myCareer TEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Introduction

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Outline

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Objectives

- During this course, you will learn:
  - Java 8 Languages and Features
  - Java Design Patterns
  - Building and Deploying Applications and Web Services with Spring Boot
  - Test-Driven Development
  - Web Security Foundations
  - Working with Git
  - Maven Essentials
- How to build systems using these technologies secured by tests
  - This will be via a case study

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Approach

- This course is workshop-based
  - You will work in teams to develop a algorithmic trading engine (case study project)
  - There are hands-on exercises for which you will work on individually
  - Today and tomorrow you will receive instructor-led training to cover core technologies for which your case study project is dependent, in addition to interactive class exercises to facilitate the development of a case study project
- While you will be presented with fundamentals of some subjects
  - Be expected to research and go beyond what is provided in the materials when working on the case study
- Your instructor will help when needed—please ask for help/advice whenever required!
  - Your instructor will also act as product owner for the case study

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Case Study: Expectations

- You will use test-driven development to build the core business logic for an algorithmic trading platform—this functionality will be made available via a REST-based web service
- All of the coding will be done by means of test-driven development
- Java 8 language features, in particular stream processing, will be used
- Best programming features and design patterns will be used to create a solid foundation for the engine and the web service
- Web security will be emphasized from design through implementation
- Maven will be used as the build tool and Git as the repository for the code base

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Case Study: Deliverable

- On the final day of this program, each group will be given the opportunity to demonstrate solutions in the form of a case study presentation
- While it is a group presentation, you will be assessed individually based on the role you played in developing the solution(s)
- Your instructor is available as a coach and product owner for the end deliverable; take advantage of your resources!

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features:
Unit Testing, Design Patterns, and RESTful Services

Chapter 1: Lambda Expressions and Functional Interfaces

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will introduce:

- Default methods in interfaces
- Lambda expressions
- Method references
- Functional interfaces

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

➤ **Default Methods**

Lambda Expressions

Method References

Functional Interfaces

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

New Features in Java 8

- Major new features added to Java include:
  - Lambda expressions and functional interfaces
  - Stream API for bulk data operations
  - Time API
  - Default and static methods in interfaces
  - Improvements to collection API
  - Improvements to concurrency API
  - More…
- This course primarily focuses on lambda expressions and streams
  - Other features are also covered, such as the Date Time library

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Java 8 Changes

- As well as major changes to the language, Java 8 has introduced many smaller changes
- In particular, to the standard APIs
  - For example, to the collection classes
    - To make them easier to use
    - To improve performance
- Many of the interfaces have been changed with extra methods added
  - This could have broken existing code bases that use these interfaces
    - Would have made the adoption of Java 8 challenging
- Java solved this problem by providing default methods to interfaces
  - A new feature in Java 8

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Default Methods

- It is possible in Java 8 to add concrete methods to a Java interface without breaking existing implementation using '`default`' methods

v1 of API

```
public interface ShapeStatistics {
  double calcPerimeter();

}

public class Square implements ShapeStatistics {
  public double calcPerimeter() {
    …
  }
}
```

v2 of API

```
public interface ShapeStatistics {
          double calcPerimeter();
  default double calcArea() { … }
}

public class Square implements ShapeStatistics {
  public double calcPerimeter() {
    …
  }

  public double calcArea() {
    …
  }
}
```

Newly introduced method is declared as '`default`' to ensure 'Square' class is not getting broken as a result

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Comparator` Interface in Java 8

- `Comparator` is an example of an interface that has been extended in Java 8
  - Many static methods added
  - Many default methods added
- Provide solutions for simple tasks such as:
  - Sorting in reverse order
    - No need to write separate comparators now!
- We will see these methods in use in the next few sections

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Default Methods

➤ **Lambda Expressions**

Method References

Functional Interfaces

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Lambda Expressions

- Anonymous classes are often used in Java
  - More compact syntax than defining an explicit class used just once
- Consider the following simple example:

```
class SimpleRunnable implements Runnable{
  @Override
  public void run() {
    System.out.println("Running Thread");
  }
}
```

```
public class ThreadDemo {
  public static void main(String[] args) {
    Thread thread = new Thread(new SimpleRunnable());
    thread.start();
  }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Lambda Expressions (continued)

- The example can be reworked using an anonymous class

```
public class ThreadDemo {
  public static void main(String[] args) {
    new Thread(new Runnable(){
      @Override
      public void run() {
        System.out.println("Running Anonymous Thread");
      }
    }).start();
  }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lambda Expression Solution

- Lambdas can be used to provide a neater solution
- For example, reworking thread example on the previous slide results in the following code:

```
public class ThreadDemo {
  public static void main(String[] args) {
    new Thread(()->System.out.println(”Lambda Thread")).start();
  }
}
```

Lambda expression

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Lambda Expressions

- Lambda expression is an **implementation** of an interface with a **single** abstract method
  - Essentially, a simplification of anonymous classes
- Can be considered as an anonymous method
  - More compact syntax than traditional Java method
    - No name, modifiers, or return type
      - In some cases, no parameter type(s)
- Use case
  - Where a method will only be used once
  - Can be passed as parameters to other methods
  - Callbacks
- General syntax is:
  - `(Parameter List) -> Body of method`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lambda Expression Syntax

- A lambda expression consists of:
  1. A parameter list
  2. Followed by an arrow `->`
  3. Followed by a function body

- A zero argument lambda expression example:
  - `() -> System.out.println("Lambda")`

- A one argument lambda expression example:

  -
    ```
    (String currency) ->
    System.out.printf("Currency is %s%d", currency)
    ```

- A lambda expression with more than one argument:

  -
    ```
    (String fromCurrency, String toCurrency) ->
      System.out.printf("Converting from %s to %s %n",
                 fromCurrency,toCurrency)
    ```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lambda Expression Syntax (continued)

- The type of parameters can be omitted if the type can be inferred by the compiler

No type specified

```
(currency) ->
      System.out.printf("Currency is %s%d", currency)
```

- The parentheses on a parameter list are optional if there is only one parameter

No parentheses

```
currency -> System.out.printf("Currency is %s%d", currency)
```

- If a lambda expression has a return value, no type needs to be specified

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Multiple Line Lambda Expressions

- A lambda expression may have more than one statement
  - Statements must be enclosed in a block { }

```
public class ThreadDemo {
  public static void main(String[] args) {
    new Thread(()-> {
      System.out.println("Lambda Thread Line One")
      System.out.println("Lambda Thread Line Two")
    }).start();
  }
}
```

Multiple line lambda expression requires { }

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Lambda Expression Example

- Consider sorting the following list of strings:

```
List<String> currencies =
    Arrays.asList("USD", "JPY", "EUR", "HKD", "INR", "AUD");
```

- To sort the list using `Collections.sort(List l, Comparator c)`

```
interface Comparator {
   int compare(T o1, T o2);
}
```

- Using a lambda expression we can write:

```
Collections.sort(currencies,
     (String a, String b) -> { return a.compareTo(b);});
```

Lambda expression

- This can be further simplified to:

```
Collections.sort(currencies, (a,b)-> a.compareTo(b));
```

Lambda expression

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Internal Iteration

- Consider printing the elements of the following collection to the console

```
List<String> currencies =
    Arrays.asList("USD", "JPY", "EUR", "HKD", "INR", "AUD");
```

- We have to do two things:

1. Write a loop to iterate over the collection
2. Print each element to the console

```
for(String currency: currencies){
  System.out.println(currency);
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Internal Iteration (continued)

- Every time we want to iterate over a collection, we have to write the loop
  - Lots of repetitive code
  - Usually only the work we want to do on the elements that changes
- Java 8 has modified the Java 8 `Iterable` interface
  - Added new method `forEach`
    - Allows collections to provide iteration *internally*
      - User just supplies work to be done on each element
- Loop on previous slide can be rewritten as:

```
currencies.forEach(c-> System.out.println(c));
```

- This feature is known as *internal iteration*

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 1.1: Working with Lambda Expressions

HANDS-ON EXERCISE

- Please turn to the Exercise Manual and complete Exercise 1.1: Working with Lambda Expressions

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Default Methods

Lambda Expressions

➤ **Method References**

Functional Interfaces

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Domain Classes

- To further explain Java 8 features, we will use the following classes:

```java
public abstract class Order {
    private Currency currency;
    private double amount;
    private Side side;

    public Order(Currency currency, double amount, Side side) {
        this.currency = currency; this.amount = amount; this.side = side;
    }
    public abstract boolean match(Order order);

    public Currency getCurrency() {
        return currency;
    }
    public double getAmount() {
        return amount;
    }
    public Side getSide() {
        return side;
    }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Domain Classes (continued)

```
public class MarketOrder extends Order {
  @Override
  public boolean match(Order order) {
   …
  }
  public MarketOrder(Currency currency, double amount, Side side) {
    super(currency, amount, side);
  }
}
```

```
public class LimitOrder extends Order {
  private double limit;

  public LimitOrder(Currency currency, double amount, Side side, double limit) {
    super(currency, amount, side);
    this.limit = limit;
  }
  @Override
  public boolean match(Order order) {
    …
  }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Method References

- Lambda expressions are an implementation of the single abstract method in a functional interface
- Often the expression simply calls a concrete method in an existing class
- Consider the example below
  - The lambda expression just calls the `println` method

```
Currencies.forEach(c-> System.out.println(c));
```

- Using a *method reference*, the code can be further simplified
  - `println` will be passed the current value in the collection on each iteration

```
currencies.forEach(System.out::println);
```

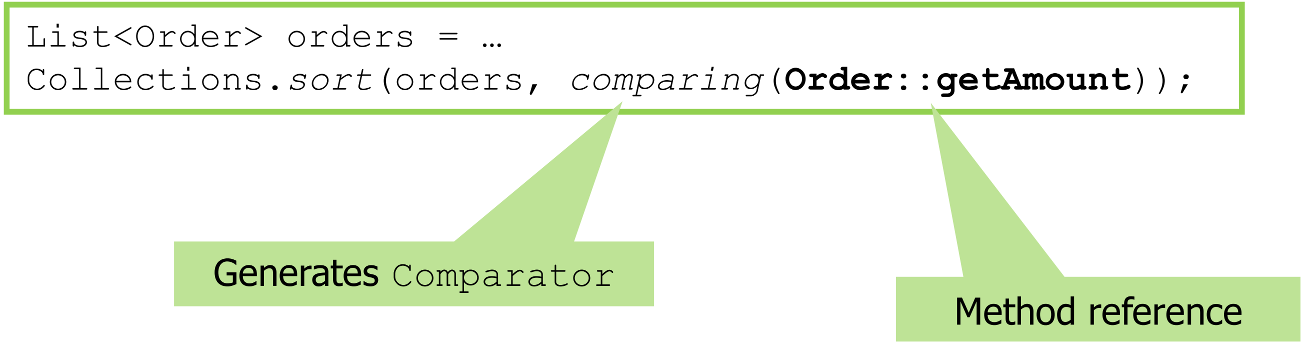
Method reference

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Method References

- Method references allow reuse of existing method definitions
  - They can be passed just like lambda expressions

- Consider sorting a collection of Orders by price
  - Using the `Comparator`s `comparing()` method, we can write:
    - `Comparing()` will generate a `Comparator` using the value returned by method whose reference is supplied



```
List<Order> orders = …
Collections.sort(orders, comparing(Order::getAmount));
```

Generates `Comparator`

Method reference

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Types of Method References

- There are four types of method references
  - Static
  - Bound instance
  - Unbound instance
  - Constructor
- Static references are created using `ClassName::staticMethodName`
- Bound instance references are created using `objectReference::methodName`
- Unbound instance references are created using `ClassName::methodName`
- Constructor references are created using `ClassName::new`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Default Methods

Lambda Expressions

Method References

➤ **Functional Interfaces**

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Functional Interfaces

- Many interfaces in Java have just one abstract method
  - Known as functional interfaces
  - For example: `Runnable`, `Comparator`
- Results in a class being written to contain the method
  - Can use a lambda expression instead
- A lambda expression can be supplied wherever an implementation of a functional interface is required
  - The lambda expression will be matched to the abstract method

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Defining Functional Interfaces

- It is possible to explicitly define a functional interface
  - Compiler will check that interface meets functional interface requirements
- Use `@FunctionalInterface` to explicitly define functional interface
  - Annotation is optional

Compiler enforces functional interface requirements

```
@FunctionalInterface
public interface Transferable{
   void transfer(Broker targetExchange);
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Built-in Functional Interfaces

- Java 8 has many built-in functional interfaces
  - In the package `java.util.function`
- Often used with enhancements to collection classes
  - Make filtering and processing of data simpler
- We will examine some of these functional interfaces now, including:
  - `Predicate`
  - `Consumer`
  - `Function`
- *Note:*
  - Everything in Java is a class and Lambdas have to be ‘wrapped’ within Functional Interfaces
    - Therefore, there are so many different Functional Interfaces – they differ only by the method signatures

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Predicates

- Consider the following collection of Orders
  – The collection contains a mixture of BUY and SELL side orders

```
List<Order> orders = …
```

- Lets assume we want to print out all the BUY side orders

```
for(Order order : orders){
  if(order.getSide() == BUY){
    System.out.println(order);
  }
}
```

- Now consider printing all the SELL side orders

```
for(Order order : orders){
  if(order.getSide() == SELL){
    System.out.println(order);
  }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Introducing Predicates (continued)

- There is a lot of duplication in the previous slide
  - The `Predicate` interface can help us reduce this
- `Predicate` interface defines one method
  - Evaluates argument and returns ‘true’ or ‘false’
  - Usually implemented using a Lambda expression
  - Common use case: filtering elements in the collection

```
public interface Predicate<T>{
   boolean test(T t);
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using `Predicate`

- To prevent duplication of previous code, can use lambdas
  - Write a method that receives a predicate
  - Method will print any item in the list that matches the predicate

```
static void evaluate(List<Order> orders,
                     Predicate<Order> predicate) {
    for(Order order: orders){
        if (predicate.test(order)) {
            System.out.println(order);
        }
    }
}
```

Apply predicate

Supply predicates

```
evaluate(orders, o -> o.getSide() == BUY);
evaluate(orders, o -> o.getSide() ==SELL);
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

The `Consumer` Functional Interface

- This interface is used when an operation is to be performed on a single input argument

```
public interface Consumer<T>{
   void accept(T t);
}
```

- Enables general methods to be written that apply work to collections
  - Such as:
    - Persisting items in collection
    - Printing items

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Consumer` Example

- The following code will call `accept` on any supplied `Consumer`

```
static void evaluate(List<Order> orders,
                           Consumer<Order> consumer){
    for(Order order : orders) {
        consumer.accept(order);
    }
}
```

Apply consumers

- Here, we just supply two different consumers

```
evaluate(orders, o -> System.out.println(o.getAmount()));
evaluate(orders, o -> System.out.println(o.getCurrency()));
```

Supply consumers

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Further Simplifying the Consumer Example

- The `Iterable` interface have been enhanced with a `forEach` method
- Signature of the method is:

```
void forEach(Consumer<? super T> action)
```

- And the default implementation behaves as:

```
for (T t: this)
   action.accept(t);
```

- The example on the previous slide can be simplified to:

```
static void evaluate(List<Order> orders,
                     Consumer<Order> consumer){
  orders.forEach(consumer);
}
```

Apply consumer

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

The `Function` Functional Interface

- Represents a unary function
  - Performs a function on a single argument of type T
  - Returns a result of type R

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

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Function` Example

- The following example shows a simple function defined to calculate the average value of the orders

Receives `List<Order>` and returns `Double`

```
Function<List<Order>, Double> averageOrder = x -> {
  double total = 0.0;
  for(Order order: x) {
    total+= order.getAmount();
  }
  return total/x.size();
};
```

Invoke function

```
System.out.println(averageOrder.apply(orders));
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composing Functions

- The Function interface has default methods that return a `Function`
  - Allows functions to be chained
    - To create processing/transformation pipelines

- `Function` **`andThen`**`(Function after)`
  - The `after` function is applied after the calling function

- `Function` **`compose`**`(Function before)`
  - The `before` function is applied first, and then the calling function

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composing Function Example

- The following shows the use of compose and `andThen`

```
Function<Integer,Integer> addOne = x -> x+1;
Function<Integer,Integer> multiplyByTwo = x -> x*2;

Function<Integer,Integer> andThenExample =
                              addOne.andThen(multiplyByTwo);
Function<Integer,Integer> composeExample =
                              addOne.compose(multiplyByTwo);

System.out.println(andThenExample.apply(10));
System.out.println(composeExample.apply(10));
```

Apply processing chain

- What two values are output to the console when this code runs?

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composing `Comparators`

- Earlier, we sorted orders by amount with the following code:

```
List<Order> orders = …
Collections.sort(orders, Comparator.comparing(Order::getAmount));
```

Generates `Comparator`

Method reference

- What if we wanted to sort the orders in decreasing amount?
  – The `Comparator` interface provides a default method `reversed()`

Reverses sort order

```
List<Order> orders = …
Collections.sort(orders, Comparator.comparing(Order::getAmount).reversed());
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chaining `Comparators`

- Assume we sort orders by amount, but we get two orders of the same amount
  - In this case, we want the orders to be further sorted by side (SELL or BUY)
- The `Comparator` provides a default method `thenComparing()` that allows chaining

```
import static java.util.Comparator.*;
…
List<Order> orders = …
Collections.sort(orders, comparing(Order::getAmount)
                             .thenComparing(Order::getSide));
```

Chain comparator

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Further Functional Interfaces

- There are a number of other functional interfaces available
  - `UnaryOperator<T>`
  - `BinaryOperator<T>`
  - `Supplier<T>`
  - Many more
- We will see some more later in the course

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 1.2: Working with Functional Interfaces

HANDS-ON EXERCISE

- Please turn to the Exercise Manual and complete Exercise 1.2: Working with Functional Interfaces

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Default Methods

Lambda Expressions

Method References

Functional Interfaces

➤ **Chapter Summary**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have introduced:

- Default methods in interfaces
- Lambda expressions
- Method references
- Functional interfaces

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Chapter 2: Java Streams

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will introduce:

- Java 8 streams
- How to filter streams
- Collect results of processing pipelines
- Run stream processing in parallel
- Create streams

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

➤ **Introducing Streams**

Filtering Streams

Stream Terminal Operations

Parallel Processing and Stream Creation

Processing Streams

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Stream Example

- Consider the code example from the previous chapter

```
static void evaluate(List<Order> orders,
                     Predicate<Order> predicate){
    for(Order order : orders){
        if (predicate.test(order)) {
            System.out.println(order);
        }
    }
}
```

Apply predicate

Supply predicates

```
evaluate(orders, o -> o.getSide() == BUY);
evaluate(orders, o -> o.getSide() == SELL);
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Stream Example (continued)

- The code on the previous slide can be restructured/improved
- Consider what the code is doing
  - Filtering all those orders that match a supplied predicate
    - Print those orders to the console
- A more elegant solution can be provided using streams

```
static void evaluate(List<Order> orders,
                     Predicate<Order> predicate){
   orders.stream().filter(predicate).forEach(System.out::println);
}
```

List supplies a stream

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Java 8 Streams

- Streams allow collections of data to be manipulated in a declarative way
  - You specify *what* you want to do
    - As opposed to *how* to implement what you want to do
- For example, in the code below we specify that we want to:
  - Filter the data
  - Iterate over it

```
static void evaluate(List<Order> orders,
                     Predicate<Order> predicate){
   orders.stream().filter(predicate).forEach(System.out::println);
}
```

Declarative approach—
what not how

- Operators on a stream can be chained together
  - To create complicated data processing pipelines

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

What Are Streams?

- A stream is “*a sequence of elements from a source that supports data processing operations*”
- Source – streams consume data from a source
  - For example: collections, arrays, I/O devices
- Data processing operations
  - Streams support operators such as:
    - Filter based on a predicate
    - Sort
    - Find
    - Match
    - Etc.

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Comparing Streams and Collections

- Collections are in memory structures that hold all the data
  - Streams do not store their elements—they are computed on demand
    - Can be an infinite source of data
- Collections require external iteration
  - Streams provide internal iteration
- Streams are consumable—can only use stream once
  - Have to be recreated to access data again
- Stream operations are lazy when possible for performance reasons

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Introducing Streams

➤ **Filtering Streams**

Stream Terminal Operations

Parallel Processing and Stream Creation

Processing Streams

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Working with Streams

- Streams provide two types of operations
  1. Intermediate
  2. Terminal

- Intermediate operations return another stream
  - Allows creation of processing pipelines

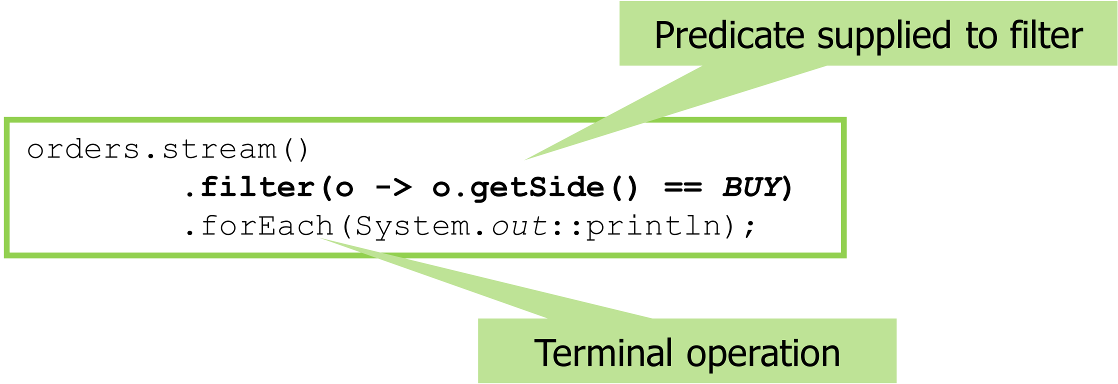
- Terminal operations produce a result from a processing pipeline

- Working with streams involves three stages
  1. Create a source stream
  2. Add a chain of intermediate operations
  3. Add a terminal operation to the end of the pipeline
     - The terminal operation actually executes the stream pipeline

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Filtering Streams

- Streams can be filtered based on a predicate
- `filter(Predicate<T>)`
  - The method returns a `Stream<T>`
- Consider printing all the buy side orders



Predicate supplied to filter

```
orders.stream()
         .filter(o -> o.getSide() == BUY)
         .forEach(System.out::println);
```

Terminal operation

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Limiting and Skipping

- It is possible to restrict the stream processing to the first `n` elements
  - Use the `limit(n)` method

First 10 elements of stream only

```
orders.stream()
          .filter(o -> o.getSide() == BUY)
          .limit(10)
          .forEach(System.out::println);
```

- `skip(n)` method allows for the first n elements to be skipped

Skip first 10 elements of stream

```
orders.stream()
          .filter(o -> o.getSide() == BUY)
          .skip(10)
          .forEach(System.out::println);
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Sorting Streams

- `Stream` provides a `sorted()` method for sorting streams
- By default it returns stream items in natural order
  - If they implement `Comparable`
- Or, method takes a `Comparator`

```
orders.stream()
        .filter(o -> o.getSide() == BUY)
        .sorted(comparing(Order::getAmount))
        .forEach(System.out::println);
```

Sorted by amount of order

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Transforming Streams

- Streams can be transformed using the `map()` function
  - The function returns a stream
    - Elements in returned stream are the result of applying the supplied function to source stream

- In the example below, the `getAmount()` method is called on each element in the stream
  - The resulting stream is the values returned from the calls to `getAmount()`

```
orders.stream()
        .filter(o -> o.getSide() == BUY)
        .map(Order::getAmount)
        .forEach(System.out::println);
```

New stream of just the amount of each order

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Types of Streams

Introducing Streams

Filtering Streams

➤ **Stream Terminal Operations**

Parallel Processing and Stream Creation

Processing Streams

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Generating Results from Streams

- Functions are provided that allow a single result to be generated from a stream of data
  - These are terminal operations
- `reduce()` operation is one example
  - Takes two arguments
    - Initial value
    - Binary function to be called
      - First parameter is current partial result
      - Second parameter is next data item
- The example at right finds the total amount of all buy side orders

```
double total = orders.stream()
        .filter(o -> o.getSide() == BUY)
        .map(Order::getAmount)
        .reduce(0.0, (a,b) -> a+b);
```

`0.0` is initial value for result

`a` represents partial result, `b` current stream data value

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Generating Results from Streams (continued)

- For a stream of orders, assume that the `map()` operation returns the values: 1.0, 2.0, 3.0

```
double total = orders.stream()
         .filter(o -> o.getSide() == BUY)
         .map(Order::getAmount)
         .reduce(0.0, (a,b) -> a+b);
```

- The processing of reduce would proceed as follows:
  - The lambda expression passed to `map()` is applied to each value in the stream

```
double total =  0.0
       total = (0.0,1.0) -> a+b
       total = (1.0,2.0) -> a+b
       total = (3.0,3.0) -> a+b
```

This is pseudo code

Current total

Next value in stream

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Other Terminal Functions

- Streams also provide methods for calculating a single result from stream data
  - `min()`
  - `max()`
  - `count()`
- `min()` and `max()` both take a comparator as an argument

```
double numberOfOrders = orders.stream()
         .filter(o -> o.getSide() == BUY)
         .count();
```

Counts number of data items in stream

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Numeric Streams

- For processing streams of numbers, three streams are provided
  - `IntStream`, `DoubleStream`, `LongStream`
- These provide convenience operations such as `max`, `min`, `average`, `sum`

```
double total = orders.stream()
        .filter(o -> o.getSide() == BUY)
        .mapToDouble(Order::getAmount)
        .sum();
```

Returns a `DoubleStream`

- `IntStream` and `LongStream` provide range method for generating a range of integers

```
IntStream.range(1,10).forEach(System.out::println);
```

Generates stream 1 to 9

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Collector` Terminal Operations

- `collect()` allows multiple values from a stream to be 'collected'
  - Can be saved in a result variable
- Uses `Collectors` class to gather actual data
- Consider wanted a `List<Order>` that contains only buy side orders from a stream of orders

```
List<Order> buySideOrders = orders.stream()
        .filter(o -> o.getSide() == BUY)
        .collect(Collectors.toList());
```

Convert stream to list

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Collectors` Operations

- `Collectors` class provides a number of operations
- For example, the operation `groupingBy()`
  - Forms groups of elements of a stream that have common characteristics
    - For example, the value of a property
- The example below groups the stream of orders by their order side

```
Map<Side, List<Order>> filteredOrders =
  orders.stream()
          .collect(Collectors.groupingBy(Order::getSide));
```

Grouped results

Group orders by side

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Short Circuit Evaluation of Streams

- Stream operations are referred to as *lazy* operations
  - They are only evaluated when a terminal operation is invoked on the pipeline
  - Enables intermediate operations to be merged, if possible
    - Leads to more efficient processing of streams

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> oddSquares =
      numbers.stream()
          .filter(n->{
                  System.out.println("filtering " + n);
                  return n % 2 != 0;
              })
          .map(n->{
                  System.out.println("mapping " + n);
                  return n*n;
              })
          .limit(3)
          .collect(toList());
```

What is the output of the following program?

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Short Circuit Evaluation of Streams (continued)

- The output of the previous program is as shown below
- The stream is only partly processed because of the limit(3)
  - Short circuit valuation is performed

```
filtering 1
mapping 1
filtering 2
filtering 3
mapping 3
filtering 4
filtering 5
mapping 5
```

Only first 5 items in stream are processed

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Introducing Streams

Filtering Streams

Stream Terminal Operations

➤ **Parallel Processing and Stream Creation**

Processing Streams

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Parallel Streams

- Streams can be processed sequentially or in parallel
- A parallel stream breaks the stream into chunks
  - Each chunk is processed with a different thread
- Parallel stream is created using the `parallelStream()` method
  - `parallel()` method on an existing sequential stream can also be called
    - Results in a parallel stream in processing pipeline

```
List<Order> orders = …

double total = orders.parallelStream()
            .filter(o -> o.getSide() == BUY)
            .mapToDouble(Order::getAmount)
            .sum();
```

Stream processed in parallel

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Streams

- Streams can be created in a variety of ways
- Using the static `Stream.of` method

```
Stream<String> currencies = Stream.of("USD", "EUR", "JPY");
```

- Creating a stream from an array

```
int [] numbers = {1,2,3,4,5,6,7,8,9,10};
IntStream integers = Arrays.stream(numbers);
```

- A stream can be created from a file

```
try (Stream<String> lines = Files.lines(Paths.get("orders.csv"))){
    lines.forEach(System.out::println);
}
catch(IOException e){
    System.out.println(e.toString());
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Streams From Functions

- The Streams API provides two static methods for creating streams
  - `iterate()` and `generate()`
  - Allow creation of infinite streams—no fixed size
- `iterate()` takes a seed and a UnaryOperator to be applied to each new value it produces
- `generate()` takes a supplier

```
Stream<Order> tradeStream = Stream.generate(() -> {
        return createNextOrder();
    });



tradeStream.limit(10).forEach(System.out::println);
```

User-defined method, source of objects for stream

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 2.1: Working with Streams

HANDS-ON EXERCISE

- Please turn to the Exercise Manual and complete Exercise 2.1: Working with Streams

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Introducing Streams

Filtering Streams

Stream Terminal Operations

Parallel Processing and Stream Creation

➤ **Processing Streams**

Chapter Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Finding and Matching

- A common stream processing pattern is to determine if elements in the stream match a condition
- Stream interface provides operations to enable this to be performed
  - `anyMatch`
  - `allMatch`
  - `noneMatch`
- All operations take a predicate as an argument and return a `boolean`

All are buy side orders

```
boolean allBuySide = orders.stream()
         .allMatch(o -> o.getSide() == BUY);
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Optional<T> Class

- Introduced in Java 8
- A container that may or may not contain a non-null value
  - Helps avoid `null` checks and `NullpointerException`s
- Methods of `Optional<T>` include:
  - `isPresent()`
    - Returns true if the `Optional<T>` contains an instance of `T`
  - `get()`
    - Returns the contained object if there is one
      - or throws `NoSuchElementException`
  - `ifPresent()`
    - Executes a block of code if there is an instance of `T` in the `Optional<T>`
- We will see an example of its use next

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

FindFirst and FindAny

- Both operations return an optional reference
  - Contents based on whether item has been found

```
Optional<Order> orders = orders.stream()
          .filter(o -> o.getSide() == BUY)
            .findAny();
```

Find any buy side order

- In the above, the Optional can be used to test if a buy side order was present
  - Can then apply an operation to object found

```
orders.stream()
     .filter(o -> o.getSide() == BUY)
     .findAny()
     .ifPresent(System.out::println);
```

Print order if found

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Reusing Streams

- Streams cannot be reused
- As soon as a terminal operation is called, the stream is closed

```
Stream<String> stream =
          Stream.of("a","b","c","d","e")
                   .filter(s-> s.startsWith("a"));
stream.anyMatch(s -> s.startsWith("a"));
stream.noneMatch(s -> s.startsWith("b"));
```

Exception as stream closed

- If a stream is required to be reused, can use a stream supplier

```
Supplier<Stream<String>> streamSupplier =
          () -> Stream.of("a","b","c","d","e")
                  .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> s.startsWith("a"));
streamSupplier.get().noneMatch(s -> s.startsWith("b"));
```

OK now as new stream available

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Revisiting Collectors

- Collect is a terminal operation that is extremely useful
  - Transform stream into a List, Set, or Map
- Collect uses a Collector to collect data
  - `Collectors` class provides built-in collectors
    - For example, to convert a stream to a `List`, `Set`, or `Map`
    - Also to perform groupings, averaging, statistics calculations
- The example at right groups orders by `Side` and places them in a `Map`

```
import static java.util.stream.Collectors.*;

List<Order> orders = …
Map<Side, List<Order>> ordersBySide =
      orders
        .stream()
        .collect(groupingBy(Order::getSide));

System.out.println("Orders grouped by side ");
ordersBySide.forEach((side, o)->
                    System.out.printf("%s:  %s%n", side, o));
```

Key in map will be `Side`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

More on `Collectors`

- Provides many implementations of `Collector` with useful stream reductions
  - Averaging
  - Counting
  - Grouping
  - Joining
  - Maximum
  - Minimum
  - Statistical summaries
  - Summing
- Take a look at the JavaDoc for the `Collectors` class

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Collectors Summing Example

- Problem:
  - Given a stream of orders, determine the total monetary amount of orders per currency
- Solution:
  - We need to group the orders by currency
    - Then sum the amount of each order per currency

```
List<Order> orders = …

Map<Currency, Double> orderTotalByCurrency =
   orders
   .stream()
   .collect(groupingBy(Order::getCurrency,
                       summingDouble(Order::getAmount) ));

System.out.println("\nOrder total per currency ");
orderTotalByCurrency.forEach((c, a)->
          System.out.printf("%s: total order value %.2f%n", c, a));
```

Sum of orders per currency

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Collectors` Averaging Example

- Problem:
  - Given a stream of orders, determine the average amount of an order
- Solution:
  - Use the `averagingDouble()` method of `Collectors`
    - Returns a `Collector` that produces mean of double-valued function applied to stream elements

```
List<Order> orders = …

Double averageOrderAmount =
          orders
             .stream()
             .collect(averagingDouble(o-> o.getAmount()));

System.out.printf("%nAverage amount of each order is %.2f %n",
                                                              averageOrderAmount);
```

Calculates average from stream

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Collectors` Summarizing Example

- Problem:
  - Given a stream of orders:
    - Determine maximum, minimum, average order amount
    - Number of orders
    - The total amount of all orders

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Collectors` Summarizing Example (continued)

- Solution:
  - Use the `summarizingDouble()` method of `Collectors`
    - Returns a `Collector` that produces summary statistics of a double-valued function applied to stream elements

```
List<Order> orders = …

DoubleSummaryStatistics amountSummary =
        orders
          .stream()
          .collect(summarizingDouble(o-> o.getAmount()));

System.out.printf("Order Amount Summary %s %n", amountSummary);
```

Calculates summary statistics from stream

```
Order Amount Summary DoubleSummaryStatistics{count=10,
sum=35800000.000000, min=1000000.000000, average=3580000.000000,
max=9800000.000000}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`Collectors` Maximum Example

- Problem:
  - Given a stream of orders, determine maximum order value of each currency
- Solution
  - Use the `maxBy()` method of `Collectors`
    - Returns a `Collector` that produces the maximal element in the stream
    - Maximum element determined by a supplied comparator
    - Returned value is an `Optional`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Collectors Maximum Example (continued)

Maximum order

```
List<Order> orders = …

Map<Currency, Optional<Order>> maxOrderByCurrency =
          orders
          .stream()
          .collect(groupingBy(Order::getCurrency,
                             maxBy(comparing(Order::getAmount)) ));




System.out.println("\n\nMaximum order per currency ");
maxOrderByCurrency.forEach((c, o)->
          System.out.printf("%s: maximum order value %.2f%n", c,
                              o.orElse(0.0));
```

Display maximum amount for each currency

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 2.2: Further Working with Streams

- Please turn to the Exercise Manual and complete Exercise 2.2: Further Working with Streams

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`flatMap` Operation

- Consider using stream processing to determine the number of unique words in a file
- The code below may seem like a solution

Split on whitespaces

```
Files.lines(Paths.get("test.txt"))
        .map(line -> line.split("\\s+"))
        .distinct()
        .forEach(System.out::println);
```

- However, the result of the above is as follows:

String representation of `String[]`

```
[Ljava.lang.String;@7229724f
[Ljava.lang.String;@4c873330
[Ljava.lang.String;@119d7047
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`flatMap` Operation (continued)

- The problem on the previous slide is because of the lambda passed to `map()`
  - It returns an array of `String`s
  - `map()` then returns a `Stream<String[]>`
    - We actually want a `Stream<String>` to be returned

```
Files.lines(Paths.get("test.txt"))
          .map(line -> line.split("\\s+"))
          .distinct()
          .forEach(System.out::println);
```

Returns `Stream<String[]>`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`flatMap()`

- `flatMap()` returns a stream
  - Replaces elements of input stream with contents generated by supplied mapping function

```
Files.lines(Paths.get("test.txt"))
         .map(line -> line.split("\\s+"))
         .flatMap(Arrays::stream)
         .distinct()
         .forEach(System.out::println);
```

Returns `Stream<String>`

- `flatMap()` in the example above processes a `Stream<String[]>`
  - Uses `Arrays::stream` to generate `Stream<String>` from `Stream<String[]>`
    - It *flattens* the stream

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Another `flatMap` Example

- Consider the following class:

```
class Trader{
  private int id;
  private List<Order> orders = new ArrayList<>();
  …
  public Trader(int id) {
    this.id = id;
  }

  public List<Order> getOrders() {
    return orders;
  }

  public void addOrder(Order order) {
    orders.add(order);
  }
  …
}
```

Each trader has a `List<Order>`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Traders and Orders

- Using streams, we can create traders as follows:

```
List<Trader> traders = new ArrayList<>();

IntStream
  .range(1,4)
  .forEach(i -> traders.add(new Trader(i)));


  traders.forEach( t->
     IntStream
          .range(1,6)
          .forEach(o -> t.addOrder(new Order())));
```

Create traders

Add orders to traders

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`flatMap` Example

- How can we count the total number of orders across all traders?

```
long totalNumberOrders =
    traders.stream()
          .map(t->  (t.getOrders()))
          .count();
System.out.printf("Total number of orders is %d%n",
                                        totalNumberOrders);
```

Returns
`Stream<List<Order>>`

- The above code will print the number of `Stream <List<Order>>`
  – Will be equal to the number of traders

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`flatMap` Example (continued)

- To count the total number of orders across all traders, we need to use `flatMap()`
  - To *flatten* the orders into a stream

Returns
`Stream<Order>`

```
long totalNumberOrders =
    traders.stream()
          .flatMap(t->  (t.getOrders().stream()))
          .count();
System.out.printf("Total number of orders is %d%n",
                                        totalNumberOrders);
```

- The above code will print the number of orders

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Introducing Streams

Filtering Streams

Stream Terminal Operations

Parallel Processing and Stream Creation

Processing Streams

➤ **Chapter Summary**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have introduced:

- Java 8 streams
- How to filter streams
- Collect results of processing pipelines
- Run stream processing in parallel
- Create streams

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Chapter 3: REST Web Services with Spring Boot

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will:

- Introduce REST Web Services
- Introduce Spring Boot
- Learn how to write REST Web services using Spring Boot

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

➤ **Web Services**

HTTP and JSON

RESTful Services

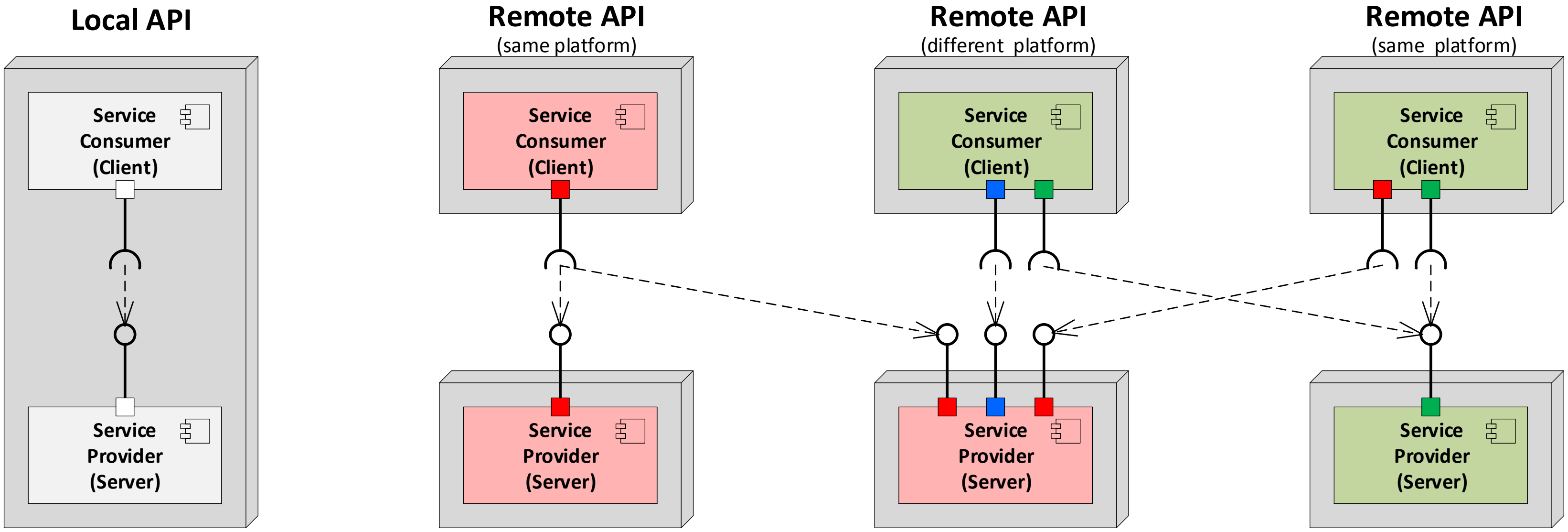
Spring Boot

Exercise

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Application Programming Interface (API)

- API is a set of clearly defined methods of **communication** between various software components


Local API
Service Consumer (Client)
Service Provider (Server)
Remote API
(same platform)
Service Consumer (Client)
Service Provider (Server)
Remote API
(different platform)
Service Consumer (Client)
Service Provider (Server)
Remote API
(same platform)
Service Consumer (Client)
Service Provider (Server)

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Web Services

- Web Service
  - Cross-platform way to integrate applications
  - Application functionality exposed over network, typically over WWW
  - Communication protocols: usually HTTP, but can use other protocols such as ESMTP, message queues, etc.
- They provide great interoperability and extensibility
- They are loosely coupled
  - Can be combined to build complex applications
  - Components can be developed in different languages on different architectures

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Types of Web Services

- **S**imple **O**bject **A**ccess **P**rotocol (SOAP) Web Services
  - Interfaces defined using Web Services Description Language (WSDL)
  - Messages are exchanged in XML
- **Re**presentational **S**tate **T**ransfer (RESTful) Web Services
  - Lightweight infrastructure which is completely stateless
  - Implementations require minimal tooling

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

SOAP vs. REST: Typical Use Cases

- **S**imple **O**bject **A**ccess **P**rotocol (SOAP) Web Services
  - RPC style of integration (**verb-first**)
  - System to System integration within a single enterprise or across enterprises
  - Presence or need for enterprise-wide integration standards (primarily WS-Security)
  - Strong formal service contracts and formal governance (in most cases)
  - Service consumers are known and very often formal agreements

- **Re**presentational **S**tate **T**ransfer (RESTful) Web Services
  - ‘Document’ CRUD style of integration (**noun-first**)
  - Client (Browser) to System as well as System to System integration
  - Many ‘unknown’ consumers (client apps for Yahoo, Google, etc.—any Internet service)
  - Need for high adaptability and flexibility

- RESTful Web Service typical implementation:
  - **JSON over HTTP**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Web Services

➤ **HTTP and JSON**

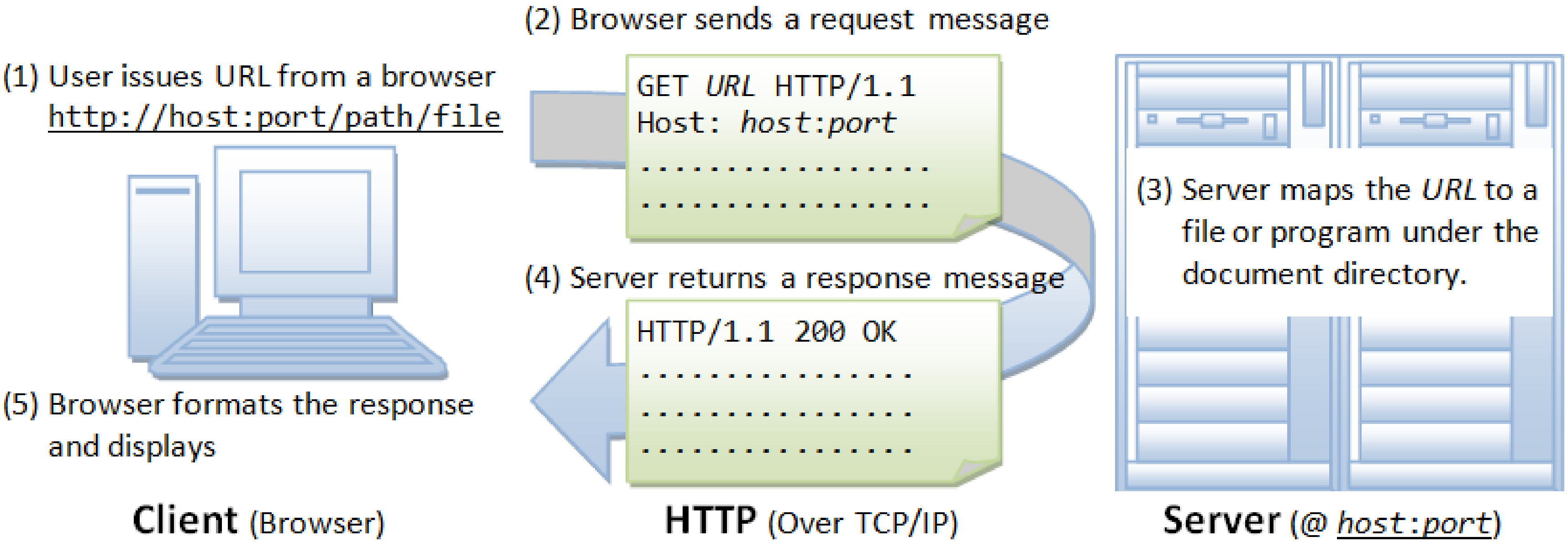
RESTful Services

Spring Boot

Exercise

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

HTTP – HyperText Transfer Protocol



(1) User issues URL from a browser
http://host:port/path/file

(2) Browser sends a request message

```
GET URL HTTP/1.1
Host: host:port
..................
..................
```

(3) Server maps the *URL* to a file or program under the document directory.

(4) Server returns a response message

```
HTTP/1.1 200 OK
.................
.................
.................
```

(5) Browser formats the response and displays

Client (Browser)

HTTP (Over TCP/IP)

Server (@ *host:port*)

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

JSON

- **J**ava**S**cript **O**bject **N**otation (JSON)
  - A lightweight data-interchange format derived from the ECMAScript (JavaScript)
  - Syntax defined in ECMA-404 – The JSON Data Interchange Standard
  - Easy for humans to read and write, easy for machines to parse and generate

- JSON is built on two structures
  - Object (map): a collection of `name:value` pairs separated by comma
    - `{"key1":"value1", "key2":"value2"}`
  - Array (list): a collection of ordered `values` separated by comma
    - `["value1", "value2", "value3"}]`

- JSON `values` can be:
  - Strings (`"string1"`)
  - Numbers (`10, 3.141, 2.5E6`)
  - Boolean (`true` or `false`)
  - `null`
  - Another Object or Array (map of lists, list of maps, map of maps, list of lists)

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

JSON – Combining Objects and Arrays

- Objects and Arrays can be combined:
  - Family members aggregated by last name

```
{"Smiths": ["John", "Jane"],
 "Jones" : ["Ann", "Dave", "Rob"]}
```

  - List of individuals (with ‘firstName’ – optional)

```
[
  {"lastName": "Doe"                       },
  {"lastName": "Smith", "firstName": "John"},
  {"lastName": "Smith", "firstName": "Jane"}
]
```

  - List of individuals (with ‘firstName’ – optional)

```
{"building1": {"1A": {"Smiths": ["John", "Jane"      ]},
                     {"Jones" : ["Ann", "Dave", "Rob"]}},
              {"2A": {"Kramer": ["Cosmo"             ]}}
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Web Services

HTTP and JSON

➤ **RESTful Services**

Spring Boot

Exercise

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST – High-Level Overview

- **Re**presentative **S**tate **T**ransfer—REST or ReST
  - A software architecture style
  - Guidelines and best practices for creating scalable web services
  - Described in Roy Fielding's doctoral thesis
- Typically communicates over HTTP
- Common data exchange format – JSON
- REST was developed by W3C in parallel with HTTP 1.1
  - The World Wide Web is an implementation of REST
- There is no official standard for REST APIs
  - REST is an architectural style
  - SOAP is a protocol which has standards
  - REST usually uses standards such as HTTP, URI, JSON, XML

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST Principles

- Application domain model (resources) are manipulated using standard set of actions
- Resources are identified by **U**niform **R**esource **I**dentifiers (URIs) and organized into collections in a tree-like structure
  - E.g.: http://mydealership.com/locations/**{*locId*}**/cars/**{*carId*}**
- Actions are normally represented via HTTP operations applied to **any** part of URI
  - `GET`, `POST`, `DELETE`, `PUT`, `PATCH`, etc.
- Data can be exchanged in various formats, though most common ones are **JSON** and XML
- Interactions are stateless
  - Actions are used to change the state of the resource one at a time
  - Each call is normally independent from each other
- Errors are handled via HTTP status codes
  - **200**: OK; **404**: Resource Not Found; **400**: Bad Request; **201**: New Resource Created

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST Operations – `GET`

- `GET` operation is a safe method and has no side effects (‘R’ in the CRUD)
  – Server-side content is unchanged

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com



GET /inventory/cars        HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"    : "honda",
 "licPlate": "BDK032",
 "invId"    : 1}

HTTP/1.1 200 OK
[{"model"    : "honda",
  "licPlate": "BDK032",
  "invId"    : 1},
 {"model"    : "toyota",
  "licPlate": "GAV101"
  "invId"    : 2}]
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST Operations – POST

- POST operation is used to create resources ('C' in the CRUD)
  - Normal practice is to return a handler (id) to the created resource

Request

```
POST  /inventory/cars         HTTP/1.1
Host: mydealership.com
{"model"    : "ford",
 "licPlate": "KYE903"}




GET  /inventory/cars/3        HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 201 Created
{"model"    : "ford",
 "licPlate": "KYE903",
 "invId"    : 3}
                              OR, simply,
{"invId"    : 3}


HTTP/1.1 200 OK
{"model"    : "ford",
 "licPlate": "KYE903",
 "invId"    : 3}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST Operations – `PUT`

- `PUT` is an idempotent operation used to replace existing resource or create one if it doesn't exist ('C' and 'U' in the CRUD)
  - Resource is replaced as a 'whole'

Request

```
GET  /inventory/cars/1      HTTP/1.1
Host: mydealership.com


PUT  /inventory/cars/1      HTTP/1.1
Host: mydealership.com
{"model"    : "tesla",
 "licPlate": "AAA001"}

GET  /inventory/cars/1      HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"    : "honda",
 "licPlate": "BDK032", … }

HTTP/1.1 200 OK
    (with optional mirroring back of
     updated resource)


HTTP/1.1 200 OK
{"model"    : "tesla",
 "licPlate": "AAA001", … }
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST Operations – `PATCH`

- `PATCH` is an operation used to update existing resource (‘U’ in the CRUD)
  - Only some attributes of the resource are updated
  - Not used too often due to ambiguity of operation to be used (default is ‘update’)

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com


PATCH /inventory/cars/1    HTTP/1.1
Host: mydealership.com
{"model":    "tesla"}


GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"    : "honda",
 "licPlate": "BDK032", … }

HTTP/1.1 200 OK
    (with optional mirroring back of
     updated resource)


HTTP/1.1 200 OK
{"model"    : "tesla",
 "licPlate": "BDK032", … }
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

REST Operations – `DELETE`

- `DELETE` is an idempotent operation used to delete existing resource (‘D’ in the CRUD)

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com


DELETE /inventory/cars/1   HTTP/1.1
Host: mydealership.com



GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"   : "honda",
 "licPlate": "BDK032", … }

HTTP/1.1 200 OK
    (with optional mirroring back of
     deleted resource)


HTTP/1.1 404 Not Found
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

RESTful API: Best Practices

- Use correct HTTP method names

Resource	`GET` read	`POST` create	`PUT` update	`DELETE` delete
`/cars`	Returns a list of cars	Create a new car	Bulk update of cars	Delete all cars
`/cars/711`	Returns a specific car	Method not allowed (405)	Updates a specific car	Deletes a specific car

- Use nouns, not verbs, in the URI
  - That is, do **NOT** use `/addCar`, `/deleteCar`, `/updateCar`
- Use URI query parameters for filtering, sorting, field selection, or pagination

```
GET /cars?color=red&seats=2
          &sort=manufacturer,model
          &fields=manufacturer,model,id,color
          &offset=10&limit=5
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

RESTful API: Best Practices (continued)

- Version your API to avoid breaking existing clients when API changes
  - http://mydealership.com/api/**v1**/inventory/cars/1
    - Use a simple ordinal number
    - Avoid dot notation such as 2.5
- Use correct HTTP status codes to communicate both success and failures
  - See https://tools.ietf.org/html/rfc7231 for details; keep in mind that industry practice might deviate occasionally
  - Duplicate HTTP status code in the body of the Response message:

```
HTTP/1.1 404 Not Found
{"Message": "Not Found",
 "Code"    : 404}

HTTP/1.1 201 Created
{"Message": "Created",
 "Code"    : 201}
```

```
HTTP/1.1 200 OK
{"Response" : {"model"    : "honda",
               "licPlate": "BDK032",
               "invId"    : 1},
 "Message"  : "Ok",
 "Code"     : 200}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

RESTful API: Best Practices (continued)

- Build the API with consumers in mind
  - Make sure hierarchy is easy to navigate for your target clients/application domain
  - Add filtering, sorting, pagination capabilities
- Create two endpoints per resource
  - The resource collection (e.g., `/cars`)
  - Individual resource within the collection (e.g., `/cars/{`***carId***`}`)
- Alternate resource names with IDs as URL nodes where needed

```
/LEVEL 1              /LEVEL 2                  /LEVEL 3  / …
--------------------------------------------------------------
/locations/{locId}    /cars        /{carId}
                      /staff       /{empId}
                      /sales       /{yyyymmdd}
/employees/{empId}
/accounts /{accountId}/transactions/{txnId}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Richardson Maturity Model

- Dr. Leonard Richardson developed a model that breaks down the principal elements of a REST approach into three steps
  - http://martinfowler.com/articles/richardsonMaturityModel.html

- Model defines four maturity levels of RESTful API
  - Level 0:
    - RPC-style API, usually with a single endpoint
  - Level 1 – Resources:
    - Resources are introduced; multiple endpoints based on the structured URI
  - Level 2 – HTTP Verbs:
    - Same as Level 1 + HTTP verbs to distinguish between operations
  - Level 3 – Hypermedia Controls:
    - HATEOS (Hypertext As The Engine Of Application State) – 'Discoverable' API
    - Response message contains **WHAT** we can do next and **HOW** to do it
      - Think hyperlinks on HTML pages

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

HATEOS API Example

Level 2

```
GET /locations/1/staff

HTTP/1.1 200 OK
[{"firstName": "David",
  "lastName" : "Bowie",
  "empId"    : 1},
[{"firstName": "Annie",
  "lastName" : "Lennox",
  "empId"    : 2}]

GET /employees/2
{"firstName": "Annie",
 "lastName" : "Lennox",
 "empId"    : 2,
 "dateHired": "2010-09-24", … }
```

Level 3

```
GET /locations/1/staff

HTTP/1.1 200 OK
[{"data": {"firstName": "David",
           "lastName" : "Bowie",
           "empId"    : 1},
  "URIs": {"details"  : "/employees/1"}},
 {"data": {"firstName": "Annie",
           "lastName" : "Lennox",
           "empId"    : 2},
  "URIs": {"details"  : "/employees/2"}}]

GET /employees/2
{"data": {"firstName" : "Annie", … },
 "URIs": { … }}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 3.1: Create URI Resource Hierarchy

GROUP EXERCISE

- Break up into groups
- Pick a subject domain
  - Can be anything: HR system, car dealership, inventory system, etc.
- Create a hierarchy of resources following best practices
- CHALLENGE!
  - Is there any other way to navigate your subject domain?

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Web Services

HTTP and JSON

RESTful Services

➤ **Spring Boot**

Exercise

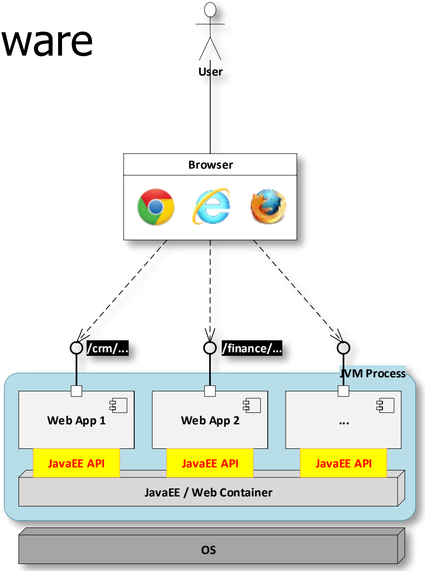
© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Web Application

- Web Application is a client–server computer program where:
  - The client (including the user interface and client-side logic) runs in a web browser
  - The server produces dynamic content (such as HTML pages) based on user actions

- Java Web Applications are managed and executed by special middleware called '**JavaEE container**' or '**Web/Servlet Container**'

- Web Container is a runtime environment for web application which handles:
  - Network connectivity
  - Lifecycle management
  - Application security
  - Concurrency
  - Transactions
  - Etc.



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Traditional JavaEE Frameworks vs. Spring Boot

Traditional Frameworks

- Pick favorite MVC framework
- Download additional libraries
  - Make sure to use the right version
  - Add Spring framework if needed
- Compile and create WAR file
- Install and configure application server
- Deploy WAR file to application server

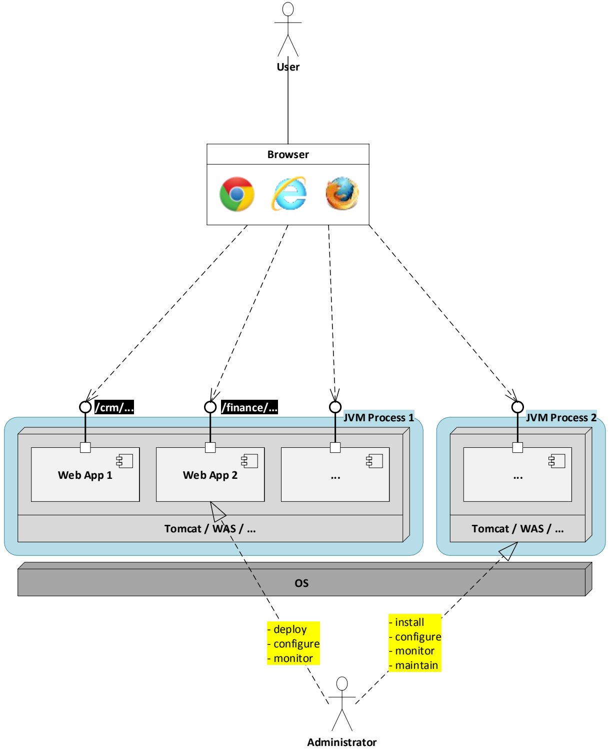
Spring Boot

- Put Spring Boot library in project dependencies
- Implement web application to conventions of Spring Boot
- Compile and run!

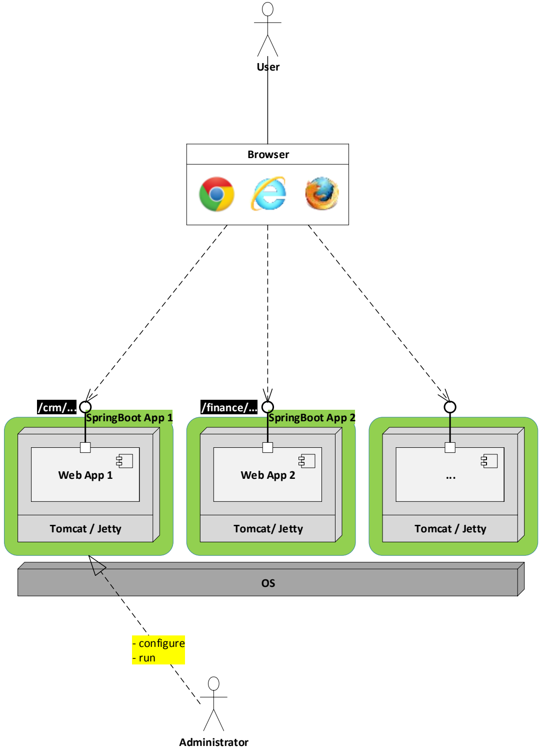
© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Traditional vs. Spring Boot Deployment

Traditional Deployment



Spring Boot Deployment



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot

- Makes it easy to create stand alone applications
  - Very little configuration required
  - Spring and third-party libraries included
- Some of the key features include:
  - Applications begin with main method
  - Embed Tomcat, Jetty, or Undertow directly in application
  - Starter POMs provided simplify Maven configuration
  - Automatically configures Spring whenever possible
  - Provides production-ready metrics, health checks, and externalized configuration
- An ‘accelerator’ to build applications fast
  - Spring MVC, Spring Security, and other Spring libraries can be used WITHOUT Spring Boot

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

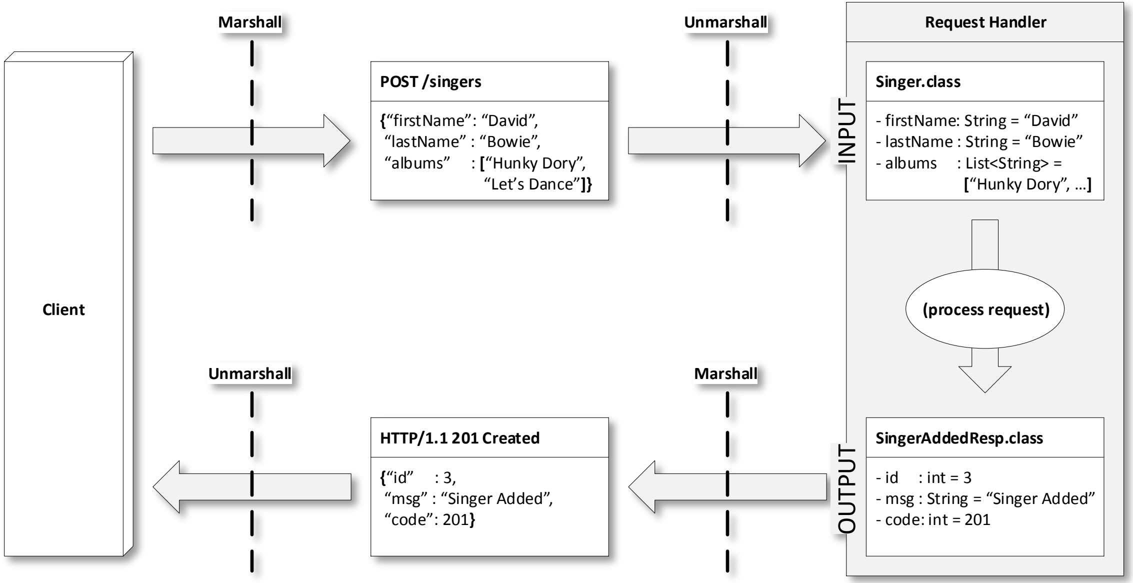
RESTful Service Implementation

- Java RESTful services are deployed within JavaEE or Web Container
  - Same as Web Applications
- Major differences between Web Application and RESTful Service implementations:
  - Data exchange format
    - HTML **vs.** JSON/XML/…
  - Frameworks used
    - SpringMVC/Struts2 **vs.** DropWizard/Restlet
      - Some frameworks, such as PlayFramework or **Spring Boot**, can be used for both
  - Specifications adhered to
    - ServletAPI **vs.** JAX-RS
      - Some implementations are ‘specification agnostic’, but follow common ‘request dispatch’ pattern
  - Client implementation
    - Browser **vs.** RESTful client (i.e., another application)

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Request/Response (De-)Serialization (from)to JSON

- Web Application is using JSP, JSF, or other templating engines to generate HTML response
- Spring Boot framework is using special libraries to automatically convert Java objects (POJOs – **P**lain **O**ld **J**ava **O**bjects) into JSON/XML and vice versa



POST /singers

```
{"firstName": "David",
 "lastName" : "Bowie",
 "albums"   : ["Hunky Dory",
               "Let's Dance"]}
```

Singer.class

```
- firstName: String = "David"
- lastName : String = "Bowie"
- albums   : List<String> =
             ["Hunky Dory", …]
```

HTTP/1.1 201 Created

```
{"id"   : 3,
 "msg" : "Singer Added",
 "code": 201}
```

SingerAddedResp.class

```
- id   : int = 3
- msg : String = "Singer Added"
- code: int = 201
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Hello World with Spring Boot

- Spring Boot provides a parent POM and also starter projects
  - Have dependencies required for application type
    - For example, starter Web has dependencies for Spring MVC and REST applications

```
<parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.5.4.RELEASE</version>
</parent>

<dependencies>
       <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-web</artifactId>
       </dependency>
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Simple Service

- The service will return the string “Hello World!” when requested

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(HelloApplication.class, args);
    }
}
```

Entry point of application

```
@RestController
public class HelloService {

    @RequestMapping("/hello")
    String home() {
        return "Hello World!";
    }
}
```

`/hello` routed to this method

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Simple Service Explained

- `@RestController` indicates class represents one or more endpoints
- `@RequestMapping` defines routing information for the services
- `@SpringBootApplication` tells Spring to detect dependencies
  - Configure application based on these dependencies
  - Equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes
- The `main` method
  - Delegates work to `SpringApplication`
    - Bootstraps application starting Tomcat
- *Gotcha:* make sure `@SpringBootApplication` bean is located in the package at the 'top'/'above' other annotated beans

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Running a Spring Boot Application

- The application can be started using a Maven run goal
  - Provided by the starter parent POM
  - Can choose port number Tomcat starts on
    - Default port is 8080

-
  ```
  mvn -Dserver.port=9090 spring-boot:run
  ```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

A Currency Service

- Following examples show a service returning currency data
  - Data will be serialized into JSON:
    - `["USD", "CAD", "GBP"]`

```
@RestController
public class CurrencyService {
  private final Logger log = LoggerFactory.getLogger(this.getClass());

  @RequestMapping(value="/currencies", method = RequestMethod.GET)
  public List<Currency> getCurrencies(){
    return Arrays.asList(Currency.values());
  }
}
```

Accessed by `/currencies` and `HTTP GET` only

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Receiving Client Data

- JSON data sent from client will be marshalled to Java Objects
  - `@RequestBody` indicates data posted from client
  - Unmarshalling happens automatically

```
{
 "currency": "EUR",
 "amount"  : 11,
 "side"    : "BUY"
}
```

```
public class MarketOrder {
  private Currency currency;
  private int      amount;
  private Side     side;
}
```

```
@RestController
public class OrderService {
  private final Logger log = LoggerFactory.getLogger(this.getClass());

  @RequestMapping(value="/order", method = RequestMethod.POST)
  public void addOrder(@RequestBody MarketOrder order){
    // process order
    log.info("Order received "+ order);
  }
}
```

POJO with properties matching JSON property names

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Processing Request Parameters and Path Variables

- `@RequestParam("<param name>")`
- `@PathVariable <named uri segment>`
- Example:

`/cars/711?fields=model`

```
...
  @RequestMapping(value="/cars/{carId}", method = RequestMethod.GET)
  public Car getCarDetails(@PathVariable("carId")  int    carId,
                          @RequestParam("fields") String fieldsToReturn){
    // return car model
    log.info("Car details ("+ fieldsToReturn +") returned");
    return carRepository.findCar(carId);
  }
...
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`RequestMapping` Annotation Shortcuts

- `RequestMapping` shortcut annotations (Spring Boot v1.4 and higher)
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`
  - `@PatchMapping`
- Most attributes can be applied both at type (`@RestController`) and method levels
  - `GET /inventory/cars/711`

```
@RestController
@RequestMapping("/inventory")
public class CarInventoryService {

  @GetMapping("/cars/{carId}")
  public Car getCarDetails(@PathVariable int carId){...}
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Content Negotiation

- The REST controllers can accept and respond with data in different formats
  - Controller inspects ‘Content-Type’ and ‘Accept’ headers set by the client and decides whether it can process the request in ‘Content-Type’ format and respond in the format indicated by ‘Accept’ header:

Request (Content-Type = application/json,
Accept = application/xml,
application/json)

```
POST /cars                     HTTP/1.1
Host          : mydealership.com
Content-Type: application/json
Accept        : application/xml,
                application/json
{"model"    : "ford",
 "licPlate": "KYE903"}
```

Controller (consumes = application/json,
produces = application/xml)

```
HTTP/1.1 201 Created
Content-Type: application/xml
<invId>3</invId>
```

Controller (consumes = application/json,
produces = application/json)

```
HTTP/1.1 201 Created
Content-Type: application/json
{"invId": 3}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Content Negotiation (continued)

- Controller capabilities are defined via ‘produces’ and ‘consumes’ attributes of `@RequestMapping` annotation

```
@RequestMapping(value="/cars", method = RequestMethod.POST,
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.TEXT_XML_VALUE},
        consumes=MediaType.APPLICATION_JSON_VALUE)
public Car createCarRecord(){
  return new Car(…);
}
```

Produces JSON (default) or XML

Consumes JSON

- To serialize data into XML, add the following dependency to pom:

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exception Handling

- Any unhandled exception causes the server to return an HTTP 500 response

```
{ "timestamp": 1516773431477,
  "status"   : 500,
  "error"    : "Internal Server Error",
  "exception": "com.artilekt.bank.business.AccountNotFoundException",
  "message"  : "Account [eebb2ced] not found",
  "path"     : "/accounts/eebb2ced" }
```

- There are two ways to customize exception handling
  - Per exception
    - By annotating custom exceptions with `@ResponseStatus` annotation
  - Globally
    - By creating classes annotated with `@ControllerAdvice` annotation

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exception Handling Customization – Per Exception

- Annotate custom exceptions with `@ResponseStatus` and define HTTP error code

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class AccountNotFoundException extends RuntimeException {
    ...
}
```

- Exceptions thrown from within your code …

```
public Account findAccountByNumber(String accountNumber) {
    Account acc = dao.getAccount(accountNumber);
    if (acc == null)
        throw new AccountNotFoundException("Account ["+accountNumber+"] not found");
    return acc;
}
```

- … would be automatically converted to JSON

```
{ "timestamp": 1516940765026,   "status": 404,   "error": "Not Found",
  "exception": "com.artilekt.bank.business.AccountNotFoundException",
  "message"  : "Account [eebb2ced] not found",   "path": "/accounts/eebb2ced" }
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exception Handling Customization – Global

- To fully customize error response, define `@ControllerAdvice` class(es)

```
@ControllerAdvice
public class ClientExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ ClientDuplicateException.class })
    protected ResponseEntity<GenericErrorResponse> handleConflict(RuntimeException ex,
                                                                  WebRequest request) {
        GenericErrorResponse response = new GenericErrorResponse();
        response.setErrorCode("Client Duplicate");
        response.setErrorMessage(ex.getMessage());

        return new ResponseEntity<>(response, HttpStatus.CONFLICT);
    }
}
```

```
public class GenericErrorResponse {
    private String errorCode;
    private String errorMessage;
}
```

```
{
  "errorCode"   : "Client Duplicate",
  "errorMessage": "Client [Client [id = a001]]
                   already exists" }
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Actuator

- Includes a number of features that let you monitor and manage your application
- Endpoints are made available over HTTP; for example:
  - `/beans` – lists all Spring beans in the application
  - `/configprops` – list of all `@ConfigurationProperties`
  - `/metrics` – list of metrics for the application
  - `/health` – application health information
  - Many more
- Enabled by including the following dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot DevTools

- Spring Boot DevTools improves the development-time experience when working on Spring Boot applications
  - **Automatic Restart** of application whenever files on the classpath change
  - **Live Reload** triggers a browser refresh when a resource is changed
    - Requires browser plugin
  - **Global Settings** properties defined in `~/.spring-boot-devtools.properties` file which will apply to *all* Spring Boot applications on your machine that use devtools
  - **Remote Applications** enable 'live' deployment of updates to remote server as well as remote debugging
  - **H2 Web Console** to view content of in-memory H2 database, available at `/h2-console`
    - http://www.h2database.com/html/quickstart.html#h2_console

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot DevTools (continued)

- To include devtools support, simply add the module dependency to your build

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <optional>true</optional>
   </dependency>
</dependencies>
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing Services with Postman

- Postman is a tool that can be used to test REST services
  - Enables messages to be configured
  - Service responses to be viewed
  - Allows to create ‘collections’ of requests, similar to ‘SOAP UI’ test suites
    - Use this to ‘replay’ messages during service development
- Your instructor will now demonstrate Postman
- RestAssured is a library for writing tests for REST services
  - Provides DSL that supports *given-when-then* structure
  - Details found at `rest-assured.io`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Details

- Full details of Spring Boot can be found at:
  - http://docs.spring.io/spring-boot/docs/current/reference/
- Skeleton Spring Boot project generator:
  - http://start.spring.io/

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Web Services

HTTP and JSON

RESTful Services

Spring Boot

➤ **Exercise**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 3.2

Hands-On Exercise

- In Eclipse, go to the project named `spring-boot-rest`
- Write a service that is accessed by the URL `/currencies` using a HTTP `GET`
  - The service should return a `List` of all the currencies available
    - The currencies available are defined in the enumeration `Currency`
      - `Currency` is found in the package `algo.trader.domain`
- Write a class named Application which will start your code as a Spring Boot application
  - It should have a main method
  - Run the application and verify that the currencies service works as expected
    - Start the application on port 9090
    - Use Postman as the client to test the service
- Add a new service that will receive a `MarketOrder` JSON object at the URL `/order`
  - Test the service using Postman

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 3.2 (continued)

- Modify your service that returns the currencies to use content negotiation
  - The service should return JSON (default) or XML, based on the accept HTTP header present in the request
- Enable Spring Boot Actuator on your project and examine the following endpoints:
  - beans
  - configprops
  - Metrics
- Bonus
  - Implement other actions for the order (cancel placed order, update existing order, list all placed orders)

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have:

- Introduced REST Web Services
- Introduced Spring Boot
- Learned how to write REST Web services using Spring Boot

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features:
Unit Testing, Design Patterns, and RESTful Services

Chapter 4: Securing REST Endpoints with Spring Security

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will:

- Introduce Spring Security
- Secure REST endpoints using Spring Security

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

➤ **Spring Boot Security**

Exercise

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Security

- Spring Security provides a rich set of security features
  - HTTP BASIC authentication headers
  - HTTP Digest authentication headers
  - HTTP X.509 client certificate exchange
  - LDAP
  - Form-based authentication
  - OpenID authentication
  - Much more …
    - https://projects.spring.io/spring-security/
- If Spring Security is found on the class path, then HTTP Basic authentication will be applied to all endpoints

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Setting Up Spring Security

- Spring Security must be added to the build file

```
…
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
…
```

POJO with properties matching JSON property names

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Configuring Spring Security

- Configuring Spring Security for authentication requires two steps:
  1. Defining an authentication manager
     - Defines where username, password, and role information is stored
       - In memory
       - JDBC
       - LDAP
  2. Configure which URLs are restricted
- Configuration of both steps is possible in Java

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Step 1: Defining an Authentication Manager

- Define a method that configures `AuthenticationManagerBuilder`
- For memory manager define username, password, and role(s)

Provides configuration for Spring Security

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
                                                    throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("john").password("smith").roles("USER")
                .and()
            .withUser("admin").password("admin").roles("USER", "ADMIN");
    }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Authentication Manager

- When an authentication manager is configured, the following process occurs:

1. On receipt of a username and password, Spring Security verifies that they are valid
2. If valid, then the list of roles for that user is obtained and a security context created
3. Security context is used to determine if user has permission – correct role – to access endpoint
   - If no roles configured, a valid username and password combination are enough to access the endpoint

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Step 2: Configure URL Access

- Spring's `HttpSecurity` class provides a DSL for configuring URL access
  - Allows URL patterns defined using Ant style pattern matchers

```
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
     http
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
           .and()
        .authorizeRequests()
           .antMatchers("/audit/**").hasRole("ADMIN")
           .antMatchers("/testobjects/**").hasAnyRole("USER", "ADMIN")
           .antMatchers("/echo").authenticated()
           .anyRequest().permitAll()
           .and()
        .httpBasic()
           .and()
        .csrf().disable()
        .headers().frameOptions().disable();
}
```

Don't create session

Access for specific role

Any authenticated user

Non-authenticated access

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring and OAuth 2.0

- Spring security provides support for OAuth 2.0
- Example provided at https://spring.io/guides/tutorials/spring-boot-oauth2/

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

Spring Boot Security

➤ **Exercise**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 4.1: Securing REST Services with Spring Security

HANDS-ON EXERCISE

- In Eclipse, go to the project named `spring-boot-rest`
- Add Spring Security to the project
- Run the application and access the service `/currencies` using Postman
  - Can you access it?
- Configure Spring Security so that the service accessed by the URL `/currencies` using a HTTP `GET` is accessible to all users
- Use Postman to verify that the service can now be accessed
  - Also verify that the service `/order` with a HTTP `GET` cannot be accessed
- Configure Spring Security to use an in memory authentication manager to provide a valid user account
- Use Postman to verify that the service `/order` with a HTTP `GET` can now be accessed

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have:

- Introduced Spring Security
- Secured REST endpoints using Spring Security

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Chapter 5: Effective Unit Testing

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will:

- Consider what makes good tests
- Review basic Java unit testing and Spring Framework integration testing
- Introduce frameworks for end-to-end testing of RESTful services

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

➤ **What Makes Good Tests**

Basic Java Unit Testing and Spring Framework Integration Testing

End-to-End Testing of RESTful Services

Supplementary Testing Frameworks

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Different Views of Testing

- Acceptance testing verifies implementation of user stories
  - Indicate whether software is broken
  - Help build the **right code**
    - What the customer wants
- Unit and integration testing verifies individual components
  - Indicate where software is broken
  - Help build the **code right**
    - From a developer perspective
      - Maintainable, extensible

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Overall Aim of Testing Code

- Test-driven development is often aiming for test-first code
  - Real aim is **Self-testing Code**
    - We should always be able to test the code
- Not always possible to write tests for code before writing the code
  - In these scenarios, it’s OK to write the tests after
- Most important thing is to test EARLY
  - As soon as it is practical to do so

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Good Unit Tests

- Are **fully automated**; i.e., write code to test code
- Offer good coverage of the code under test, **including discontinuities and error-handling paths**
- Express the intent of the code under test – they do more than just check it
  - Behavioral specifications, where functionality is illustrated by example
- Should focus on and express the interface contract, not the private implementation
- Should aim to demonstrate and test **one** behavior only
  - The behavior should be clear from the test name
    - E.g., 'depositPositiveAmount', 'overdrawAccountWithZeroInitialBalance'
    - **not** 'test1', 'test2' or simply 'testDeposit' or 'testWithdraw'
  - The behavior should be reflected in the assertions in the test
  - But there may be more than one actual assertion within the test

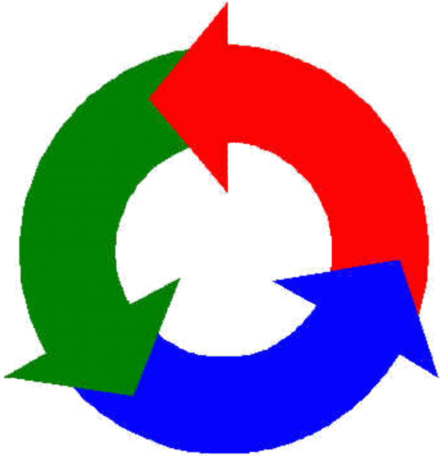
© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Problematic Tests

- Problematic test styles include:
  - *Monolithic tests*: all test cases in a single method; e.g., *Test* or *Main*
  - *Ad hoc tests*: test cases arbitrarily scattered across test functions; e.g., *Test1*, *Test2*, ...
  - *Procedural tests*: test cases bundled into a test method that correspond to target method; e.g., *testFoo* tests *foo*

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

TDD Cycle



Red

Write a failing test for a new feature

Green

Write enough code to pass the test

Refactor

Simplify, consolidate, and generalize the code

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

What Tests to Write

- There are a number of things that should appear in tests
  - Simple cases, because you have to start somewhere
  - Common cases, using equivalence partitioning to identify representatives
  - Boundary cases, testing at and around
  - Contractual error cases; i.e., test rainy-day as well as happy-day scenarios
- There are a number of things that should not appear in tests
  - Do not assert on behaviors that are standard for the platform or language—the tests should be on your code
  - Do not assert on implementation specifics
    - A comparison may return 1 but test for > 0
      - Or incidental presentation—spacing, spelling, etc.

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

What Makes Good Tests

➤ **Basic Java Unit Testing and Spring Framework Integration Testing**

End-to-End Testing of RESTful Services

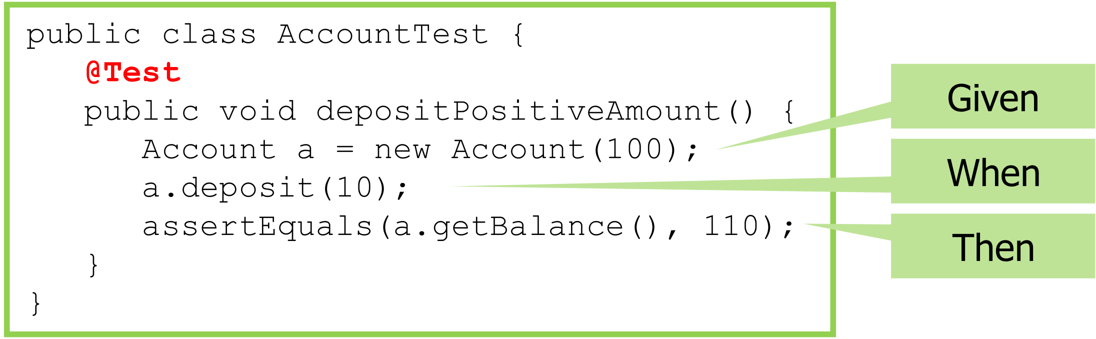
Supplementary Testing Frameworks

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Structure of a Test

- A test case should have linear flow: arrange, act, assert
  - ***Given***: declare and set up data
  - ***When***: perform the action to be tested
  - ***Then***: assert desired outcome

- Note that these sections may vary depending on what is being tested
  - But tests should rarely lack assertions
    - Assert-less tests are likely to be very brittle!



```
public class AccountTest {
    @Test
    public void depositPositiveAmount() {
        Account a = new Account(100);
        a.deposit(10);
        assertEquals(a.getBalance(), 110);
    }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Testing for Exceptions

- **Do not** catch exceptions in your test code when testing for ‘happy path’ scenarios
  - If exception is thrown, JUnit will **fail** the test, which is desired behavior
- **Do** test for ‘expected’ exceptions when testing for ‘alternative flows’:
  - If expected exception is thrown, JUnit will **pass** the test
  - JUnit will **fail** test if exception does **not** occur

```
public class CrmTest {
   @Test(expected = ClientDuplicateException.class)
   public void addDuplicateClient() {
      crm.addClient(JOHN_SMITH);
      crm.addClient(KATE_SMITH);
      crm.addClient(JOHN_DOE);
      crm.addClient(JOHN_DOE);
   }
}
```

Trying to add client that already exists

```
public class CRM {
  …
  public void addClient(Client client) {
    if (contains(client))
      throw new ClientDuplicateException(
        "Client ["+client+"] already exists");
    clients.add(client);
  }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Integration Tests

- Unit tests can be used to test Spring beans – spring context will be initialized by the testing framework
  - NB: make sure ‘classes’ attribute is referring to all classes needed to construct context OR get Spring to scan for ALL beans with the package hierarchy (see next slide)

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {Bank.class, …})
public class BankTest {
   @Autowired
   private Bank bank;

   @Test
   public void validateCreatedAccounts() {
      bank.openAccount(100);
      assertEquals(1, bank.getTotalNumberOfAccounts());
   }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Integration Tests – Other Useful Annotations

- To initialize Spring context with ALL beans found in (sub)packages of BankApplication.class, while resetting context after each test method invocation

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_EACH_TEST_METHOD)
@ActiveProfiles("test")
public class BankTest {
   @Autowired
   private Bank bank;

   @Test
   public void validateCreatedAccounts() {
      bank.openAccount(100);
      assertEquals(1, bank.getTotalNumberOfAccounts());
   }

   @Configuration
   @ComponentScan(basePackageClasses = BankApplication.class)
   public static class TestConfig {}
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

What Makes Good Tests

Basic Java Unit Testing and Spring Framework Integration Testing

➤ **End-to-End Testing of RESTful Services**

Supplementary Testing Frameworks

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Test

- Spring Boot Test provides simple, yet powerful support for testing
- The tests need to be run with the SpringRunner test runner
- The `@SpringBootTest` annotation provides all Spring Boot Support
  - Runs in web container if web components are included
  - Initializes all beans in the hierarchy
  - Injects dependencies into tests
- For more information on integration testing:
  - https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html
  - https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Integration Test

- @SpringBootTest attributes:
  - webEnvironment: NONE, DEFINED_PORT, RANDOM_PORT, MOCK
    - Whether to startup web container or not; default is MOCK
  - classes: normally points to the class annotated with @SpringBootApplication
    - If not defined, Spring Boot will ‘walk up’ the package hierarchy till first @SpringBootApplication bean

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = BankApplication.class)
public class BankTest {
   @Autowired
   private Bank bank;

   @Test
   public void validateCreatedAccounts() {
      bank.openAccount(100);
      assertEquals(1, bank.getTotalNumberOfAccounts());
   }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Testing REST Services

- Rest Assured is a DSL for testing rest services
- Write tests in a *Given-When-Then* structure
- Can run Rest Assured tests against Spring Boot REST Service
  - Allow full end-to-end tests to be executed
- Need to set port Spring Boot Web container starts on
  - Tell Rest Assured which port to connect to for running tests
    - Port defined in `application.properties`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Spring Boot Rest Test Initialization

Port from `application.properties`

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.DEFINED_PORT,
                                           classes = { Application.class})
public class CurrencyServiceTest {
  @Value("${local.server.port}")
  private int serverPort;

  @Before
  public void init(){
    RestAssured.port = serverPort;
  }
  …
```

Inject port to use to configure Rest Assured

Tell Rest Assured about port

```
server.port=9090
```

`application.properties`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Rest Assured Testing

- The test below reads: Given I accept JSON, When I access endpoint /currencies, I expect the HTTP status code `200 OK`

```
@Test
public void allCurrenciesReturned(){
        given().
          accept(MediaType.APPLICATION_JSON_VALUE).
        when().
          get("/currencies").
        then().
          statusCode(HttpStatus.SC_OK);
}
```

Access using `HTTP GET`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Accessing the JSON Body

- `/currencies` returns a JSON array: [“`USD`”, “`GBP`”, “`EUR`”, “`JPY`”]
- Response object provides access to response received

```
@Test
public void allCurrenciesReturned(){
  Response response =
        given()
          .accept(MediaType.APPLICATION_JSON_VALUE).
        when()
          .get("/currencies").
        then()
          .statusCode(HttpStatus.SC_OK).
        and()
          .extract().response();
  Currency[] jsonResponse = response.as(Currency[].class);
  assertArrayEquals(jsonResponse, Currency.values());
}
```

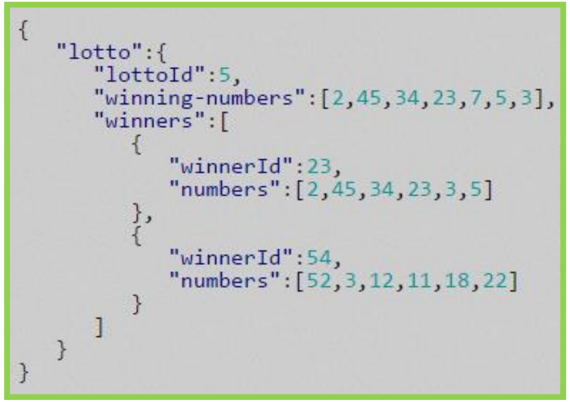
Extract response body as array of Currency enumerations

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Accessing the JSON Body (continued)

- JsonPath/XMLPath to evaluate response

```
@Test
public void lotto_returns_200_with_expected_id_and_winners() {
    when().
        get("/lotto/{id}", 5).
    then().
        statusCode(200).
        body("lotto.lottoId", equalTo(5),
             "lotto.winners.winnerId", containsOnly(23, 54)); }
```



```
{
   "lotto":{
      "lottoId":5,
      "winning-numbers":[2,45,34,23,7,5,3],
      "winners":[
         {
            "winnerId":23,
            "numbers":[2,45,34,23,3,5]
         },
         {
            "winnerId":54,
            "numbers":[52,3,12,11,18,22]
         }
      ]
   }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

More Rest Assured

- Rest Assured allows access to the JSON or XML body of a service
  - Uses JSONPath and XMLPath
- Full details can be found at http://rest-assured.io/

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Concepts

What Makes Good Tests

Basic Java Unit Testing and Spring Framework Integration Testing

End-to-End Testing of RESTful Services

➤ **Supplementary Testing Frameworks**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Unit Testing Support

- There are numerous other libraries used instead of or together with JUnit:
  - Assertion frameworks
    - Hamcrest
    - AssertJ
    - Truth
  - Mocking frameworks
    - Mockito
    - Easymock
    - Jmock
  - Alternative frameworks
    - TestNG
    - Spock
  - Miscellaneous
    - JUnit rules
    - JUnitParams
  - Testing mulithreaded applications
    - Awaitility
    - ConcurrentUnit

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Hamcrest Matchers

- Hamcrest is a framework for writing declarative match criteria
- Used by several testing frameworks
  - JUnit, jMock, Mockito, etc.
- In JUnit used in conjunction with `Assert`'s `assertThat`
- Hamcrest provides a built in set of matchers
  - Matchers can be combined

```
String s = "What is there not to like about TDD ? ";

assertThat(s, containsString("TDD"));
assertThat(s, not(containsString("not")));
```

Hamcrest matcher

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Built-in Matchers

- Hamcrest `Matchers` class has a number of built-in matchers
  - `containsString(String substring)`
  - `endsWith(String substring)`
  - `equalTo(T operand)`
  - `greaterThan(T value)`
  - `lessThan(T value)`
  - `is(T Value)    - a wrapper for readability`
  - `instanceof(T class)`
  - `isIn(T[] param)`
  - `isOneOf(T... elements)`
  - `allOf(Matcher ... matchers)`
  - Many more

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Built-in Matchers Example

```
@Test
public void addingFourPlusFive() throws Exception {
   assertThat((4+5), is(equalTo(9)));
}


@Test
public void allMatchersMustBeTrue() throws Exception {
    assertThat("Hello", is( allof( notNullValue(),
                            instanceof(String.class),
                            equalTo("Hello") )));
}
```

Matcher

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

AssertJ

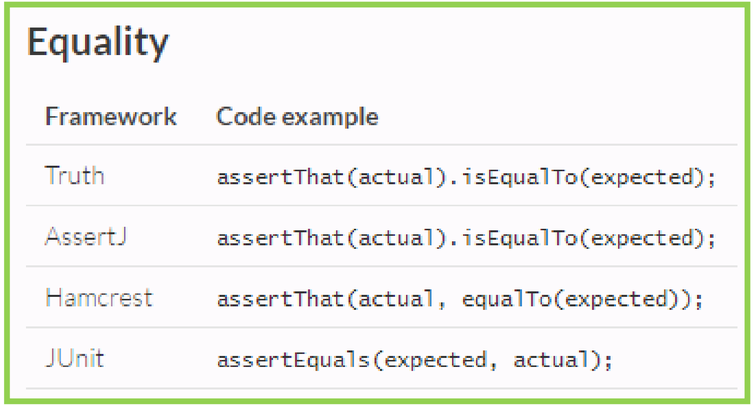
- Similar to Hamcrest, provides a fluent interface for writing assertions
  - Improves test code readability

```
// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");
```

- For more information:
  - Example of AssertJ can be found in the project `demos`
    - Package `com.demo.assertj`

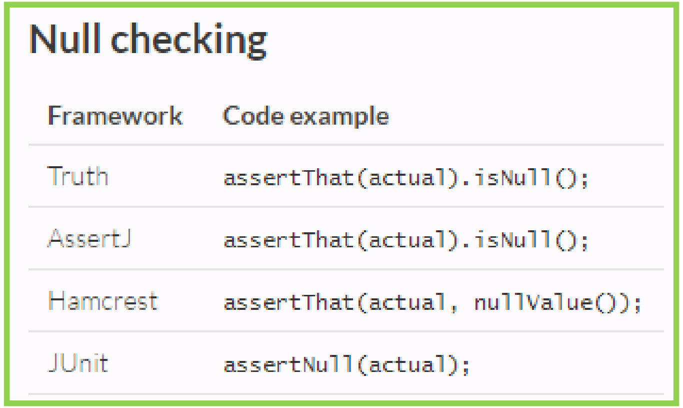
© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Assertion Frameworks Comparison



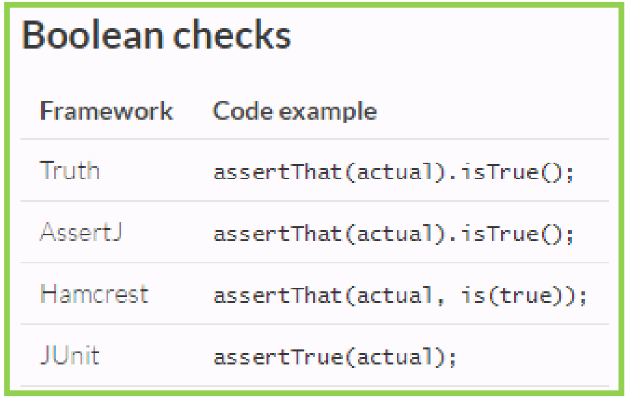
Equality

Framework	Code example
Truth	`assertThat(actual).isEqualTo(expected);`
AssertJ	`assertThat(actual).isEqualTo(expected);`
Hamcrest	`assertThat(actual, equalTo(expected));`
JUnit	`assertEquals(expected, actual);`



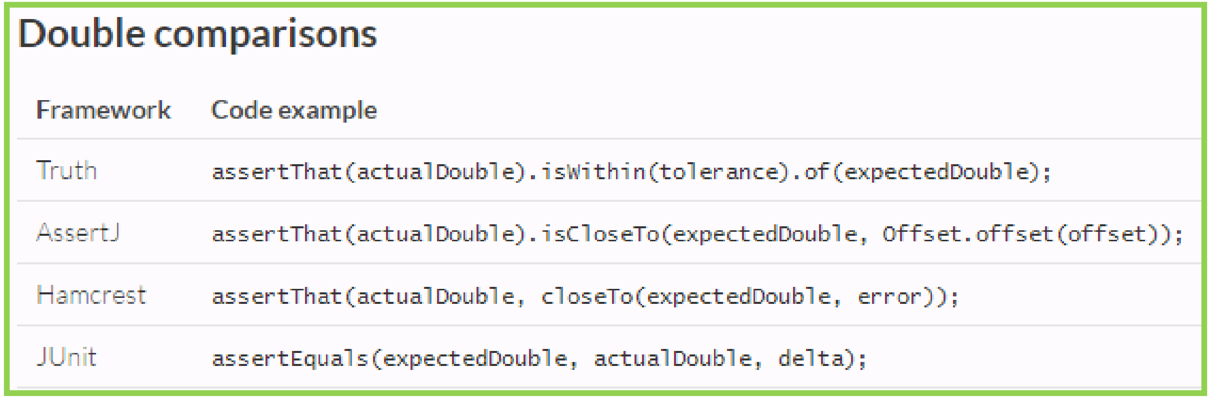
Null checking

Framework	Code example
Truth	`assertThat(actual).isNull();`
AssertJ	`assertThat(actual).isNull();`
Hamcrest	`assertThat(actual, nullValue());`
JUnit	`assertNull(actual);`



Boolean checks

Framework	Code example
Truth	`assertThat(actual).isTrue();`
AssertJ	`assertThat(actual).isTrue();`
Hamcrest	`assertThat(actual, is(true));`
JUnit	`assertTrue(actual);`



Double comparisons

Framework	Code example
Truth	`assertThat(actualDouble).isWithin(tolerance).of(expectedDouble);`
AssertJ	`assertThat(actualDouble).isCloseTo(expectedDouble, Offset.offset(offset));`
Hamcrest	`assertThat(actualDouble, closeTo(expectedDouble, error));`
JUnit	`assertEquals(expectedDouble, actualDouble, delta);`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

JUnit Rules

- Rules allow the ‘reuse’ of setup/teardown code across unit tests
- Some built-in rules are provided
  - `TempFolder`
    - Will create a temporary folder before each test
    - Remove the folder after each test
  - `ExternalResource`
    - Sets up external resources before a test
    - Guarantees to tear it down after each test
- For more information:
  - Details can be found at https://github.com/junit-team/junit/wiki/Rules
  - Example of rules can be found in the project `demos`
    - Package `com.demo.rules`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom JUnit Rules

```
public class SystemOutRule implements TestRule {
   public Statement apply(final Statement base, final Description desc) {
      return new Statement() {
         @Override
         public void evaluate() throws Throwable {
            System.out.println("Before " + desc.getClassName()+"."+ desc.getMethodName());
            base.evaluate();
            System.out.println("After " + desc.getClassName()+"."+desc.getMethodName());
         }
      };
   }
}
```

```
public class SystemOutRuleTest {
   @Rule
   public SystemOutRule systemOutRule = new SystemOutRule();
   @Test
   public void exampleToShowSystemOutRule() {
      System.out.printf("In example Test to show rules %n");
   }
}
```

OUTPUT:

```
Before SystemOutRuleTest…
In example Test to show…
After SystemoutRuleTest…
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

JUnit Params

- Provides a way to run parameterized tests
  - At the method level
    - Not like class level of JUnit parameterized test runner

```
@Test
@Parameters({"17, false",
             "22, true" })
public void personIsAdult(int age, boolean valid) throws Exception {
   assertThat(new Person(age).isAdult(), is(valid));
}
```

- For more information:
  - Details can be found at https://github.com/Pragmatists/junitparams
  - Example of Junit params can be found in the project `demos`
    - Package `com.demo.params`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Write Tests with Benefits

- If you are to write tests, they must provide value
  - They are expensive to write and maintain!
- Unit and integration tests are for you, the developer
  - They must provide *value* to your team
    - Confidence code works as you intend
    - Confidence that code changes have not broken functionality
  - So, always question what value does the test provide
- Do not write tests simply because you think there should be a test
  - Without a good reason for that test
- Do not mock excessively
  - Usually happens because you are using the version 1 definition of a unit
  - Mock only public APIs that are tested elsewhere

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Write Real Tests

- Fact is most people write too many tests
- Test behavior not implementation
  - More effective tests
  - Less test code
- A behavior can span many classes at implementation level
  - Tests should be unaware of this
  - Can talk to the database and other resources
- Stop using the word Unit on your tests!
  - As long as a test has no dependency on other tests, it’s OK
- Consider these tests as DEVELOPER tests
  - Allow you to confidently stand over your code

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Use Code Coverage Wisely

- Testing is never finished
  - We merely choose to stop at some point
- Coverage is often used as the stopping point
  - It is not a good measure for this purpose
- Test coverage is a negative metric
  - Indicates what is missing or not covered
    - Not how good it is
- A low test coverage is useful information to developers
  - Testing has not been performed on parts of the product
- A high test coverage is much less useful
  - Says nothing about the type of testing
    - Or the outcome of the testing

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have:

- Consider what makes good tests
- Review basic Java unit testing and Spring Framework integration testing
- Introduce frameworks for end-to-end testing of RESTful services

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Appendix

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Asynchronous Tests

- Awaitility provides support for asynchronous tests
- Provides a DSL enabling concise tests to be written

```
@Test
public void testTwo(){
   messageService.publish(new Node());
   messageService.publish(new Node());
   messageService.publish(new Node());

   await().untilCall( to(nodeRepository).size(), equalTo(3) );
}
```

Add element to `nodeRepository` in a separate thread

Block main thread till condition is met

- For more information:
  - Details can be found at https://github.com/jayway/awaitility
  - Example of asynchronous tests can be found in the project `demos`
    - Package `com.demo.async`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Concurrent Tests

- ConcurrentUnit is a toolkit for testing multithreaded code

```
@Test
public void shouldDeliverMessages() throws Throwable {
  final Waiter waiter = new Waiter();

  messageBus.registerHandler(message -> {
    waiter.assertEquals(message, "foo");
    waiter.resume();
  };

  messageBus.send("foo");
  messageBus.send("foo");
  messageBus.send("foo");

  // Wait for resume() to be called 3 times
  waiter.await(1000, 3);
}
```

Check messages from a separate thread

Send message from main thread

Block main thread till condition is met

- For more information:
  - Details can be found at https://github.com/jhalterman/concurrentunit
  - Example of concurrent tests can be found in the project `demos`
    - Package `com.demo.concurrent`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Writing Your Own Matchers

- Possible to define your own matchers

- Developing a custom matcher requires:
  1. Developing custom matcher class
     - Extend `TypeSafeMatcher`
  2. Provide static factory method
     - Creates instance of `TypeSafeMatcher`

- Use custom matcher in tests as you would use built-in matchers

- Extend `org.hamcrest.TypeSafeMatcher` and override
  - `boolean matchesSafely(T item)`
  - `void describeTo(Description)`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Writing Your Own Matchers (continued)

- Optional: to customize failure messages, override
  - `void describeMismatchSafely(T item, Description description)`
  - Common message pattern:
    - "Expected: ${describeTo}, but ${describeMismatchSafely}"
- Usually provide static factory method to simplify matcher use

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom Matchers Example Usage

- Example will develop a matcher named `endsWith`
  - Determines if String ends with user-supplied argument

```
import static com.company.MyMatchers.endsWith;

class SomeTests{
  @Test
  public void testNameEndsWith(){
    assertThat("Athlone", endsWith("ne"));
  }
}
```

User-defined factory class

Custom matcher

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom Matcher Factory Class

- Code shows example factory class
  - Creates matcher objects in static methods
    - Enables symbolic names to be defined for custom matchers

```
package com.company;
import org.hamcrest.Factory;

public class MyMatchers {
  private MyMatchers() {}

  @Factory
  public static EndsWithMatcher endsWith(String end) {
      return new EndsWithMatcher(end);
  }
}
```

Prevent instances being created

Return instance of custom matcher

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom Matcher Class

```
import org.hamcrest.Description;
import org.hamcrest.TypeSafeMatcher;

public class EndsWithMatcher extends TypeSafeMatcher<String>{
  private String end;

  public EndsWithMatcher(String end) { this.end = end; }

  public boolean matchesSafely(String value) {
     if (value.endsWith(end))  return true;
     return false;
  }

  public void describeTo(Description description){
     description.appendText("string ending with " + end);
  }

  protected void describeMismatchSafely(String value, Description description) {
     description.appendText("found ‘" + value + "’ string instead");
  }
}
```

What is expected

What was actually found

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Custom Matcher Operation

- Consider the `assertThat` method

T or super classes of T

```
assertThat(T actual, Matcher<? super T> matcher);
```

- And its use with the developed custom matcher

```
assertThat("Athlone", endsWith("ne"));
```

Returns custom matcher object, "`ne`" passed to constructor

- Steps performed by `assertThat` are:
  1. Receives a `String` and a matcher
  2. Invokes `matchesSafely` passing `String` as parameter
     - In this example, passes "Athlone"
  3. If `matchesSafely` returns true, test passes, else fails

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 5.1

GROUP EXERCISE

- Write a custom Hamcrest matcher that will check that a String begins and ends with the same character
- Your test will be of the form:

```
assertThat("MyStringM",startandEndCharacterIs("M"));
```

- **Bonus:** Write a custom matcher that will check that all items in an array of integers meets a user-supplied matcher condition

```
Integer [] data = {0,2,4};
assertThat(data, eachAndEveryIntegerIs(greaterThan(0)));
```

Built-in matcher

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareer TEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Chapter 6: Design Patterns

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will:

- Introduce Design Patterns for writing maintainable code

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

What Is a Design Pattern?

- Defined by Christopher Alexander (1977)

> “Each pattern **describes a problem** which occurs **over and over** again in our environment, and then describes the **core of the solution** to that problem, in such a way that you can **use the solution a million times over**, without ever doing it the **same way twice**.”

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Principles

- The following two design principles will help you design/write better software
  1. Identify those aspects of the system that vary and separate them from what stays the same
  2. Program to an interface(s), not an implementation

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Design Pattern Catalog

- Original software design patterns were defined in the text:
  - *Design Patterns, Elements of Reusable Object-Oriented Software*
    - Gamma, Helm, Johnson, and Vlissides
- Defined a catalogue of 23 patterns
  - Classified as:
    - Creational
    - Structural
    - Behavioral
- We will discuss a few to provide examples

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

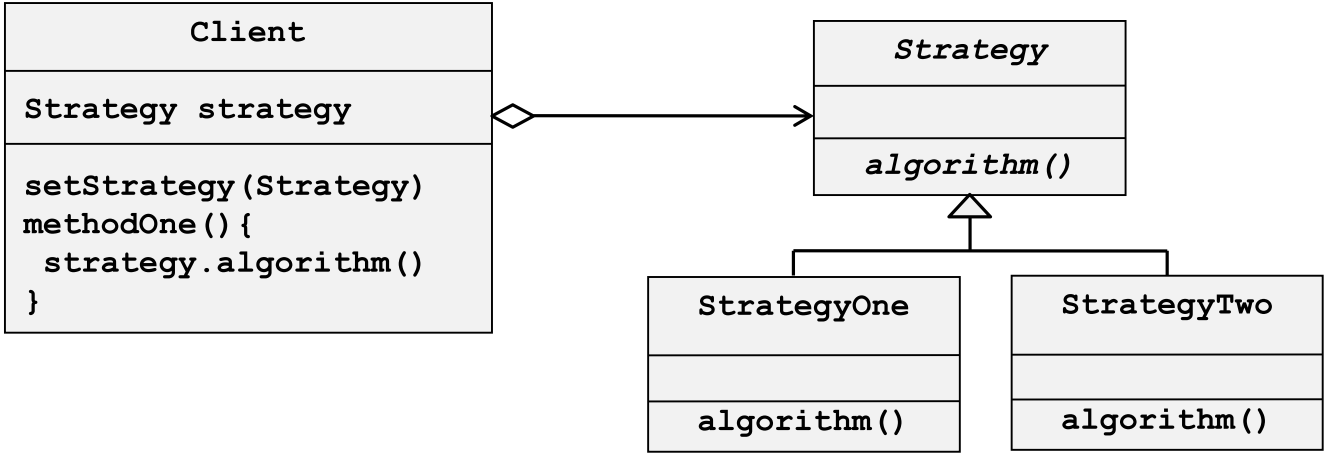
Benefits of Design Patterns

- Using design patterns has many benefits
  - Starting point towards a solution
  - Shared vocabulary amongst development team
  - Flexible solutions
- **IMPORTANT!**
  - Do not use patterns for their own sake—use with purpose!

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

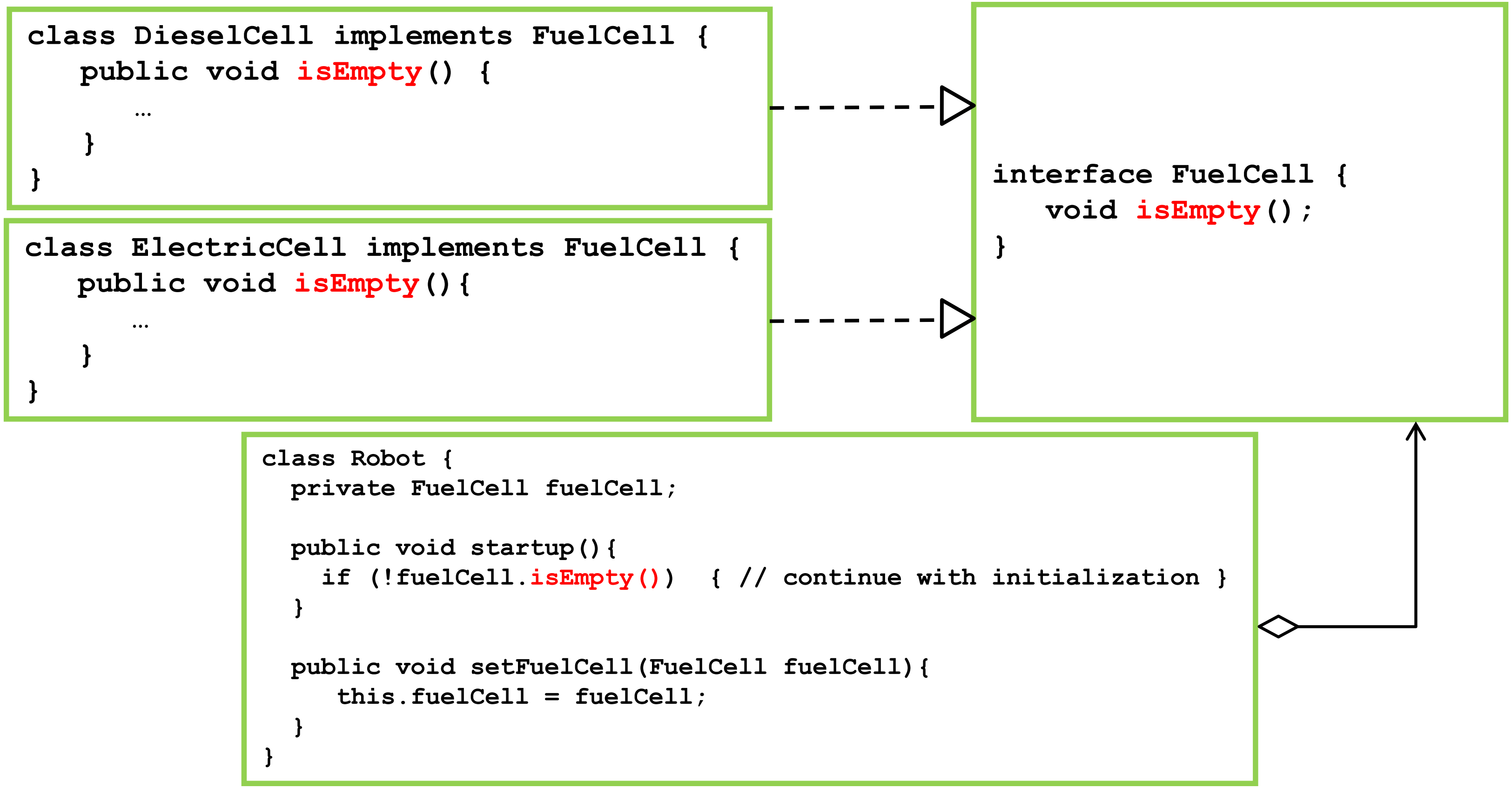
Strategy Design Pattern

- Problem description:
  - Define a family of algorithms, encapsulate each one, make them interchangeable
  - Strategy lets the algorithm vary independently from clients that use it
  - Very common pattern in Spring Framework



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Strategy Example



```
class DieselCell implements FuelCell {
   public void isEmpty()  {
      …
   }
}
```

```
class ElectricCell implements FuelCell {
   public void isEmpty(){
      …
   }
}
```

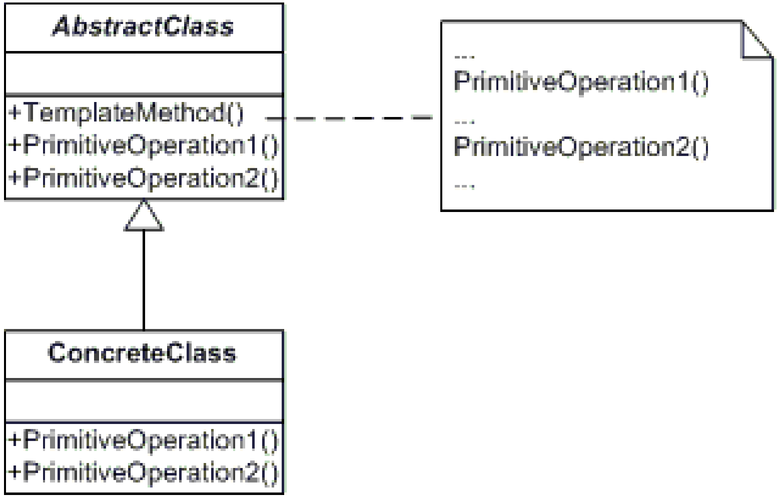
```
interface FuelCell {
   void isEmpty();
}
```

```
class Robot {
  private FuelCell fuelCell;

  public void startup(){
    if (!fuelCell.isEmpty())  { // continue with initialization }
  }

  public void setFuelCell(FuelCell fuelCell){
     this.fuelCell = fuelCell;
  }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

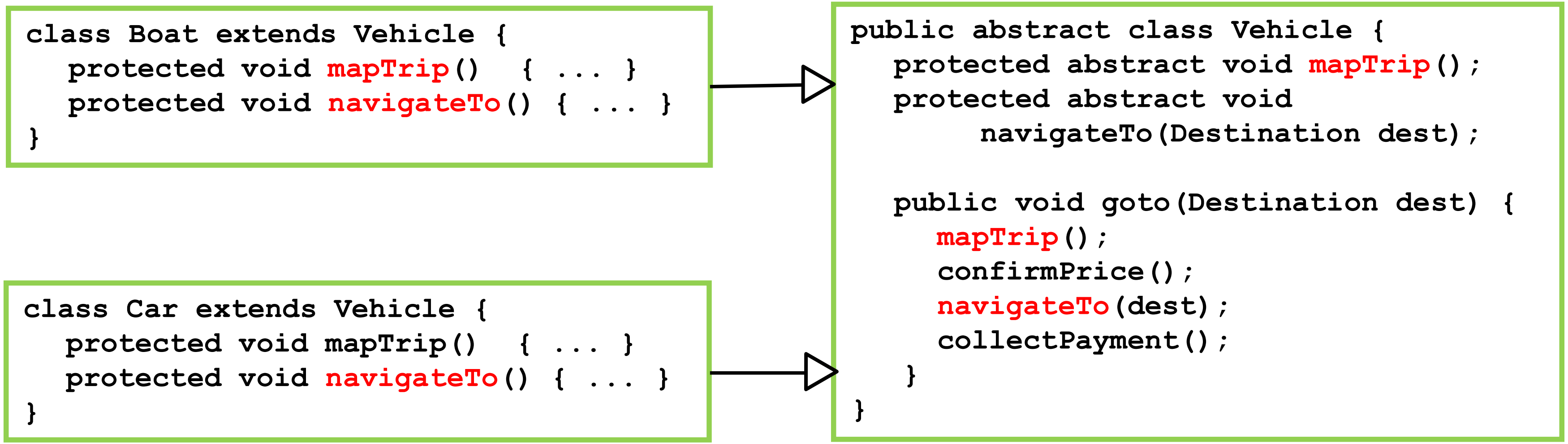
Template Method Design Pattern

- Problem description:
  - Common algorithm flow (template method) with customizable steps (operations)



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Template Method Design Pattern



```
class Boat extends Vehicle {
   protected void mapTrip()   { ... }
   protected void navigateTo() { ... }
}
```

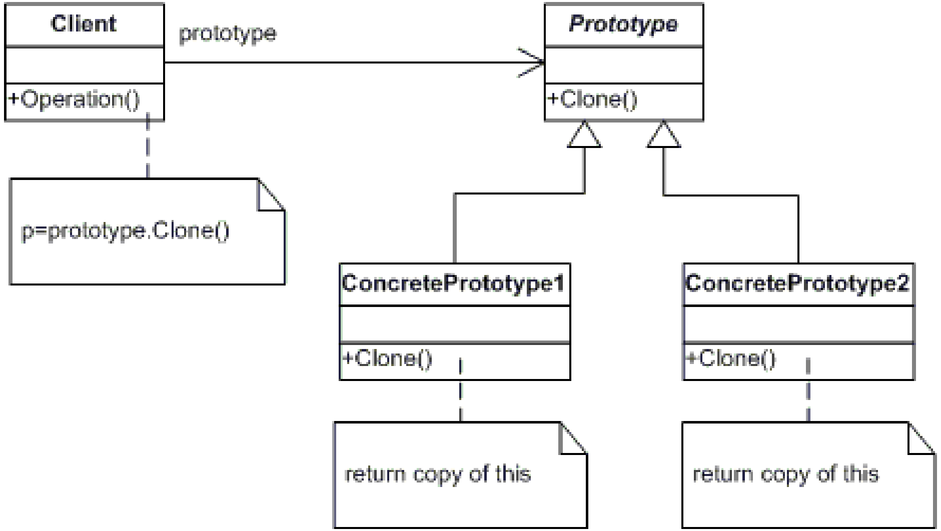
```
class Car extends Vehicle {
   protected void mapTrip()   { ... }
   protected void navigateTo() { ... }
}
```

```
public abstract class Vehicle {
   protected abstract void mapTrip();
   protected abstract void
         navigateTo(Destination dest);

   public void goto(Destination dest) {
      mapTrip();
      confirmPrice();
      navigateTo(dest);
      collectPayment();
    }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Clone Design Pattern

- Problem description:
  - Create a replica of an object which would evolve and manage independently from the prototype
  - Built-in pattern in Java SDK since v1.0



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Clone Design Pattern

```
class Car implements Cloneable {
   private CarModel  model;
   private String    licPlate;
   private Color     color;
   private CarMake   make;
   private boolean   isClone = false;
   ...
   public Car clone() throws CloneNotSupportedException {
      Car car = (Car) super.clone();
      car.isClone = true;
   }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

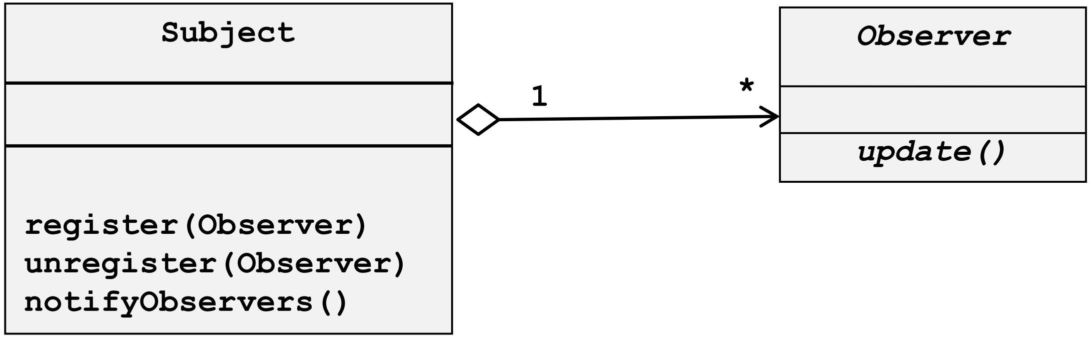
Observer Pattern

- Problem description:
  - There is a one-to-many relationship between objects
  - When the state of one object is updated, dependent objects should be informed
  - The dependent objects may change over time
- Solution:
  - An *Observer* registers interest with the *Subject* of interest
  - When state of *Subject* changes:
    - *Subject* notifies registered *Observer(s)*

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observer Pattern Class Diagram

- List of *Observers* changes over application lifetime
- *Observers* register and un-register themselves



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observer Example

```
abstract class ObservableRobot {
  private List<Observer> observers;

  public void register(Observer o){
    observers.add(o);
  }

  public void unRegister(Observer o){
    observers.remove(o);
  }

  protected void notifyObservers(){
    for(Observer observer : observers)
       observer.update();
  }
}
```

Abstract parent class

Observers register interest

Observers unregister interest

Called by child on change of state

```
interface Observer {
   void update();
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observer Example (continued)

```
class Robot extends ObservableRobot {
  public void move(int x, int y){
     // work to move robots
     notifyObservers();
  }
}
```

Notify observers on move

```
class RobotObserver implements Observer{

  public void update(){
     // robot changed
     …
  }
}
```

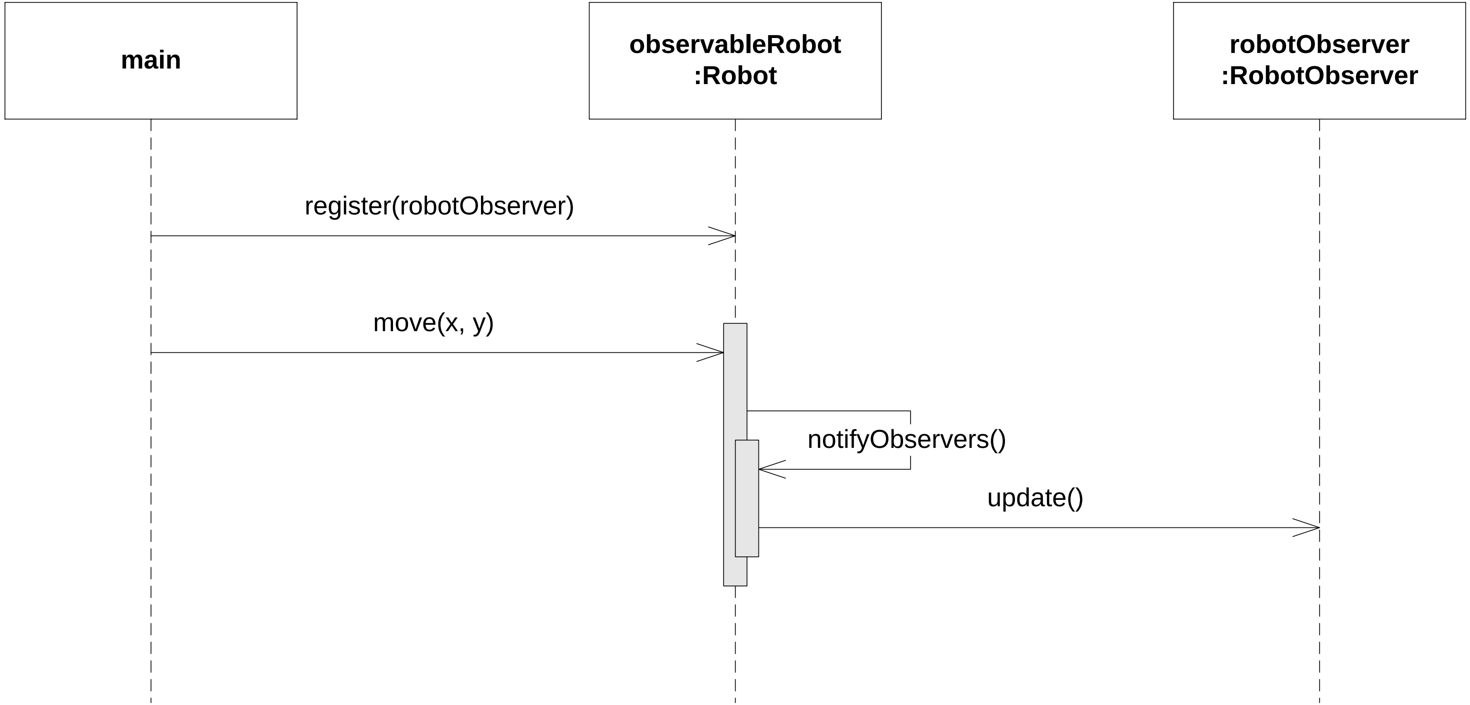
Receive updates from robot

```
Robot observableRobot = new Robot();
RobotObserver robotObserver = new RobotObserver();
observableRobot.register(robotObserver);
observableRobot.move(5, 10);
```

Register interest

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observer Sequence



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

`java.util.Observer` and `java.util.Observable`

- Java has built-in support for the observer design pattern
  - Observer interface
  - Observable abstract class

- `Observer` interface has one method
  - `update(Observable observable, Object o)`
    - First parameter is object reference that made call to update
      - So observer can call observable back for data
    - Second parameter allows observable to pass data to observer
      - Will be null if no data passed

- Observable class provides all methods for managing observers
  - And also for updating observers

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

java.util.Observerable Example

```
public class SimpleRobot extends Observable implements Robot {
    private String name;

    public SimpleRobot(String name){
        this.name = name;
    }

    public String getName(){ return name;   }

    @Override
    public void move(int x, int y) {
       System.out.format("Robot %s has moved x: %d y: "+ "
                              %d %n",name, x,y);
        RobotData robotData = new RobotData(x,y,name);
        setChanged();
        notifyObservers(robotData);
    }
}
```

Inherit observable implementation

Send notification to registered observers

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

java.util.Observer Example

```
public class RobotObserver implements Observer {
    private static int totalX, totalY;

    @Override
    public void update(Observable robot, Object data) {
        RobotData = (RobotData)data;
        totalX += robotData.getX();
        totalY += robotData.getY();
        System.out.printf("RobotObserver has "+"
                                   heard that: %s", robotData);
    }
    public static int getTotalX(){
        return totalX;
    }
    public static int getTotalY(){
        return totally;
    }
}
```

Observer keeps a running total of distance moved by robot it is observing

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Observer Example Usage

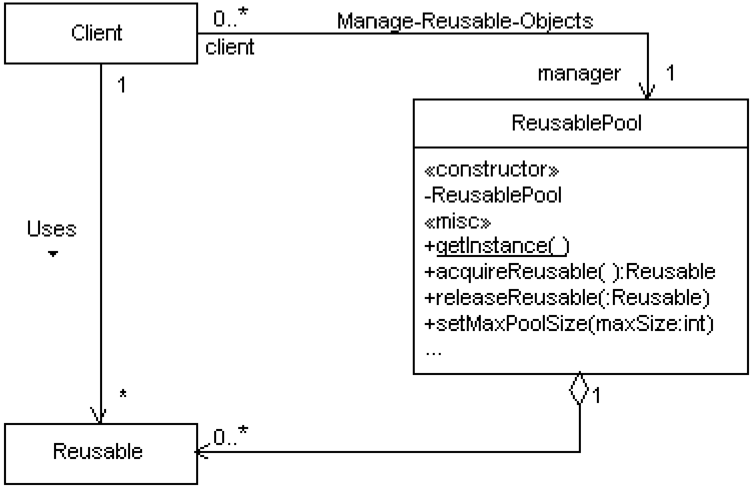
```
public static void main(string[] args){
    System.out.printf("The total moved is x: %d,  y: %d %n",
        RobotObserver.getTotalX(),RobotObserver.getTotalY());

    Robot robotOne = new SimpleRobot();
    RobotObserver = new RobotObserver();

    robotOne.addObserver(robotObserver);

    robotOne.move(10,20);
    robotOne.move(12,30);

    System.out.printf("The total moved is x: %d,  y: %d %n",
       RobotObserver.getTotalX(), robotObserver.getTotalY());
}
```

Register observer

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

ObjectPool Design Pattern

- Problem description:
  - reuse objects that are expensive to create



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

ObjectPool Design Pattern

```
public class ConnectionPool {
   private Connection[] connections = new Connection[10];
   private boolean[]    isLeased    = new boolean[10];

   private ConnectionPool() {
      initConnections(connections);
   }

   public Connection getConnection() {
      return leaseAvailableConnection(); // scan through ‘isLeased’ array and return
                                         // first available Connection
   }

   public void closeConnection(Connection conn) {
      closeLease(conn);                  // cleanup Connection object; reset isLeased flag
   }
}
```

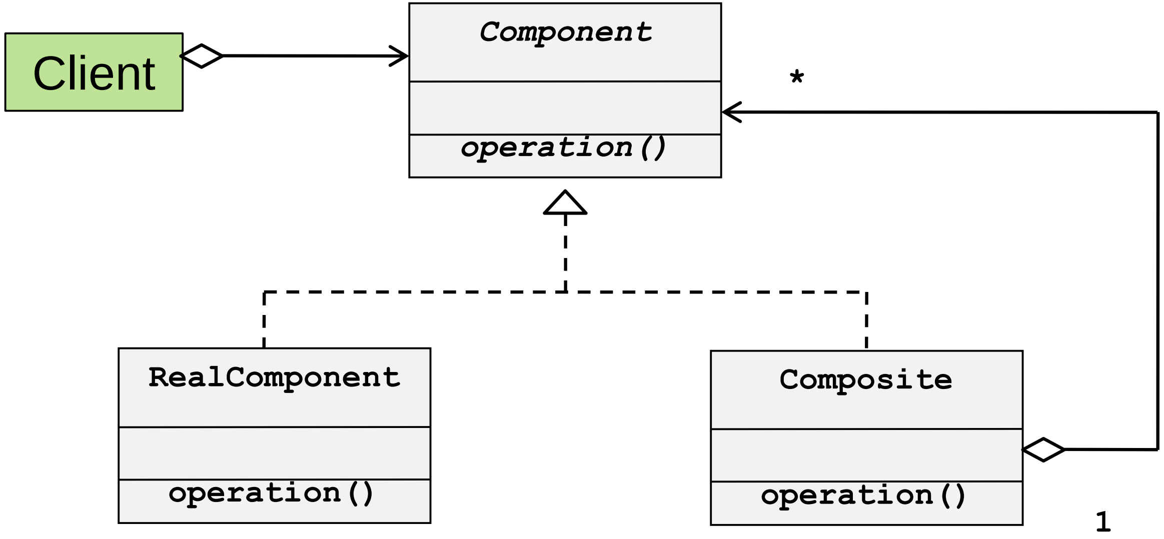
© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composite Pattern

- Problem description:
  - Require to handle single classes and groups of classes in a uniform way
  - Compose objects into tree structures to represent part-whole hierarchies
- Solution:
  - Composite lets clients treat individual objects and compositions of objects uniformly

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composite Pattern Class Diagram



© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Composite Pattern Code Example

```
class RobotGroup implements Robot {
  private List<Robot> children
          = new ArrayList<>();

  public void add(Robot robot){
    children.add(robot);
  }
  public void remove(Robot robot){
    children.remove(robot) ;
  }
  public void move(int x, int y){
    for(Robot robot : children)
       robot.move(x,y);
  }
}
```

```
interface Robot {
   public void move(int x, int y);
}
```

```
class MarsRover implements Robot {

  public void move(int x, int y){
    …
  }
}
```

```
Robot robotOne = new MarsRover();
Robot robotTwo = new MarsRover();
RobotGroup group = new RobotGroup();
group.add(robotOne);
group.add(robotTwo);
group.move(10,20);
```

Build composite from individual robots

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Singleton

- Class from which only one instance can be created

```
public class SingletonRobot implements Robot {
  private final static Robot instance = new SingletonRobot();

  private SingletonRobot(){   }

  public static Robot getInstance(){
    return instance;
  }

  public void move(int x, int y){
    …
  }
}
```

Single instance

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Factory Classes

- Create objects internally
  - Client requests objects without knowing what type is created
    - Receives a reference to an interface
  - Factory can be configured with which classes to create
    - Via external files or annotations

```
public class RobotFactory {

   public static Robot createRobot(){
       if(Math.random() > 0.5){
           return new MarsRover();
       } else{
           return singletonRobot.getInstance();
       }
   }
}
```

Client does not know implementation class

```
Robot robot = RobotFactory.createRobot();
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 6.1: Java Design Patterns

HANDS-ON EXERCISE

- Please turn to the Exercise Manual and complete Exercise 6.1: Java Design Patterns

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Annotations

- Java supports annotations

```
class SimpleRobot implements Robot {
    @Override
          public void move(int x, int y){
             …
    }

    @Deprecated
    public  accelerate(int toSpeed) {
      …
    }
}
```

- Tools can interrogate annotations for data
  - Perform tasks based on data
    - For example, create deployment descriptors
  - Issue compile time warnings

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

User-Defined Annotations

- Annotations are simple to create
- For example, write your own test framework
  - Define an annotation that can be used to mark unit test classes
  - Note that there are standard Java annotations helping to define this annotation

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyTest { }
```

Define new annotation `MyTest`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using User-Defined Annotations

- Once defined, simply refer to the annotation
  - Should precede other modifiers (e.g., private/public)

User-defined annotation

```
@MyTest
public class TestRobot {
   …

}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Annotations with Parameters

- Annotations can be parameterized
  - Any type of value can be used
  - Default values may be specified

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Log {
    String level() default "OFF";
}
```

Parameter for annotation

```
public class SimpleRobot implements Robot {

    @Log (level = "DEBUG")
    public void move(int x, int y) {

    }
}
```

Parameter set to DEBUG

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Accessing Annotation Parameters

- Annotation parameter values can be read using reflection

```
@Log (level = "DEBUG")
public void move(int x, int y) { … }
```

```
Method[] methods = Class.forName("SimpleRobot").getMethods();
for (Method m : methods) {
    if (m.isAnnotationPresent(Log.class)) {
        Annotation a = m.getAnnotation(Log.class);
        for (Method annotationMethod : a.getClass().getMethods()) {


          if ("level".equals(annotationMethod.getName())) {
          Object[] params = new Object[]{};
          String level = (String) annotationMethod.invoke(a, params);
          if ("DEBUG".equalsIgnoreCase(level)) {
            //detected that debug logging is enabled
          }
        }
    }
}
```

Load Class object for SimpleRobot

Check if annotation is on class

Get methods of annotation

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Annotations: A Context

- We have just introduced annotations
  - How you can define your own
  - How you can use reflection to determine:
    - If annotations are present at runtime
      - Gain access to annotation attributes at runtime
- Later in the chapter, we will introduce a practical example of using custom annotations
  - For profiling methods

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Flexible Software

- Using concrete class implementations in an application can cause tight coupling

```
class Client {
    private ServiceImpl service = new ServiceImpl();

    public void clientMethod() {
        service.operation();
    }
}
```

Client tightly coupled to service

- Client classes become dependent upon service implementations
  - Changing the service implementation becomes hard
  - Reconfiguring the application for different deployments required

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Program to Interfaces

- Referencing an interface reduces coupling to any concrete classes
  - Allows implementation to be initialized dynamically

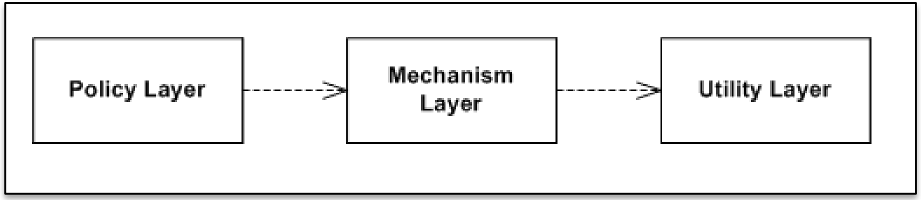
```
class Client {
    private Service service;

    public void clientMethod() {
       service.operation();
    }
}
```

Service is now an interface

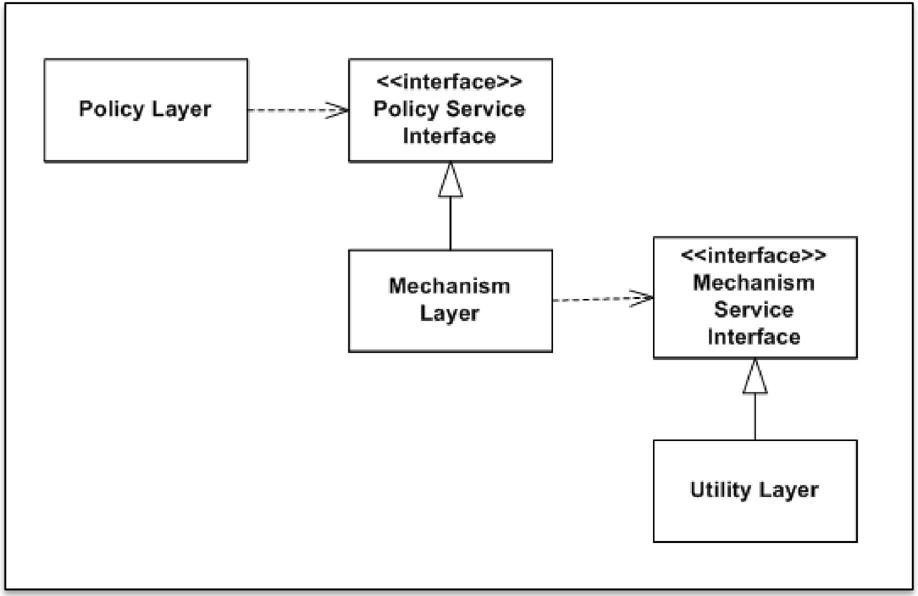
- The Service reference must still be initialized though
  - There are many common techniques for implementing this

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dependency Inversion Principle



- Tight Coupling (Traditional Layers Pattern)


Policy Layer
<<interface>> Policy Service Interface
Mechanism Layer
<<interface>> Mechanism Service Interface
Utility Layer

- Loose Coupling (Ownership Inversion)

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Factory Design Pattern Revisited

- Factories allow the choice of which concrete types an application uses to be encapsulated
- Client code delegates construction to the factory
  - Factory could read configuration file to allow for different deployments

```java
class Client {
    private Service service = ServiceFactory.createService();

    public void clientMethod() {
       service.operation();
    }
}
```

Factory creates concrete implementation

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Inversion of Control

- Inversion of Control (IoC) frameworks eliminate explicit object construction
  - Also known as Dependency Injection frameworks
- The framework will:
  - Read configuration information
  - Create object instances according to this information
  - Set up all references between the required instances
- The developer provides methods for the framework to use
  - Constructor
  - Setter methods

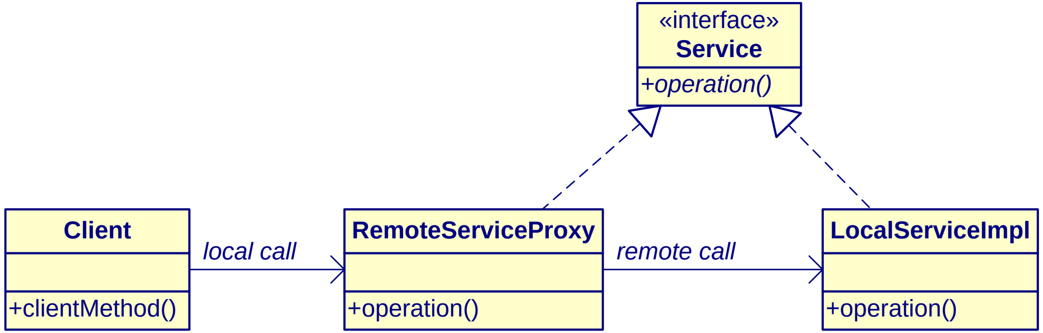
Factory will create Service and then call `setService()` when it creates the Client object

```
class Client {
    private Service service;

    public void setService(Service s) {
        this.service = service;
    }
}
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Using Proxies

- The Proxy Design pattern replaces a service with a surrogate object
  - Surrogate implements the same interface as the expected service
- The surrogate will usually add extra behavior to the service
  - For example, to provide network calls to access a remote service

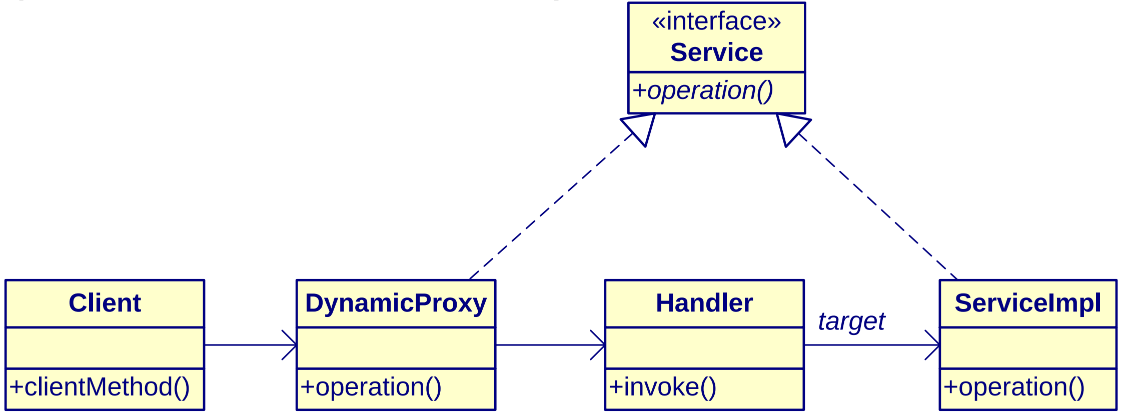


- Proxies have many other uses too
  - Encrypt and decrypt method parameters
  - Measure performance of calls on methods

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dynamic Proxies

- Java provides support for proxying of any interface
  - Interfaces to be proxied can be determined dynamically
  - Developer implements a handler class only



- To utilize dynamic proxies, the developer must:
  1. Implement the handler class
  2. Instantiate a proxy which uses that handler

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Implementing the Handler Class

- Implement the Handler interface
  - Add behavior before and/or after the target object’s method is called
  - For example, to log calls to a method

```
import java.lang.reflect.InvocationHandler;

public class LoggingProxyHandler implements InvocationHandler {
    private Object target;   // Any object

    public LoggingProxyHandler(Object target) {
        this.target = target;
    }
    public Object invoke(Object proxy, Method method, Object[] args){
       Log.info("about to call method " + method.getName());
       Object result = method.invoke(target, args);


       Log.info("completed call to method " + method.getName());
       return result;
    }
}
```

Actual object that will do the main work

Call proxied object to do the main work

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Instantiate Proxy

- To create a proxy, you must specify:
  - The class loader of the target service you are proxying
  - The interfaces that you wish to proxy on the service
  - A handler instance that the proxy will use
    - This will refer to the target service instance

```
Service targetService = new ServiceImpl();
LoggingProxyHandler handler = new
    LoggingProxyHandler(targetService);
Service proxy = (Service) Proxy.newProxyInstance(
               Service.class.getClassLoader(),
                           new Class[] {Service.class},
                           handler);
```

Actual object that will be proxied

Interface proxy is to implement

- Good places to create proxies include:
  - Within a factory method

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Example: Factory Using Proxy and Annotations

- We will now show an example of a factory that creates Robots
- The factory has the ability to create a proxy to a robot that:
  - Logs the execution time of its methods
- The factory will create a proxy to a robot if:
  - The robot class has the annotation `@Profile`
    - `@Profile` is a user-defined annotation
- The steps required to do this are:
  1. Define `@Profile` annotation
  2. Define handler that will do the profiling
  3. Create factory that potentially creates proxy

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Step 1: Define Profile Annotation

- The annotation needs to:
  - Be available at runtime
  - Be applied at class level

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Profile { }
```

Define new annotation named Profile

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Step 2: Define Handler

- The handler will be called by the proxy
  - Handler performs profiling

```
public class Profiler implements InvocationHandler {

      private Object target;  // Actual robot reference

      public Profiler(Object target){ this.target = target; }

      @Override
      public Object invoke(Object proxy, Method method,
                             Object[] arguments)throws Throwable {
        long start = System.nanoTime();
        Object result = method.invoke(target, arguments);
        long end = System.nanoTime();
        System.out.format("Method %s took %f seconds to run%n%n",
                method.getName(), ((end-start)/1000000000.0));
        return result;
      }
}
```

Call actual target method

Log execution time

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Step 3: Define Factory

- Factory will create robot and check `for @Profile` annotation

```
public class RobotFactory {
  public static Robot createRobot() throws IllegalArgumentException,
                   IllegalAccessException, InvocationTargetException{
    Robot robot = new SimpleRobot();
    if(robot.getClass().isAnnotationPresent(Profile.class)){



       Profiler profiler = new Profiler(robot));

       robot = (Robot)Proxy.newProxyInstance(profiler.getClass().
                 getClassLoader(),new Class[]{Robot.class}, profiler);
    }
    return robot;

  }
}
```

Is `@Profile` present on robot class?

Create proxy because annotation was found

Proxy returned if `@Profile` found on robot class or else plain robot instance

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Exercise 6.2: Implementing the Proxy Design Pattern

- Please turn to the Exercise Manual and complete Exercise 6.2: Implementing the Proxy Design Pattern

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have:

- Introduced Design Patterns
  - Aided the writing of maintainable code

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Chapter 7: Overview of Git

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will:

- Provide a high-level view of Git

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Role of Git

- The purpose of Git is to manage a project
  - Or a set of files as they change over time
    - This information is stored in a *repository*
- The software is known as a version control system
  - It records changes to files
    - Allows specific versions to be recalled on demand
- The concept is not new
  - Many products have been around a long time
    - Subversion, Perforce, ClearCase, etc.

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Why Git?

- The older version control systems are centralized
  - Server keeps all versions in one place
  - Clients ‘check-out’ the files they want to work on
    - Modify them on the local machine
  - Then ‘merge’ them back into central server
- Git is one of a new breed of version control systems that are distributed (mercurial is another well-known one)
  - No ***requirement*** to have central server that keeps all changes in one place
    - For all practical reasons, central server(s) still used to synchronize ‘local’ repositories
  - Each client fully mirrors the repository when they check out
  - Allows to implement MYRIADS of various workflows
    - Centralized, feature branch, gitflow, forking, etc.

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Advantages of Distributed Version Control

- Has many advantages over a centralized version control system
  - Speed – all files are local
  - Flexibility of working
    - Can have many branches (thousands)
      - Easy to switch between them
  - Ease of creating and merging branches

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Working with Git

- Git has three main states for the files it manages
  - *Modified*
    - File changed, but not committed
  - *Staged*
    - File has been marked as changed, ready for next commit
  - *Committed*
    - Stored safely in local database
- A simple workflow for a developer is:

  modify file → stage → commit

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Working with Git: Creating a New Repository

- A new repository is created from the command line by typing:
  - `git init`
- This creates `.git` directory in the current directory
  - This directory has the Git repository in it
- The status of the repository can be checked by typing:
  - `git status`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Staging and Committing

- To move file to staging, use:
  - `git add file(s)`
- This moves the file(s) to staging ready for commit
- Once files are staged, they can be committed
  - Every commit requires a message
- To commit a file, use:
  - `git commit -m "message describing changes"`
- A history of commits can be displayed by:
  - `git log`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Retrieving Previous Versions of a File

- Each commit has a hashcode listed with it
  - Displayed when you run `git log`
- To revert to a file from a previous commit, use:
  - `git checkout hashcode filename`
    - Where hash code is the code of the commit that has the file version you want
- You can then carry on working with the previous version

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Git Branches

- Branches enable multiple versions of a project to exist
  - They are given names to make them easier to track
  - `master` is the default branch name Git uses
- Enable separation of experimentation from production code
- Convention is that the `master` branch is the main line of code

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Creating Branches

- New branches are easily created
  - `git branch name_of_branch  hashcode_of_commit_to_branch_from`
- Work can continue as normal in the new branch
- Branches can be listed using the command:
  - `git branch`
- Current branch will be asterisked

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Merging Changes

- When two branches are to be merged:
  - Check out to the branch you want to merge into
    - `git checkout branch_name`
- Use the merge command to merge changes
  - `git merge branch_name_to_merge_with`
- If there are conflicts, the same file has been modified in the two branches
  - Merge will fail
  - Conflicts must be manually resolved

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Useful Resources and Tips

- https://www.atlassian.com/git/tutorials/comparing-workflows

- Scott Chacon talks
  - https://www.youtube.com/watch?v=ZDR433b0HJY
  - https://www.youtube.com/watch?v=xbLVvrb2-fY

- Key things to remember

1. **"Empty your cup"**
   - Forget all that you know about VCS if your experience is only with centralized VCS
2. **Understand underlying principles of Git/decentralized VCS**
   - Beam me up, Scotty! (see above)
3. **Practice makes perfect**

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Main Commands

- `git clone {git url}`
- `git add {files | ./}`
- `git commit -m '{comment}'`
- `git push`
- `git pull`
- `git fetch`
- `git merge`
- `git checkout {branch}`
- `git checkout -b {branch}`

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have:

- Provided a high-level view of Git

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Chapter 8: Introduction to Maven

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Objectives

In this chapter, we will:

- Introduce Maven and its role in Java development

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

The Java Development Process

- Consider a typical Java project
  - Uses third-party libraries—these must be fetched (jar files)
  - Tests written
  - Code developed
  - Resources created—configuration files, images
  - Class path set to include third-party libraries and project code
  - Project settings set for which directory has source code
    - Where to place class files
- When project is complete, files will need packaging in a jar file
  - With all related files to be included
- All this is time consuming and potentially error prone

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

What Is Maven?

- Tool for simplifying the Java development and build process
- Defines a common project structure
- Manages project dependencies—third-party libraries
- Enables quality metrics to be determined from code
  - Uses third-party tools for this

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Maven Projects

- Each project is built from a template
  - Known as an archetype
    - Many archetypes provided as standard
      - Plain Java applications
      - Web applications
      - Enterprise JEE applications
- Template defines project contents
  - Which third-party libraries should be included by default
- Project can be created from the command line

```
>mvn archetype:generate -DgroupId=com.mycompany.app -
DartifactId=my-app -DarchetypeArtifactId=maven-archetype-
quickstart -DinteractiveMode=false
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Maven Project Structure

- Maven defines a common directory structure for each project
- Ensures consistency across projects
- Allows Maven to have default settings for:
  - Which directories to compile code in
  - Which is test code
  - Where resources are
  - Where to place class files
- All details of the project are contained in a file named `pom.xml`

```
project root
  src
     main
        java
        resources
     test
        java
        resources
  target
     classes
     test-classes
pom.xml
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

POM File Structure

- POM file is the core configuration file for Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
           http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>first-maven-project/name>
  <url>http://maven.apache.org</url>
   <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.8.2</version>
       <scope>test</scope>
      </dependency>
    </dependencies>
</project>
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

POM File Structure (continued)

- `GroupId` indicates company or group that created project; e.g., apache
- `ArtifactId` is the project name; e.g., junit
- `Version` contains the version number for the project
- `Packaging` defines the type of output to be generated
  - Java has jar, war, and ear files
- `Name` is a display name used by IDE's
- `URL` is the URL of the project website, if it exists
- The combination of `GroupId`, `ArtifactId`, and `Version` define:
  - The co-ordinates of the project
    - Used by others who want to access/use the jar/war/ear file

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Managing Dependencies

- One of the great benefits of Maven is that it manages project dependencies
- Third-party libraries are automatically fetched and added to:
  - Class path for build
  - jar/war/ear file for packaging, if appropriate
- When Maven is installed, each user will have a local database of jar files
  - Known as local Maven repository
    - Installed in user's home directory in folder `.m2/repository`
- When a project requires a jar file(s), Maven looks:
  - In the local Maven repository
    - If its there, Maven uses that one
    - If not, it fetches it from central repository
      - And places it in the local one for future use

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dependencies

- Each public library (jar) is registered with a global Maven repository
- It is given its co-ordinates
  - `GroupId`, `ArtifactId`, `Version`
- These allow Maven to locate the jar file
  - Either in the user's local repository or in the global repository
- Dependencies are added to the `pom.xml` file

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Dependency Configuration

- Dependency configuration is performed in `pom.xml`
- The fragment below shows the configuration to include JUnit in the project
- The scope element defines where the dependency is needed
  - Compile, test, or runtime

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      …
    <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.8.2</version>
       <scope>test</scope>
      </dependency>
    </dependencies>
</project>
```

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Running Maven

- Maven can be run from the command line or from Eclipse
  - Eclipse requires a plugin such as m2e
- Maven has a number of commands; for example:
  - Package – packages project as a jar/war/ear
  - Test – runs tests
- When a command is executed, a number of phases are executed
  - These ensure the command can run successfully

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Maven Phases

- The phases Maven runs through when a command is issued are:
  - *Validate* – make sure all information and files are in place
  - *Compile* – main source code
  - *Test* – compiles and then executes test code
  - *Package* – creates project jar/war/ear file

- Maven can be further configured to:
  - Run integration tests
  - Run quality test with tools such as:
    - Cobertura for test code coverage analysis
    - Sonar for code quality metrics
  - Deploy the code for execution

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Maven Installation

- To install Maven to run from the command line:
  1. Download the Maven distribution (`maven.apache.org`)
  2. Unzip the distribution
  3. Set environment variables:
     a) `M2_HOME` to where you unzipped Maven
     b) `Add M2_HOME/bin` to your `PATH`
- That's it—you can start using Maven
- Comprehensive documentation is provided online
  – maven.apache.org

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Chapter Summary

In this chapter, we have:

- Introduced Maven and its role in Java development

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Course Summary

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Summary

- During this course, you have learned:
  - Java 8 Languages and Features
  - Java Design Patterns
  - Building and Deploying Applications and Web Services with Spring Boot
  - Test-Driven Development
  - Web Security Foundations
  - Working with Git
  - Maven Essentials
- How to build systems using these technologies secured by tests
  - Via a case study

© 2019 Copyright ROI Training, Inc. All rights reserved. Not to be reproduced without prior written consent.