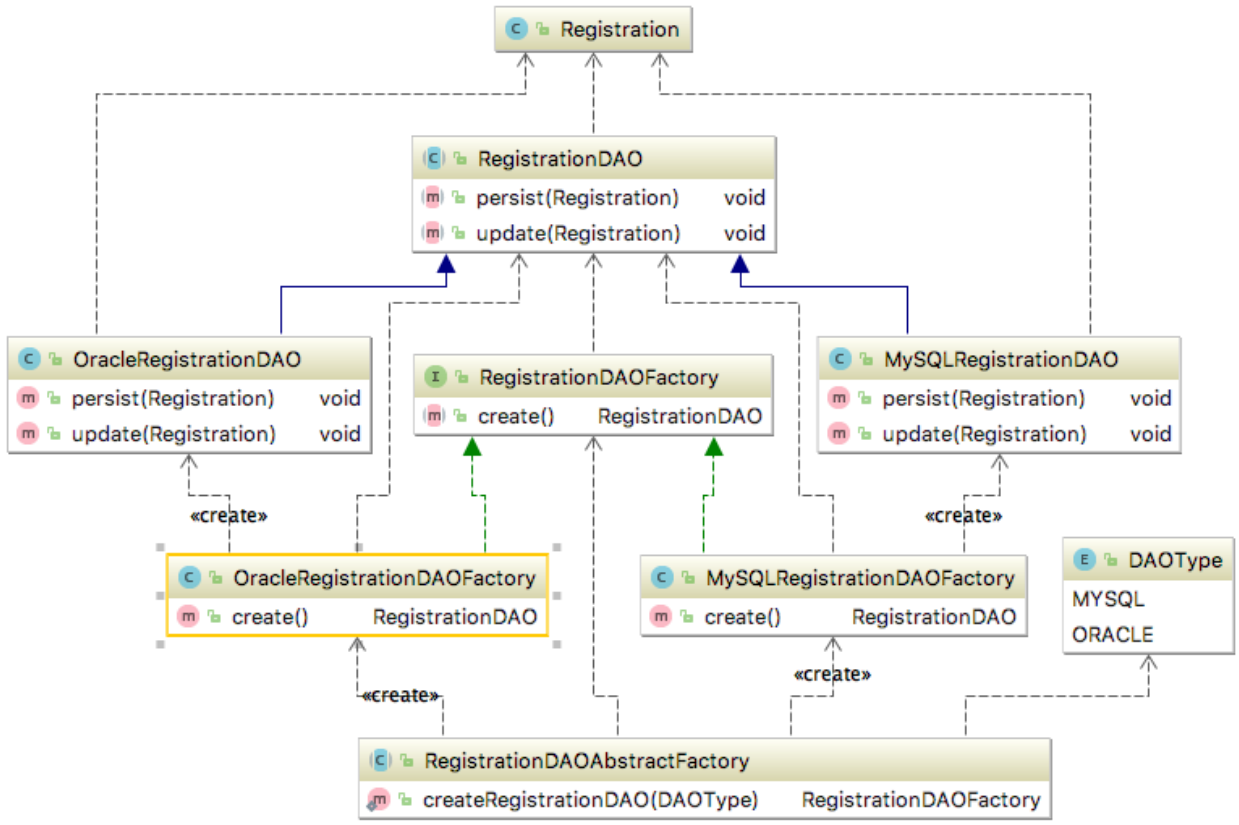
AbstractProduct – An abstract class or interface describing the general behavior of the resource that will be used by the application.

ConcreteFactory – A class derived from the abstract factory . It implements create methods for one or more concrete products.

ConcreteProduct – A class derived from the abstract product, providing an implementation for a specific resource or operating environment.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Demo Diagram



Abstract Factory Advantages

- Flexibility, the client is independent of how the products are created
- Application is configured with one of multiple families of products
- Objects need to be created as a set, in order to be compatible
- Provide a collection of classes and you want to reveal just their contracts and their relationships, not implementation

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Disadvantages

- An ill-defined abstraction can make things difficult later

Abstract Factory: AbstractFactory

```
public abstract class RegistrationDAO {
    public abstract void persist(Registration registration);
    public abstract void update(Registration registration);
}
```

Abstract Factory: ConcreteFactory1

```
public class OracleRegistrationDAOFactory extends RegistrationDAOFactory
{
   public RegistrationDAO create() {
      return new OracleRegistrationDAO();
   }
}
```

Abstract Factory: ConcreteFactory2

```
public class MySQLRegistrationDAOFactory extends RegistrationDAOFactory
{
   public RegistrationDAO create() {
      return new MySQLRegistrationDAO();
   }
}
```

Abstract Factory: Abstract Product

```
public abstract class RegistrationDAO {
   public abstract void setDataSource(DataSource dataSource);
   public abstract void persist(Registration registration);
   public abstract void update(Registration registration);
}
```

Abstract Factory: Concrete Product

```
public class MySQLRegistrationDAO extends RegistrationDAO {
   private DataSource dataSource;

   public MySQLRegistrationDAO() { }

   @Override
   public void setDataSource(DataSource dataSource) {..}

   @Override
   public void persist(Registration registration) {..}

   @Override
   public void update(Registration registration) {..}
}
```

Behavioral Patterns

State Pattern

State Pattern Properties

Type: Behavioral
Level: Object

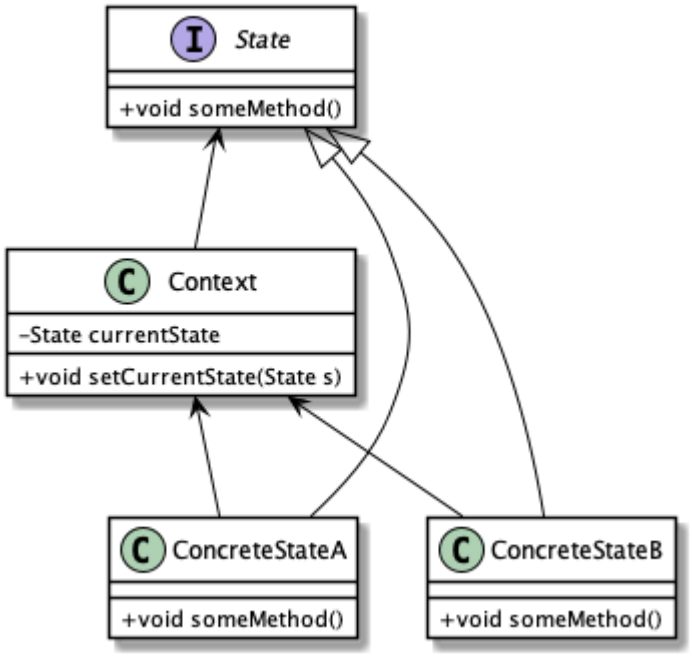
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Pattern Purpose

- Easily change an object's behavior at runtime.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Pattern Canonical Diagram



State Pattern Ingredients

Context – Keeps a reference to the current state, and is the interface for other clients to use. It delegates all state-specific method calls to the current State object.

State – Defines all the methods that depend on the state of the object.

ConcreteState – Implements the State interface, and implements specific behavior for one state.

State Pattern Advantages

- Behavior depends on its state and the state changes frequently
- Methods have large conditional statements that depend on the state of the object

- You need clarity on the change of state by focusing on the small segmentation
- Transitions are explicit and known
- States can be shared

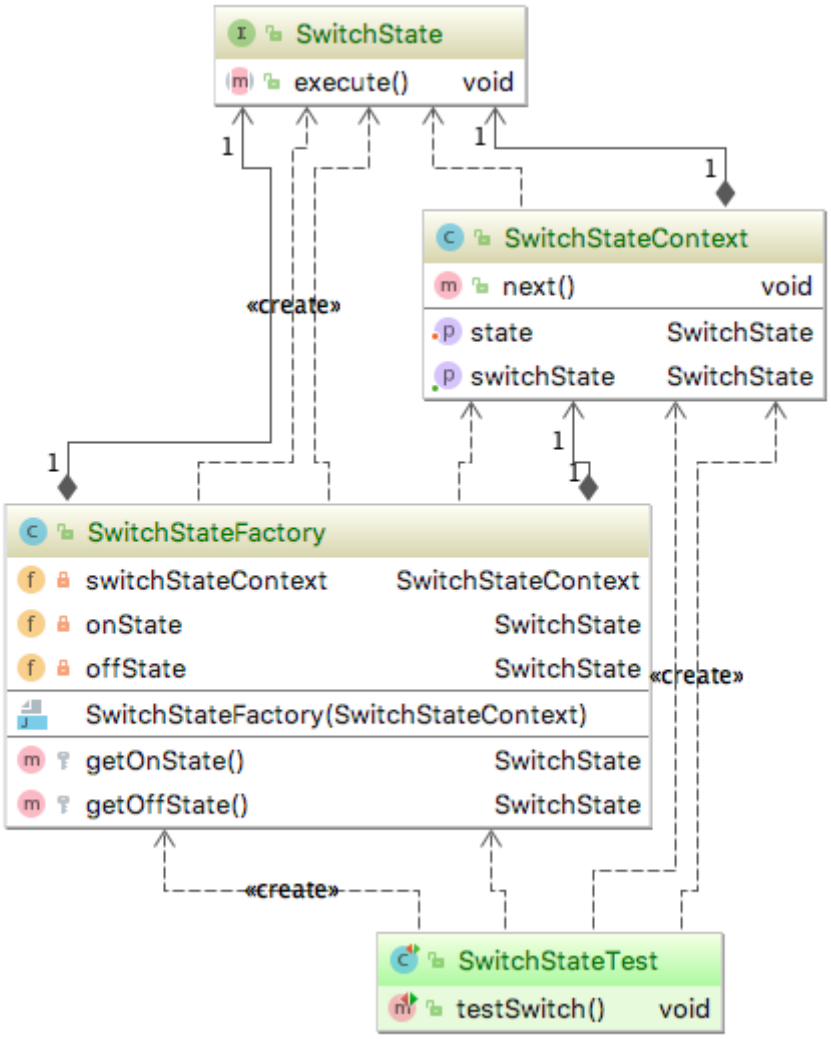
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Pattern Disadvantage

- Number of classes can increase
- Requires mutability, and therefore care in multi-threading

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Demo Diagram



State: Defining `State`

```
public interface SwitchState {
    void execute();
}
```

State: Create an `onState` with a `StateFactory`

The following is setting up an on state in a factory

```
public class SwitchStateFactory {
    private SwitchStateContext switchStateContext;
    private SwitchState onState;

    ...

    public SwitchStateFactory(SwitchStateContext switchStateContext) {
        this.switchStateContext = switchStateContext;
    }

    protected SwitchState getOnState() {
        if (this.onState == null) {
            this.onState = new SwitchState() {
                @Override
                public void execute() {
                    switchStateContext.setState(getOffState());
                }

                @Override
                public String toString() {
                    return "on";
                }
            };
        }
        return onState;
    }
}
```

State: Create an `offState` with a `StateFactory`

The following is setting up an off state in the `StateFactory`

```
public class SwitchStateFactory {
    private SwitchStateContext switchStateContext;
    private SwitchState offState;

    ...

    public SwitchStateFactory(SwitchStateContext switchStateContext) {
        this.switchStateContext = switchStateContext;
    }

    protected SwitchState getOffState() {
        if (this.offState == null) {
            this.offState = new SwitchState() {
                @Override
                public void execute() {
                    switchStateContext.setState(getOnState());
                }

                @Override
                public String toString() {
                    return "off";
                }
            };
        }
        return offState;
    }
}
```

State: Context

```
public class SwitchStateContext {
    private SwitchState switchState;

    public void setState(SwitchState switchState) {
        this.switchState = switchState;
    }

    public SwitchState getSwitchState() {
        return switchState;
    }

    public void next() {
        switchState.execute();
    }
}
```

Strategy Pattern

Strategy Pattern Properties

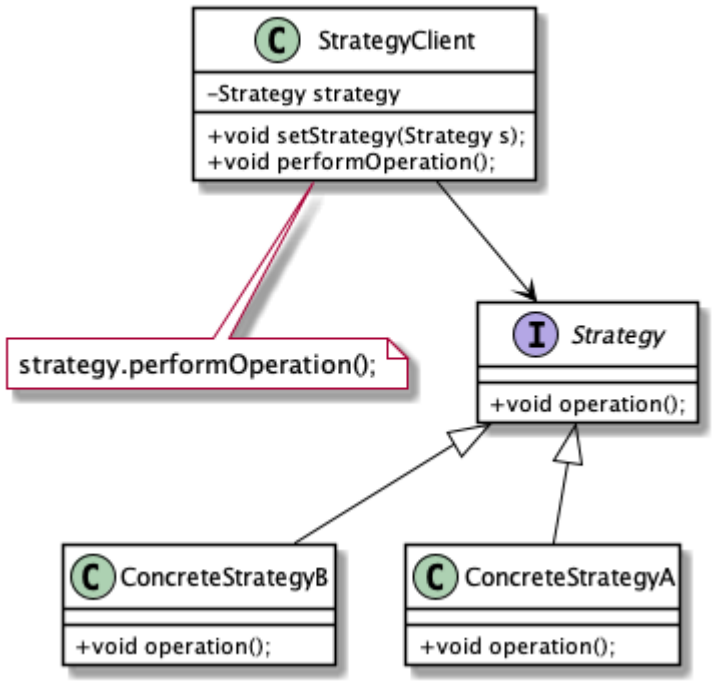
Type: Behavioral
Level: Component

Strategy Purpose

To define a group of classes that represent a set of possible behaviors. These behaviors can then be flexibly plugged into an application, changing the functionality on the fly.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Canonical Diagram



Strategy Ingredients

StrategyClient – This is the class that uses the different strategies for certain tasks. It keeps a reference to the Strategy instance that it uses and has a method to replace the current Strategy instance with another Strategy implementation.

Strategy – The interface that defines all the methods available for the StrategyClient to use.

ConcreteStrategy – A class that implements the Strategy interface using a specific set of rules for each of the methods in the interface.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Advantages

- You have a variety of ways to perform an action.
- You might not know which approach to use until runtime.
- You want to easily add to the possible ways to perform an action.
- You want to keep the code maintainable as you add behaviors.

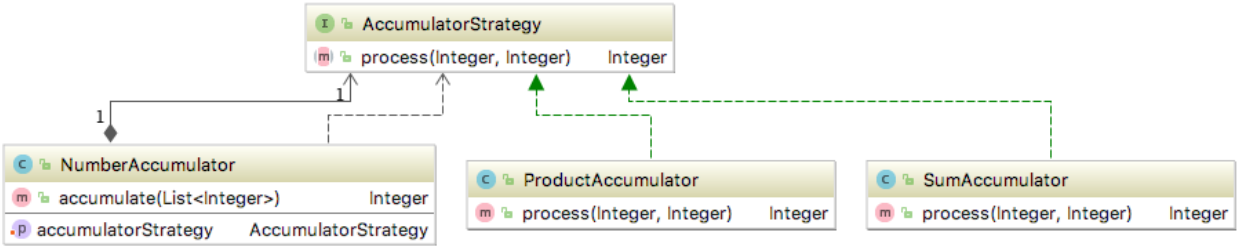
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Disadvantages

- Forethought and planning is required
- Identifying a strategy that is generic enough for this pattern

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Demo Diagram



Strategy: Strategy Interface

```
public interface AccumulatorStrategy {
    public Integer process(Integer a, Integer b);
}
```

Strategy: One Concrete Strategy

```
public class NumberAccumulator {
    private AccumulatorStrategy strategy;

    public void setAccumulatorStrategy(AccumulatorStrategy strategy) {
        this.strategy = strategy;
    }

    public Integer accumulate(List<Integer> integers) {
        if (integers.size() == 1) return integers.get(0);
        return strategy.process(integers.get(0),
            accumulate(integers.subList(1, integers.size())));
    }
}
```

Strategy: Another Concrete Strategy

```
public class ProductAccumulator implements AccumulatorStrategy {
    public Integer process(Integer a, Integer b) {
        return a * b;
    }
}
```

Strategy: Yet Another Concrete Strategy

```
public class SumAccumulator implements AccumulatorStrategy {
    public Integer process(Integer a, Integer b) {
        return a + b;
    }
}
```

Strategy: Use of the Strategy Pattern

```
NumberAccumulator numberAccumulator = new NumberAccumulator();
numberAccumulator.setAccumulatorStrategy(new ProductAccumulator());

List<Integer> integers = new ArrayList<Integer>();
integers.add(1);
integers.add(2);
integers.add(3);
integers.add(4);
```

Strategy: Use of the Strategy Pattern using Lambdas (Java 8+)

The following is the same as the above, but using lambdas and functional programming, much of the boilerplate is evaporated away

Using lambdas as a Strategy Pattern

```
List.of(1,2,3,4).stream().reduce((total, next) -> total + next);
```

Renders:

```
Optional[10]
```

Chain of Responsibility Pattern

Chain of Responsibility Properties

Type: Behavioral
Level: Component

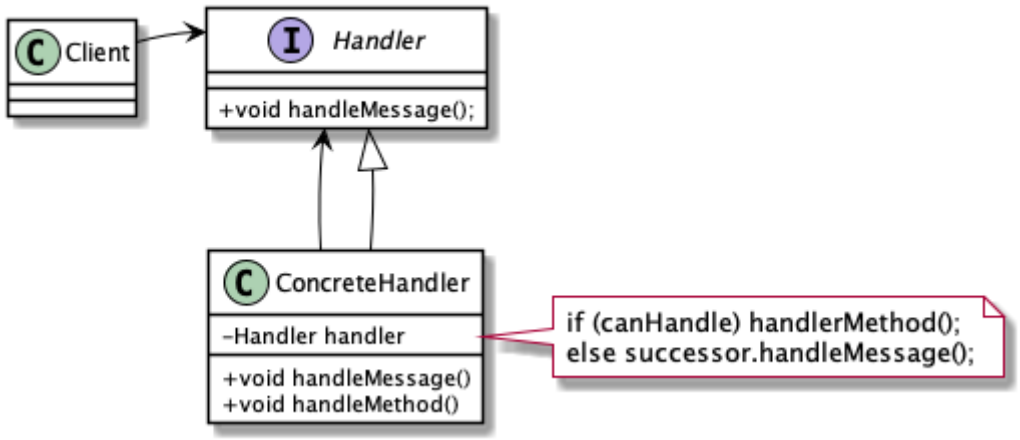
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Purpose

To establish a chain within a system, so that a message can either be handled at the level where it is first received, or be directed to an object that can handle it.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Canonical Diagram



Chain of Responsibility Ingredients

Handler – The interface that defines the method used to pass a message to the next handler. That message is normally just the method call, but if more data needs to be encapsulated, an object can be passed as well.

ConcreteHandler – A class that implements the Handler interface. It keeps a reference to the next Handler instance inline. This reference is either set in the constructor of the class or through a setter method. The implementation of the handleMessage method can determine how to handle the method and call a handleMethod, forward the message to the next Handler or a combination of both.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Advantages

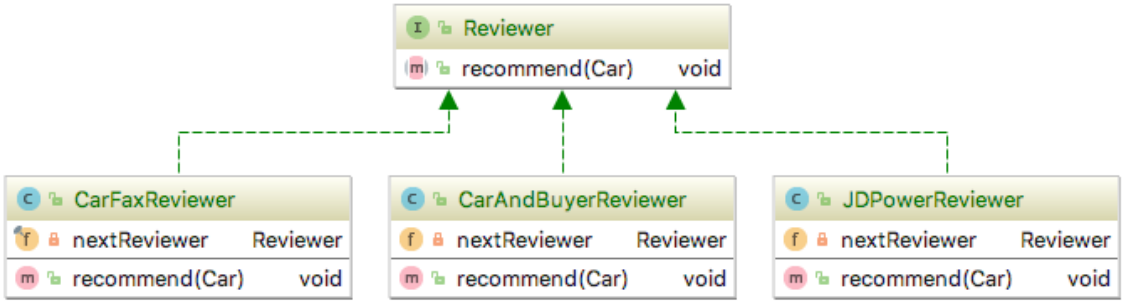
- There is a group of objects in a system that can all potentially respond to the same kind of message

- Offers complex message handling
- Messages must be handled by one of several objects within the system.
- Messages follow the “handle or forward” model—that is, some events can be handled at the level where they are received or produced, while others must be forwarded to some other object.

Chain of Responsibility Disadvantages

- Difficult to test and debug
- Possible dropped message if not handled

Chain of Responsibility Demo Diagram



Chain of Responsibility: Interface of a Model

```
import java.util.List;

public interface Car {
    String getMake();
    String getModel();
    int getYear();

    boolean powerSteering();
    boolean driverAirBags();
    boolean passengerAirBags();
    boolean seatHeaters();
    boolean seatCoolers();
    boolean driveLaneAssist();
    boolean rearCamera();
    void addRecommendation(String name);
    List<String> getRecommendations();
}
```

Chain of Responsibility: Handler

```
public interface Reviewer {
    void recommend(Car car);
}
```

Chain of Responsibility: Concrete Handler

```
public class CarAndBuyerReviewer implements Reviewer {

    private Reviewer nextReviewer;

    public CarAndBuyerReviewer(Reviewer nextReviewer) {
        this.nextReviewer = nextReviewer;
    }

    public CarAndBuyerReviewer() {
        this.nextReviewer = null;
    }

    @Override
    public void recommend(Car car) {
        if (car.passengerAirBags() && car.driverAirBags())
           car.addRecommendation("Car and Buyer");
        if (nextReviewer != null)
           nextReviewer.recommend(car);
    }
}
```

Chain of Responsibility: Another Concrete Handler

```
public class CarFaxReviewer implements Reviewer {

    private final Reviewer nextReviewer;

    public CarFaxReviewer(Reviewer nextReviewer) {
        this.nextReviewer = nextReviewer;
    }

    public CarFaxReviewer() {
        this.nextReviewer = null;
    }

    @Override
    public void recommend(Car car) {
        if (car.driverAirBags())
           car.addRecommendation("CarFax");
        if (nextReviewer != null)
           nextReviewer.recommend(car);
    }
}
```

Chain of Responsibility: Yet Another Concrete Handler

```
public class JDPowerReviewer implements Reviewer {

    private Reviewer nextReviewer;

    public JDPowerReviewer(Reviewer reviewer) {
        this.nextReviewer = reviewer;
    }

    public JDPowerReviewer() {
        this.nextReviewer = null;
    }

    @Override
    public void recommend(Car car) {
        if (car.rearCamera() && car.driveLaneAssist() &&
            car.powerSteering())
            car.addRecommendation("JD Power");
        if (nextReviewer != null) nextReviewer.recommend(car);
    }
}
```

Chain of Responsibility: Running

```
CarFaxReviewer carFaxReviewer = new CarFaxReviewer();
JDPowerReviewer jdPowerReviewer = new JDPowerReviewer(carFaxReviewer);
CarAndBuyerReviewer carAndBuyerReviewer = new CarAndBuyerReviewer
(jdPowerReviewer);

carAndBuyerReviewer.recommend(car);

System.out.println(car.getRecommendations());
```

Command Pattern

Command Pattern Properties

Type: Behavioral
Level: Object

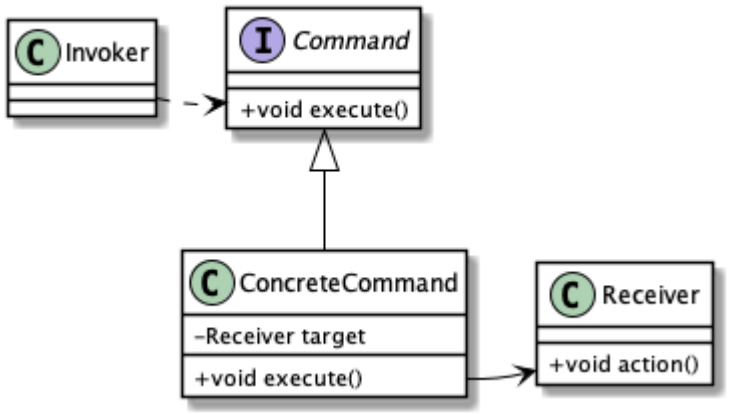
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Command Pattern Purpose

- To wrap a command in an object so that it can be stored, passed into methods, and returned like any other object.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Command Pattern Canonical Diagram



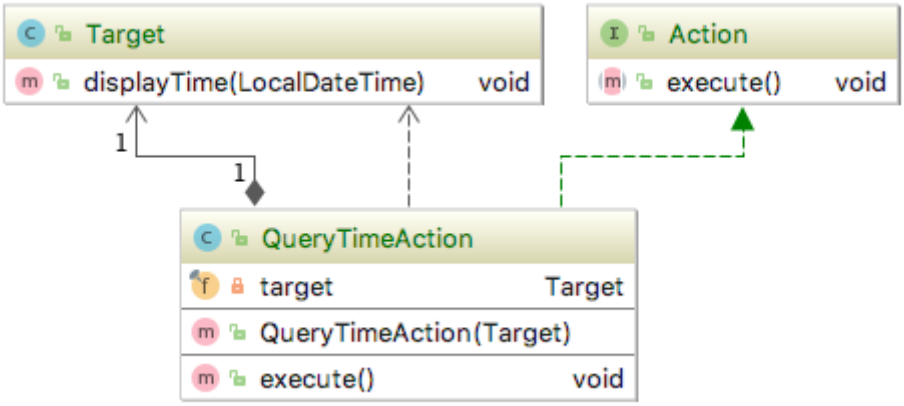
Command Pattern Advantages

- Decoupling the source or trigger of the event from the object that has the knowledge to perform the task.
- Sharing Command instances between several objects.
- Allowing the replacement of Commands and/or Receivers at runtime.
- Making Commands regular objects, thus allowing for all the normal properties.
- Easy addition of new Commands; just write another implementation of the interface and add it to the application.

Command Pattern Disadvantages

- Not beneficial with too few commands

Command Demo Diagram



Command: The command interface

```
public interface Action {
    public void execute();
}
```

Command: The command target

```
import javax.swing.<strong>;
import java.awt.</strong>;
import java.time.LocalDateTime;

public class Target {
    public void displayTime(LocalDateTime localDateTime) {
        JFrame jFrame = new JFrame("Title");
        JPanel contentPane = new JPanel(new FlowLayout());
        JLabel jLabel = new JLabel("The time is: " +
           localDateTime.toString());
        contentPane.add(jLabel);
        jFrame.setContentPane(contentPane);
        jFrame.pack();
        jFrame.setVisible(true);
        jFrame.setDefaultCloseOperation(
           WindowConstants.DISPOSE_ON_CLOSE);
    }
}
```

Command: A command action

The nice thing about the command pattern is that it is easy to add runtime actions!

```
import java.time.LocalDateTime;

public class QueryTimeAction implements Action{

    private final Target target;

    public QueryTimeAction(Target target) {
        this.target = target;
    }

    @Override
    public void execute() {
        target.displayTime(LocalDateTime.now());
    }
}
```

Command: Bringing it together

Using a `main` method or a dependency injection container:

```
Map<String, Action> commandMap = new HashMap<>();
Target target = new Target();
Action action = new QueryTimeAction(target);
commandMap.put("showTime", action);
commandMap.get("showTime").execute();
```

Command: Bringing it together with Java 8 lambdas

Using a `main` method or a dependency injection container:

```
Map<String, Action> commandMap = new HashMap<>();
Target target = new Target();
Action action = () -> {
   target.displayTime(LocalDateTime.now());
};
commandMap.put("showTime", action);
commandMap.get("showTime").execute();
```

Iterator Pattern

Iterator Pattern Properties

Type: Behavioral, Object
Level: Component

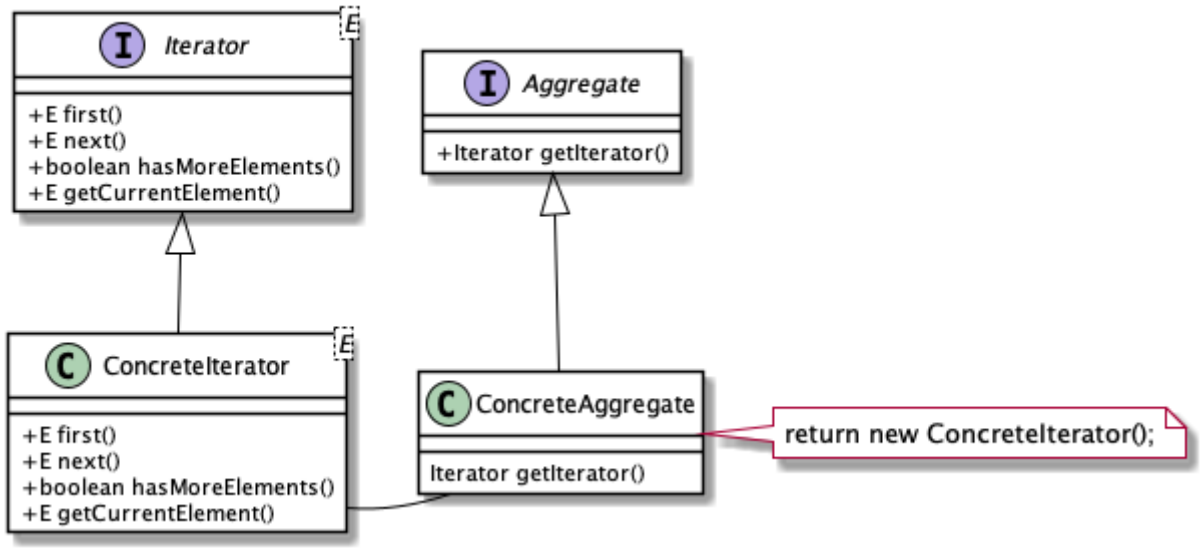
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Iterator Purpose

To provide a consistent way to sequentially access items in a collection that is independent of and separate from the underlying collection.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Iterator Canonical Diagram



Iterator Ingredients

Iterator – This interface defines the standard iteration methods. At a minimum, the interface defines methods for navigation, retrieval and validation (`first`, `next`, `hasMoreElements` and `getCurrentItem`)

ConcreteIterator – Classes that implement the `Iterator`. These classes reference the underlying collection. Normally, instances are created by the `ConcreteAggregate`. Because of the tight coupling with the `ConcreteAggregate`, the `ConcreteIterator` often is an inner class of the `ConcreteAggregate`.

Aggregate – This interface defines a factory method to produce the `Iterator`.

ConcreteAggregate – This class implements the Aggregate, building a `ConcreteIterator` on demand. The `ConcreteAggregate` performs this task in addition to its fundamental responsibility of representing a collection of objects in a system. `ConcreteAggregate` creates the

`ConcreteIterator` instance.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Iterator Advantages

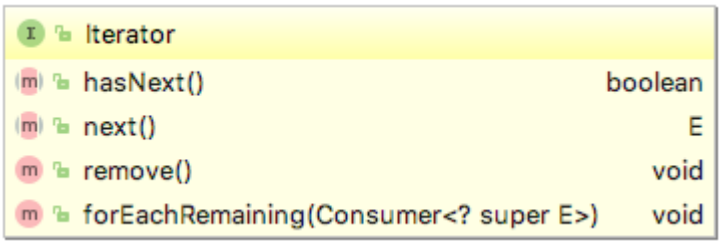
- A uniform interface for traversing a collection
- Not to tied to the implementation of the collection
- Enabling several clients to simultaneously navigate within the same underlying collection.
- You can think of an Iterator as a cursor or pointer into the collection

Iterator Disadvantages

- They give the illusion of order to unordered structures

Iterator Demo Diagram

JDK 8+ UML Diagram



Iterator: Java's implementation (Java 8+)

```
public interface Iterator<E> {
    boolean hasNext();

    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }

    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

The above is the Java 8+ signature

Use of Java's Iterator

```
var stringList = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
Iterator<String> iterator = stringList.iterator();

String value1 = iterator.next();
String value2 = iterator.next();

assertEquals(value1, "Foo"); //true
assertEquals(value2, "Bar"); //true
```

Java Iterator Trick Question

What is `value1` and `value2`?

```
var stringList = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
String value1 = stringList.iterator().next();
String value2 = stringList.iterator().next();
```

Using Java Iterator with `while`

```
var iterator = List.of("Foo", "Bar", "Baz", "Qux", "Quux").iterator();
var result = new ArrayList<String>();
while(iterator.hasNext()) {
    result.add(iterator.next());
}
assertEquals("[Foo, Bar, Baz, Qux, Quux]",result.toString());
```

Using Java Iterator with Java 5 for loop

If a collection extends from `Iterable` it can be placed in a loop

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var result = new ArrayList<String>();
for (String s : list) {
    result.add(s);
}
assertEquals("[Foo, Bar, Baz, Qux, Quux]",result.toString());
```

Using Java's `ListIterator`

- An iterator for lists that traverses the list in either direction
- Modify the list during iteration
- Obtain the iterator's current position in the list

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var listIterator = list.listIterator();
listIterator.next();
listIterator.next();
listIterator.next();
listIterator.previous();
listIterator.previous();
listIterator.next();
assertEquals(listIterator.next(), "Baz");
```

Using Java's `Spliterator`

- Iterator that "splits" perhaps for parallel purposes
- Traverse elements individually using `tryAdvance`
- Traverse elements sequentially in bulk using `forEachRemaining`
- `Spliterator` can be queried with characteristics about the underlying collection

Using `forEachRemaining` with a `Spliterator`

`forEachRemaining` iterates all elements using given `Consumer`

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var split1 = list.spliterator();
var split2 = split1.trySplit();
split1.forEachRemaining(x -> System.out.println("S1 " + x));
split2.forEachRemaining(x -> System.out.println("S2 " + x));
```

Will result in:

```
S1 Baz
S1 Qux
S1 Quux
S2 Foo
S2 Bar
```

Using `tryAdvance` with a `Spliterator`

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var split1 = list.spliterator();
var split2 = split1.trySplit();
split1.tryAdvance(x -> System.out.println("S1 " + x));
split2.tryAdvance(x -> System.out.println("S2 " + x));
```

Will result in:

```
S1 Baz
S2 Foo
```

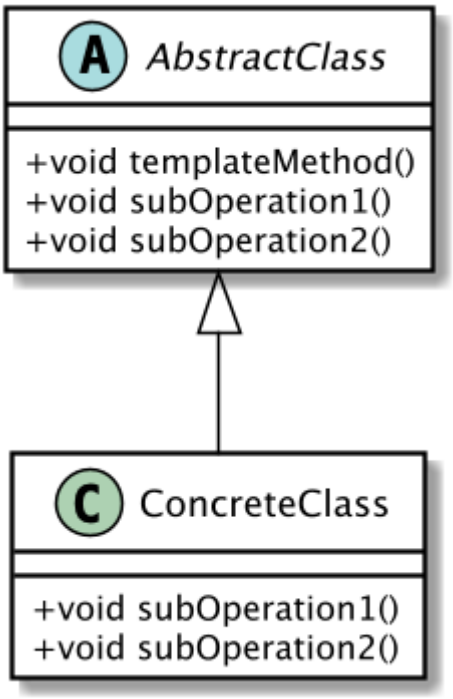
Template Method Pattern

Template Method Properties

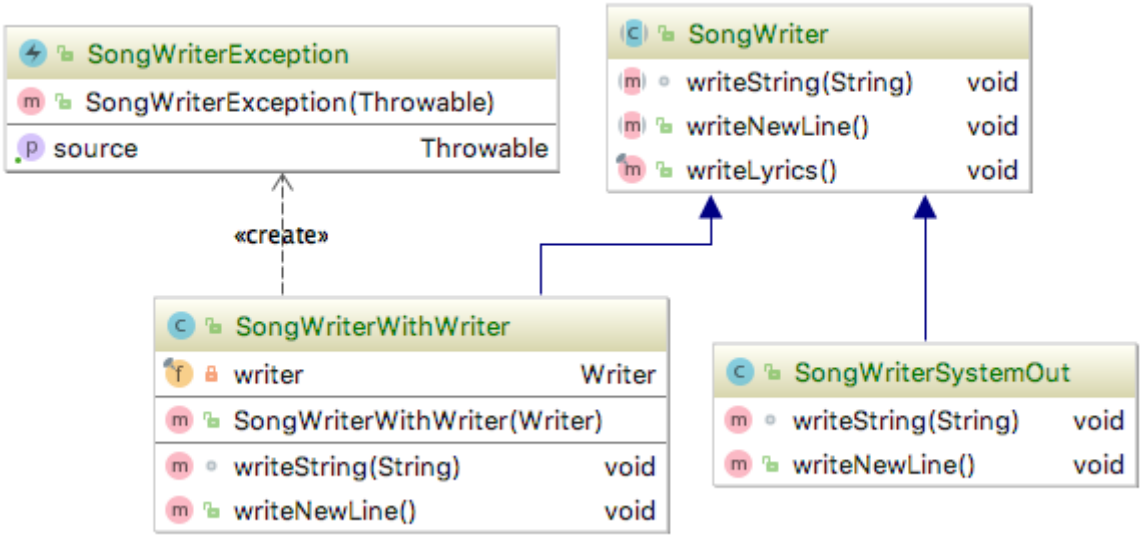
Type: Behavioral Level: Object

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Template Method Canonical Diagram



Template Method Diagram



Template Method Purpose

- To provide a method that allows subclasses to override parts of the method without rewriting it.
- To provide a skeleton structure for a method
- Allow subclasses to redefine specific parts of the method.
- To centralize pieces of a method that are defined in all subtypes of a class
- Always have a small difference in each subclass.
- Control which operations subclasses are required to override.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Template Method: The `abstract` class

```
public abstract class SongWriter {
    abstract void writeString(String str) throws SongWriterException;
    public abstract void writeNewLine() throws SongWriterException;

    public final void writeLyrics() throws SongWriterException {
        writeString("I see trees of green");
        writeNewLine();
        writeString("red roses too, I see them bloom");
        writeNewLine();
        writeString("for me and you");
        writeNewLine();
        writeString("and I think to myself");
        writeNewLine();
        writeString("what a wonderful world");
        writeNewLine();
    }
}
```

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Template Method: One possible implementation

```
import java.io.IOException;
import java.io.Writer;

public class SongWriterWithWriter extends SongWriter {
    private final Writer writer;

    public SongWriterWithWriter(Writer writer) {
        this.writer = writer;
    }

    @Override
    void writeString(String str) throws SongWriterException {
        try {
            writer.write(str);
        } catch (IOException e) {
            throw new SongWriterException(e);
        }
    }

    @Override
    public void writeNewLine() throws SongWriterException {
        try {
            writer.write("\n");
        } catch (IOException e) {
            throw new SongWriterException(e);
        }
    }
}
```

Template Method: Another possible implementation

```
public class SongWriterSystemOut extends SongWriter {
    @Override
    void writeString(String str) throws SongWriterException {
        System.out.println(str);
    }

    @Override
    public void writeNewLine() throws SongWriterException {
        System.out.println();
    }
}
```

Template Method can be done with `default` methods

```
public interface LineItem {
    public float getCost();
    public float getQuantity();
    public float getDiscount();
    public default float getSubtotal() {
        return (getCost() * getQuantity()) * getDiscount();
    }
}
```

Mediator Pattern

Mediator Pattern Properties

Type: Behavioral
Level: Component

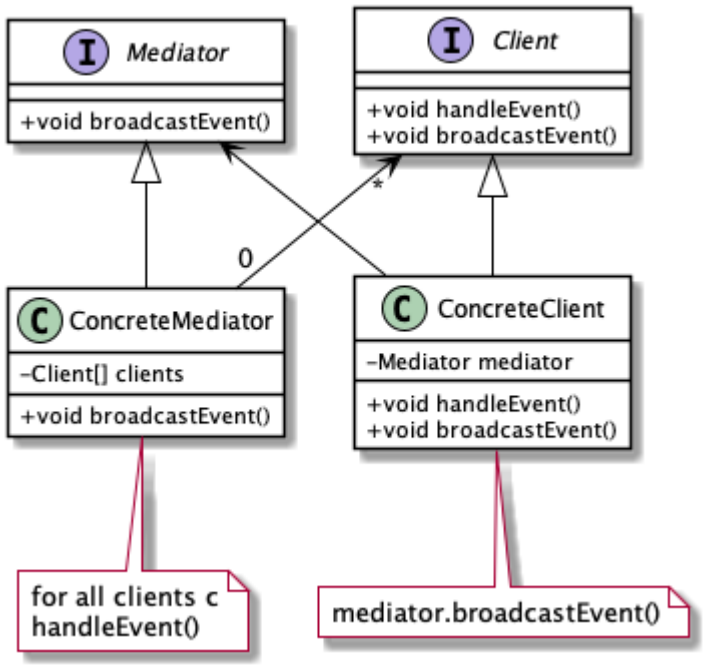
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Purpose

Simplify communication among objects in a system by introducing a single object that manages message distribution among the others

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Canonical Diagram



Mediator Ingredients

Mediator – The interface that defines the methods clients can call on a `Mediator`.

ConcreteMediator – The class that implements the `Mediator` interface. This class mediates among several client classes. It contains application-specific information about processes, and the `ConcreteMediator` might have some hardcoded references to its clients. Based on the information the Mediator receives, it can either invoke specific methods on the clients, or invoke a generic method to inform clients of a change or a combination of both.

Client – The interface that defines the general methods a `Mediator` can use to inform client instances.

ConcreteClient – A class that implements the `Client` interface and provides an implementation to each of the client methods. The `ConcreteClient` can keep a reference to a `Mediator` instance to inform colleague clients of a change (through the `Mediator`).

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Advantages

- There are complex rules for communication among objects in a system (often as a result of the business model).
- You want to keep the objects simple and manageable.
- You want the classes for these objects to be redeployable, not dependent on the business model of the system.
- The individual components become simpler and easier to deal with, since they no longer need to directly pass messages to each other.
- Components are more generic, no longer need to contain logic to deal with their communication with other components
- Communications strategy becomes easier, since it is now the exclusive responsibility of the mediator

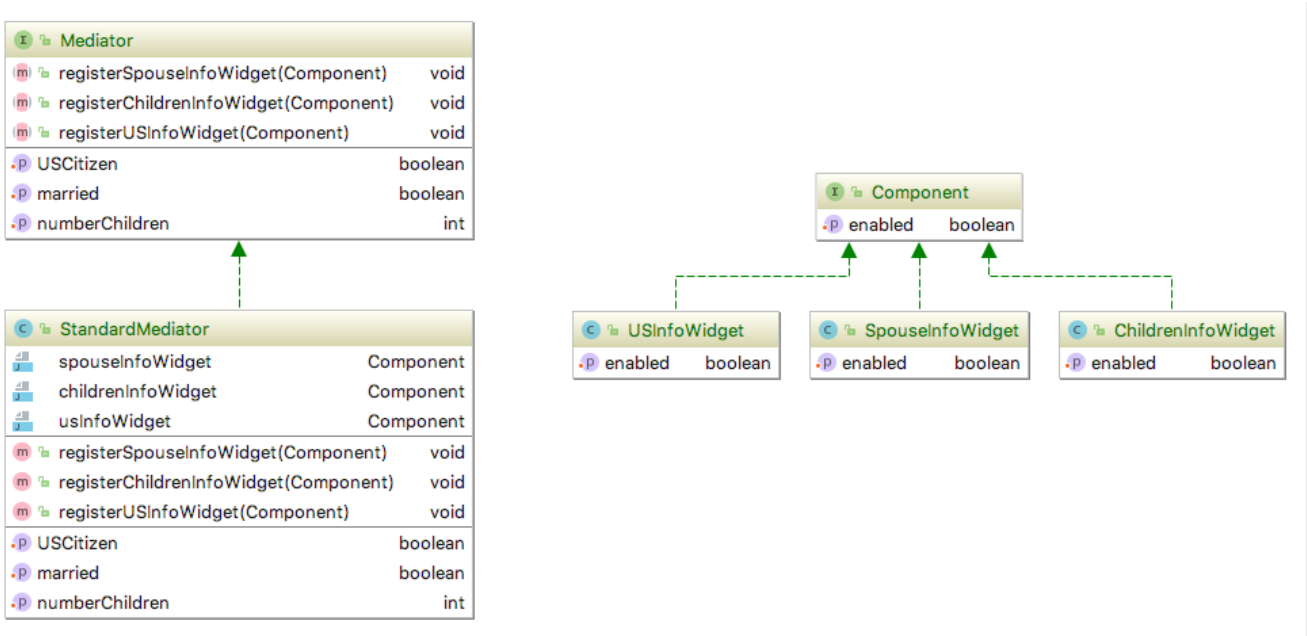
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Disadvantages

- The Mediator is often application specific and difficult to redeploy
- Testing and debugging complex Mediator implementations can be challenging
- The Mediator's code can become hard to manage as the number and complexity of participants increases

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Demo Diagram



Mediator: The Mediator Interface

```
public interface Mediator {
    void setUSCitizen(boolean isUSCitizen);
    void setNumberChildren(int children);
    void setMarried(boolean isMarried);
    void registerSpouseInfoWidget(Component component);
    void registerChildrenInfoWidget(Component component);
    void registerUSInfoWidget(Component component);
}
```

Mediator: The Mediator Implementation Registration

```
public class StandardMediator implements Mediator {
    private Component spouseInfoWidget;
    private Component childrenInfoWidget;
    private Component usInfoWidget;

    @Override
    public void registerSpouseInfoWidget(Component component) {
        this.spouseInfoWidget = component;
    }

    @Override
    public void registerChildrenInfoWidget(Component component) {
        this.childrenInfoWidget = component;
    }

    @Override
    public void registerUSInfoWidget(Component component) {
        this.usInfoWidget = component;
    }
}
```

Mediator: The Mediator Implementation Activation

```
public class StandardMediator implements Mediator {
    private Component spouseInfoWidget;
    private Component childrenInfoWidget;
    private Component usInfoWidget;

    //Switches

    @Override
    public void setUSCitizen(boolean isUSCitizen) {
        if (usInfoWidget != null) {
            if (isUSCitizen) {
                usInfoWidget.setEnabled(true);
            } else {
                usInfoWidget.setEnabled(false);
            }
        }
    }

    @Override
    public void setNumberChildren(int children) {
        if (childrenInfoWidget != null) {
            if (children > 0) {
                childrenInfoWidget.setEnabled(true);
            } else {
                childrenInfoWidget.setEnabled(false);
            }
        }
    }

    @Override
    public void setMarried(boolean isMarried) {
        if (spouseInfoWidget != null) {
            if (isMarried) {
                spouseInfoWidget.setEnabled(true);
            } else {
                spouseInfoWidget.setEnabled(false);
            }
        }
    }
}
```

Memento Pattern

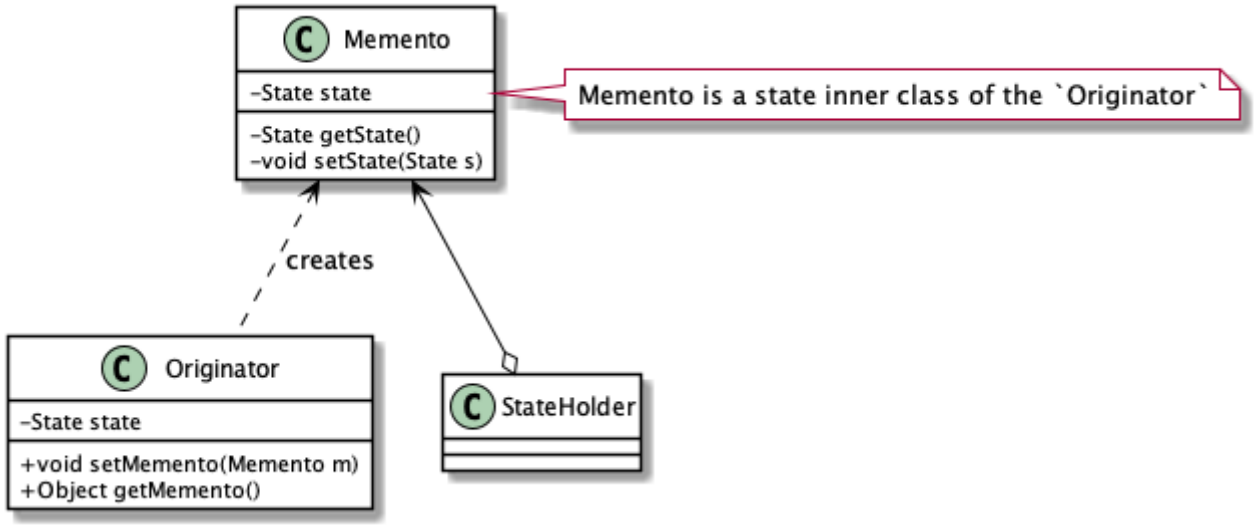
Memento Pattern Properties

Type: Behavioral
Level: Object

Memento Purpose

- To preserve a "snapshot" of an object's state
- Object can return to its original state without having to reveal its content to the rest of the world

Memento Canonical Diagram



Memento Advantages

- A snapshot of the state of an object should be taken.
- That snapshot is used to recreate the original state.
- Doesn't not expose internal state

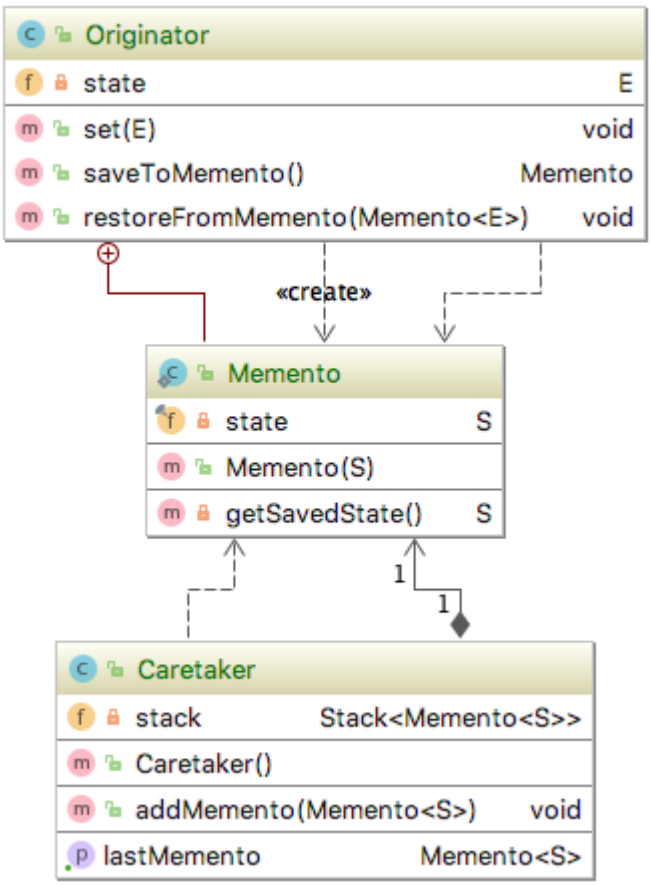
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Memento Disadvantages

- Expensive Storage Overtime
- Requires thought with large graphs

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Memento Demo Diagram



Memento: Originator

```
public class Originator<E> {
    private E state;
    // The class could also contain
    // additional data that is not part of the
    // state saved in the memento..

    public void set(E state) {
        this.state = state;
        System.out.println("Originator: Setting" +
                           " state to " + state);
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento<>(this.state);
    }

    public void restoreFromMemento(Memento<E> memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after" +
                           "restoring from Memento: " +
                           state);
    }

    //Memento Declaration...
}
```

Source: https://en.wikipedia.org/wiki/Memento_pattern

Memento: Memento inside the Originator

```
public class Originator<E> {

    //Memento declaration and methods in previous slide

    public static class Memento<S> {
        private final S state;

        public Memento(S stateToSave) {
            state = stateToSave;
        }

        // accessible by outer class only
        private S getSavedState() {
            return state;
        }
    }
}
```

Source: https://en.wikipedia.org/wiki/Memento_pattern

Memento: StateHolder

- `Caretaker` is a state holder will hold the different `Memento`
- You can use whatever collection to recall the previous state

```
import java.util.Stack;

public class Caretaker<S> {
    private Stack<Originator.Memento<S>> stack;

    public Caretaker() {
        this.stack = new Stack<>();
    }

    public void addMemento(Originator.Memento<S> memento) {
        this.stack.push(memento);
    }

    public Originator.Memento<S> getLastMemento() {
        return this.stack.pop();
    }
}
```

Observer Pattern

Observer Pattern Properties

Type: Behavioral
Level: Component

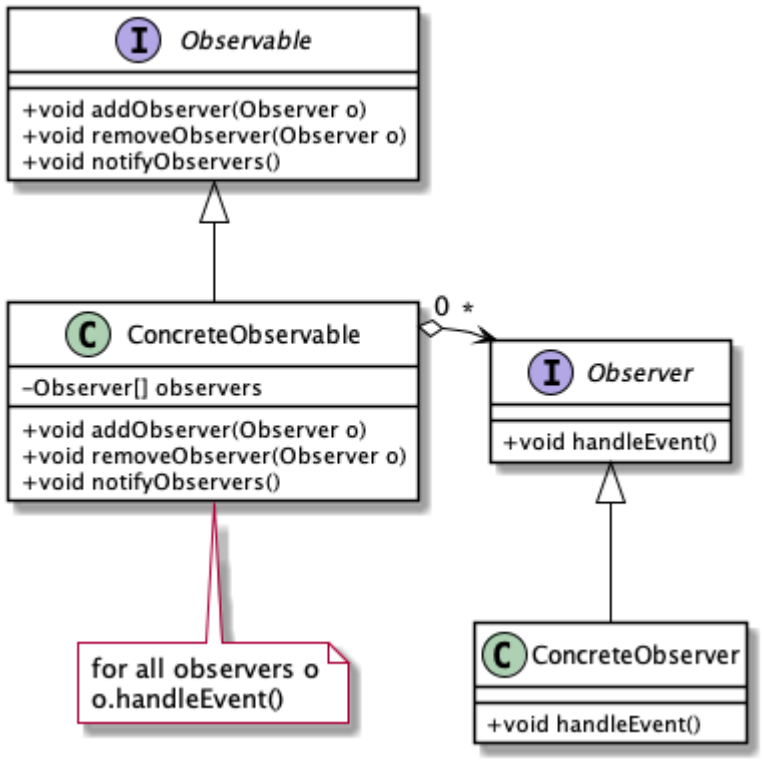
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Observer Pattern Purpose

To provide a way for a component to flexibly broadcast messages to interested receivers.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Observer Canonical Pattern



Observer Ingredients

Observable – The interface that defines how the observers/clients can interact with an `Observable`. These methods include adding and removing observers, and one or more notification methods to send information through the `Observable` to its clients.

ConcreteObservable – A class that provides implementations for each of the methods in the `Observable` interface. It needs to maintain a collection of Observers.

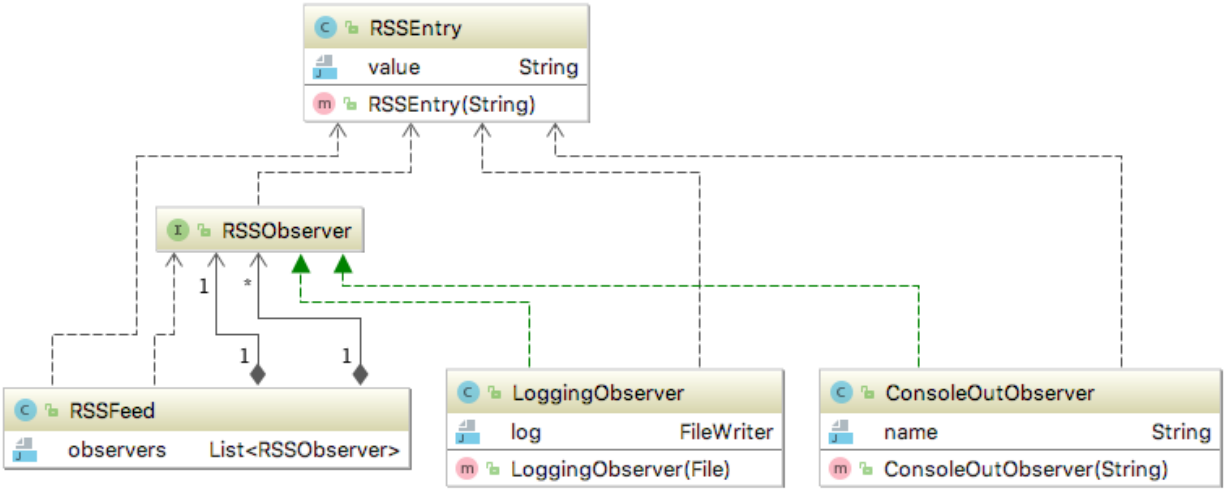
The notification methods copy (or clone) the `Observer` list and iterate through the list, and call the specific listener methods on each `Observer`.

Observer – The interface the `Observer` uses to communicate with the clients.

ConcreteObserver – Implements the `Observable` interface and determines in each implemented method how to respond to the message received from the `Observable`.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Observer Demo Diagram



Observer: Object

```
public class RSSEntry {

    private String value;

    public RSSEntry(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

Observer: Concrete Observable

```
public class RSSFeed {

    private List<RSSObserver> observers = new ArrayList<RSSObserver>();

    public void broadcast(RSSEntry entry) {
        for (RSSObserver observer : observers) {
            observer.update(entry);
        }
    }

    public RSSObserver addObserver(RSSObserver observer) {
        observers.add(observer);
        return observer;
    }

    public void removeObserver(RSSObserver observer) {
        observers.remove(observer);
    }
}
```

Observer: Observer

```
public interface RSSObserver {

    void update(RSSEntry entry);
}
```

Observer: Concrete Observer

```
public class LoggingObserver implements RSSObserver {

    private FileWriter log;

    public LoggingObserver(File log) {
        try {
            this.log = new FileWriter(log);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void update(RSSEntry entry) {

        try {
            log.write(entry.getValue());
            log.write('\n');
            log.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Observer: Another Concrete Observer

```
public class ConsoleOutObserver implements RSSObserver {

    private String name;

    public ConsoleOutObserver(String name) {
        this.name = name;
    }

    public void update(RSSEntry entry) {
        System.out.println(name + " : " + entry.getValue());
    }
}
```

Structural Patterns

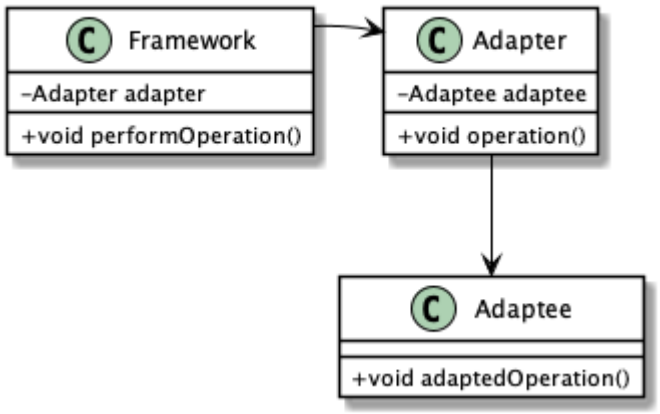
Adapter Pattern

Adapter Pattern Purpose

- To act as an intermediary between two classes
- Converting the interface of one class so that it can be used with another

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Adapter Pattern Canonical Diagram



Adapter Pattern Advantages

- Code Reuse
- Apply a different interface
- Translate code from another language

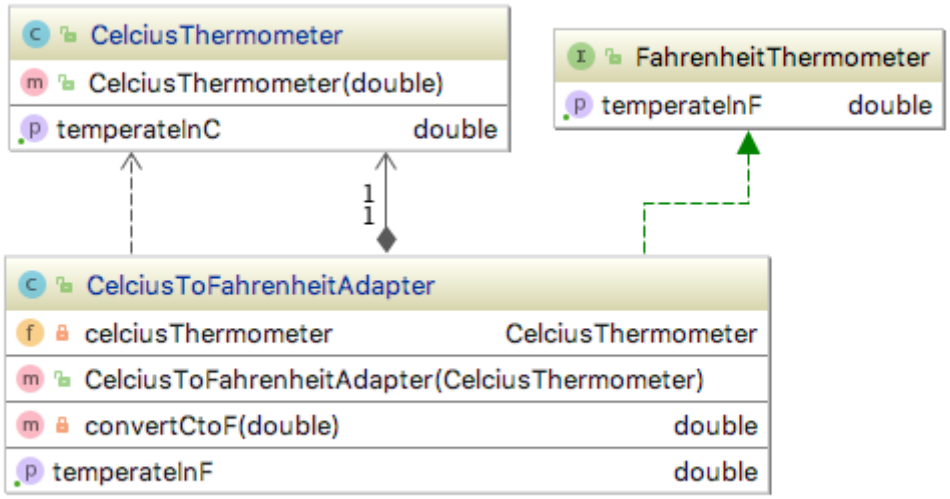
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Adapter Pattern Disadvantages

- The parameters may not be the same
- May require more work if the methods are substantial

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Adapter Pattern Demo Diagram



Adapter Pattern: The Target (Adaptee)

```
public class CelciusThermometer {

    private double temp;

    public CelciusThermometer(double temp) {
        this.temp = temp;
    }

    public double getTemperateInC() {
        return temp;
    }
}
```

Adapter Pattern: The Adapter

```
public class CelciusToFahrenheitAdapter
   implements FahrenheitThermometer {

    private CelciusThermometer celciusThermometer;

    public CelciusToFahrenheitAdapter
              (CelciusThermometer celciusThermometer) {
        this.celciusThermometer = celciusThermometer;
    }

    public double getTemperateInF() {
        return convertCtoF(celciusThermometer.getTemperateInC());
    }

    private double convertCtoF(double c) {
        return (c * 9 / 5) + 32;
    }

}
```

Adapter Pattern: The Adapter's Interface

```
public interface FahrenheitThermometer {
    double getTemperateInF();
}
```

Bridge Pattern

Bridge Pattern Properties

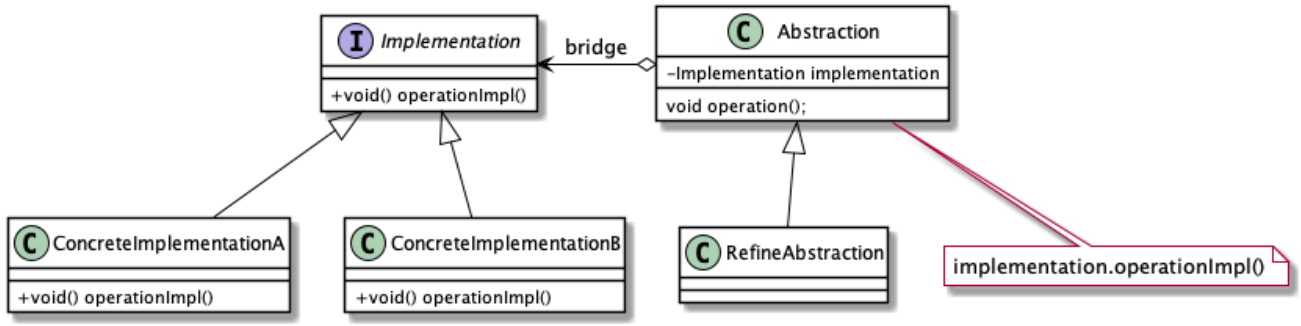
Type: Structural, Object
Level: Component

Bridge Pattern Purpose

- To divide a complex component into two separate but related inheritance hierarchies:
  - The functional Abstraction
  - The internal implementation
- This makes it easier to change either aspect of the component

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Bridge Pattern Canonical Diagram



Bridge Pattern Advantages

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don’t affect the client.

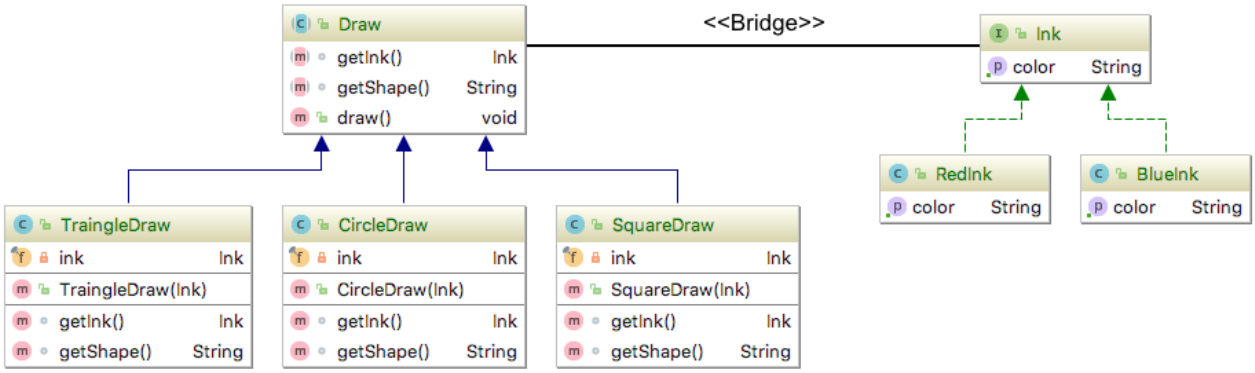
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Bridge Pattern Disadvantages

- Useful in graphics and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity and components

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Bridge Demo Diagram



Bridge: Abstraction

```
public abstract class Draw {

    abstract Ink getInk();
    abstract String getShape();

    public void draw() {
        System.out.println("Drawing a " + getShape() +
                           " with " + getInk().getColor()
                            + " ink");
    }
}
```

Bridge: Refine Abstraction

```
public class SquareDraw extends Draw {

    private final Ink ink;

    public SquareDraw(Ink ink) {
        this.ink = ink;
    }

    @Override
    Ink getInk() {
        return ink;
    }

    @Override
    String getShape() {
        return "square";
    }
}
```

Bridge: Implementation

```
public interface Ink {
    String getColor();
}
```

Bridge: Concrete Implementation

```
public class RedInk implements Ink {
    @Override
    public String getColor() {
        return "red";
    }
}
```

Bridge: Another Concrete Implementation

```
public class BlueInk implements Ink {
    @Override
    public String getColor() {
        return "blue";
    }
}
```

Composite Pattern

Composite Properties

Type: Structural, Object **Level:** Component

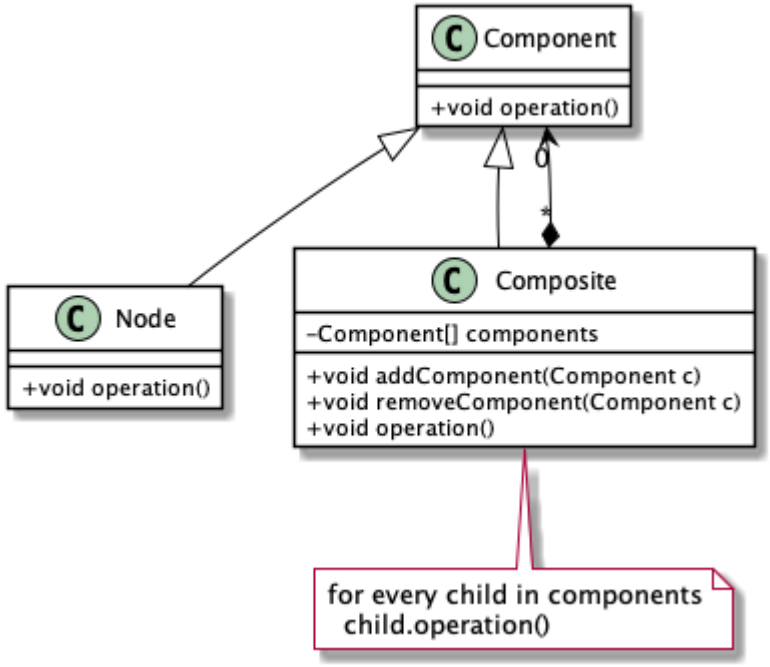
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Composite Purpose

- To develop a flexible way to create hierarchical tree structures of arbitrary complexity
- While enabling every element in the structure to operate with a uniform interface

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Composite Pattern Canonical Diagram



Composite Ingredients

Component – The `Component` interface defines methods available for all parts of the tree structure. `Component` may be implemented as abstract class when you need to provide standard behavior to all of the sub-types. Normally, the component is not instantiable; its subclasses or implementing classes, also called nodes, are instantiable and are used to create a collection or tree structure.

Composite – This class is defined by the components it contains; it is composed by its components. The `Composite` supports a dynamic group of `Components` so it has methods to add and remove `Component` instances from its collection. The methods defined in the `Component` are implemented to execute the behavior specific for this type of `Composite` and to call the same method on each of its nodes. These `Composite` classes are also called branch or container classes.

Leaf – The class that implements the `Component` interface and that provides an implementation for

each of the `Component` 's methods. The distinction between a Leaf class and a `Composite` class is that the Leaf contains no references to other `Components`. The Leaf classes represent the lowest levels of the containment structure.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

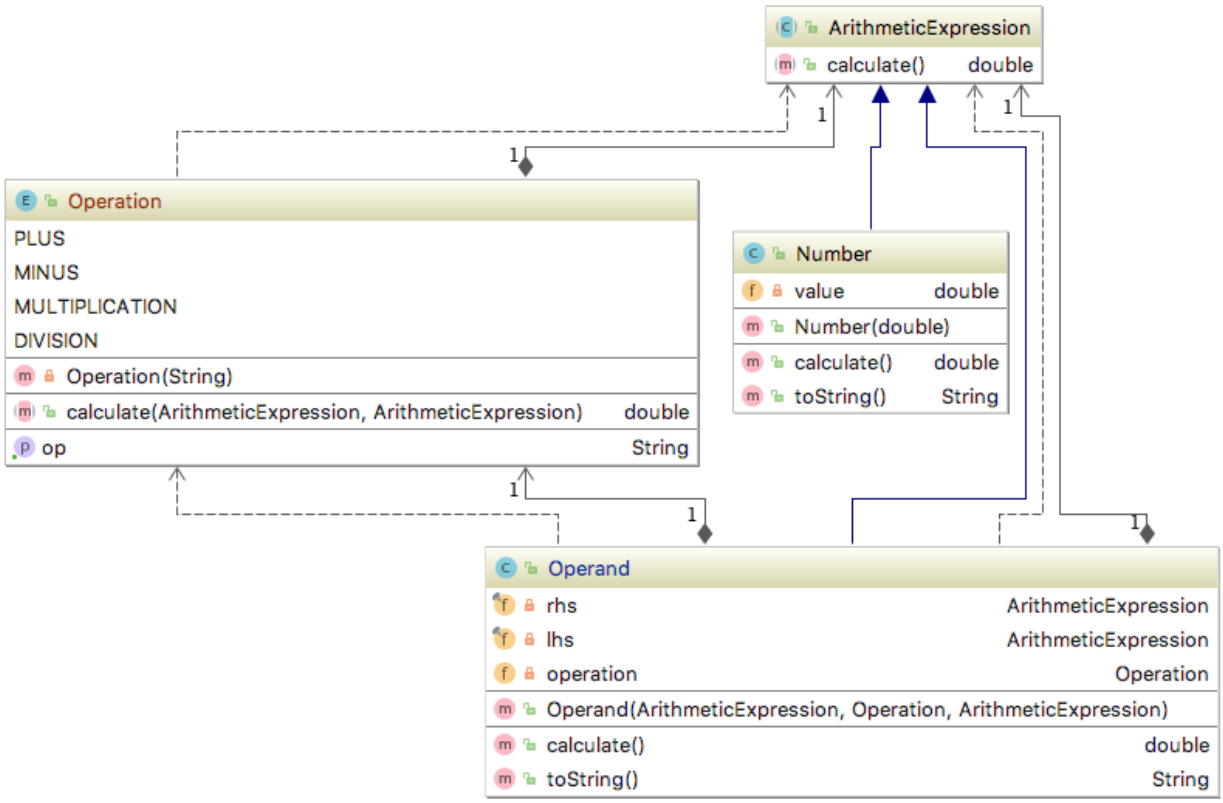
Composite Advantages

- Users perceive a unified structure
- Users can also add or remove components
- Great for
  - UI Development
  - Organizational Charts
  - Schedules
  - Outlines

Composite Disadvantages

- Because it is so dynamic, the Composite pattern is often difficult to test and debug
- It normally requires a more sophisticated test/validation strategy that is designed around the concept of the whole-part object hierarchy
- Requires full advance knowledge of the structure being modeled

Composite Pattern Demo Diagram



Composite: Component

```
public abstract class ArithmeticExpression {
    public abstract double calculate();
}
```

Composite: Leaf

```
public class Number extends ArithmeticExpression {

    private double value;

    public Number(double value) {
        this.value = value;
    }

    @Override
    public double calculate() {
        return value;
    }

    @Override
    public String toString() {
        return Double.toString(value);
    }
}
```

Composite: Composite

```
public class Operand extends ArithmeticExpression {
    private final ArithmeticExpression rhs;
    private final ArithmeticExpression lhs;
    private Operation operation;

    public Operand(ArithmeticExpression rhs,
                   Operation operation,
                   ArithmeticExpression lhs) {
        this.rhs = rhs;
        this.lhs = lhs;
        this.operation = operation;
    }

    @Override
    public double calculate() {
        return operation.calculate(rhs, lhs);
    }

    @Override
    public String toString() {
        return "( " + rhs.toString() + " " +
                      operation.getOp() + " " +
                      lhs.toString() + " )";
    }
}
```

Composite: Utility Class

```
public enum Operation {
    PLUS("+") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() + lhs.calculate();
        }
    },
    MINUS("-") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() - lhs.calculate();
        }
    },
    MULTIPLICATION("*") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() * lhs.calculate();
        }
    },
    DIVISION("/") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() / lhs.calculate();
        }
    };
}
```

Composite: Utility Class Continued

```
public enum Operation {

    //...

    private String op;

    private Operation(String op) {
        this.op = op;
    }

    public abstract double calculate(ArithmeticExpression rhs,
ArithmeticExpression lhs);

    public String getOp() {
        return op;
    }
}
```

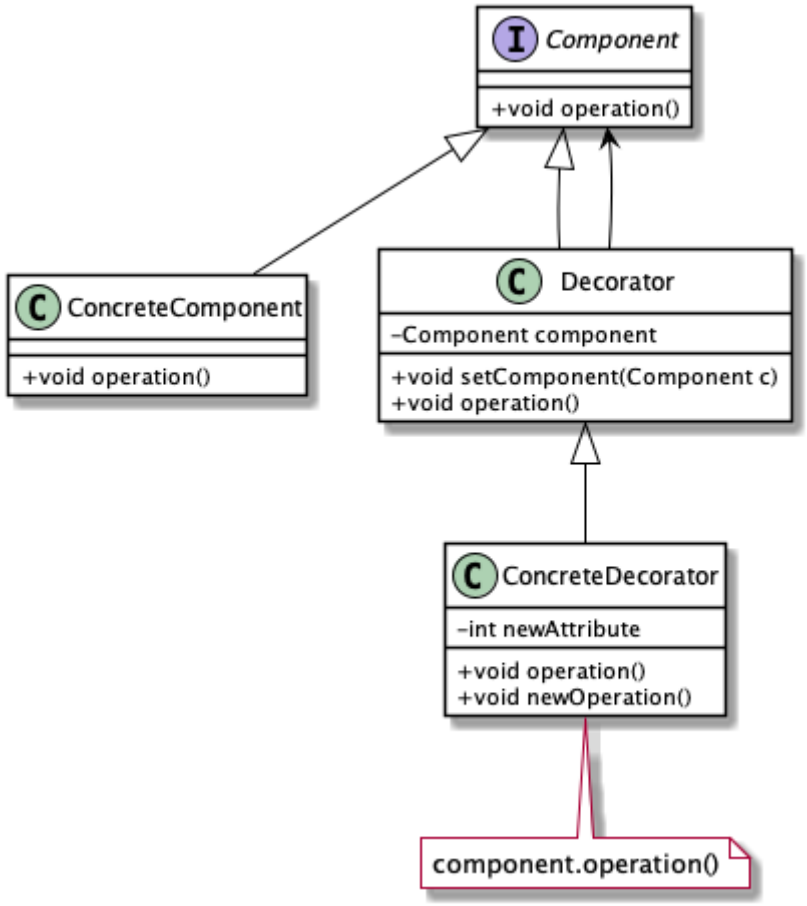
Decorator Pattern

Decorator Pattern Purpose

- Also known as a Wrapper
- To provide a way to flexibly add or remove component functionality without changing its external appearance or function

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Canonical Diagram



Decorator Pattern Advantages

- "Delegation over Inheritance" - Effective Java Josh Bloch
- Produce classes with plugin capabilities
- You want to make dynamic changes that are transparent to users, without the restrictions of subclassing
- Offers the opportunity to easily adjust and augment the behavior of an object during runtime
- Can reduce memory by reusing layers

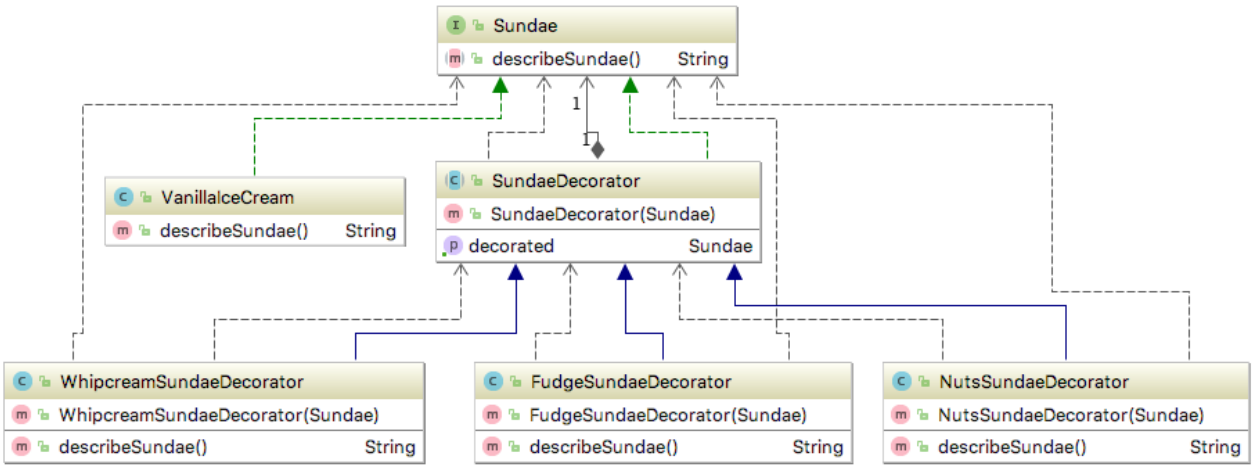
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Pattern Disadvantages

- Can produce large number of layers
- Debugging and testing can be difficult
- Can be slow if done incorrectly

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Demo Diagram



Base Interface

```
public interface Sundae {
    String describeSundae();
}
```

The Decorator

```
public abstract class SundaeDecorator implements Sundae {

    private Sundae decorated;

    public SundaeDecorator(Sundae decorated) {
        this.decorated = decorated;
    }

    public Sundae getDecorated() {
        return decorated;
    }
}
```

The Base Class

```
public class VanillaIceCream implements Sundae {
    public String describeSundae() {
        return "Vanilla Ice Cream";
    }
}
```

Sample Decorator Layer

```
public class WhipcreamSundaeDecorator extends SundaeDecorator {
    public WhipcreamSundaeDecorator(Sundae sundae) {
        super(sundae);
    }

    public String describeSundae() {
        return "Whipcream " + getDecorated().describeSundae();
    }
}
```

Another Sample Decorator Layer

```
public class NutsSundaeDecorator extends SundaeDecorator {
    public NutsSundaeDecorator(Sundae sundae) {
        super(sundae);
    }

    public String describeSundae() {
        return "Nuts " + getDecorated().describeSundae();
    }
}
```

And Yet, Another

Notice this one, this one changes the end result

```
public class FudgeFilterDecorator extends SundaeDecorator {
    public FudgeFilterDecorator(Sundae cherryOnTopDecorator) {
        super(cherryOnTopDecorator);
    }

    @Override
    public String describeSundae() {
        return getDecorated().describeSundae().replaceAll("Fudge",
"xxxxx");
    }
}
```

Running a Decorator

```
Sundae sundae = new NutsSundaeDecorator(
        new FudgeSundaeDecorator(
                new WhipcreamSundaeDecorator(
                        new VanillaIceCream()
                )
        )
);

System.out.println(sundae.describeSundae());

//add a cherry

Sundae cherryOnTopDecorator = new CherryOnTopDecorator(sundae);

System.out.println(cherryOnTopDecorator.describeSundae());

Sundae filteredFudge = new FudgeFilterDecorator(cherryOnTopDecorator);

System.out.println(filteredFudge.describeSundae());
```

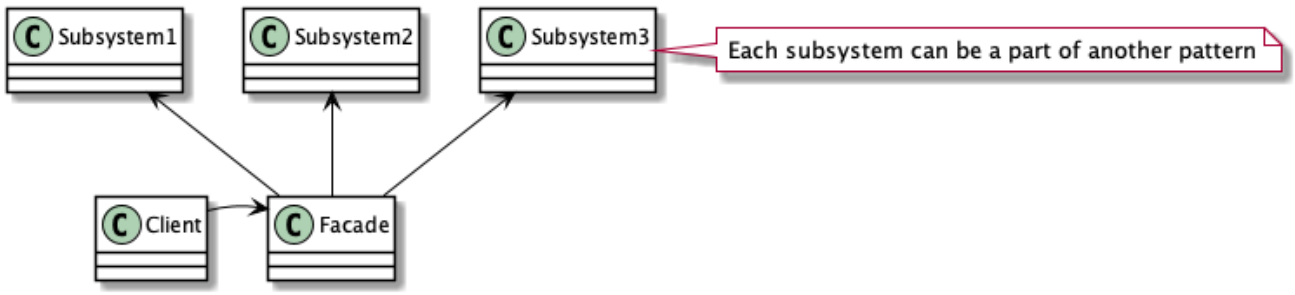
Facade Pattern

Facade Pattern Properties

Type: Structural
Level: Component

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Pattern Canonical Diagram



Facade Ingredients

Facade – The class for clients to use. It knows about the subsystems it uses and their respective responsibilities. Normally all client requests will be delegated to the appropriate subsystems.

Subsystem – This is a set of classes. They can be used by clients directly or will do work assigned to them by the `Facade`. It does not have knowledge of the `Facade`; for the subsystem the `Facade` will be just another client.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Purpose

- To provide a simplified interface to a group of subsystems or a complex subsystem
- Reduce coupling between clients and subsystems.
- Layer subsystems by providing Facades for sets of subsystems.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Advantages

- Protects the client with an overabundance of options, parameters, and setup methods
- One request can be translated to multiple subsystems
- Promotes low coupling between subsystems

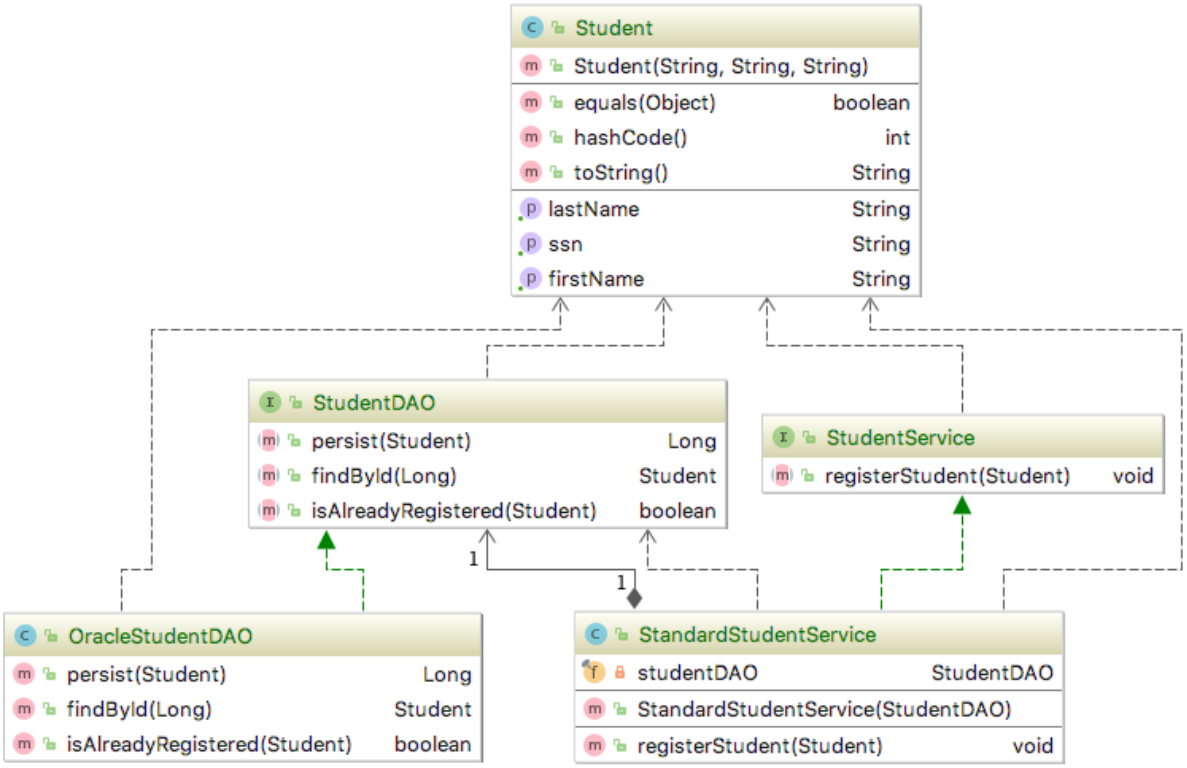
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Disadvantages

- Can be difficult to debug if subsystems gets too high

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Demo Diagram



Facade: The Facade's Interface

```
public interface StudentService {
    public void registerStudent(Student student);
}
```

Facade: The Facade's Implementation

Here, we create a "complication", albeit a small one.

```
public class StandardStudentService implements StudentService {

    private final StudentDAO studentDAO;

    public StandardStudentService(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    @Override
    public void registerStudent(Student student) {
        if (!studentDAO.isAlreadyRegistered(student)) {
            studentDAO.persist(student);
        }
    }
}
```

Facade: The Facade's Dependency Interface

```
public interface StudentDAO {
    public Long persist(Student student);
    public Student findById(Long id);
    public boolean isAlreadyRegistered(Student student);
}
```

Facade: The Facade's Dependency Implementation

```
public class OracleStudentDAO implements StudentDAO {
    @Override
    public Long persist(Student student) {
      //insert lots of database code
    }

    @Override
    public Student findById(Long id) {
      //insert lots of database code
    }

    @Override
    public boolean isAlreadyRegistered(Student student) {
      //insert lots of database code
    }
}
```

Proxy Pattern

Proxy Pattern Properties

Type: Structural
Level: Component

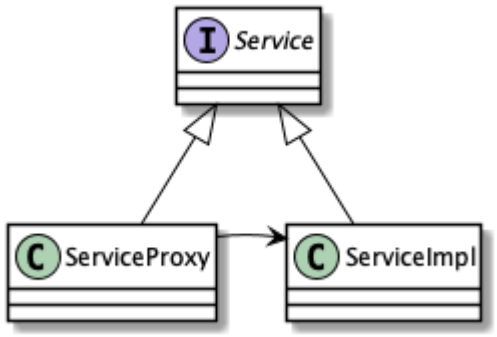
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Purpose

To provide a representative of another object, for reasons such as access, speed, or security.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Canonical Diagram



Proxy Ingredients

Service – The interface that both the proxy and the real object will implement.

ServiceProxy – ServiceProxy implements `Service` and forwards method calls to the real object (`ServiceImpl`) when appropriate.

ServiceImpl – The real, full implementation of the interface. This object will be represented by the Proxy object.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Advantages

- Kind of an adapter pattern, but for complex, remote, or both objects
- Delays creation of those expensive objects
- Can be used to constrain access based on access control

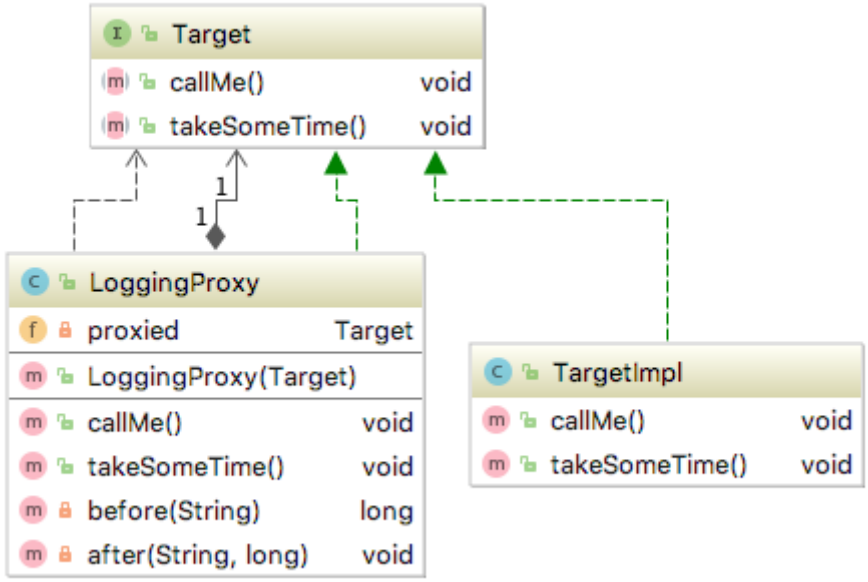
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Disadvantages

- Complicated setup
- Unnecessary for simple local reference

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Demo Diagram



Proxy: Target Interface

```
public interface Target {
    void callMe();
    void takeSomeTime();
}
```

Proxy: Target Implementation

```
public class TargetImpl implements Target {

    public void callMe() {
        System.out.println("Called");
    }

    public void takeSomeTime() {
        try {
            Thread.sleep(1000);
            System.out.println("Took some time");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Proxy: The Proxy

The difference between a Proxy and an Adapter is that the Adapter can have a different interface, a Proxy must have the same interface and perhaps some supporting methods

```
public class LoggingProxy implements Target {

    private Target proxied;

    public LoggingProxy(Target proxied) {
        this.proxied = proxied;
    }

    public void callMe() {
        long start = before("callMe()");
        proxied.callMe();
        after("callMe()", start);
    }

    public void takeSomeTime() {
        long start = before("takeSomeTime()");
        proxied.takeSomeTime();
        after("takeSomeTime()", start);
    }

    private long before(String name) {
        System.out.println("Before " + name);
        return System.currentTimeMillis();
    }

    private void after(String name, long start) {
        System.out.println("After: " + name + " took: " + (System
.currentTimeMillis() - start));
    }
}
```

Proxy: Use of a Proxy

```
Target target = new TargetImpl();

target.callMe();
target.takeSomeTime();

System.out.println("Added Proxy");
Target targetProxy = new LoggingProxy(target);

targetProxy.callMe();
targetProxy.takeSomeTime();
```

Flyweight Pattern

Flyweight Pattern Properties

Type: Structural
Level: Component

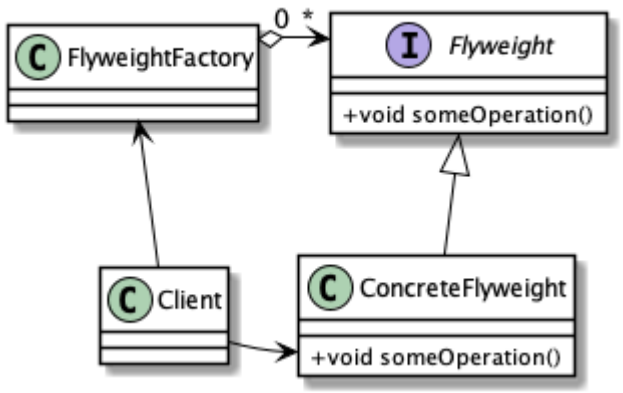
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Purpose

- Provides for sharing an object between clients
- Creating a responsibility for the shared object that normal objects do not need to consider
- An ordinary object doesn't have to worry much about shared responsibility
- Most often, only one client will hold a reference to an object at any one time
- When the object's state changes, it's because the client changed it, and the object does not have any responsibility to inform any other clients
- Sometimes, though, you will want to arrange for multiple clients to share access to an object

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Canonical Diagram



Flyweight Ingredients

Flyweight – The interface defines the methods clients can use to pass external state into the flyweight objects.

ConcreteFlyweight – This implements the `Flyweight` interface, and implements the ability to store internal data. The internal data has to be representative for all the instances where you need the Flyweight.

FlyweightFactory – This factory is responsible for creating and managing the Flyweights. Providing access to Flyweight creation through the factory ensures proper sharing. The factory can create all the flyweights at the start of the application, or wait until they are needed.

Client – The client is responsible for creating and providing the context for the flyweights. The only way to get a reference to a flyweight is through `FlyweightFactory`.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Advantages

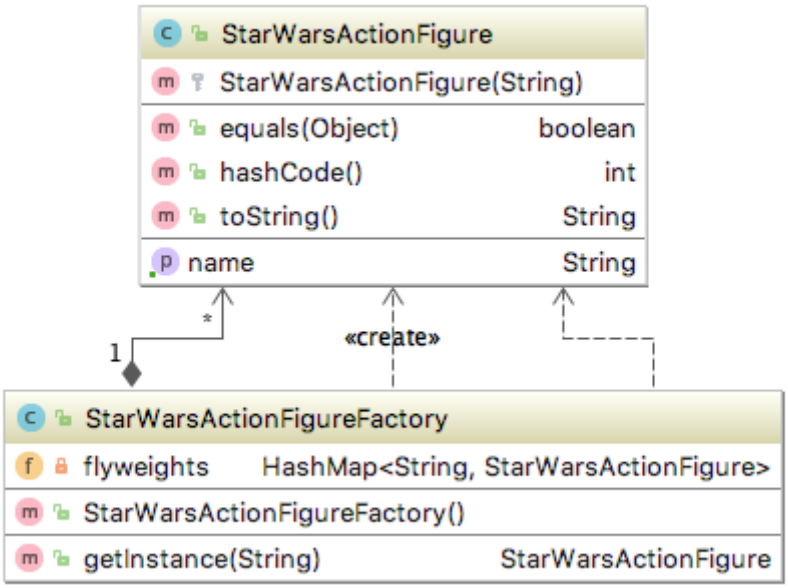
- Sharing an object among multiple clients occurs when you must manage thousands
- Provides for sharing an object between clients
- Save in memory
- Runtime can be efficient

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Disadvantages

- None

Flyweight Demo Diagram



Flyweight Factory

```
import java.util.HashMap;

public class StarWarsActionFigureFactory {
    private HashMap<String, StarWarsActionFigure> flyweights;

    public StarWarsActionFigureFactory() {
        this.flyweights = new HashMap<>();
    }

    public StarWarsActionFigure getInstance(String name) {
         if (flyweights.containsKey(name)) return flyweights.get(name);
         else {
             StarWarsActionFigure starWarsActionFigure =
                new StarWarsActionFigure(name);
             flyweights.put(name, starWarsActionFigure);
             return starWarsActionFigure;
         }
    }
}
```

Flyweight Objects

```
public class StarWarsActionFigure {
    private String name;

    protected StarWarsActionFigure(String name) {
        this.name = name;
    }

    public String getName() {return name;}

    //toString, equals, hashCode
}
```

What is new in Java

JDK 9

- JShell
- Immutable Collection Creation:
- Module System
- Process Handles
- Flow API
- Functional and Stream Additions

- HTTP2 Client

Immutable Collection API

```
List.of(3,1,2,4);
```

Module Systems

- Java Platform Module System (JPMS), or “Modules” for short
- Module - Group of closely related packages and resources along with a new module descriptor file
- Java Required Modules for "componentizing" Java and having a small JDK footprint

Process Handles

Ability to query system information like PID

```
System.out.printf("Process Starting on %s%n", ProcessHandle.current
().pid());
```

Flow API

- Reactive Extensions based on Interfaces
- Alleviates Backpressure, Request Based

Functional Additions

- `Optional:stream`
- `takeWhile` and `dropWhile` for `Streams`

```
Stream.of(1,2,3,4,5,6,7,8,9,10).takeWhile(i -> i < 5 ).forEach(System
.out::println);
```

HTTP2 Client

- Client supporting HTTP2 and some of its new features
  - HTTP/2 Multiplexing
  - HTTP/2 Header Compression
  - HTTP/2 Push

```
HttpClient client = HttpClient
    .newBuilder()
    .version(Version.HTTP_2)
    .build();
```

```
HttpResponse<String> response = client.send(
    HttpRequest
        .newBuilder(TEST_URI)
        .POST(BodyProcessor.fromString("Hello world"))
        .build(),
    BodyHandler.asString()
);
```

JDK 10

- Type inference
- `Collectors` changes
- `copyOf`

Type Inference

```
var list = List.of(12, 4, 50, 103)
```

`Collectors` Changes

Additional methods to collect a Stream into `unmodifiable` `List`, `Map` or `Set`:

```
List<Integer> evenList = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .collect(Collectors.toUnmodifiableList());
    evenList.add(4);
```

`copyOf`

```
List<Integer> copyList = List.copyOf(someIntList);
copyList.add(4);
```

`Optional.orElseThrow`

`orElseThrow` implementation for all `Optional` variants * `java.util.Optional` * `java.util.OptionalDouble` * `java.util.OptionalInt` * `java.util.OptionalLong`

`orElseThrow()` * doesn't take any argument * `throws` `NoSuchElementException` if no value is

present:

```
Integer firstEven = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .findFirst()
      .orElseThrow();
```

JDK 11

- `String` class additions
- Type inference lambda variables

`String` class additions

`repeat`

```
String output = "La ".repeat(2) + "Land";
```

`strip` - strip leading and trailing spaces * `stripLeading` - remove all leading spaces * `stripTrailing` - remove all trailing spaces

```
"\n\t  hello   \u2005".strip() //Hello
```

Takes into account Unicode blank values

`isBlank` - Determines if a `String` is indeed blank

```
"\n\t\u2005  ".isBlank()
```

Takes into account Unicode blank values

`lines` - Break a String by line terminators into a Stream of `String`

```
String multilineStr = "This is\n \n a multiline\n string.";

long lineCount = multilineStr.lines()
  .filter(String::isBlank)
  .count();
```

Type Inference Lambda Expressions

```
Function<String, String> append = (var string) -> string + " World";
String appendedString = append.apply("Hello");
```

Null Output Stream

- Similar to `/dev/null`
- Useful for a connection that doesn't go anywhere

```
InputStream inputStream = InputStream.nullInputStream();
OutputStream outputStream = OutputStream.nullOutputStream();
Reader reader = Reader.nullReader();
Writer writer = Writer.nullWriter();
```

JDK 12

- Switch Statement Preview
- `teeing Collector`

`switch` Expressions

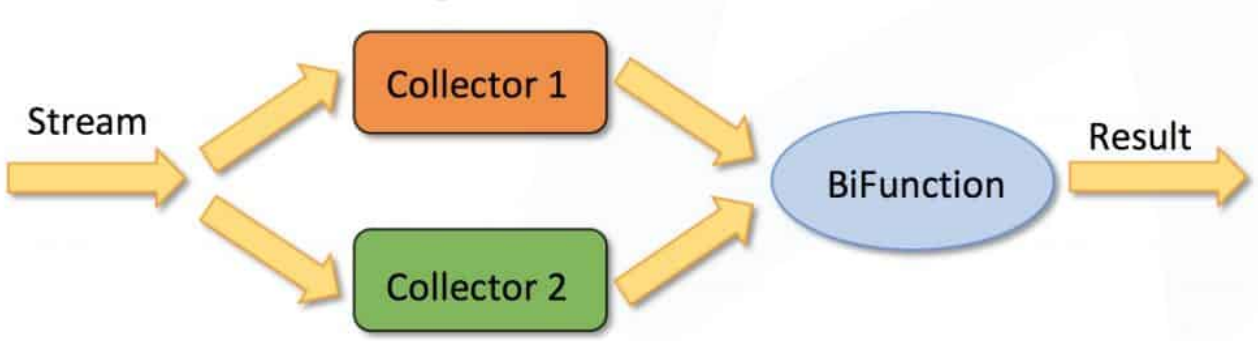
- Switch expressions (JEP 325)
- A preview language feature
  - The idea of preview language features was introduced at the start of 2018, under JEP 12.
  - Essentially, this is a way of including beta versions of new features.
  - By doing this, it is still possible to make changes based on user feedback and, in an extreme case, remove the feature altogether if it is not well received.

```
int numLetters;
switch (day) {
   case MONDAY:
   case FRIDAY:
   case SUNDAY:
       numLetters = 6;
       break;
   case TUESDAY:
       numLetters = 7;
       break;
   case THURSDAY:
   case SATURDAY:
       numLetters = 8;
       break;
   case WEDNESDAY:
       numLetters = 9;
       break;
   default:
       throw new IllegalStateException("Huh? " + day);
}
```

```
int numLetters = switch (day) {
   case MONDAY, FRIDAY, SUNDAY -> 6;
   case TUESDAY -> 7;
   case THURSDAY, SATURDAY -> 8;
   case WEDNESDAY -> 9;
   default -> throw new IllegalStateException("Huh? " + day);
};
```

`teeing` Collector

Take three arguments: * Three Collectors * BiFunction



```
double average = Stream.of(1, 4, 2, 7, 4, 6, 5)
    .collect(teeing(
        summingDouble(i -> i),
        counting(),
        (sum, n) -> sum / n));
```

JDK 13

Text Blocks Preview

Sample HTML in a Text Block

```
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

Sample SQL

```
String query = """
               SELECT <code>EMP_ID</code>, <code>LAST_NAME</code> FROM
<code>EMPLOYEE_TB</code>
               WHERE <code>CITY</code> = 'INDIANAPOLIS'
               ORDER BY <code>EMP_ID</code>, <code>LAST_NAME</code>;
               """;
```

Project Loom

- High-throughput and lightweight concurrency in Java to help simplify writing scalable software
- In Java, each thread is mapped to an operating system thread by the JVM
- Currently Java uses Context Switching
  - A bunch of CPU time is allocated to schedule the threads on the core.
  - If a thread goes to wait state (e.g., waiting for a database call to respond), the thread will be marked as *paused* and a separate thread is allocated to the CPU resource.
- Instead of allocating one OS thread per Java thread (current JVM model)
- Project Loom provides additional schedulers that schedule the multiple lightweight threads on the same OS thread.
- This approach provides better usage (OS threads are always working and not waiting) and much less context switching.

Fibers

- Lightweight, user-mode threads, scheduled by the Java virtual machine, not the operating system.
- Fibers are low footprint and have negligible task-switching overhead. You can have millions of them!

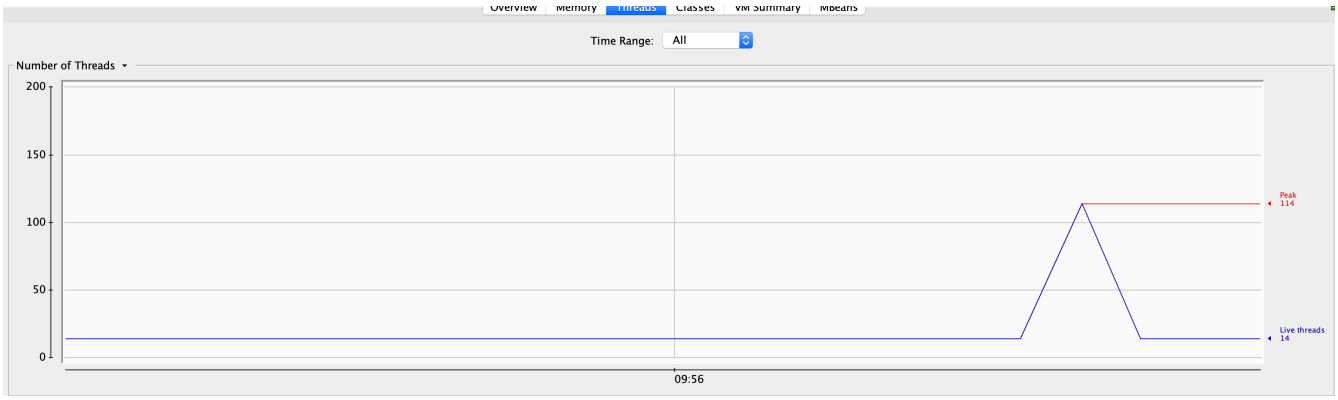
Running a `Fiber`

Running a `Fiber` would just be sending a `Runnable` to a `Fiber`

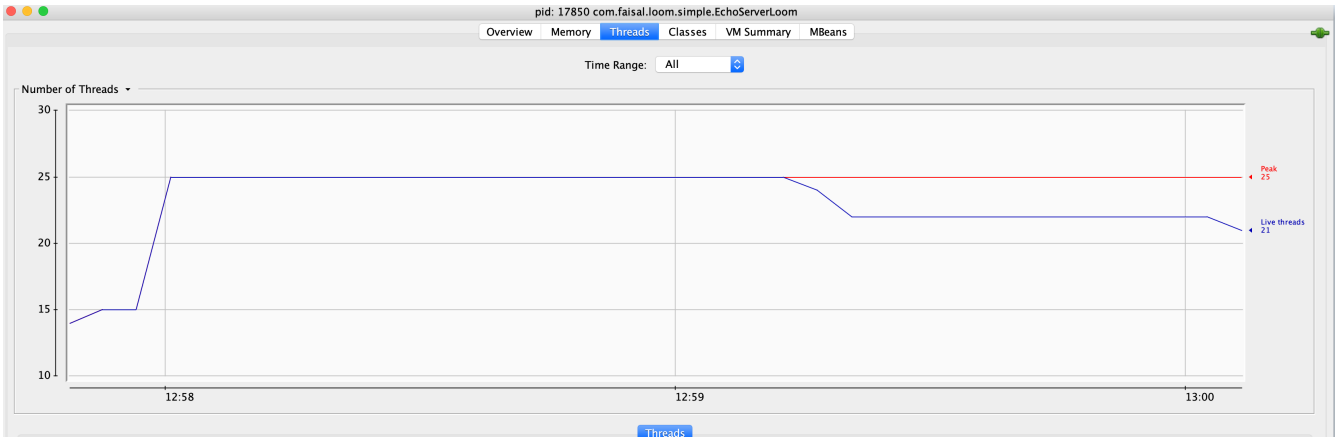
```
Fiber.schedule(runnable);
```

Showing the difference

Before Using Loom:



After Using Loom:



Source: RedHat Project Loom Blog

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