Developing RESTful Web Services in Java

Overview of Web Services

Common Elements of Distributed Computing

- Service
  - The behavior that will be executed
- Protocols
  - Agreements between service and client about what messages “on the wire” mean
  - Describe argument and return data
  - Describe what behavior to execute
- Stubs, Ties, Proxies, …
  - Provide the connection between the wire protocols and the code that implements the service

Evolution of Web Services — 1

- Intranet systems, 1960-1995 (approx)

- Network accessible systems
  - Encapsulation (origins of OO) facilitates reuse
- Stateless functionality
  - Roots of clustering; facilitates capacity and failover
- Messaging
  - Temporal and Spatial decoupling, clustering support

Evolution of Web Services — 2

- Internet / B2B systems, 1995 on

- HTTP transport
  - Cheats on reluctant firewall admins, becomes standard way to do B2B
  - Tunnelling other protocols (IIOP, RMI, etc.)
  - Modified to “pure” HTTP and became “Web Services”
  - Other transports can be/are sometimes used, e.g. SMTP

Evolution of Web Services — 3

- XML message format
  - Platform and language neutral
  - “Looks like” HTML

- HTML provides content with presentation information
- XML provides content with structural information
  - More suited to machine consumption

Evolution of Web Services — 4

- Standards address interface specification, message, and data formats
  - SOAP, WSDL, XSD
- Many standards grow to address all kinds of business needs
  - WS-SOUP
- Complexity, rigidity begin to irritate some developers—trend toward REST
  - Treat web as network of “resources”
  - HTTP provides POST, GET, PUT, DELETE (CRUD)

Origins of Service Oriented Architecture

- Tightly coupled mesh of interactions between services is abstracted to a “Mediator” pattern
  - Manages all interactions
  - Centralized workflow that can change without affecting the services
- Orchestration brings “statefulness” or history to flow management
- BPEL standardizes flow control language
  - Visual flow design tools facilitate feedback/verification by business managers

SOA Goals

- Design considerations/goals:
  - Stateless
  - Coarse grained
  - Loosely coupled
- Architecture goals
  - Scalable
  - Available
  - Maintainable
  - Reconfigurable
  - Securable
  - (Probably not externally transactional)

Key Architectural Concerns in Distributed Computing

- Security:
  - Everyone is trying to attack you
  - How many times must you log in?
- Capacity:
  - Benefits from statelessness
  - Amdahl’s law; avoid transactions
- Redundancy
  - Benefits from statelessness

Key Architectural Concerns in Distributed Computing

- Performance:
  - Network roundtrips are slow
  - Bandwidth is finite
- Reliability:
  - Partial failure creates new problems for transactional correctness

Overview of JAX-WS

Using JAX-WS

- JAX-WS provides for SOAP type web services
  - Historically several products, tools, and APIs have provided this with varying levels of complexity and standardization
- JAX-WS currently provides for WS-I Basic Profile 1.1
  - Therefore supports non-Java clients and services
- Annotation based
- Creates clients and servers

Design Approaches

- Create client support code from WSDL & XSD
  - `wsimport`
- Create WSDL & XSD from a Java implementation of a Web Service
  - Automatic on deployment / `wsgen`
- Create skeleton code to support implementation of a Web Service in Java starting from WSDL & XSD
  - `wsimport`

Creating A New WebService In Java

```
@WebService public class SmartRemark {
  private String [] remarks = {
    "Imagination is more important than knowledge.",
    "Quidquid Latine dictum sit altum videtur!"
  };

  @WebMethod public String getRemark() {
    int idx = (int)(Math.random() * remarks.length);
    return remarks[idx];
  }
}
```

Publishing A Web Service

- Options include:
  - In a Web Container
  - From a Stateless Session Bean in an EJB container
  - In Java SE directly

- Benefits of Web and EJB containers:
  - Declarative security control
  - Management/monitoring features
  - Capacity/efficiency

Publishing A WebService In Java SE

```
public class Publisher {
  public static void main(String[] args) {
    String theURL =
      "http://localhost:8888/ws/server";

    Endpoint.publish(theURL, new SmartRemark());
    System.out.println("Service is published!");
  }
}
```

Accessing Deployed WSDL

- Following deployment, WSDL is automatically generated and published:
  - In this example:

```
http://localhost:8888/ws/server?wsdl
```

Creating A Java WS Client

- Generate supporting artefacts from WSDL:

```
wsimport -keep -p <package> <wsdl-url>
```

- Resulting classes provide:
  - Java `interface` defining the service methods
  - Factory class for creating the port / stub
  - JAX-B annotated classes for arguments, returns, exceptions (faults), and exception details (as JavaBeans)
  - `ObjectFactory` for creating objects of supporting types

Working With Eclipse

- Eclipse doesn't like "foreign" classes
  1. Run `wsimport` with the `-keep` option, in a *different directory tree* entirely
  2. Delete all the `.class` files that it generates
  3. Create the destination package in the project in Eclipse
  4. Right-click on the package in Eclipse, then select Import
  5. In the wizard, open the "General" folder and choose File System, hit "Next"
  6. Browse to & select the generated package directory
  7. Select the checkbox for that directory & hit "Finish"

Creating A Java WS Client

- Find the class that offers the method:
  - **get**Xxxx**Port**()
- Create the port and call methods on it:

```
public class RemarkClient {
  public static void main(String[] args) {
    SmartRemark remark =
      new SmartRemarkService().getSmartRemarkPort();
    System.out.println("Smart Remark is “
      + remark.getRemark());
  }
}
```

WS Arguments & Return Types

- JAX-WS permits complex arguments & returns
  - They must be XSD compatible (aka JAX-B compatible)
- These do not need to be JAX-B annotated; `wsgen/wsimport` creates the JAX-B types as needed
- `wsimport` will represent `List` or array elements using a mutable `List`
  - Only a `List<?> getXxxx()` method will be provided, expect to modify the provided list, not replace it

Exceptions From Web Methods

- Exceptions may be thrown from web methods
- Exception classes should be XSD compatible
- wsgen/wsimport create exception types that use a JavaBean to represent the exception data
- This JavaBean property is called “faultInfo”, generally has a property “message”
  - `exception.getFaultInfo().getMessage()`

RESTful Web Services

Objectives

On completion you should be able to:

- Describe the nature of RESTful web services
- Describe how HTTP and its methods support REST
- Build a REST services using key features of JAX-RS

What are REST Services?

RESTful Web Services

- Data-structure oriented
  - Implied behavior through HTTP METHOD types
  - Not “strongly typed”; assumes client understands data
  - Data may be sent in XML, JSON, or other forms
- REST interactions are performed using URLs
  - Following general HTTP request structures (headers, parameters, cookies, etc.)
  - Focused on invoking / accessing resources
  - REST WS should be cacheable

Composition of HTTP

- Basic format of HTTP request / response
  - *Initial Line*
  - *Headers*
  - Blank Line
  - *Message Body*
  - Blank line

Initial Request Line

- Single line containing space delimited fields
- Used to describe request
  - HTTP method
  - Resource
    - Translated into path starting from *server root* (/)
    - May contain a *query string* depending on Method type
  - Protocol version
    - HTTP 1.0
    - HTTP 1.1

HTTP Methods

- GET
  - Retrieve information specified in URI
  - Server returns Headers and Message Body
  - Responses are cacheable
  - Typically result of an `<A href>`
  - Extra URI information passed as a query string
- POST
  - “Send” information to specified URI
  - Asks server to accept information as subordinate of specified URI
  - Information is “handled” by specified URI
  - Responses are not cacheable
  - Typically result of `<FORM>`
  - Post data sent as entity

HTTP Methods (cont.)

- PUT
  - “Send” information to specified URI
  - Server stores information as a resource under specified URI
  - Can overwrite existing content
- DELETE
  - Requests specified resource be deleted
  - Typically generates a success or a fail response

- Proposed: PATCH
  - Currently under discussion
  - Variation of PUT intended to modify only part of the resource
    - PUT overwrites the entire resource
    - PATCH modifies the elements included in the entity

HTTP Methods (cont.)

- HEAD
  - Like GET; retrieves information
  - Retrieves meta-information; server returns only Headers
- OPTIONS
  - Retrieve information about the communication options available
  - May be used by browser to determine best interaction mechanism
- TRACE
  - Similar to trace route
  - Used to determine what information is received along request chain

REST Interactions with HTTP

- HTTP methods create database like access
  - POST - Creates resource
  - GET - Reads resource
  - PUT - Updates resource
  - DELETE - Deletes resource

Example Request Initial Lines

- Accessing top-level domain
  - **http://www.host.com**
  - `GET / HTTP/1.1`
- Accessing sub-resource
  - **http://www.host.com/LEARN/j2se_overview.pdf**
  - `GET /LEARN/j2se_overview.pdf HTTP/1.1`
- Accessing dynamic resource using GET
  - **http://www.google.com/search?q=rest**
  - `GET /search?q=rest HTTP/1.1`

Initial Response Line

- Single line containing space-delimited fields
- Used to describe response
  - Protocol version
    - HTTP 1.0
    - HTTP 1.1
  - Response status (*status code*)
    - Numeric value
    - Standard or proprietary
  - Description
    - Human readable description of status code
- Five main categories of responses
  - *Information*
  - *Success*
  - *Redirection*
  - *Client Error*
  - *Server Error*

Informational Responses

- Provisional response; typically used for handshaking or negotiating
- Common status codes
    - `100` - Tells client to continue with request
    - `101` - Tells client server is trying to adopt client-suggested protocol
- Consists of
    - Initial line (*status line*)
    - Headers
    - Empty line

Successful Responses

- Signifies client request was received, understood, and accepted by server
- Common status codes
  - `200` - Tells client request succeed; response body is sent depending on type of method
  - `202` - Tells client server accepted request; but processing of request has not been completed
- Consists of
  - Status line
  - Headers
  - Empty line
  - Response body*

Redirection Responses

- Used to notify client further action is required to fulfill request
- Typically used to notify client to access resource in a different location
- Common status code
  - `301` - Resource permanently relocated
  - `307` - Temporary redirect of resource; also known as client-side redirect
- Consists of
  - Status line
  - Headers
  - Empty line

Client Error Responses

- Used to notify client that the request contained an error
- Error could be
  - `400` - Bad request; malformed request
  - `401` - Unauthorized access; the request was missing the authorization header
  - `403` - Attempted forbidden; no resolution
  - `404` - Resource specified in URI was not found
  - `405` - Method not supported
- Typically consists of
  - Status line
  - Headers
  - Empty line

Server Error Responses

- Used to notify client that server encountered an error in processing the request
- Server errors could be
  - `500` - Internal, irresolvable, unexpected problem
  - `503` - Service unavailable typically as a result of load issues
  - `505` - HTTP version not supported
- Consists of
  - Status line
  - Headers
  - Empty line

Example Response Initial Lines

- Accessing top-level domain
  - **http:// www.host.com**
  - `HTTP/1.1 200 OK`
- Accessing missing sub-resource
  - **http:// www.host.com/LEARN/dot-net_overview.pdf**
  - `HTTP/1.1 404 NOT FOUND`
- Accessing redirected resource
  - **http://www.javasoft.com**
  - `HTTP/1.1 301 Moved Permanently`

HTTP Headers

- Used to describe meta-information
  - Describe information about client capabilities
  - Describe information about server response
- Used for both requests and responses
- Represented as name/value pairs
- Follows this syntax:
  - *HeaderName: HeaderValue*
  - *HeaderValue* typically contains text data
- May use custom headers

Request Headers

- `Accept`
  - Notifies server type of data client can handle
  - Can be repeated
  - Contains comma separate list of mime-types; may use wildcards
- `Accept-Encoding`
  - Notifies server type of data encoding client can handle
  - Used for things like Zip compression
- `Accept-Language` - Desired response language
- `Host` - Specifies Internet host and port of requested resource
- `User-Agent`
  - Describes client software to server
  - Used for tracking purposes; commonly written in HTTP access logs
  - Used to generate browser specific HTML
- `Referrer` - Notifies server of where the request originated

Response Headers

- `Age`
  - Designation (estimate) of time since response was generated
  - Used with proxies
- `Server` - Describes server software used
- `Location` - Describes location client should use to fulfill request

General Purpose Headers

- Applicable to both request and response
- Secondary meta-information about interaction
- `Cache-Control`
  - Describes caching directives
  - Used by clients and proxies
- `Date` - General purpose date
- `Upgrade`
  - Used by client on request to tell server what protocol it would like to use
  - Used by server to notify of which protocols are changing

Entity Headers

- Headers used to describe entity (*message body*)
- Can be used in request and / or response
- `Allow` - Specifies which HTTP methods are supported
- `Content-Language` - Natural language of content
- `Content-Length` - Length of content
- `Content-Type`
  - Describes type of content
  - Usually MIME type representation
- `Expires` - When content is considered out of date
- `Last-Modified` - Last modification date of content

Client Request Example

- Example of client HTTP request
    - Initial Line
    - Headers
- URI used for request **http:// www.host.com**

```
GET / HTTP/1.1
Host:  www.host.com
Accept-Encoding: gzip
Accept: */*
Accept-Language: en-us, ja;q=0.62, de-de;q=0.93, de;q=0.90, fr-
fr;q=0.86, fr;q=0.83, nl-nl;q=0.79, nl;q=0.76, it-it;q=0.72,
it;q=0.69, ja-jp;q=0.66, en;q=0.97, es-es;q=0.59, es;q=0.55, da-
dk;q=0.52, da;q=0.48, fi-fi;q=0.45, fi;q=0.41, ko-kr;q=0.38
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us)
AppleWebKit/125.5.5 (KHTML, like Gecko) Safari/125.11 Web-Sniffer/
1.0.17
```

Server Response Example

- Example of HTTP response
    - Initial Line
    - Headers
- Sent from **http:// www.host.com**

```
HTTP/1.1 200 OK
Date: Tue, 16 Nov 2004 03:16:22 GMT
Server: Apache/1.3.28 (Unix) mod_watch/3.12 PHP/4.3.2 mod_ssl/2.8.15
OpenSSL/0.9.7b
X-Powered-By: PHP/4.3.2
Content-Type: text/html
```

HTTP Entity

- Considered body or payload of
  - Request
  - Response
- Entity contents may or may not exist
  - Depends on Method
  - Depends on type of response
- Content headers assist in
  - Determining length of entity
  - Type of entity
  - Encoding of entity
- Server may
  - Take action against entity (process it)
  - Apply it (store it)
  - Ignore it
- Client may
  - Take action against entity (render it)
  - Apply it (store it)
  - Ignore it

GET Request - Entity Example

- Client requesting **http://www.google.com/search?q=rest**
- Entity is empty

```
GET /search?q=rest HTTP/1.1
Host: www.google.com
Accept-Encoding: gzip
Accept: */*
Accept-Language: en-us, ja;q=0.62, de-de;q=0.93, de;q=0.90, fr-
   fr;q=0.86, fr;q=0.83, nl-nl;q=0.79, nl;q=0.76, it-it;q=0.72,
   it;q=0.69, ja-jp;q=0.66, en;q=0.97, es-es;q=0.59, es;q=0.55,
   da-dk;q=0.52, da;q=0.48, fi-fi;q=0.45, fi;q=0.41, ko-kr;q=0.38
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us)
   AppleWebKit/125.5.5 (KHTML, like Gecko) Safari/125.11 Web-
   Sniffer/1.0.17
Content-type: application/x-www-form-urlencoded
Content-length: 7
```

<entity is empty>

Post Request - Entity Example

- Client requested **http://www.google.com/search?q=rest**
- Entity contains `q=john`

```
POST /search HTTP/1.1
Host: www.google.com
Accept-Encoding: gzip
Accept: */*
Accept-Language: en-us, ja;q=0.62, de-de;q=0.93, de;q=0.90, fr-
   fr;q=0.86, fr;q=0.83, nl-nl;q=0.79, nl;q=0.76, it-it;q=0.72,
   it;q=0.69, ja-jp;q=0.66, en;q=0.97, es-es;q=0.59, es;q=0.55,
   da-dk;q=0.52, da;q=0.48, fi-fi;q=0.45, fi;q=0.41, ko-kr;q=0.38
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us)
   AppleWebKit/125.5.5 (KHTML, like Gecko) Safari/125.11 Web-
   Sniffer/1.0.17
Content-type: application/x-www-form-urlencoded
Content-length: 7

q=john
```

Response - Entity Example

- Client requested http://www.google.com/search?q=rest

```
HTTP/1.0 200 OK
Content-Type: text/html
Server: GWS/2.1
Date: Wed, 17 Nov 2004 04:09:10 GMT
Connection: Close

<html><head><meta HTTP-EQUIV="content-type" CONTENT="text/html; charset=ISO-8859-1"><title>Google Search: rest </title><style><
!--
body,td,div,.p,a{font-family:arial,sans-serif }
div,td{color:#000}
.f,.fl:link{color:#6f6f6f}
a:link,.w,a.w:link,.w a:link{color:#00c}
a:visited,.fl:visited{color:#551a8b}
a:active,.fl:active{color:#f00}
.t a:link,.t a:active,.t a:visited,.t{color:#000}
.t{background-color:#e5ecf9}
.k{background-color:#36c}
.j{width:34em}
.h{color:#36c}
.i,.i:link{color:#a90a08}
.a,.a:link{color:#008000}
.z{display:none}
div.n {margin-top: 1ex}
.n a{font-size:10pt; color:#000}
.n .i{font-size:10pt; font-weight:bold}
.q a:visited,.q a:link,.q a:active,.q {color: #00c; }
.b{font-size: 12pt; color:#00c; font-weight:bold}
.ch{cursor:pointer;cursor:hand}
.e{margin-top: .75em; margin-bottom: .75em}
.g{margin-top: 1em; margin-bottom: 1em}
//--><script>
<!--
function ss(w){window.status=w;return true;}
function cs(){window.status='';}
function ga(o,e) {return true;}
//-->
</script>
```

Introduction to JAX-RS

RESTful Web Services With JAX-RS

- JAX-RS provides APIs for REST web services
  - Not included in JAVA SE; included as part of Java EE 6
  - Annotation based development of services
  - Reference implementation (“Jersey”) available from http://jersey.java.net
  - Alternate popular implementation from JBoss community at http://www.jboss.org/resteasy
  - JAX RS currently only server side
    - Client API in development
  - Ensure JAR files are on path
- Server side is often deployed in a container
  - Jersey provides stand-alone implementation

A Trivial JAX-RS Web Service

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;

@Path("/helloworld")
public class HelloWorldResource {
  @GET @Produces("text/plain")
  @Path(“{id}”)
  public String getMessage(@PathParam("id") int id){
    return "id is " + id;
  }
}
```

Key JAX-RS Annotations

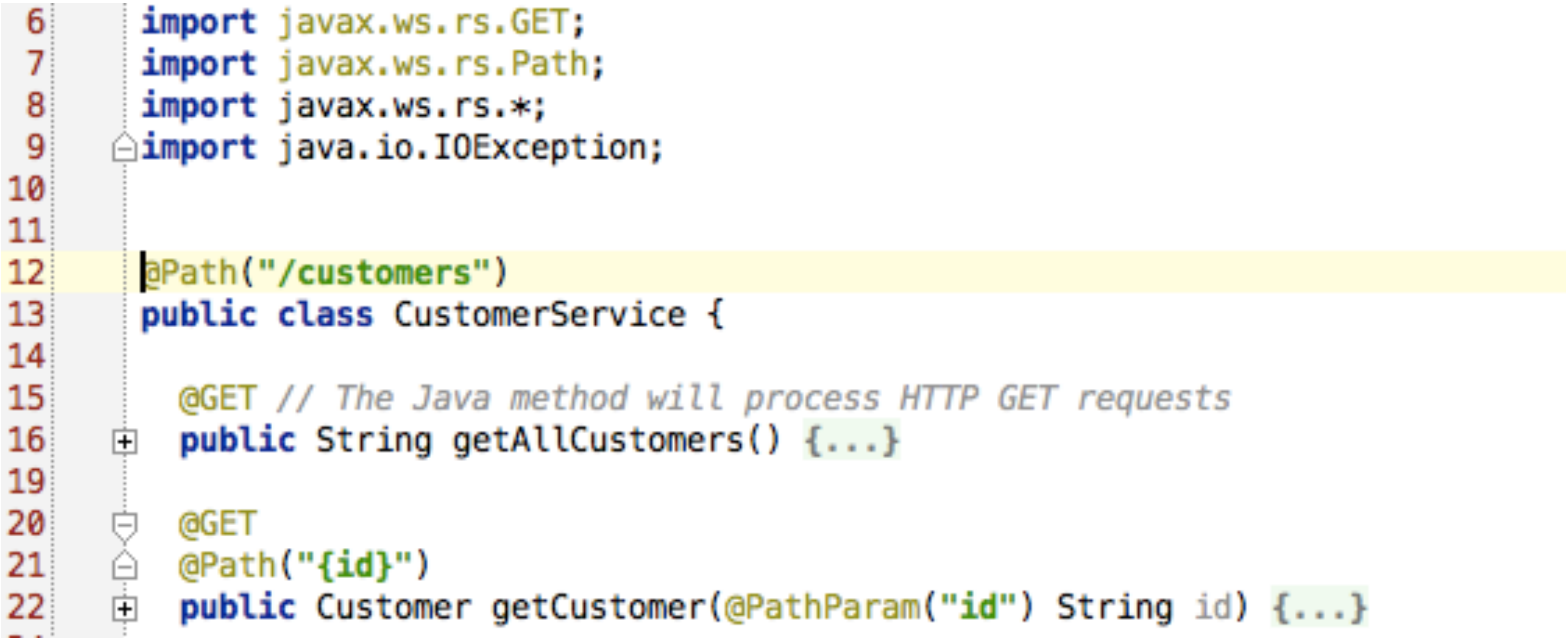
- `@Path` – defines endpoint for REST service
- HTTP request methods – apply to methods to define “request handlers”
  - `@GET`
  - `@POST`
  - `@PUT`
  - `@DELETE`
- Data transfer annotations
  - `@Produces`
  - `@Consumes`

Key JAX-RS Annotations [cont.]

- `@PathParam` – used to retrieve path variables
- `@QueryParam` – used to retrieve query string variables
- `@FormParam` – used to retrieve form parameters
- `@HeaderParam` – used to retrieve header fields and values
- `@CookieParam` – used to retrieve cookie values

@Path Annotation

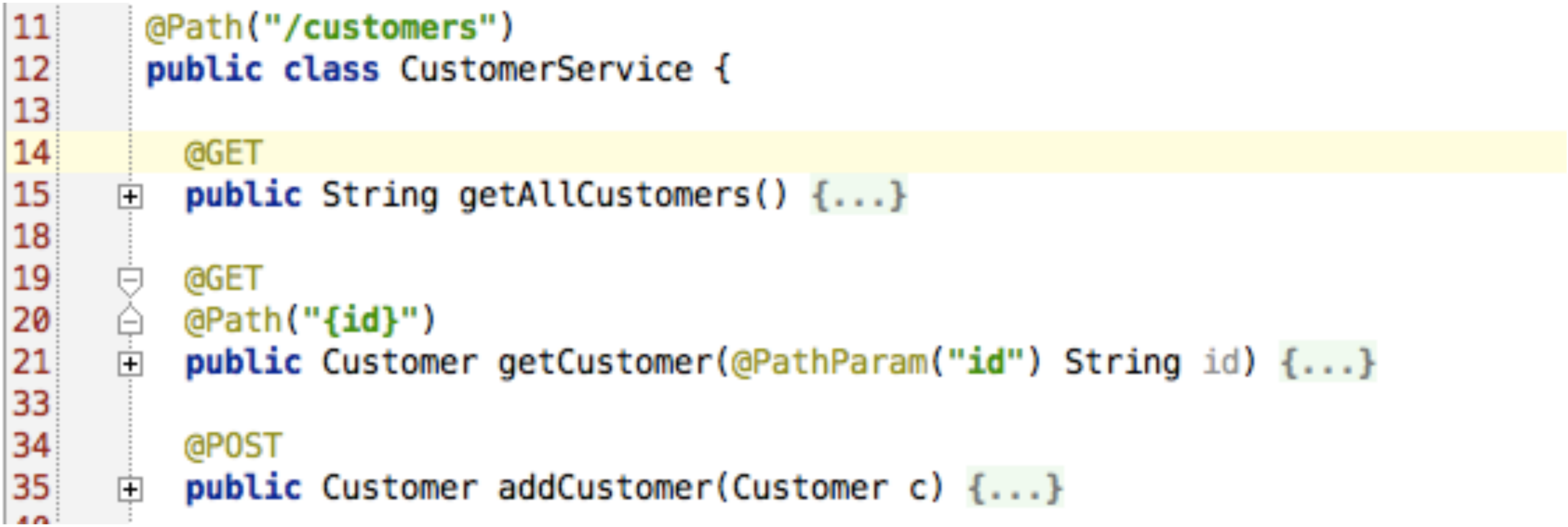
- Can be applied to:
  - Class – defines top-level path for resource
  - Method – defines specialized path for REST call
  - Can contain variable placeholders using { }



```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.*;
import java.io.IOException;


@Path("/customers")
public class CustomerService {

  @GET // The Java method will process HTTP GET requests
  public String getAllCustomers() {...}

  @GET
  @Path("{id}")
  public Customer getCustomer(@PathParam("id") String id) {...}
```

HTTP Method Annotations

- Single annotation applied to single method
- Multiple methods can have same method annotation
  - If each path is distinct or
  - If each has different @Consumes or @Produces



```
@Path("/customers")
public class CustomerService {

  @GET
  public String getAllCustomers() {...}

  @GET
  @Path("{id}")
  public Customer getCustomer(@PathParam("id") String id) {...}

  @POST
  public Customer addCustomer(Customer c) {...}
```

JAX-RS Data Type Annotations

- Data type may be specified at the class or the method level, or both
  - Class level creates a default
  - Method declaration overrides the default
  - Beware that defaults apply to any method that doesn't have explicit specification
    - This might create an input expectation for a GET handler
- `@Produces({ array of mime types})`
  - Specifies the MIME type(s) that can be returned
- `@Consumes({ array of mime types}`
  - Lists accepted input types

JAX-RS Data Type Annotations

- `MediaType` class provides constants:
  - `@Produces(MediaType.APPLICATION_JSON)`
- Or can use a String:
  - `@Produces({ “text/html”, “text/plain”})`
- `@Produces` and `@Consumes` constrain the selection of handler methods based on the HTTP request’s `Accept:` and `Content-Type:` headers

Example Data Type Configurations

- Text conversion
  - `@Produces(MediaType.TEXT_PLAIN)`
  - `@Consumes(“text/plain”)`

- XML conversion done using JAXB
  - `@Produces(MediaType.APPLICATION_XML)`
  - `@Consumes(“application/xml”)`

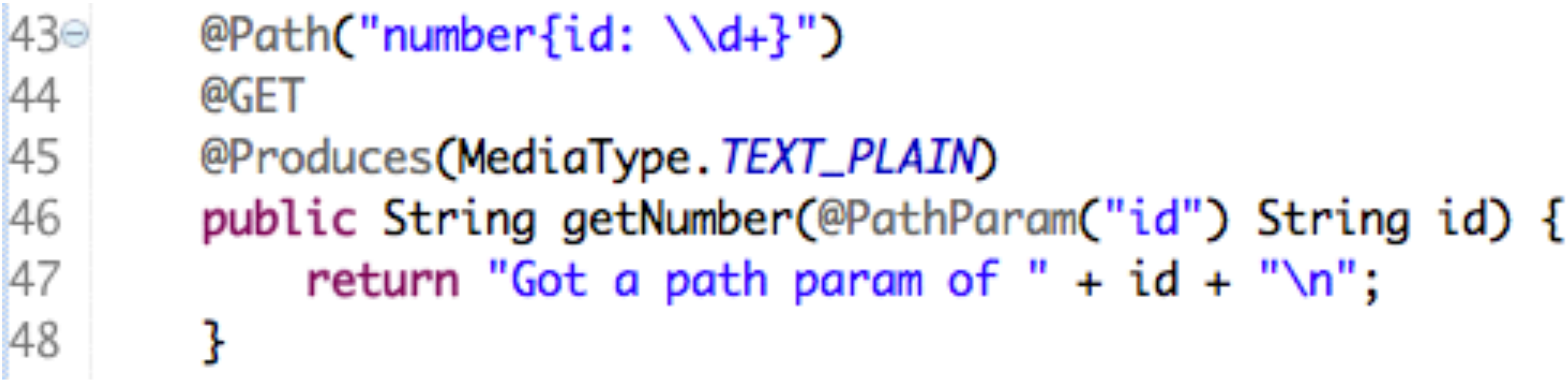
- JSON conversion is not standard in EE 6; but generally supported
  - `@Produces(“application/json”)`
  - `@Consumes(MediaType.APPLICATION_JSON)`

Path Parameter Annotation

- Client originated data may be passed into the service method via arguments
- … `method(@`**`PathParam`**`("id") int id)`
  - `PathParam` pastes the { } part of path into variable

@PathParam Regular Expressions

- Path parameters can be coerced to match regular expressions
- Remember that many regular expressions involve backslashes, which must be doubled in Java `String`s



```
@Path("number{id: \\d+}")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String getNumber(@PathParam("id") String id) {
    return "Got a path param of " + id + "\n";
}
```

PathParam and Segments

- Generally one path element, such as `@Path("{id}")` will match one “segment” of a URL, between two slash (“/”) characters
- However, if a regular expression admits slash, then the parameter might match multiple segments, so:
- `@Path("{multiple: [a-z/]*}")` matches:
  - `simple`
  - `complex/with/many/parts`

JAX-RS Parameter Annotations

- …`method(@`**`QueryParam`**`(“author”) String author)`
  - Injects a query param (…`/service?author=“Fred”`) into the service method
- …`method(@`**`FormParam`**`(“count”) String count)`
  - Injects a form param (…`/service?author=“Fred”`) into the service method

Running the Service

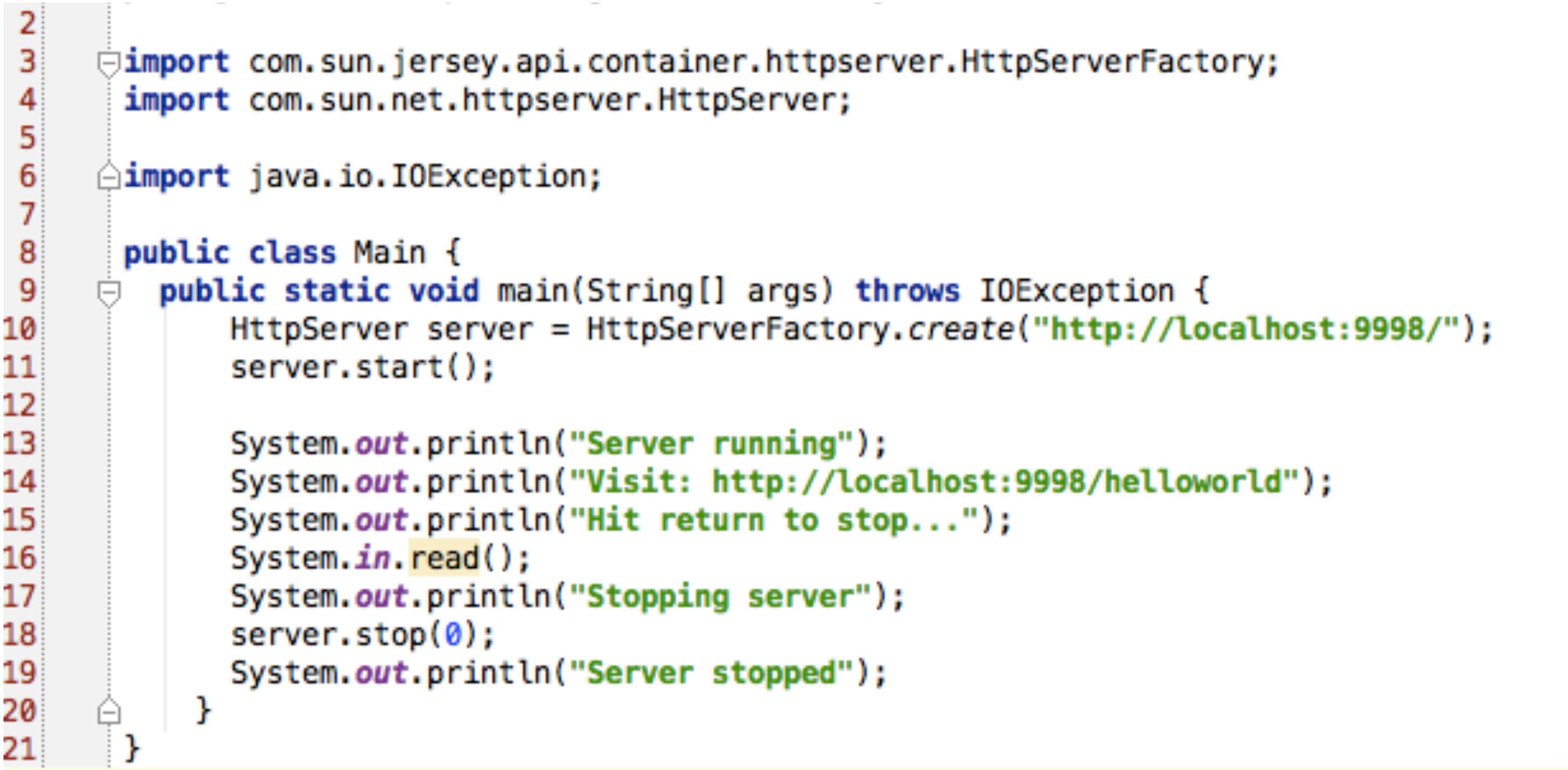
Various launch mechanisms exist for REST services

1. Use HTTP server provided by JAX-RS reference implementation (Jersey/Grizzly)

2. Deploy REST services in a Web container
   - Configure REST framework using a Servlet

3. Deploy REST services in Java EE 6 Web Container
   - Include an `Application` class in your `lib`

Launching The Service with Grizzly

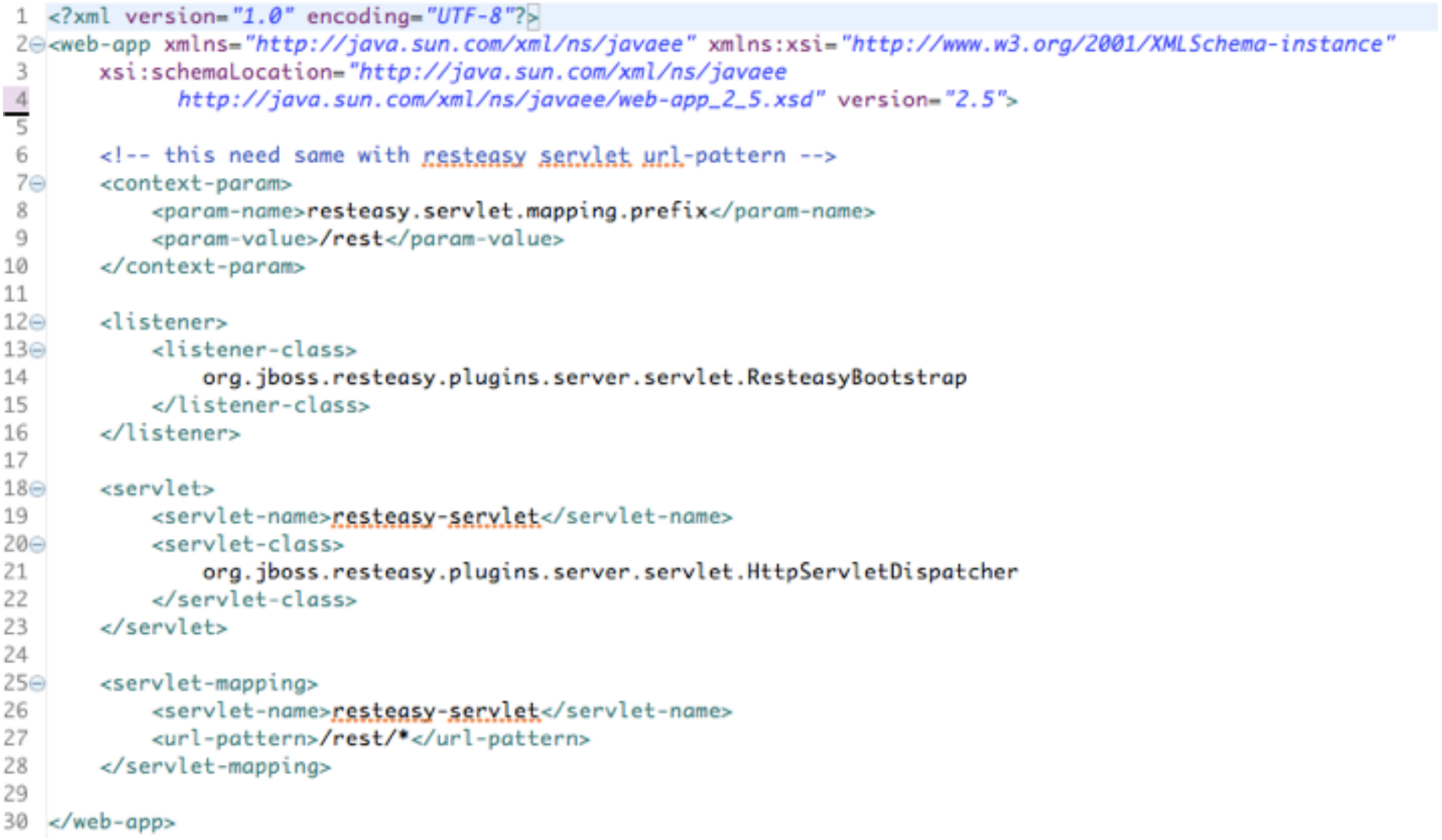


```java

import com.sun.jersey.api.container.httpserver.HttpServerFactory;
import com.sun.net.httpserver.HttpServer;

import java.io.IOException;

public class Main {
  public static void main(String[] args) throws IOException {
      HttpServer server = HttpServerFactory.create("http://localhost:9998/");
      server.start();

      System.out.println("Server running");
      System.out.println("Visit: http://localhost:9998/helloworld");
      System.out.println("Hit return to stop...");
      System.in.read();
      System.out.println("Stopping server");
      server.stop(0);
      System.out.println("Server stopped");
    }
}
```

Running Service in Container [#1]

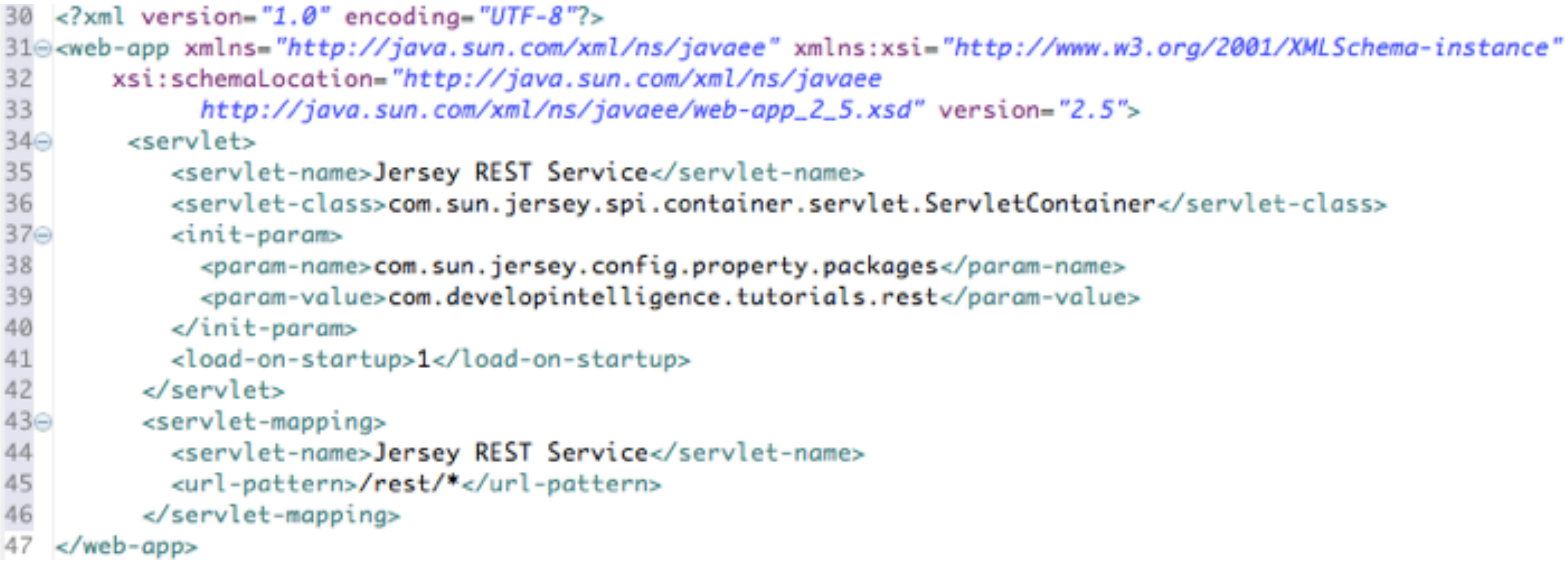
- Deploy JAX-RS implementation and service classes as part of WAR
- Configure `web.xml` to support REST
- Configure top level resources into the system
  - Using Application class
  - Or using scanning where supported

RESTEasy `web.xml` config



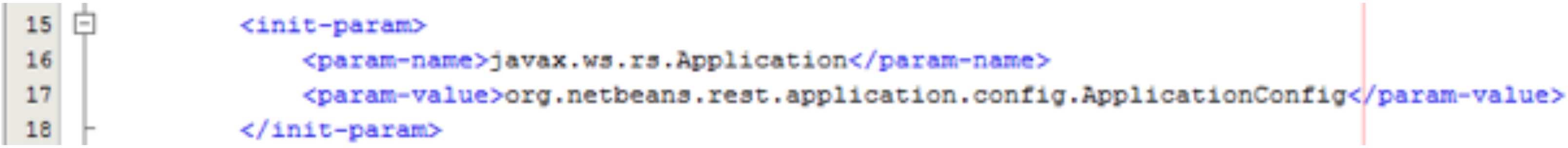
```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

    <!-- this need same with resteasy servlet url-pattern -->
    <context-param>
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/rest</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>resteasy-servlet</servlet-name>
        <servlet-class>
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>resteasy-servlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Jersey `web.xml` config



```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
     <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
          <param-name>com.sun.jersey.config.property.packages</param-name>
          <param-value>com.developintelligence.tutorials.rest</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
        <url-pattern>/rest/*</url-pattern>
      </servlet-mapping>
</web-app>
```

- For non-scanning (explicit application) use:



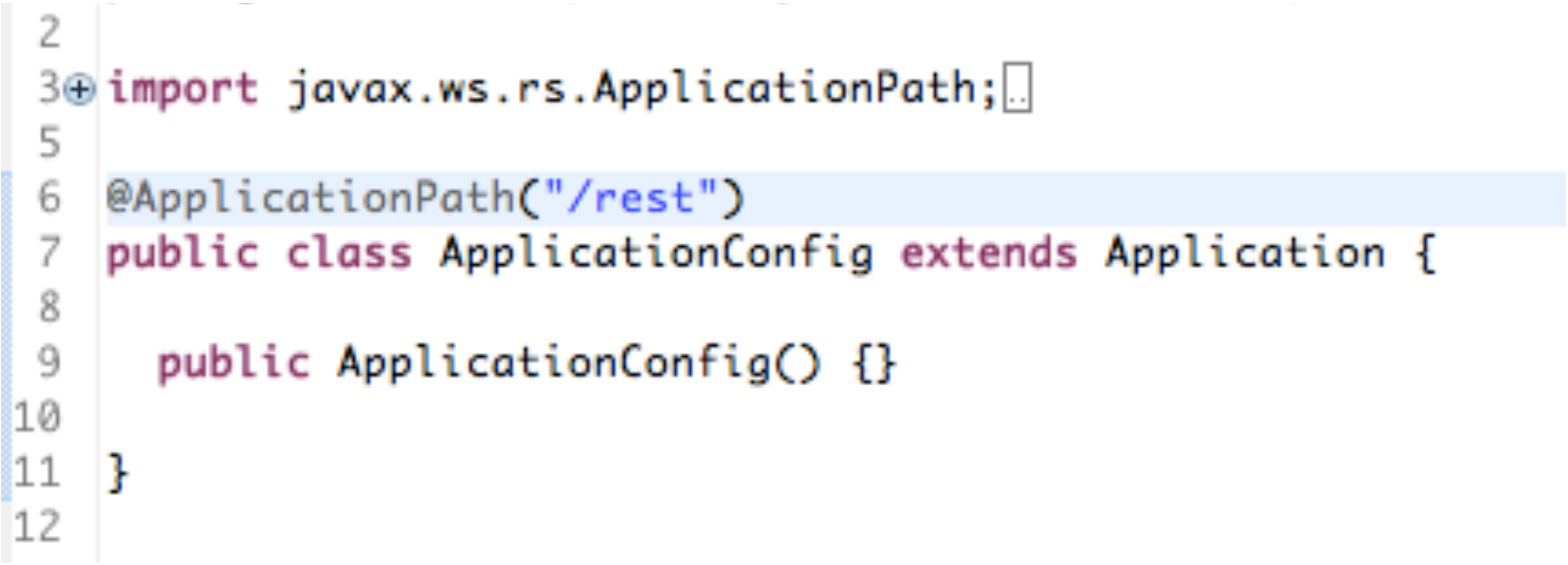
```xml
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>org.netbeans.rest.application.config.ApplicationConfig</param-value>
        </init-param>
```

Running Service in Container [#2]

- JAX-RS implementation provided by Java EE 6 containers
  - Pre-configured to support REST
  - No need to “configure” web.xml (web.xml might not exist)
  - Will automatically perform “auto-scan” to locate Application class

- Only need to specify application path for REST endpoints
  - Create a class specifically to define `@ApplicationPath`
  - Must be a subclass of `Application`

Application Config Example



```java

import javax.ws.rs.ApplicationPath;

@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

  public ApplicationConfig() {}

}

```

Lab 1

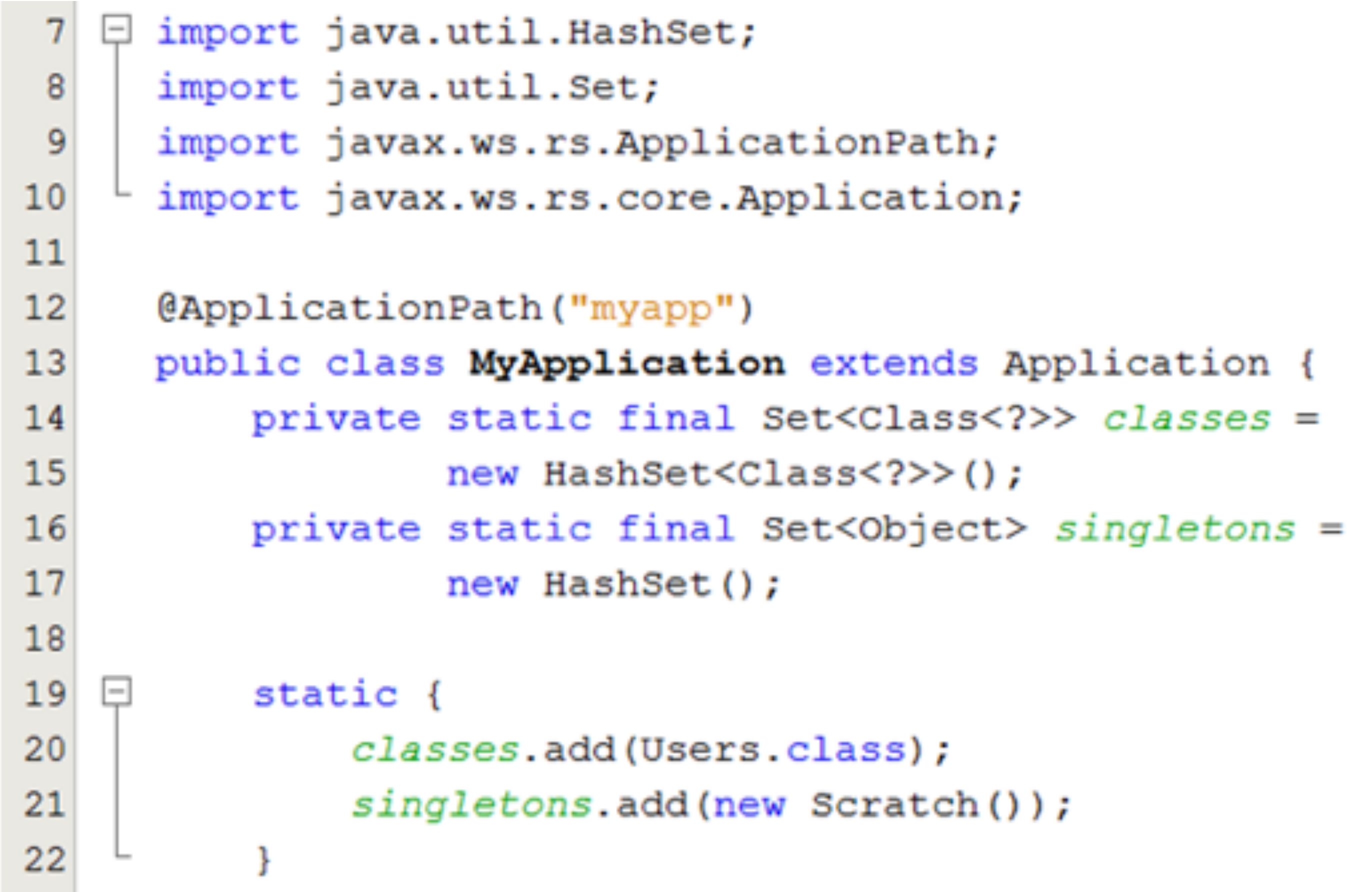
Working with JAX-RS

Resource Object Lifecycle

- Resource objects, those annotated with @Path can be created on a per-request basis, or as singletons
  - Modern Java GC ensures that per-request is not a performance issue
- Controlled by Application class
- Application class can also define a URI path element

Application Class

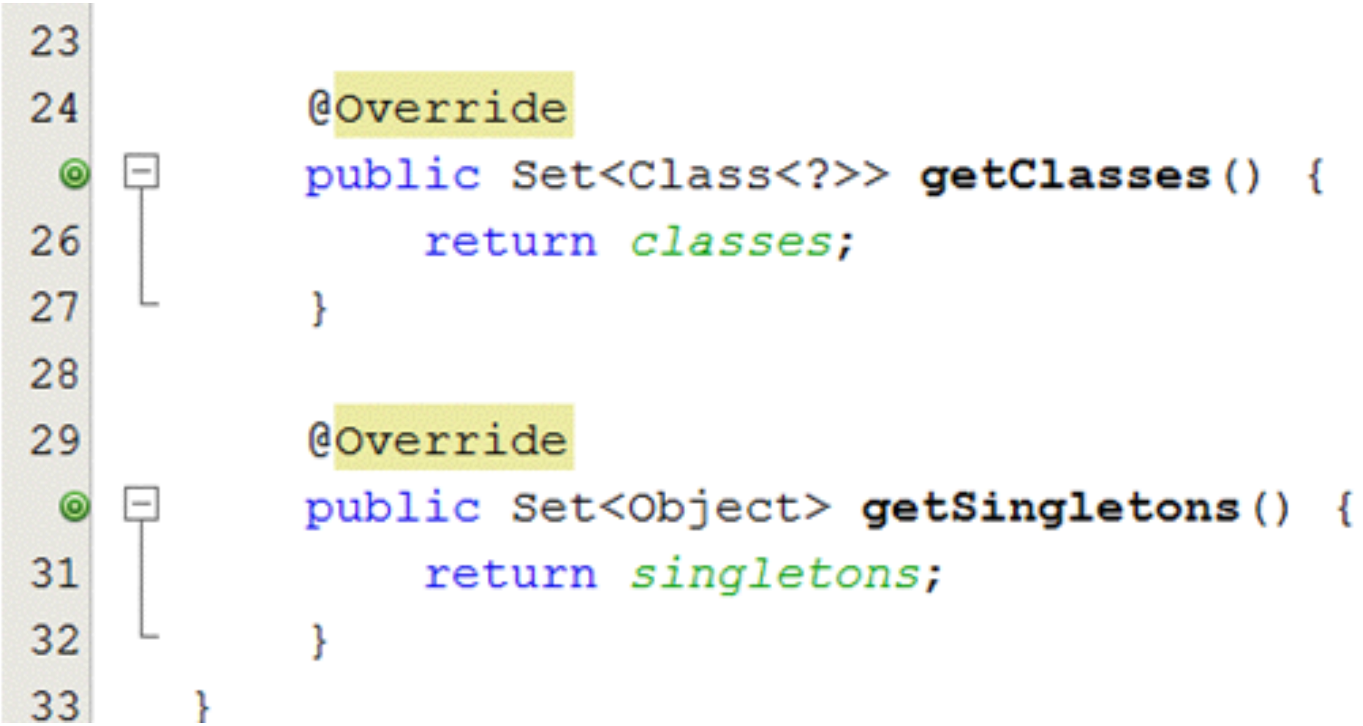
- ApplicationPath might be ignored
- Prepare Sets



```
import java.util.HashSet;
import java.util.Set;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("myapp")
public class MyApplication extends Application {
    private static final Set<Class<?>> classes =
            new HashSet<Class<?>>();
    private static final Set<Object> singletons =
            new HashSet();

    static {
        classes.add(Users.class);
        singletons.add(new Scratch());
    }
```

Application Class

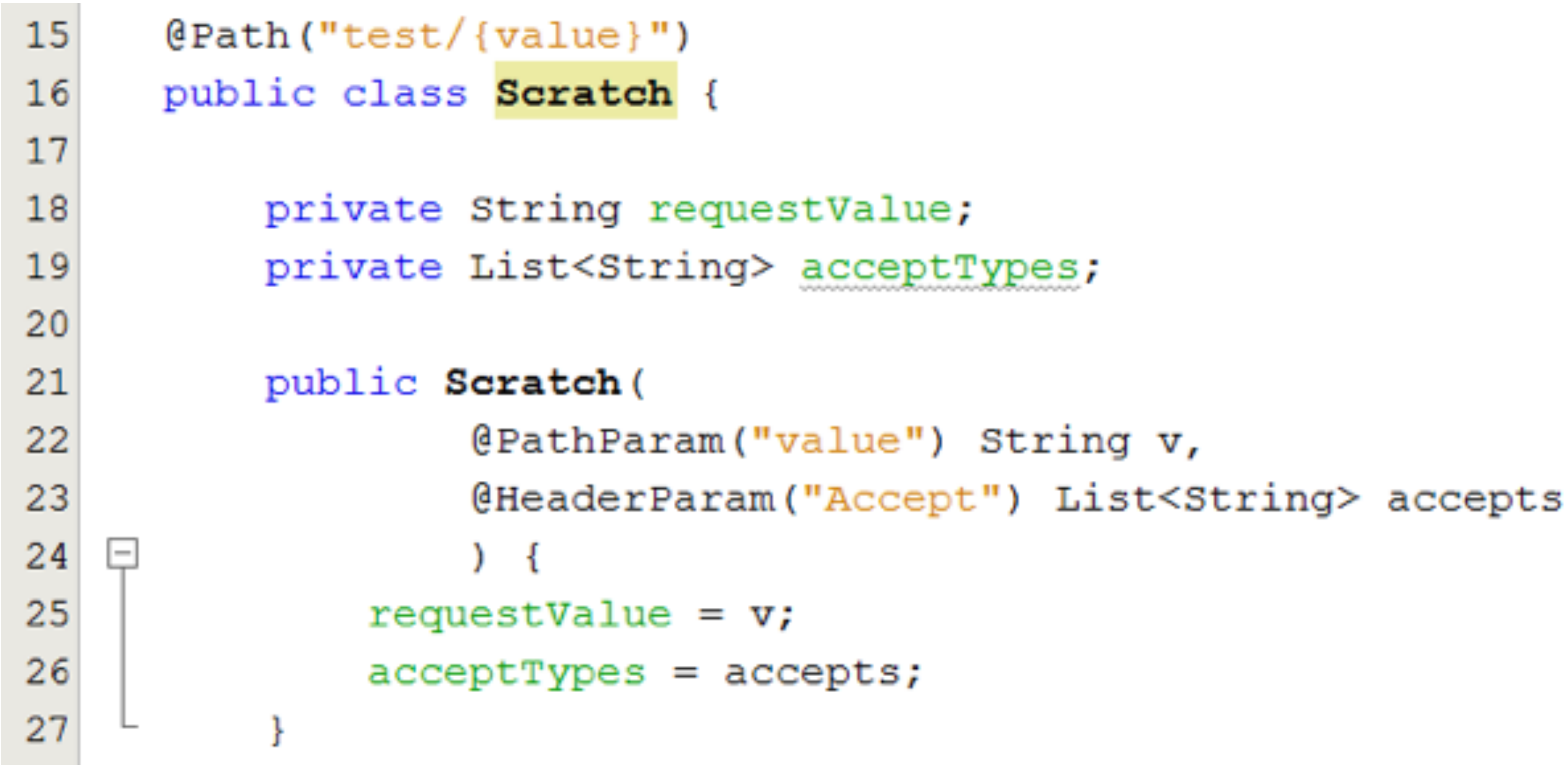
- Return the sets from `getClasses` and `getSingletons` methods



```

    @Override
    public Set<Class<?>> getClasses() {
        return classes;
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

Per-Request Injection

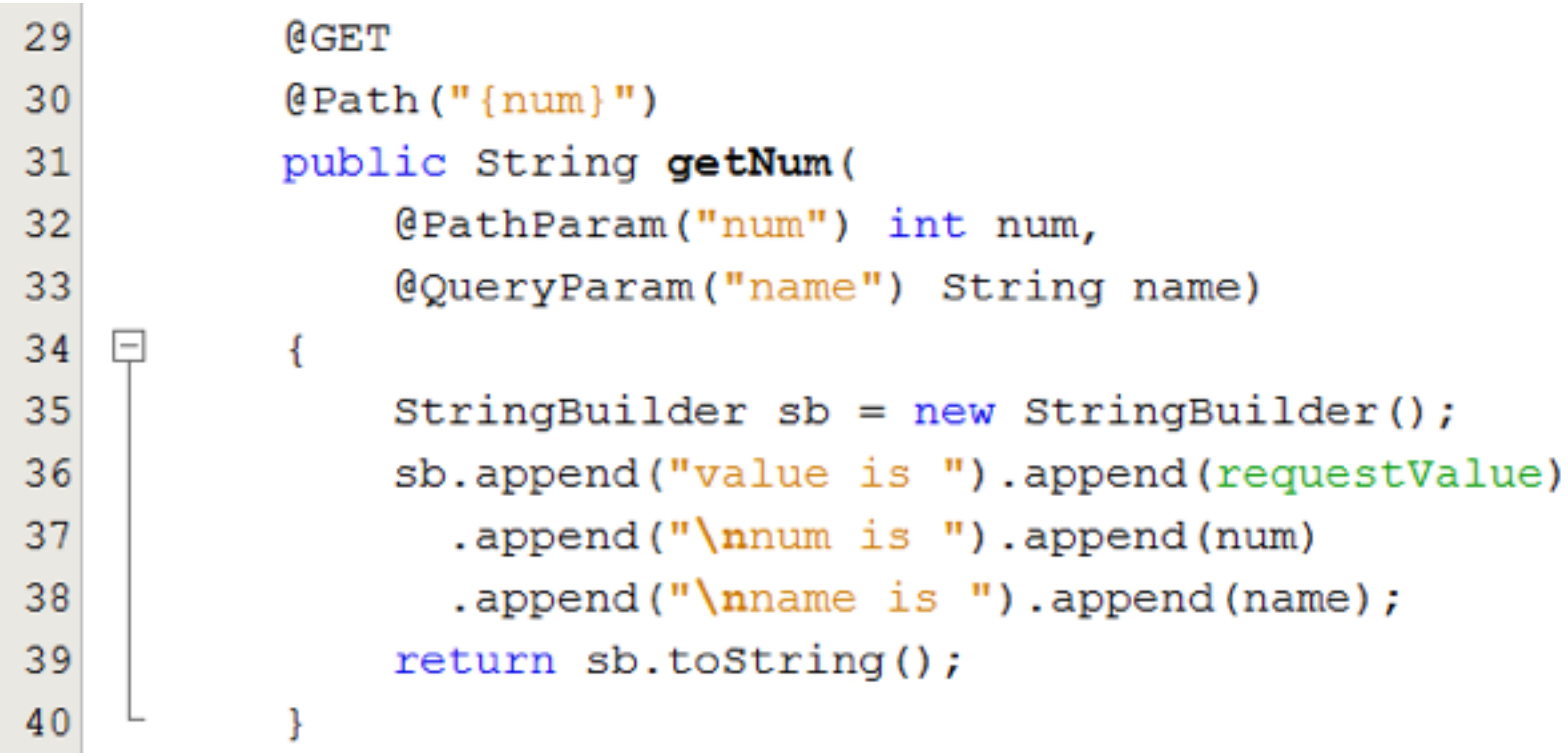
- For per-request objects, common aspects of injection can be centralized in the constructor:



```
@Path("test/{value}")
public class Scratch {

    private String requestValue;
    private List<String> acceptTypes;

    public Scratch(
            @PathParam("value") String v,
            @HeaderParam("Accept") List<String> accepts
            ) {
        requestValue = v;
        acceptTypes = accepts;
    }
```

Per-Request Injection

- Each handler method can obtain additional parameters too



```
@GET
@Path("{num}")
public String getNum(
    @PathParam("num") int num,
    @QueryParam("name") String name)
{
    StringBuilder sb = new StringBuilder();
    sb.append("value is ").append(requestValue)
      .append("\nnum is ").append(num)
      .append("\nname is ").append(name);
    return sb.toString();
}
```

What Can Be Injected?

- Injected values come from:
  - Elements of the URL (query and path parameters)
  - Values in form key/value pairs
  - Generally, textual, sources
- JAX-RS permits injection to types if it can work out how to convert text to the injected value
  - Primitive types and their wrappers (`int`, `Integer`)
  - `Constructor(String)`
  - `static fromString(String)`
  - `static valueOf(String)`
  - `List<T>`, `Set<T>`, `SortedSet<T>` where `T` is one of the above

Default Values for Injected Variables

- Annotation `@DefaultValue` permits specification of a value to be injected if the HTTP request does not provide the injectable element

```
public DataItem getJson(
   @QueryParam("b") @DefaultValue("false")
   boolean b) { …
```

- Default is always specified as a `String`
- Using `@DefaultValue` hides the omission
  - Consider injecting `String`, then `null` indicates value omitted by caller

URI Navigation Scenario #1

- Class carrying @Path annotation is a “resource”
  - Method carries @GET etc.
  - This method responds to the URI

```
/WebContextRoot
    [ /ApplicationPath ]
        /Class Annotated Path
```

← Might be ignored

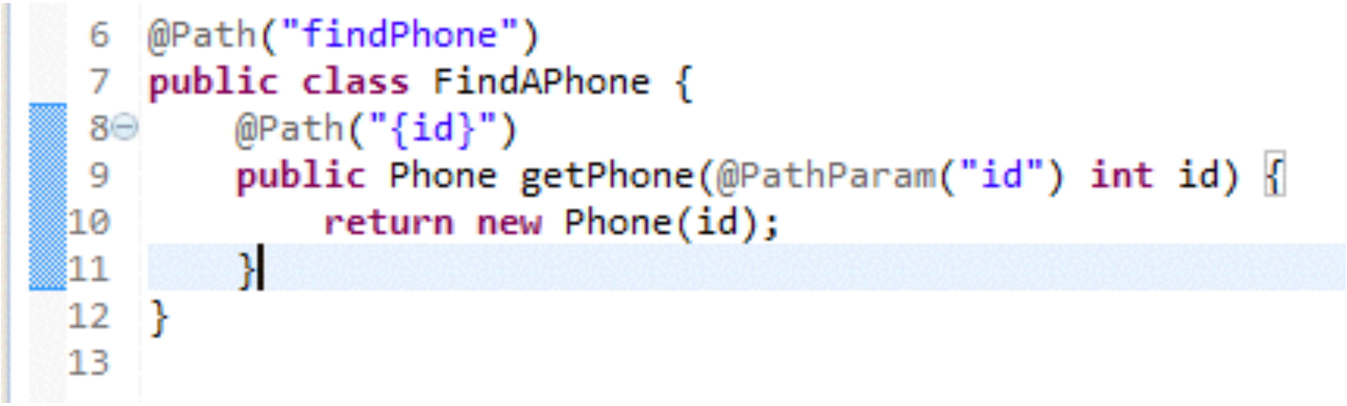
URI Navigation Scenario #2

- Class carries @Path
  - Method carries @GET etc. *and* @Path
  - This method responds to the URI

```
/WebContextRoot
   [ /ApplicationPath ] ← Might be ignored
         /Class Annotated Path
               /Method Annotated Path
```

URI Navigation Scenario #3

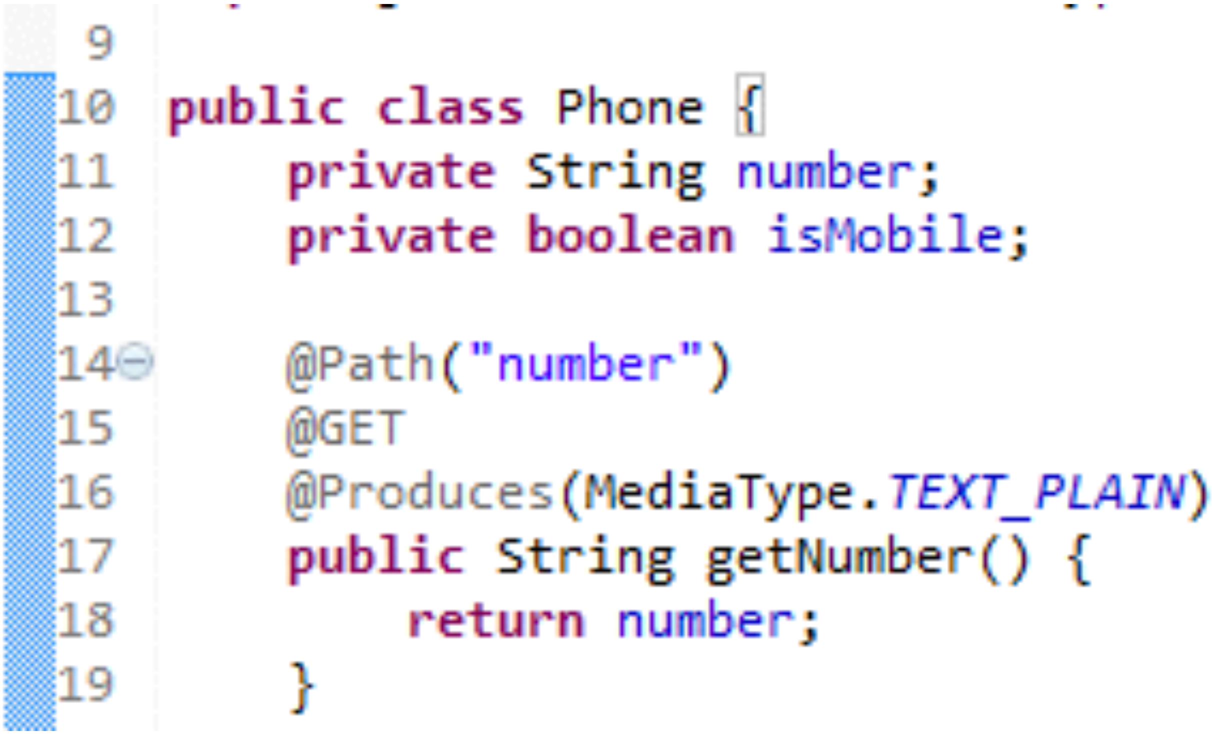
- Class carries @Path
  - Method carries @Path but *not* @GET etc.
  - Method returns an object
  - Returned object has methods annotated with @GET etc.



```java
@Path("findPhone")
public class FindAPhone {
    @Path("{id}")
    public Phone getPhone(@PathParam("id") int id) {
        return new Phone(id);
    }
}
```

URI Navigation Scenario #3

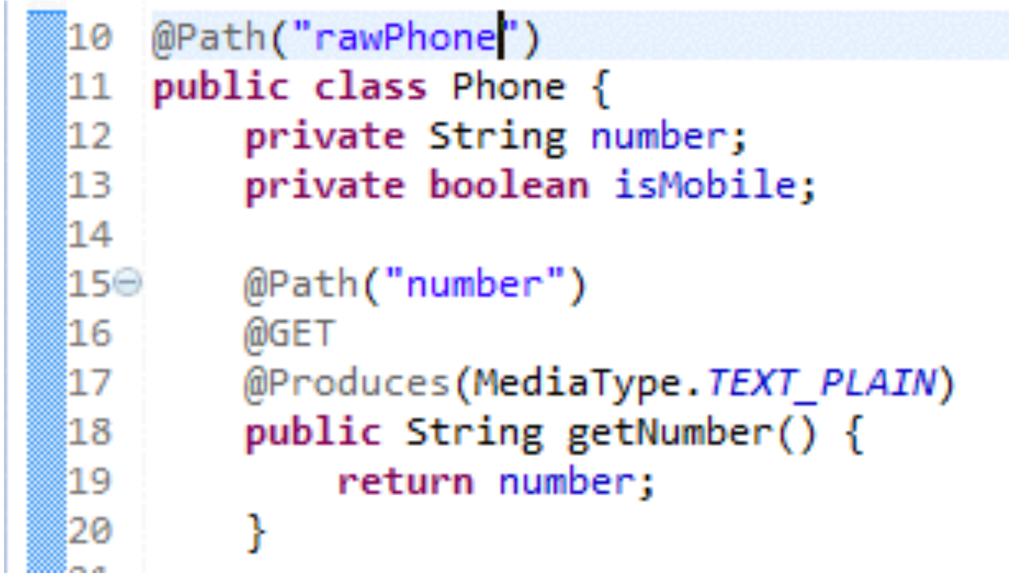
- Returned object has methods annotated with @GET etc.



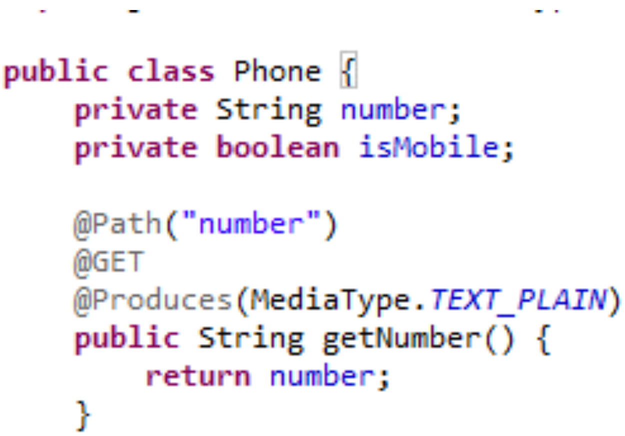
```

public class Phone {
    private String number;
    private boolean isMobile;

    @Path("number")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getNumber() {
        return number;
    }
```

URI Navigation Scenario #3

- Returned object need not be a root resource
  - It does not require annotation with @Path, but this is permitted
  - It does not have to be declared to the application, but this is permitted



```
@Path("rawPhone")
public class Phone {
    private String number;
    private boolean isMobile;

    @Path("number")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getNumber() {
        return number;
    }
```



```
public class Phone {
    private String number;
    private boolean isMobile;

    @Path("number")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getNumber() {
        return number;
    }
```

@Context Annotation

- The `@Context` annotation provides additional injection options
- `@Context` can be used to inject:
  - `UriInfo` – useful for building up URIs
  - `HttpHeaders` – useful for retrieving HTTP Header info
  - `Request` – useful for modifying request / response
  - `Providers` – access to underlying providers
  - `SecurityContext` – security information

Optional @Context Injections

- The `@Context` annotation may provide these injection options if deployed in a web container
- They give access to web container features
  - `ServletContext`
  - `ServletConfig`
  - `HttpServletRequest`
  - `HttpServletResponse`

UriInfo

- Provides access to details of the request that lead JAX-RS to this handler:
  - Path URI and elements thereof
  - Path and Query parameters
  - Resource classes used in servicing the request
  - Means of building paths for returning to the caller: `UriBuilder` objects

UriInfo—UriBuilder

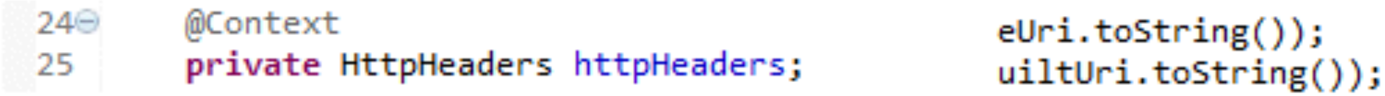
- `UriInfo` gives access to `UriBuilder`s, these:
  - Absolute URI is the URI that led to this method
  - Base URI is the protocol, host, port, web context and application root
- UriBuilder constructs/modifys URIs, allowing all elements to be specified e.g.:
  - urib.path(“customer”).path(“account”)

  → …/customer/account
  - urib.path(“customer”).queryParam(“name”, “Sheila”)

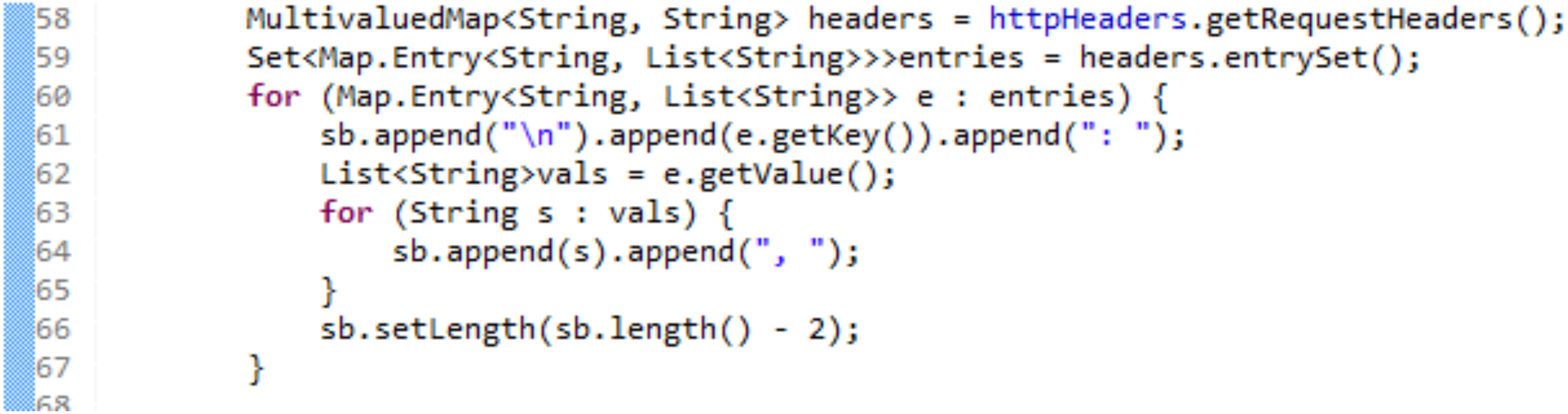
  → …/customer?name=Sheila

HttpHeaders

- Access all headers, cookies, language etc.
- Enumerate all request headers in a `MultivaluedMap`:



```
    @Context                                     eUri.toString());
    private HttpHeaders httpHeaders;             uiltUri.toString());
```



```
        MultivaluedMap<String, String> headers = httpHeaders.getRequestHeaders();
        Set<Map.Entry<String, List<String>>>entries = headers.entrySet();
        for (Map.Entry<String, List<String>> e : entries) {
            sb.append("\n").append(e.getKey()).append(": ");
            List<String>vals = e.getValue();
            for (String s : vals) {
                sb.append(s).append(", ");
            }
            sb.setLength(sb.length() - 2);
        }
```

- Prefer `@HeaderParam` for individual headers

SecurityContext

- `SecurityContext` provides:
  - `getPrincipal()`

  → gives `Princpal`, identifying user
  - `isUserInRole(String role)`

  → true if the user is in this role
  - `isSecure()`

  → true if https or similar secure channel
  - `getAuthenticationScheme()`

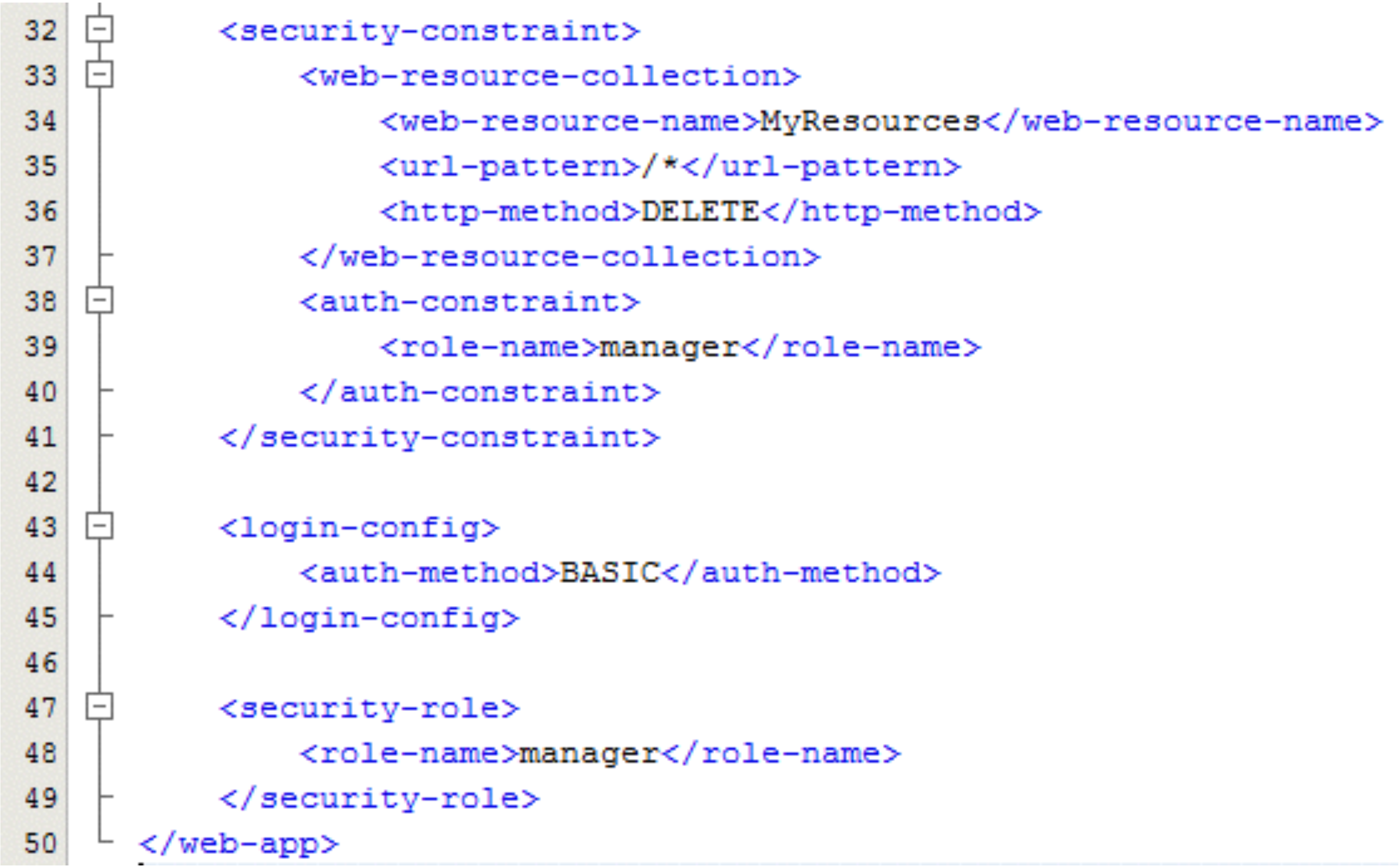
  → Indicates login method (BASIC, DIGEST, etc.)

Java EE Web Security Basics

- JAX-RS uses Java EE Web security
- Security constraints define URL ranges and HTTP methods that require authentication and the roles that may have access
- Login configuration indicates how authentication is to be performed
  - BASIC, DIGEST, FORM, CLIENT-CERT
- Roles must be declared
- Container authentication must be configured (this is server specific)

Java EE Web Security Configuration

- Configure web.xml



```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>MyResources</web-resource-name>
            <url-pattern>/*</url-pattern>
            <http-method>DELETE</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>

    <security-role>
        <role-name>manager</role-name>
    </security-role>
</web-app>
```

Servlet Context

- ServletContext provides access to container features:
  - InitParameters — Configuration from the deployment descriptor (compare with command line arguments)
  - Resources — E.g. reading files from the deployed .war file, or alongside the Java class files
  - Listeners — lifecycle events

Lab 2

Response Features

- `Response` object offers explicit control of response

- Construct using factory methods of inner class: `Response.ResponseBuilder`

- `ResponseBuilder` provides static methods that modify the response in preparation, e.g.:
  - `status(<code>)`
  - `header(<headername>, <value>)`
  - `ok`

Response Examples

- To send successful response with encoded object body:

```
return Response.ok(myJaxBObject)
  .type(MediaType.APPLICATION_XML).build();
```

- To generate a resource not found (404) response with body containing text “Not found”:

```
return Response.ok("Not found")
   .status(Response.Status.NOT_FOUND)
   .build();
```

- Note, status may be specified numerically, or using the `Response.Status` enumeration

Response Factory Methods

- `Response` has factory methods that create `ResponseBuilder` objects:
  - `ok()` — prepares 200 OK
  - `created(URI)` — prepares 201 Created
    - Puts URI in a `Location:` header
  - `noContent()` — prepares 204 No Content
  - `status(int)` — prepares the given response code
  - `temporaryRedirect(URI)` — prepares 307 Temporary Redirect
  - And others…

ResponseBuilder Methods

- Some key `ResponseBuilder` methods:
  - `.build()` method constructs the `Response` object ready for return by the method
  - `.entity(<content>)` sets the body contents
  - `.cookie(<newCookie>…)` adds cookies
  - `.header(“Name: value”)` sets a general header
  - `.type(MediaType.XXX)` sets the content type

Returning XML From JAX-RS

- Create a class that defines the data structure to be returned
  - Should follow Java Beans conventions
  - Generally requires JAXB `@XmlRootElement` on class
  - May also return `JAXBElement<Type>` wrapper around unannotated object

- Define the web method as:
  - `@Produces(MediaType.APPLICATION_XML)`
  - Returns a Java Object
  - Or, returns `Response` with entity set to the object

```
Response.ok(new MyJaxBThing(<args>)).build()
```

Handling Lists in XML Returns

- JAX-B does not automatically handle List types
- Can create a wrapper class:
  - `@XmlRootEntity public class ListOfCustomers…`
- Or can use the JAX-RS `GenericEntity`
  - Note, must create a subclass (usually anonymous)
  - `GenericEntity<List<Customer>> ge = new GenericEntity<List<Customer>>(lc){};`

More Content Types

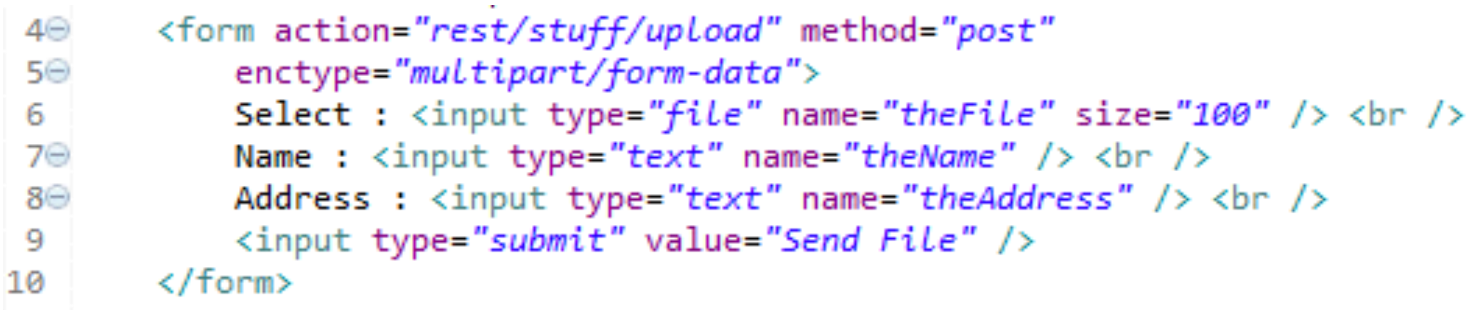
- Return of several types is handled automatically by JAX-RS:
  - Plain text — return a `String`
  - XML — return an `@XmlRootElement` or `JAXBElement<Type>`
  - Binary streams, including images — return `InputStream`, `byte[]` (or some others)
- Other types can be added as extensions using `Provider`s
  - Some additional providers might be built-in in any given implementation

More Content Types

- JSON is typically provided by Jackson library
  - Supplied with RESTEasy and GlassFish (but not with Jersey itself)
- For example, RESTEasy and Jersey include providers for `multipart/form-data`
  - These are not part of the JAX-RS standard, and implementations are not compatible

Multipart Form Data With RESTEasy

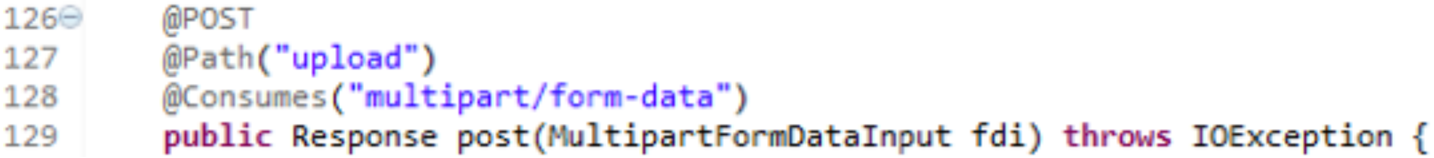
- Prepare a form to send multipart data:



```
<form action="rest/stuff/upload" method="post"
    enctype="multipart/form-data">
    Select : <input type="file" name="theFile" size="100" /> <br />
    Name : <input type="text" name="theName" /> <br />
    Address : <input type="text" name="theAddress" /> <br />
    <input type="submit" value="Send File" />
</form>
```

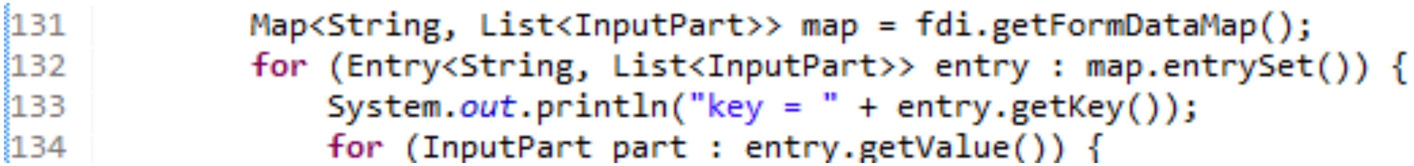
Multipart Form Data With RESTEasy

- REST handler method responds to POST, entity argument is a MultipartFormDataInput



```
@POST
@Path("upload")
@Consumes("multipart/form-data")
public Response post(MultipartFormDataInput fdi) throws IOException {
```

- MultipartFormDataInput contains a Map with all the uploaded data in InputParts

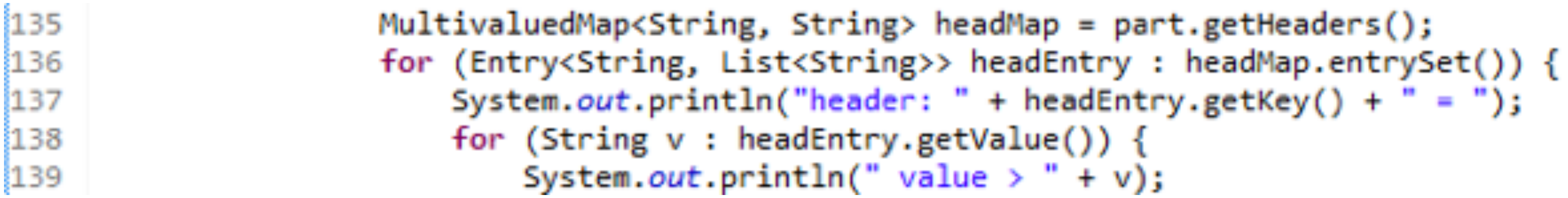


```
Map<String, List<InputPart>> map = fdi.getFormDataMap();
for (Entry<String, List<InputPart>> entry : map.entrySet()) {
    System.out.println("key = " + entry.getKey());
    for (InputPart part : entry.getValue()) {
```

  - Map key is the name of the part
  - Map value is a List<InputPart>

Multipart Form Data With RESTEasy

- Each part contains headers

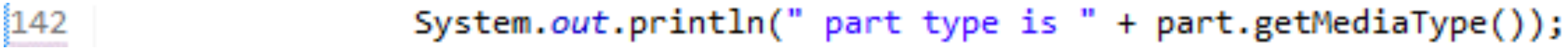


```
MultivaluedMap<String, String> headMap = part.getHeaders();
for (Entry<String, List<String>> headEntry : headMap.entrySet()) {
    System.out.println("header: " + headEntry.getKey() + " = ");
    for (String v : headEntry.getValue()) {
        System.out.println(" value > " + v);
```

- One very important header is `Content-Disposition`; this specifies the field name that this `InputPart` represents, and the file name of an uploaded file:
  - `Content-Disposition = form-data; name="theFile"; filename=“uploadfile.txt"`

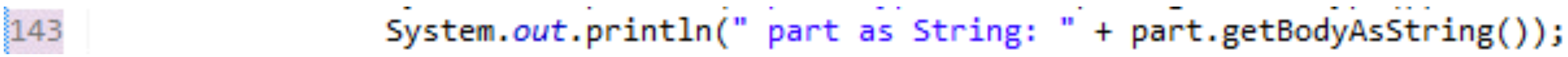
Multipart Form Data With RESTEasy

- `InputPart` knows the media type of its data:



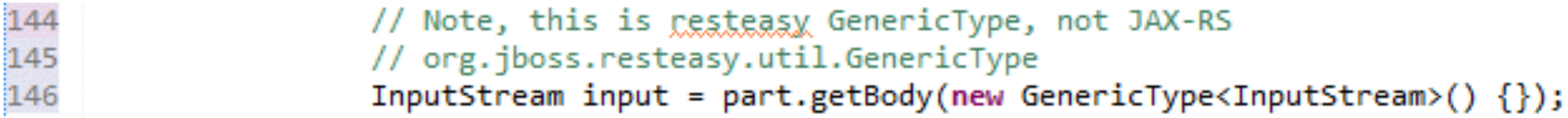
```
System.out.println(" part type is " + part.getMediaType());
```

- For text type media, a convenience method returns the content as a `String`:



```
System.out.println(" part as String: " + part.getBodyAsString());
```

- For other media types, generic conversion is possible to appropriate types:

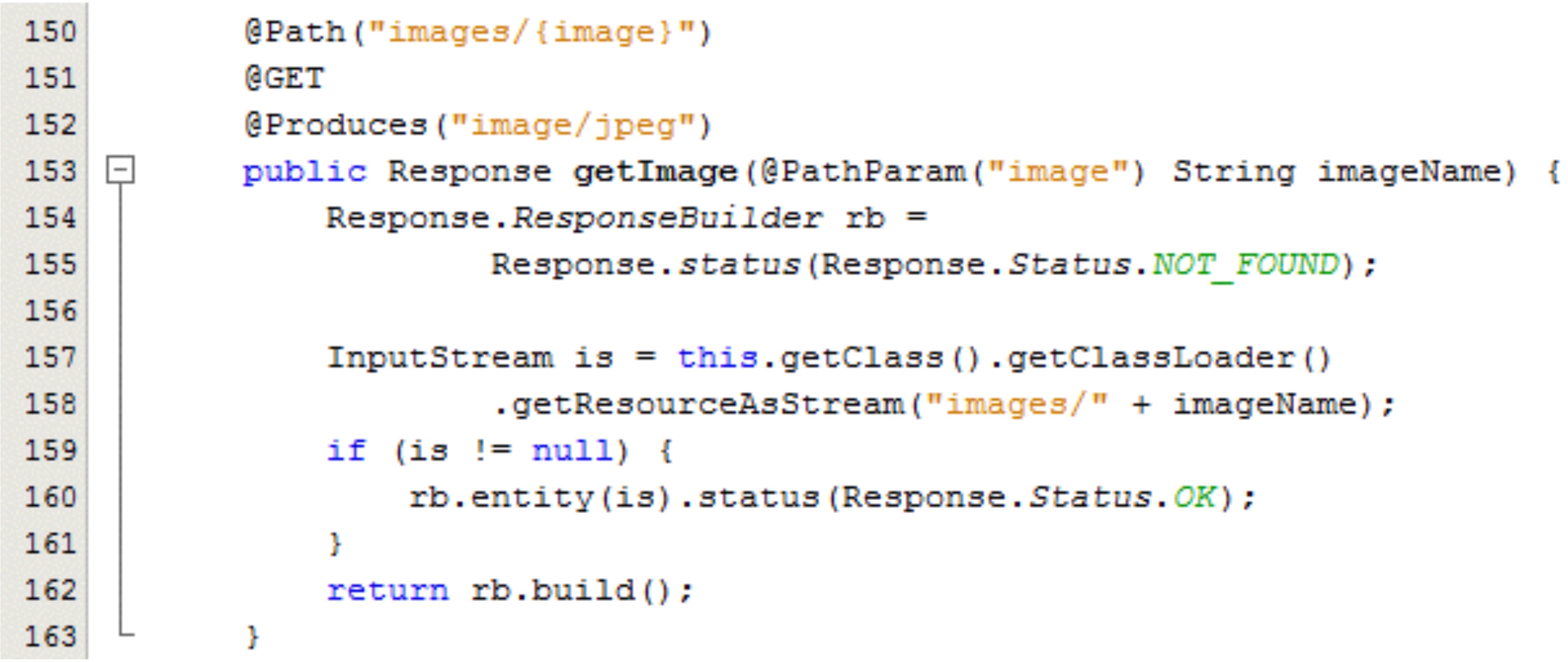


```
// Note, this is resteasy GenericType, not JAX-RS
// org.jboss.resteasy.util.GenericType
InputStream input = part.getBody(new GenericType<InputStream>() {});
```

  - Note `org.jboss.resteasy.util.GenericType`, be aware there is a `GenericType` class in JAX-RS too!

Returning InputStream

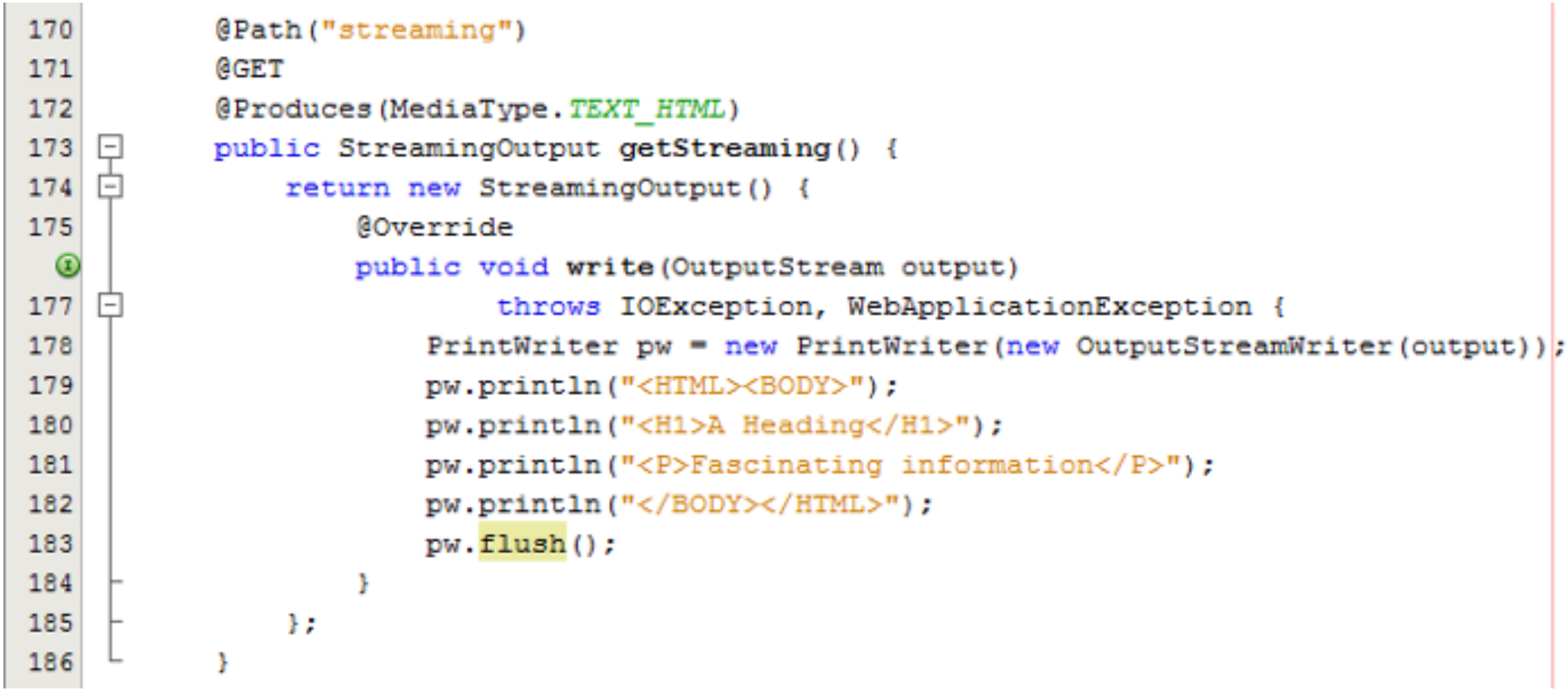
- A resource can return an `InputStream` directly:



```
@Path("images/{image}")
@GET
@Produces("image/jpeg")
public Response getImage(@PathParam("image") String imageName) {
    Response.ResponseBuilder rb =
            Response.status(Response.Status.NOT_FOUND);

    InputStream is = this.getClass().getClassLoader()
            .getResourceAsStream("images/" + imageName);
    if (is != null) {
        rb.entity(is).status(Response.Status.OK);
    }
    return rb.build();
}
```

Accessing the OutputStream

- May implement `StreamingOutput`:



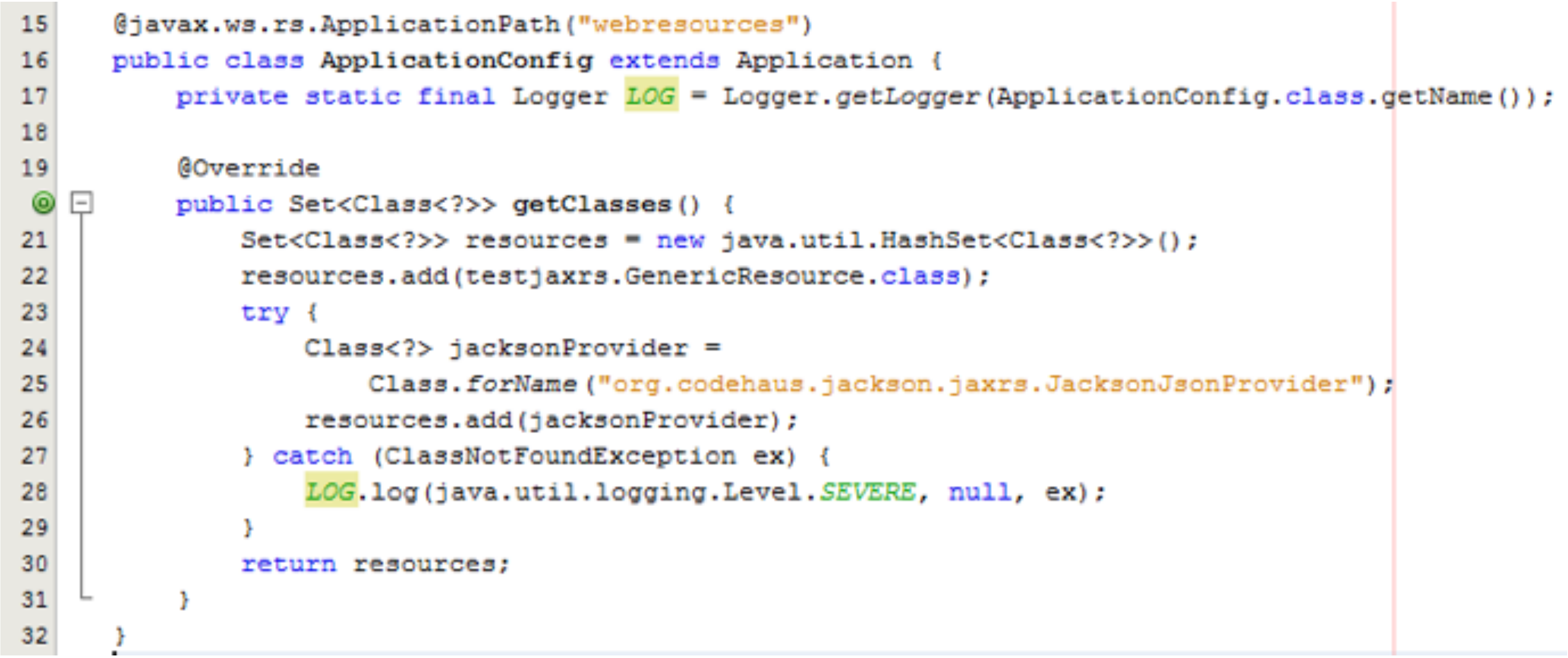
```
@Path("streaming")
@GET
@Produces(MediaType.TEXT_HTML)
public StreamingOutput getStreaming() {
    return new StreamingOutput() {
        @Override
        public void write(OutputStream output)
                throws IOException, WebApplicationException {
            PrintWriter pw = new PrintWriter(new OutputStreamWriter(output));
            pw.println("<HTML><BODY>");
            pw.println("<H1>A Heading</H1>");
            pw.println("<P>Fascinating information</P>");
            pw.println("</BODY></HTML>");
            pw.flush();
        }
    };
}
```

Handling JSON Content

- JSON is not part of the JAX-RS standarrd
- It is typically provided by “Jackson” library
  - Included and configured with RESTEasy
  - Must configured with Jersey
- Jersey does not include Jackson library, but GlassFish does
  - If using Jersey without GlassFish, must download and install, Jackson
  - This might be true of other web containers and JAX-RS implementations too
- NetBeans builds code assuming Jackson will be on the target server

Configuring Jersey to Use Jackson

- Might not have non-JAX-RS classes to compile against
  - Need not use `Class.forName` if Jackson is on build path



```
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {
    private static final Logger LOG = Logger.getLogger(ApplicationConfig.class.getName());

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<Class<?>>();
        resources.add(testjaxrs.GenericResource.class);
        try {
            Class<?> jacksonProvider =
                Class.forName("org.codehaus.jackson.jaxrs.JacksonJsonProvider");
            resources.add(jacksonProvider);
        } catch (ClassNotFoundException ex) {
            LOG.log(java.util.logging.Level.SEVERE, null, ex);
        }
        return resources;
    }
}
```

Lab 3

Receiving the Entity

- Body of the HTTP request is called the “entity”
- This is coded according to the `@Consumes` type
  - If not, then JAX-RS will not invoke the method
- A single argument to the handler method may be un-annotated:

```
public Response newCustomer(Cust c) …
```

- HTTP entity will be decoded based on the `@Consumes` type and passed into the method
- Content types handled same as for return values

Non-Standard HTTP Methods

- HTTP requests can be made with other methods, not just GET etc.
- Use this capability thoughtfully (i.e. very rarely!)
- JAX-RS supports this with two steps

Non-Standard HTTP Methods

- Define a new annotation:

```
@Target({ElementType.CONSTRUCTOR,
ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod(“ACTION")
public @interface ACTION{
}
```

- Apply the annotation to a resource method

```
@ACTION
public Response doAnAction() …
```

Custom Conversions: Provider

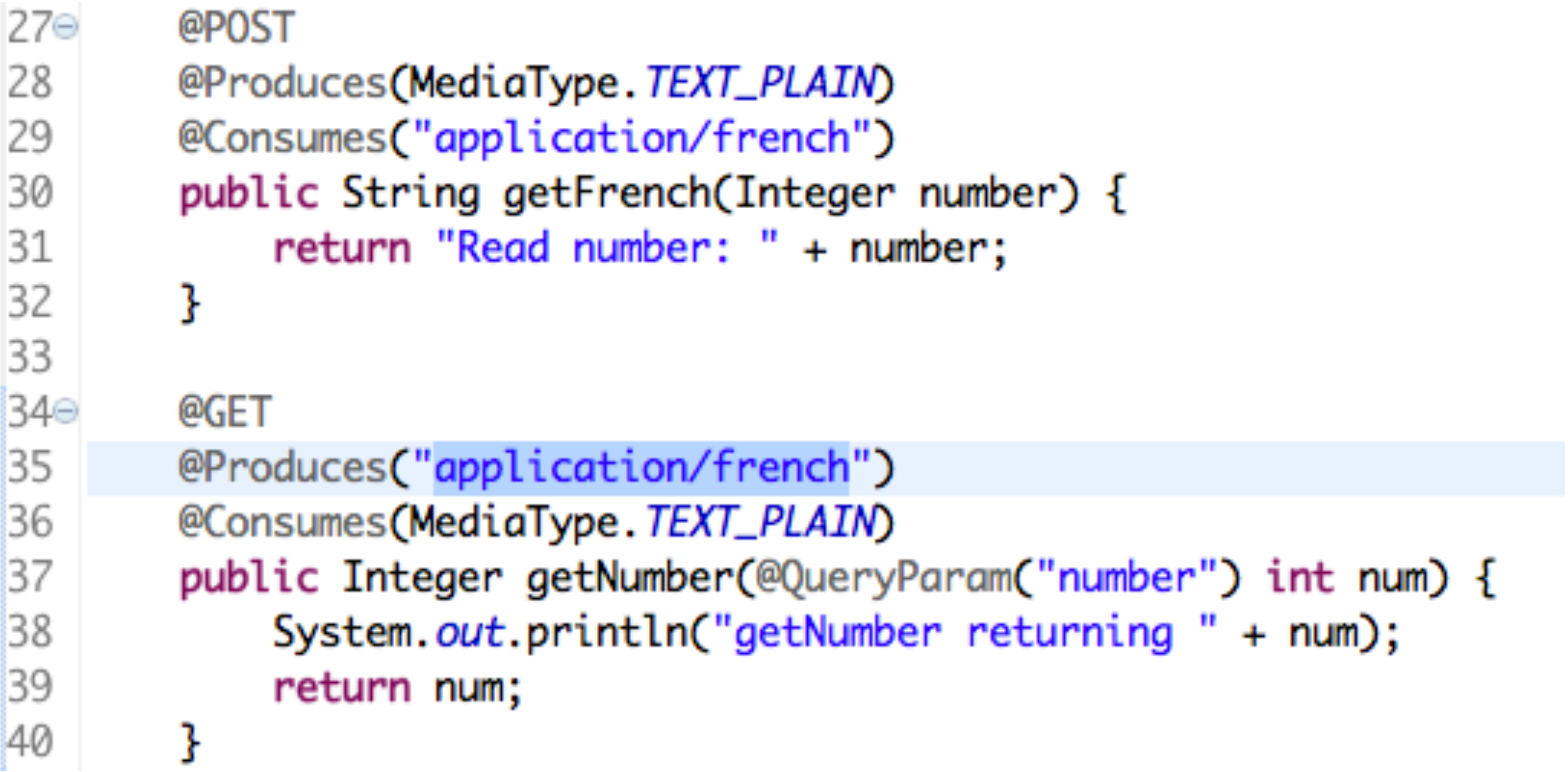
- JAX-RS allows registering handlers for converting custom types
  - Called Providers
- Parsing is performed by a `MessageBodyReader<T>`
  - `<T>` is the type being created from the input
- Annotated `@Provider`
- Included in classes or singletons reported by `Application`
- Implement `isReadable` and `readFrom` methods

Custom Conversions: Provider

- Generating output is performed by a `MessageBodyWriter<T>`
  - `<T>` is the type being converted to output
- Annotated `@Provider`
  - Included in classes or singletons reported by `Application`
  - Can be the same class as the reader
- Implement `isWriteable`, `getSize` and `writeTo` methods
  - `flush` the output in `writeTo`

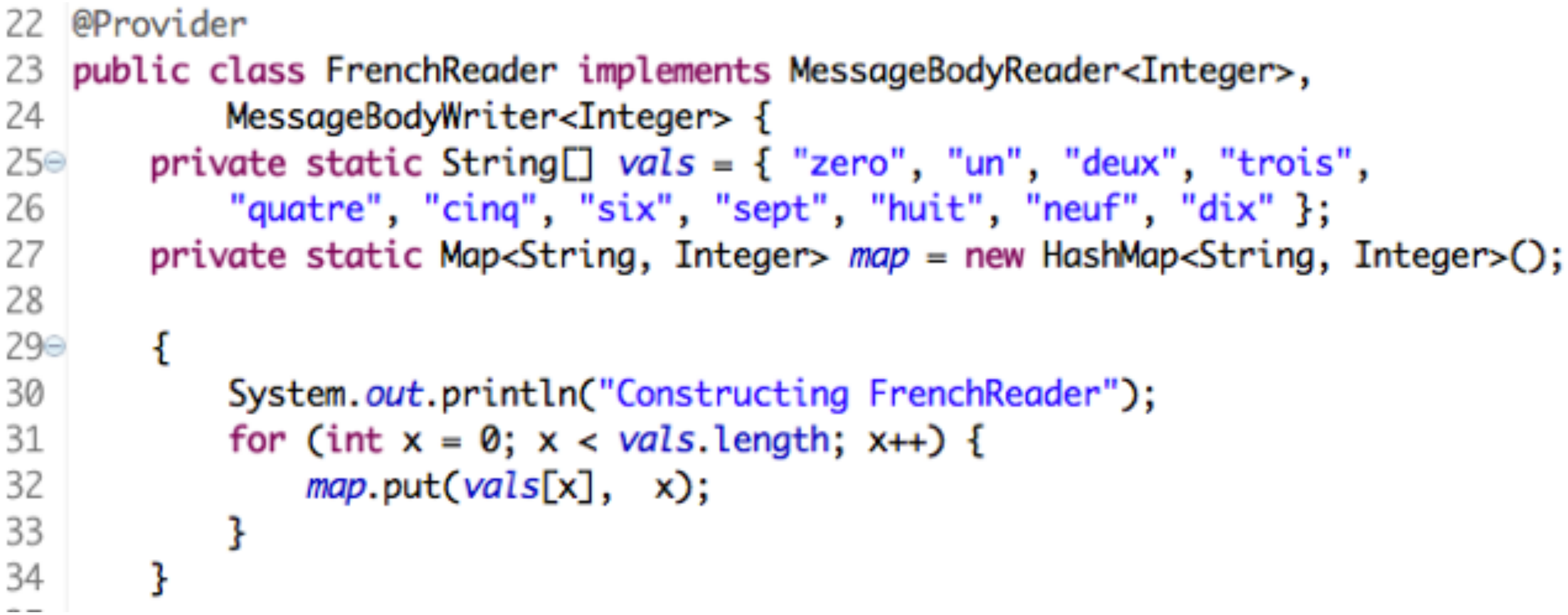
Custom Provider Example

- Custom content types may be listed
  - These will trigger search for custom `Providers`



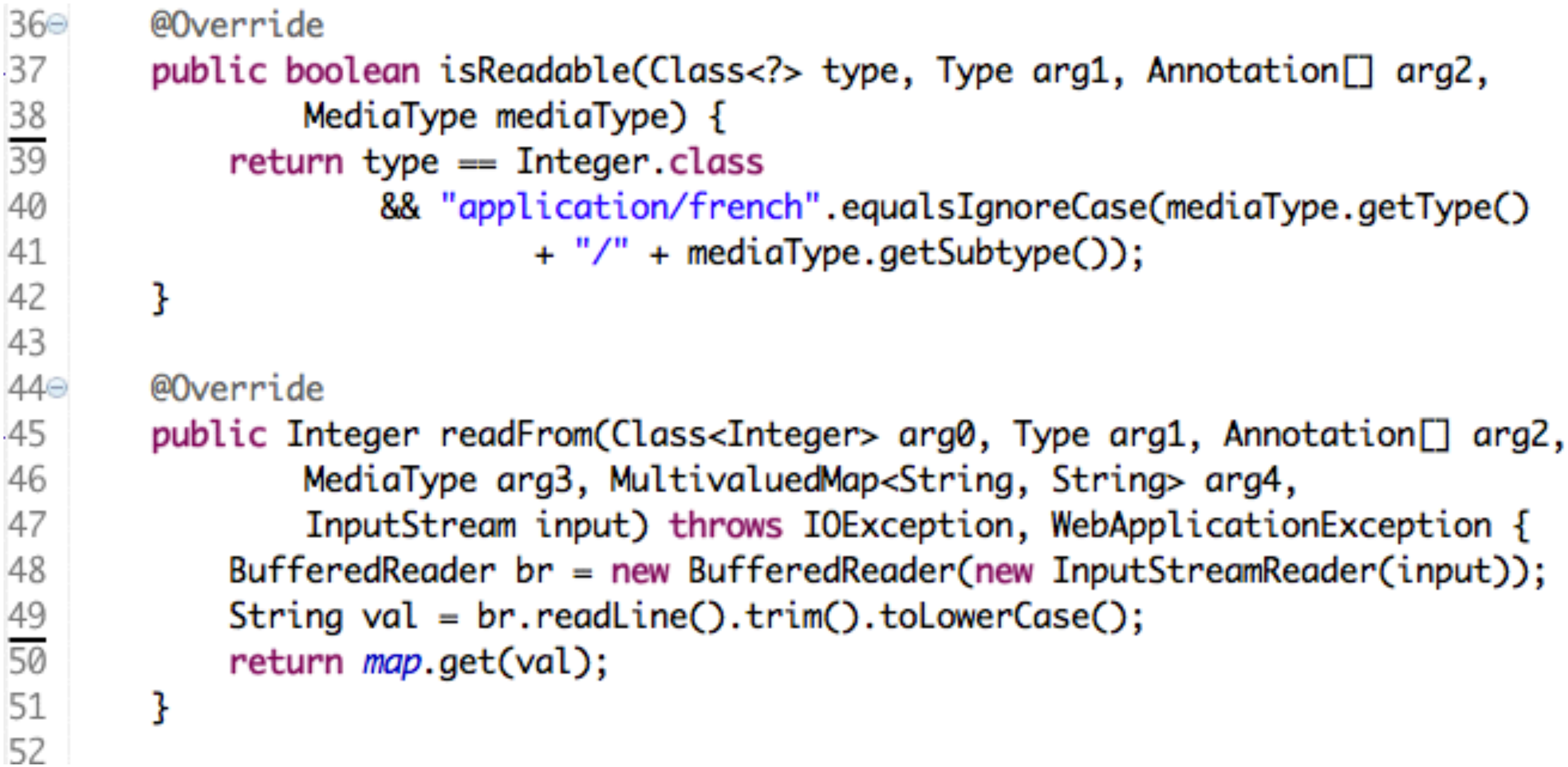
```
    @POST
    @Produces(MediaType.TEXT_PLAIN)
    @Consumes("application/french")
    public String getFrench(Integer number) {
        return "Read number: " + number;
    }

    @GET
    @Produces("application/french")
    @Consumes(MediaType.TEXT_PLAIN)
    public Integer getNumber(@QueryParam("number") int num) {
        System.out.println("getNumber returning " + num);
        return num;
    }
```

Prepare the Provider



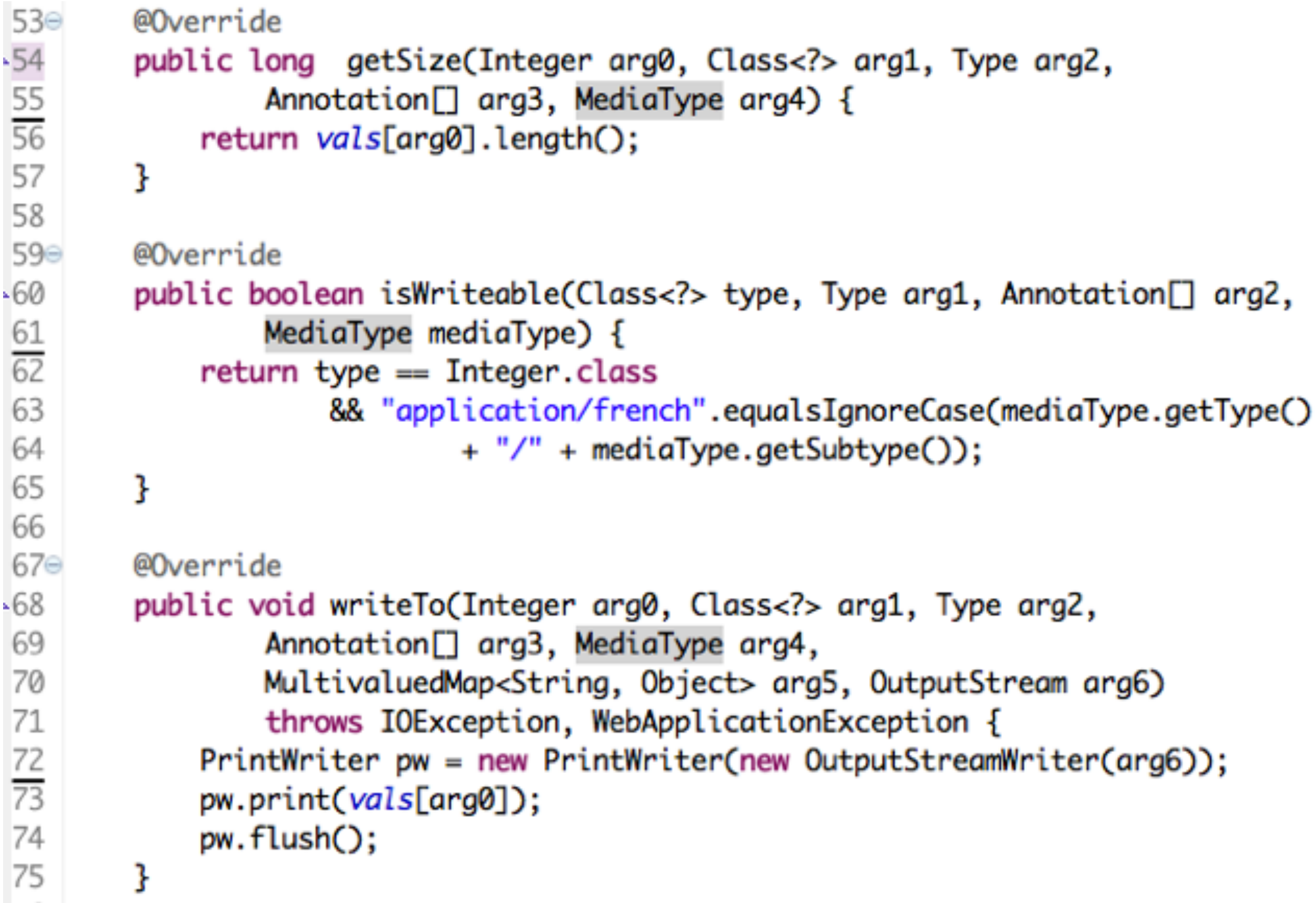
```
@Provider
public class FrenchReader implements MessageBodyReader<Integer>,
        MessageBodyWriter<Integer> {
    private static String[] vals = { "zero", "un", "deux", "trois",
        "quatre", "cinq", "six", "sept", "huit", "neuf", "dix" };
    private static Map<String, Integer> map = new HashMap<String, Integer>();

    {
        System.out.println("Constructing FrenchReader");
        for (int x = 0; x < vals.length; x++) {
            map.put(vals[x],  x);
        }
    }
```

Implement MessageBodyReader



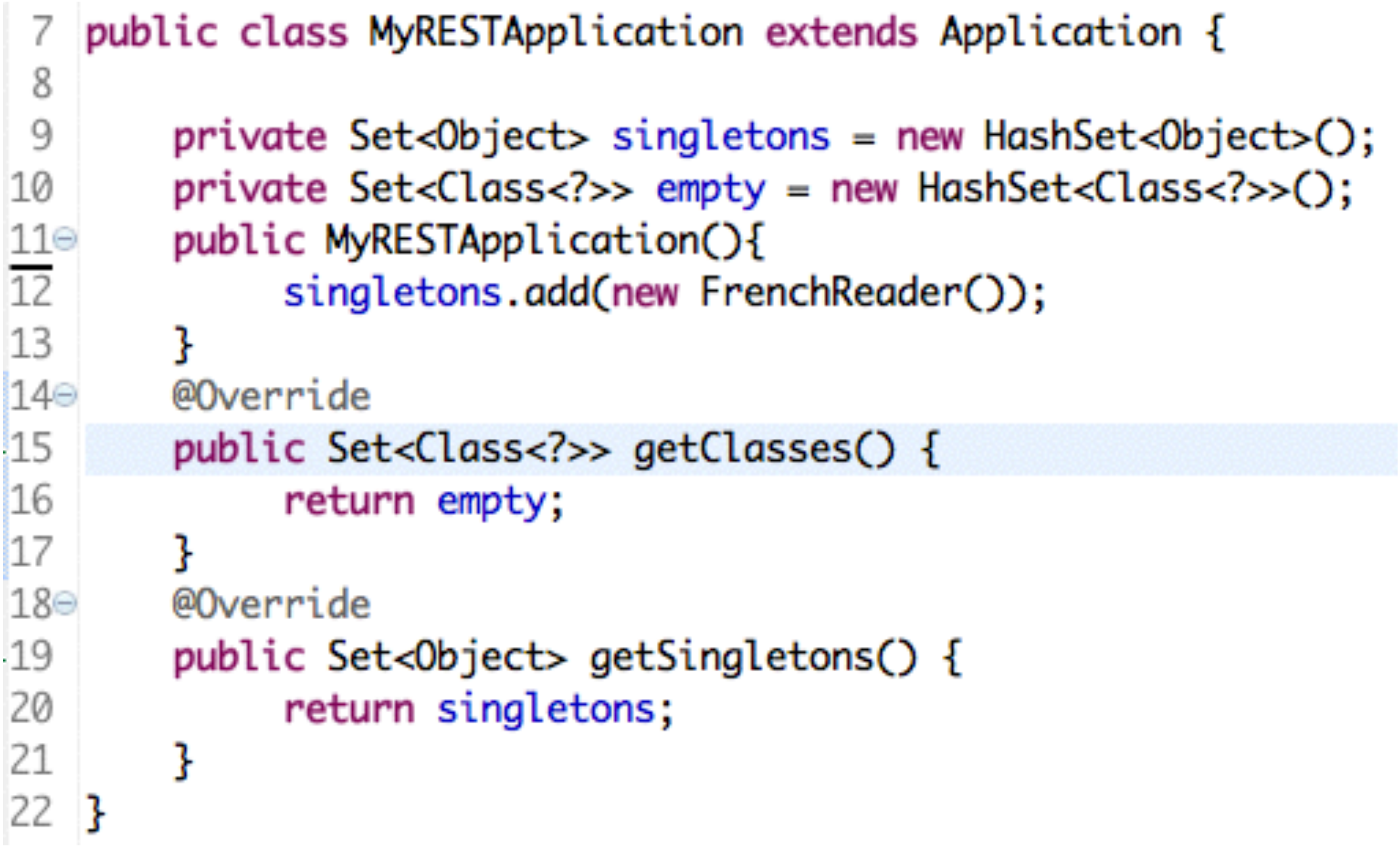
```java
    @Override
    public boolean isReadable(Class<?> type, Type arg1, Annotation[] arg2,
            MediaType mediaType) {
        return type == Integer.class
                && "application/french".equalsIgnoreCase(mediaType.getType()
                        + "/" + mediaType.getSubtype());
    }

    @Override
    public Integer readFrom(Class<Integer> arg0, Type arg1, Annotation[] arg2,
            MediaType arg3, MultivaluedMap<String, String> arg4,
            InputStream input) throws IOException, WebApplicationException {
        BufferedReader br = new BufferedReader(new InputStreamReader(input));
        String val = br.readLine().trim().toLowerCase();
        return map.get(val);
    }

```

Implement MessageBodyWriter



```
    @Override
    public long  getSize(Integer arg0, Class<?> arg1, Type arg2,
            Annotation[] arg3, MediaType arg4) {
        return vals[arg0].length();
    }

    @Override
    public boolean isWriteable(Class<?> type, Type arg1, Annotation[] arg2,
            MediaType mediaType) {
        return type == Integer.class
                && "application/french".equalsIgnoreCase(mediaType.getType()
                        + "/" + mediaType.getSubtype());
    }

    @Override
    public void writeTo(Integer arg0, Class<?> arg1, Type arg2,
            Annotation[] arg3, MediaType arg4,
            MultivaluedMap<String, Object> arg5, OutputStream arg6)
            throws IOException, WebApplicationException {
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(arg6));
        pw.print(vals[arg0]);
        pw.flush();
    }
```

Advertise The Producer



```java
public class MyRESTApplication extends Application {

    private Set<Object> singletons = new HashSet<Object>();
    private Set<Class<?>> empty = new HashSet<Class<?>>();
    public MyRESTApplication(){
        singletons.add(new FrenchReader());
    }
    @Override
    public Set<Class<?>> getClasses() {
        return empty;
    }
    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

Lab 4

Handling Errors

- Code can use ResponseBuilder to create error codes directly
- Application code can throw exceptions
  - JAX-RS will report this in a 5XX errors
- Application can throw a `WebApplicationException`
  - Can send a Response object with this for more control

Throwing a WebApplicationException

- `javax.ws.rs.WebApplicationException` is a `RuntimeException`
  - Therefore, no throws clause required
- Constructors take:
  - Status code (int)
  - `Response`
  - `Throwable`
  - `Throwable` and status
  - `Throwable` and `Response`

```
throw new WebApplicationException(
 Response.status(400).entity("Failed!").build());
```

Handling Errors

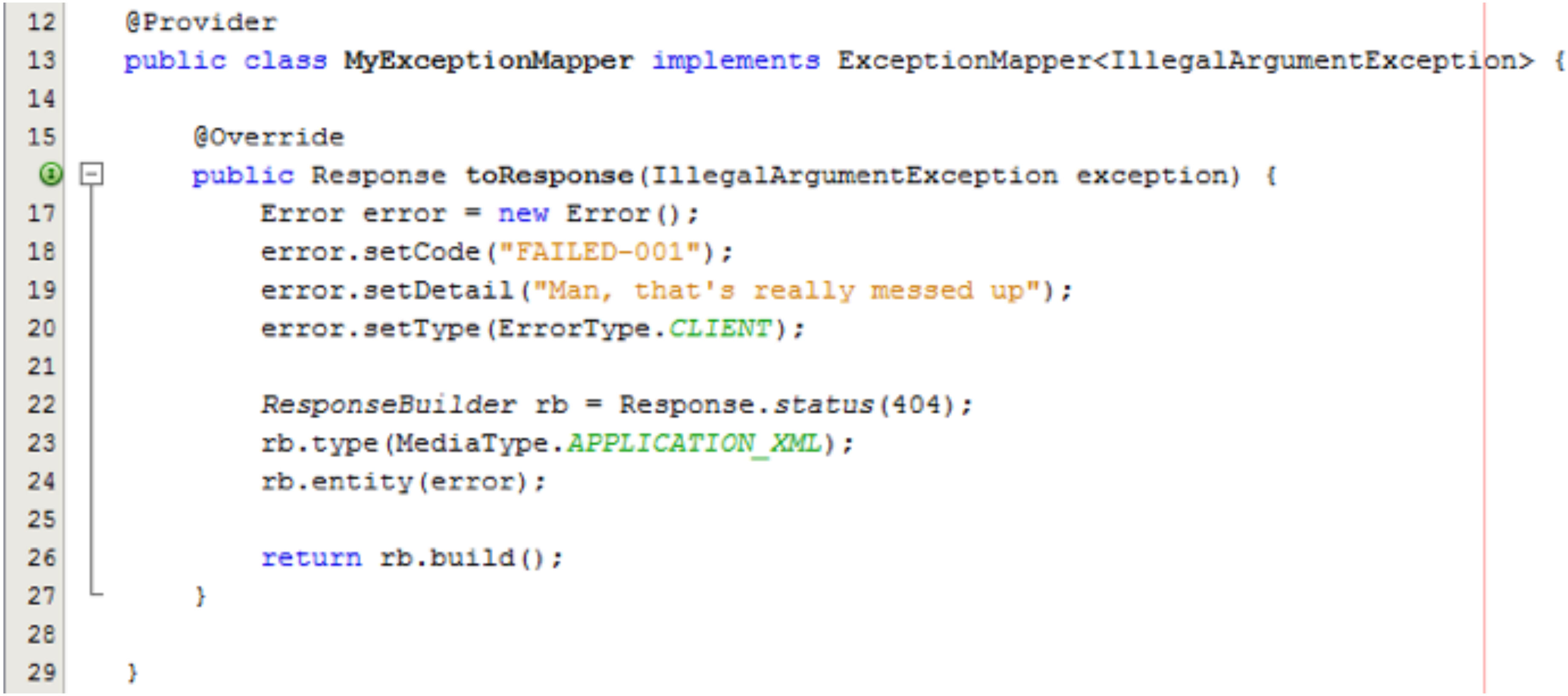
- Routing-type errors are handled by JAX-RS by default
  - E.g. unable to match URI path, HTTP method, and media types to a handler method
  - Internally, these throw exceptions, though the exception type is implementation dependent

- Control exception reporting by implementing an `ExceptionMapper`
  - Convert application exception into a `Response`
  - Also, convert system exception into a `Response`

ExceptionMapper<T>

- An `ExceptionMapper` takes control when JAX-RS infrastructure, or the application code throws an exception
- Exception mapper converts the exception object into a `Response`
- Exception mapper classes:
  - Implement ExceptionMapper<T>, where T is the class of exception to be converted
  - Annotated `@Provider`
  - Registered with the `Application` class (if used)

Implementing ExceptionMapper



```
@Provider
public class MyExceptionMapper implements ExceptionMapper<IllegalArgumentException> {

    @Override
    public Response toResponse(IllegalArgumentException exception) {
        Error error = new Error();
        error.setCode("FAILED-001");
        error.setDetail("Man, that's really messed up");
        error.setType(ErrorType.CLIENT);

        ResponseBuilder rb = Response.status(404);
        rb.type(MediaType.APPLICATION_XML);
        rb.entity(error);

        return rb.build();
    }

}
```

Introduction to Bean Validation

Bean Validation Framework

- Bean validation allows the use of annotations to specify constraints on fields of a bean

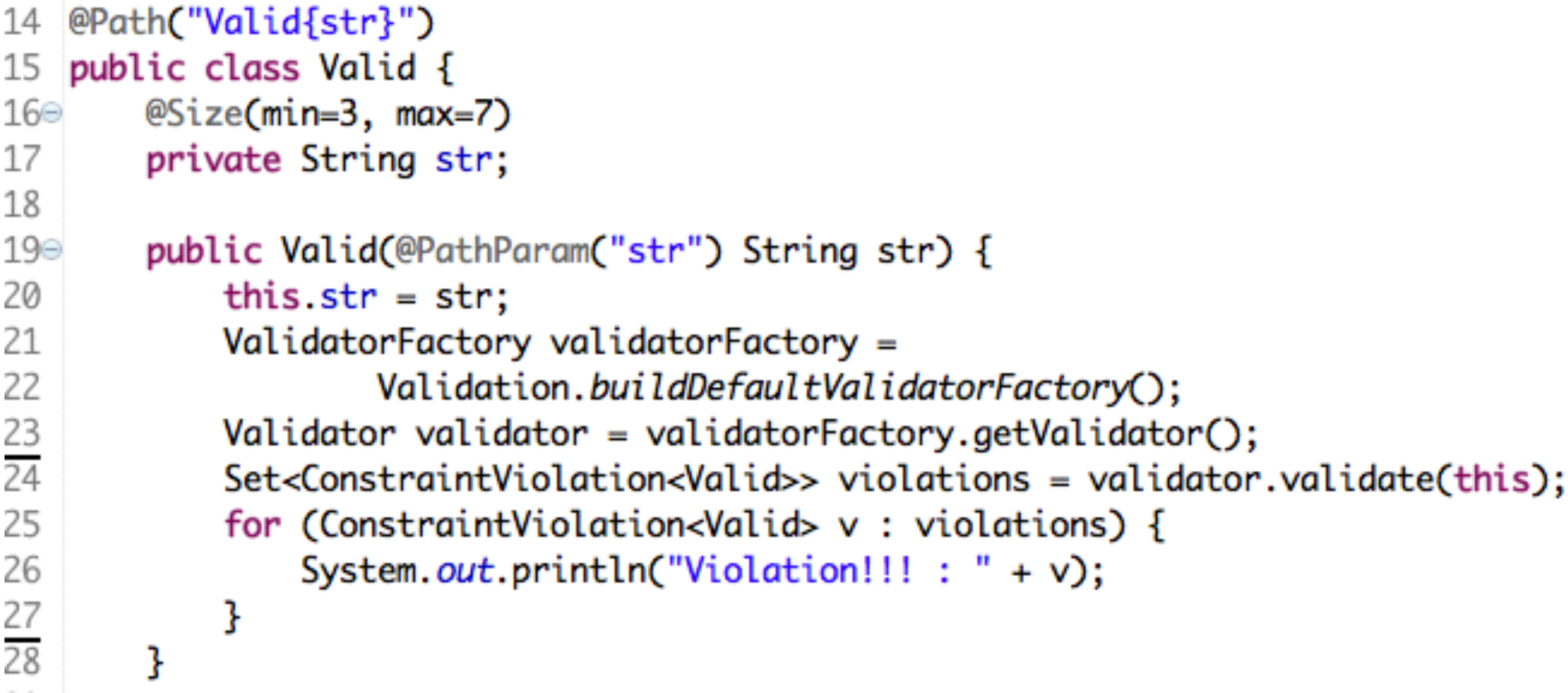
Annotation	Effect
@Null	***Element must be null***
@NotNull	***Element must not be null***
@AssertTrue	***Boolean element must be true***
@AssertFalse	***Boolean element must be false***
@Min, @Max	***Integer numeric must comply with limits***
@Size(min=X, max=Y)	***Constrains items in a String, array, Map or Collection***
@Past, @Future	***Date must be in the past or futre***
@Digits(integer=X, fractional=Y)	***Constrains digits of integers, BigDecimal, or String***
@Pattern(regexp=X)	***String must match regular expression***

Bean Validation

- Validation can also support:
  - Validation in groups, verifying partially filled beans, such as when checking values on individual pages of a wizard before the whole data set is entered
  - Custom Constraints, providing potentially complex constraints specified by programmers, and embodied in new annotations
- Validation is built into JSF, JPA, JAX-RS 2.0

Manual Bean Validation

- Validation can be invoked manually whenever needed, e.g. during Unit Testing, or after reading all the parameters of a RESTful request:



```java
@Path("Valid{str}")
public class Valid {
    @Size(min=3, max=7)
    private String str;

    public Valid(@PathParam("str") String str) {
        this.str = str;
        ValidatorFactory validatorFactory =
                Validation.buildDefaultValidatorFactory();
        Validator validator = validatorFactory.getValidator();
        Set<ConstraintViolation<Valid>> violations = validator.validate(this);
        for (ConstraintViolation<Valid> v : violations) {
            System.out.println("Violation!!! : " + v);
        }
    }
```

Automatic Validation

- In JAX-RS 2.0 Bean Validation is part of the specification
- Simply specify `@Valid` annotation on Entity body method parameter:

```
@POST
public Response doCreate(@Valid Customer c) {
…
```

ConstraintViolation<T>

- The ConstraintViolation object gives access to:
  - The invalid value
  - The affected bean(s) (root and leaf)
  - The message
    - Note that each constraint annotation can take a message property, e.g:

```
@Size(min=3, max=9,
  message=“Unacceptable length”)
```

Clients for RESTful Services

Consuming REST

- JAX-RS 2.0 Client API
- JavaScript client / AJAX

Java REST Client API

- A Java REST Client API is standard with JAX-RS 2.0
  - Included with Java EE 7
  - Client implementation with Jersey 1.x is proprietary, and does not conform to the new standard

Obtaining JAX-RS 2.0 Client

- Obtain the complete JAX-RS 2.0 Reference Implementation from:

```
https://java.net/projects/jersey/downloads/directory/
   jaxrs-2.0-ri
```

- Take the third item from the list:

```
Binary JAX-RS RI archive incl. JAX-RS API and external
dependencies.
```

  - Note this does not include Jackson JSON processing features.

- Add jars from all three subdirectories to the build path

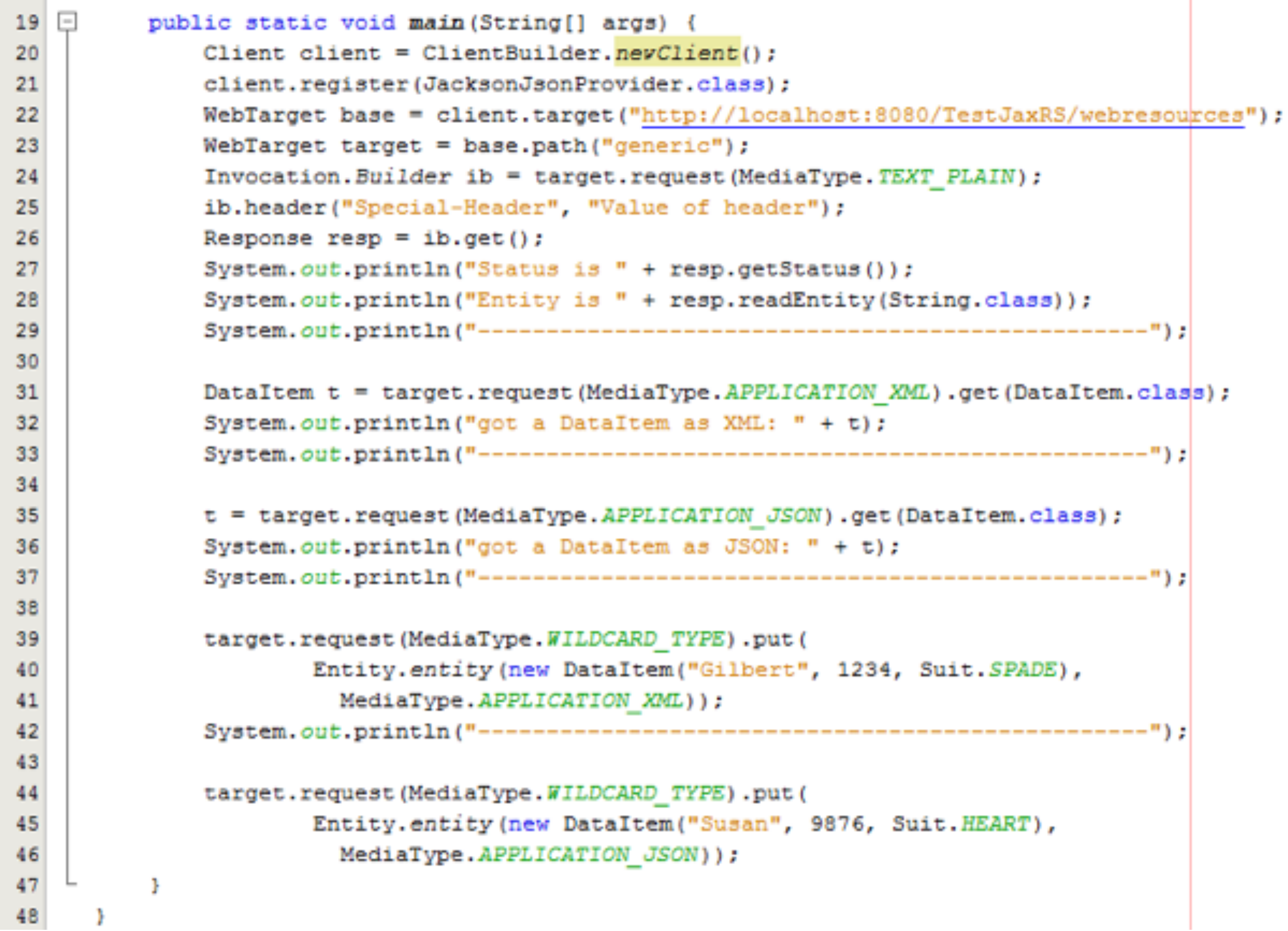
Adding Jackson for JSON Support

- Obtain the jarfiles (Base documentation calls for version 1.9.11, but this might change):
  - `jackson-asl-core`
  - `jackson-jaxrs`
  - `jackson-mapper-asl`
  - `jackson-xc`
- These can be obtained manually from subdirectories of: `repository.codehaus.org/org/codehaus/jackson`
- Once on the classpath, the `Provider` can be configured in code

Creating A Simple JAX-RS 2.0 Client

- The JAX-RS 2.0 Client API uses a builder API pattern
  - Expect lots of chained method calls
- Basic steps:
  - Create a `Client` from a `ClientBuilder`
  - Configure the `Client`
  - Create a `WebTarget` for the root of the target URI
  - Create a sub `WebTarget` to navigate to sub-resources
  - Create an `Invocation.Builder`
  - Make the call and get a `Response`
- This API has multiple approaches to most tasks

Simple Client With JSON Handling



```
    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
        client.register(JacksonJsonProvider.class);
        WebTarget base = client.target("http://localhost:8080/TestJaxRS/webresources");
        WebTarget target = base.path("generic");
        Invocation.Builder ib = target.request(MediaType.TEXT_PLAIN);
        ib.header("Special-Header", "Value of header");
        Response resp = ib.get();
        System.out.println("Status is " + resp.getStatus());
        System.out.println("Entity is " + resp.readEntity(String.class));
        System.out.println("-------------------------------------------------");

        DataItem t = target.request(MediaType.APPLICATION_XML).get(DataItem.class);
        System.out.println("got a DataItem as XML: " + t);
        System.out.println("-------------------------------------------------");

        t = target.request(MediaType.APPLICATION_JSON).get(DataItem.class);
        System.out.println("got a DataItem as JSON: " + t);
        System.out.println("-------------------------------------------------");

        target.request(MediaType.WILDCARD_TYPE).put(
                Entity.entity(new DataItem("Gilbert", 1234, Suit.SPADE),
                  MediaType.APPLICATION_XML));
        System.out.println("-------------------------------------------------");

        target.request(MediaType.WILDCARD_TYPE).put(
                Entity.entity(new DataItem("Susan", 9876, Suit.HEART),
                  MediaType.APPLICATION_JSON));
    }
}
```

WebTarget Operations

- Move down a path element

```
wt1 = wt.path(“subPath”)
```

- Add a query parameter

```
wt1 = wt.queryParam(“key”, “value”)
```

- Prepare an `Invocation.Builder ib`

```
ib = wt.request()
```

- Each of these operations creates a new object, and ***does not modify*** the original

Invocation.Builder Operations

- Add acceptable media type(s)
  - `ib.accept(MediaType…)`
- Add a header
  - `ib.header(String key, Object value)`
- Add a Cookie
  - `ib.cookie(Cookie c)`
  - `ib.cookie(String name, String value)`
- Prepare `Invocation` for an HTTP method
  - `ib.buildPost(Entity),ib.buildPut(Entity)`
  - `ib.buildGet(),ib.buildDelete()`
  - `ib.build(String methodName, Entity)`

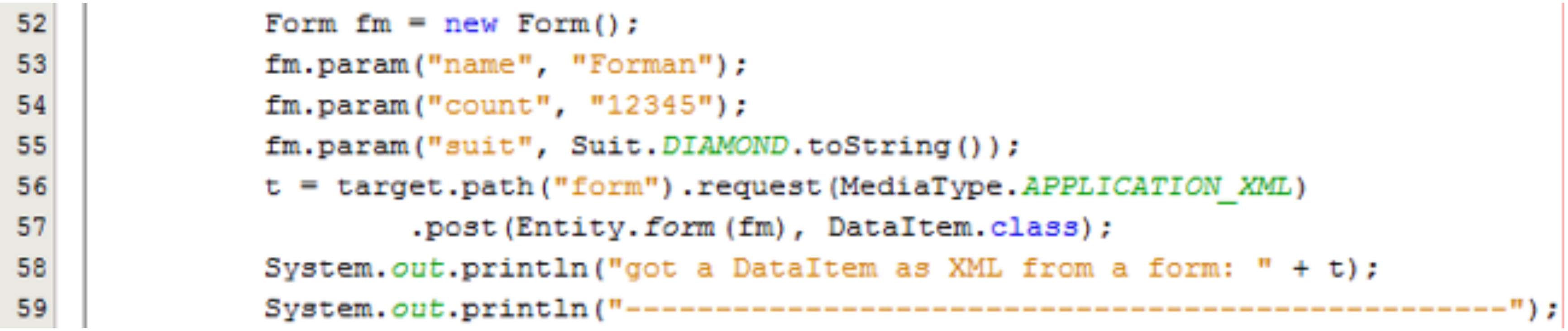
Invocation

- Invocation can be performed directly by the Invocation.Builder:
  - `ib.post(Entity)`
  - `ib.get()`
  - Etc.
- Or an `Invocation` object may be created by the `Invocation.Builder` which can then be invoked
  - `ib.buildGet().invoke();`
  - `ib.buildGet().invoke(Class<T> respType)`
- In the second case, the class indicates the desired type of the returned data

Preparing an Entity

- `Invocation.Builder` `.post()` and `.put()` can take an `Entity`
- Entity factories include:
  - `Entity.entity (obj, MediaType)`
  - `Entity.form(Form f)`
  - `Entity.html(String)`
  - `Entity.text(String)`
  - `Entity.json(obj)`
  - `Entity.xml(obj)`
- And some others

POSTing a Form

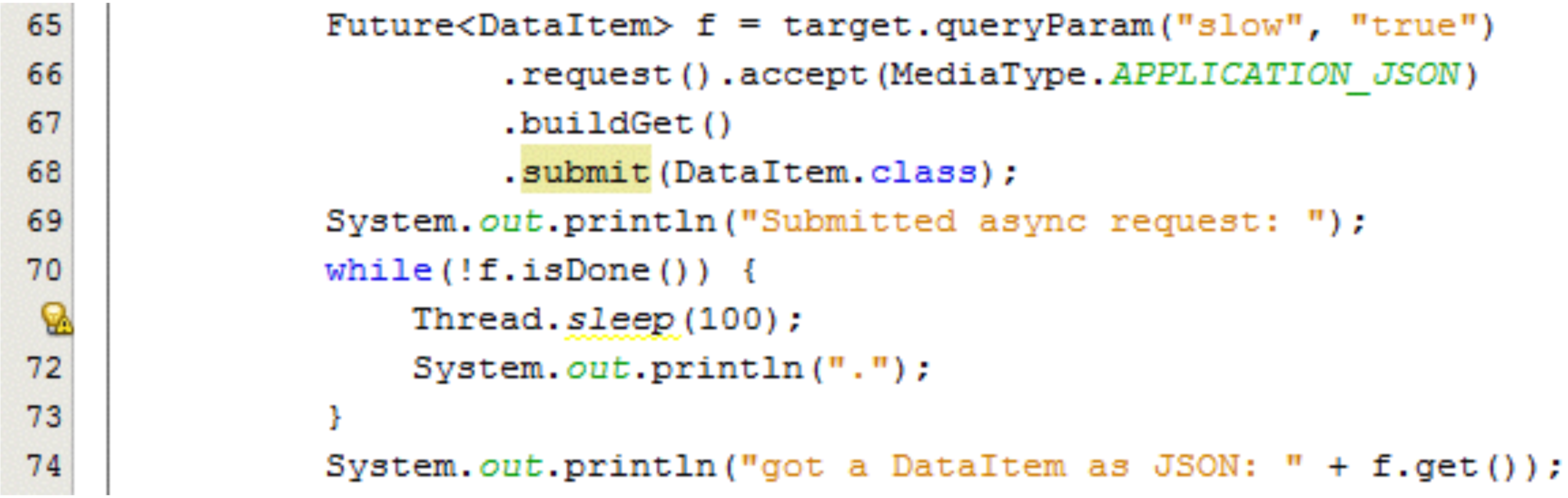


```java
Form fm = new Form();
fm.param("name", "Forman");
fm.param("count", "12345");
fm.param("suit", Suit.DIAMOND.toString());
t = target.path("form").request(MediaType.APPLICATION_XML)
        .post(Entity.form(fm), DataItem.class);
System.out.println("got a DataItem as XML from a form: " + t);
System.out.println("-------------------------------------------------");
```

Asynchronous Invocation

- Invocation object provides
  - `Future<Response> submit()`
  - `Future<T> submit(Class<T> respType)`
- These return a `Future` object, which provides two key methods:
  - `f.isDone()`
    - Returns `true` if the response has been received
  - `f.get()`
    - Returns the `Response`, or decoded `Entity` object

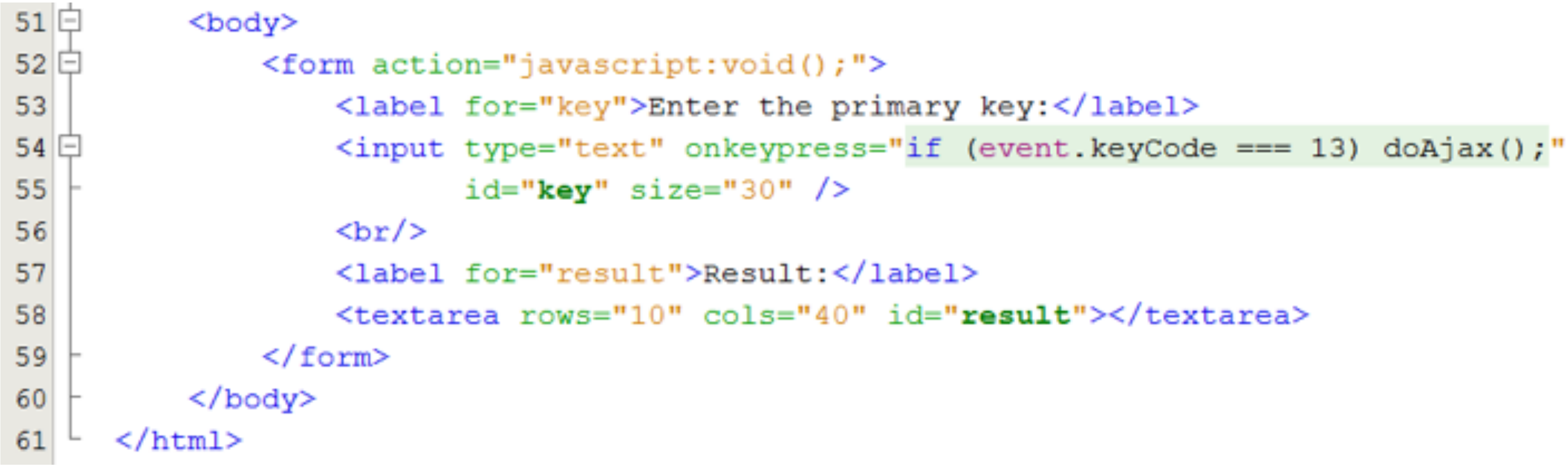
Handling a Future



```java
Future<DataItem> f = target.queryParam("slow", "true")
        .request().accept(MediaType.APPLICATION_JSON)
        .buildGet()
        .submit(DataItem.class);
System.out.println("Submitted async request: ");
while(!f.isDone()) {
    Thread.sleep(100);
    System.out.println(".");
}
System.out.println("got a DataItem as JSON: " + f.get());
```

JavaScript Client

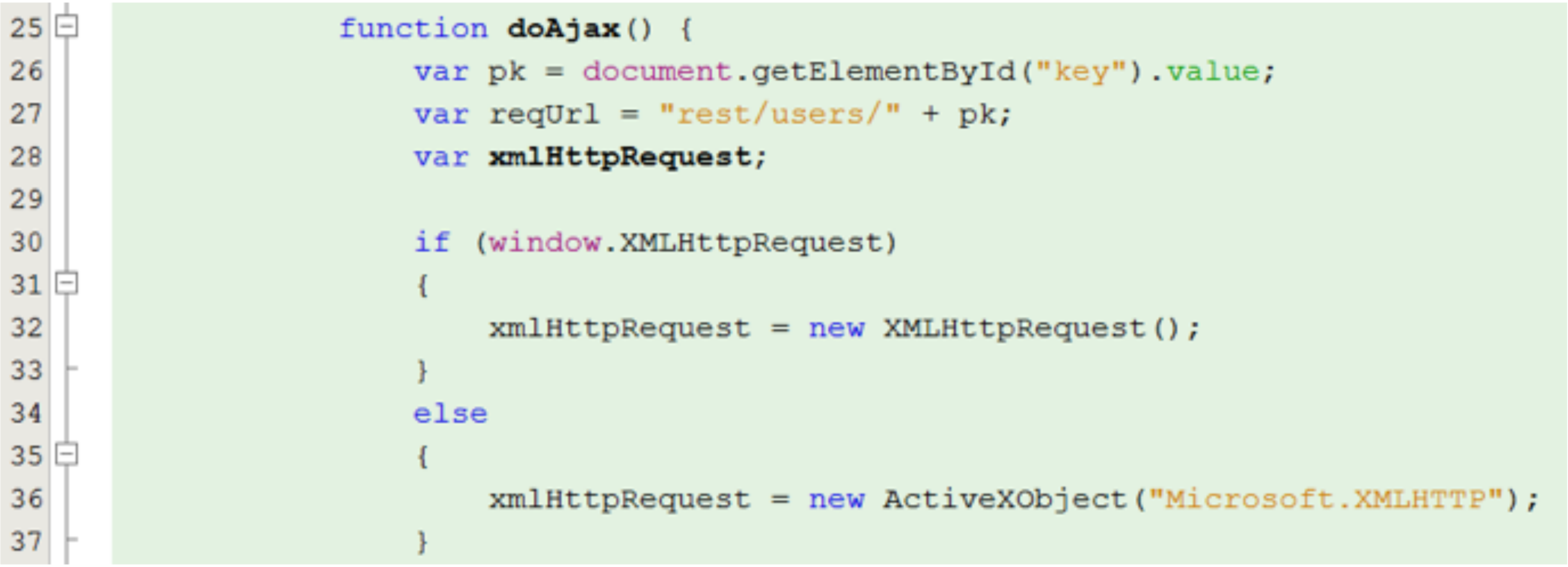
- JavaScript / AJAX code is commonplace in browsers
  - Collect input, provide a space for output, and respond to trigger event
  - Response is easiest to handle in JSON format



```
    <body>
        <form action="javascript:void();">
            <label for="key">Enter the primary key:</label>
            <input type="text" onkeypress="if (event.keyCode === 13) doAjax();"
                   id="key" size="30" />
            <br/>
            <label for="result">Result:</label>
            <textarea rows="10" cols="40" id="result"></textarea>
        </form>
    </body>
</html>
```

JavaScript Client

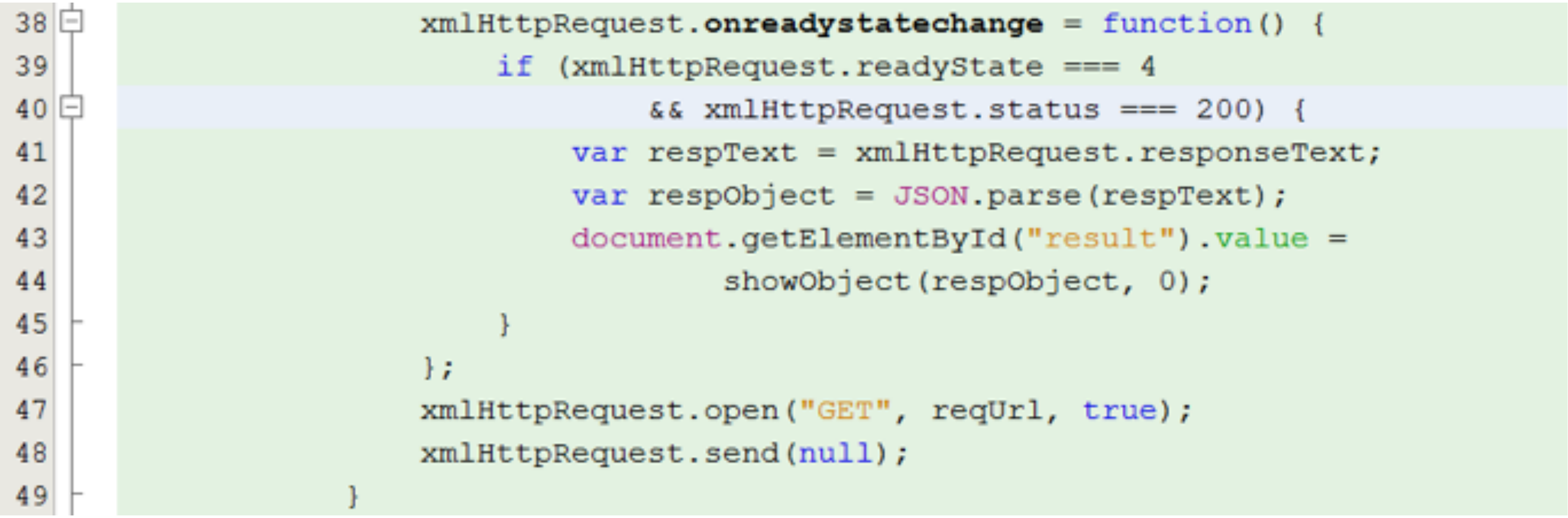
- On receiving the trigger event prepare the HTTP request:
  - Read the user input and prepare the URL
  - Determine HTTP handling behavior of this browser



```
function doAjax() {
    var pk = document.getElementById("key").value;
    var reqUrl = "rest/users/" + pk;
    var xmlHttpRequest;

    if (window.XMLHttpRequest)
    {
        xmlHttpRequest = new XMLHttpRequest();
    }
    else
    {
        xmlHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
    }
```

JavaScript Client

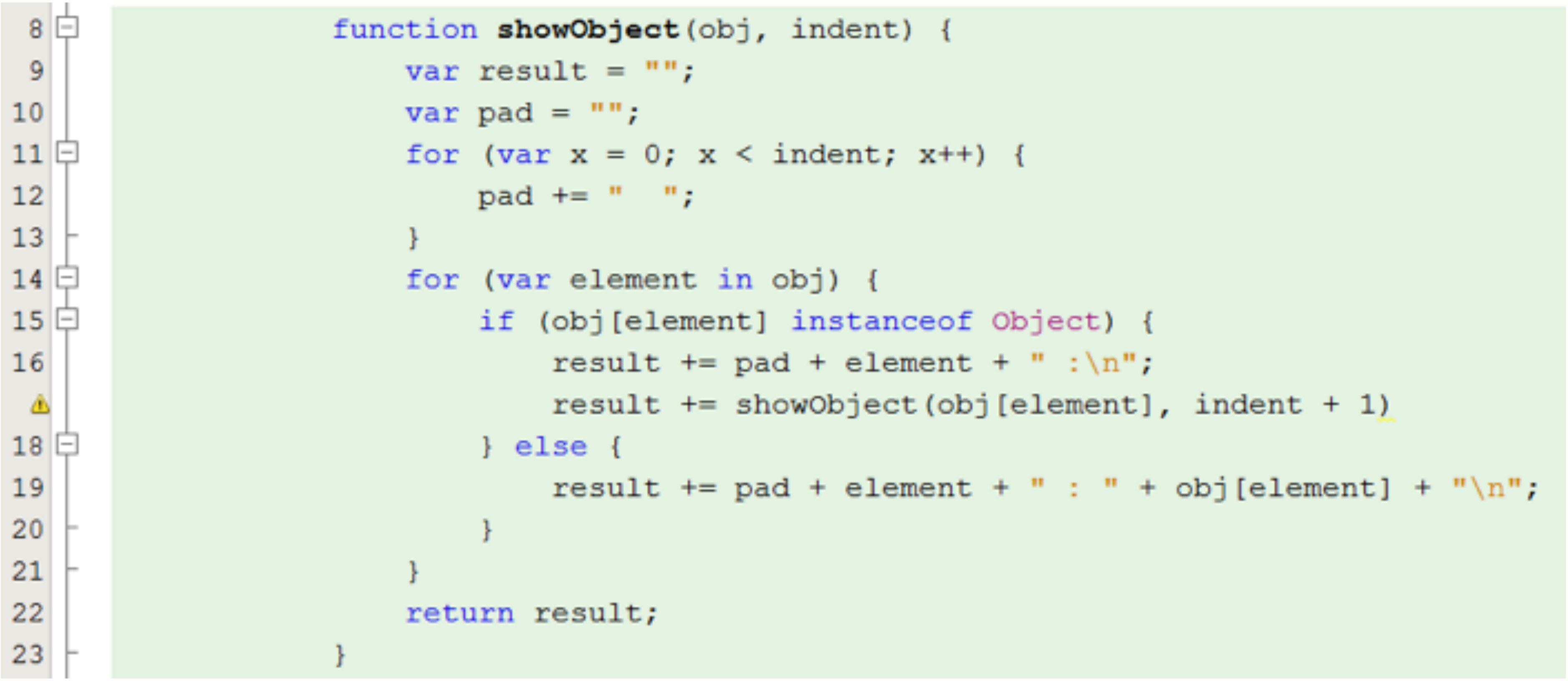
- Prepare the response handler (a callback function)
- Open and send the request
- Parse responseText using JSON.parse



```
    xmlHttpRequest.onreadystatechange = function() {
        if (xmlHttpRequest.readyState === 4
                && xmlHttpRequest.status === 200) {
            var respText = xmlHttpRequest.responseText;
            var respObject = JSON.parse(respText);
            document.getElementById("result").value =
                    showObject(respObject, 0);
        }
    };
    xmlHttpRequest.open("GET", reqUrl, true);
    xmlHttpRequest.send(null);
}
```

JavaScript Client

- JavaScript Objects are associative arrays, handling unknown object types is syntactically easy



```
function showObject(obj, indent) {
    var result = "";
    var pad = "";
    for (var x = 0; x < indent; x++) {
        pad += "  ";
    }
    for (var element in obj) {
        if (obj[element] instanceof Object) {
            result += pad + element + " :\n";
            result += showObject(obj[element], indent + 1)
        } else {
            result += pad + element + " : " + obj[element] + "\n";
        }
    }
    return result;
}
```

Testing RESTful Services

Testing REST

- CLI – curl
- Browser / Firebug / Developer tools
    - Fiddler (Windows only)
    - Chrome extensions:
        - Postman — REST client
        - Dev HTTP Client
- TestNG/JUnit, HTTPUnit

The curl Tool

- `curl` is a command line tool for transferring data with URL syntax
  - Freely available for any platform:

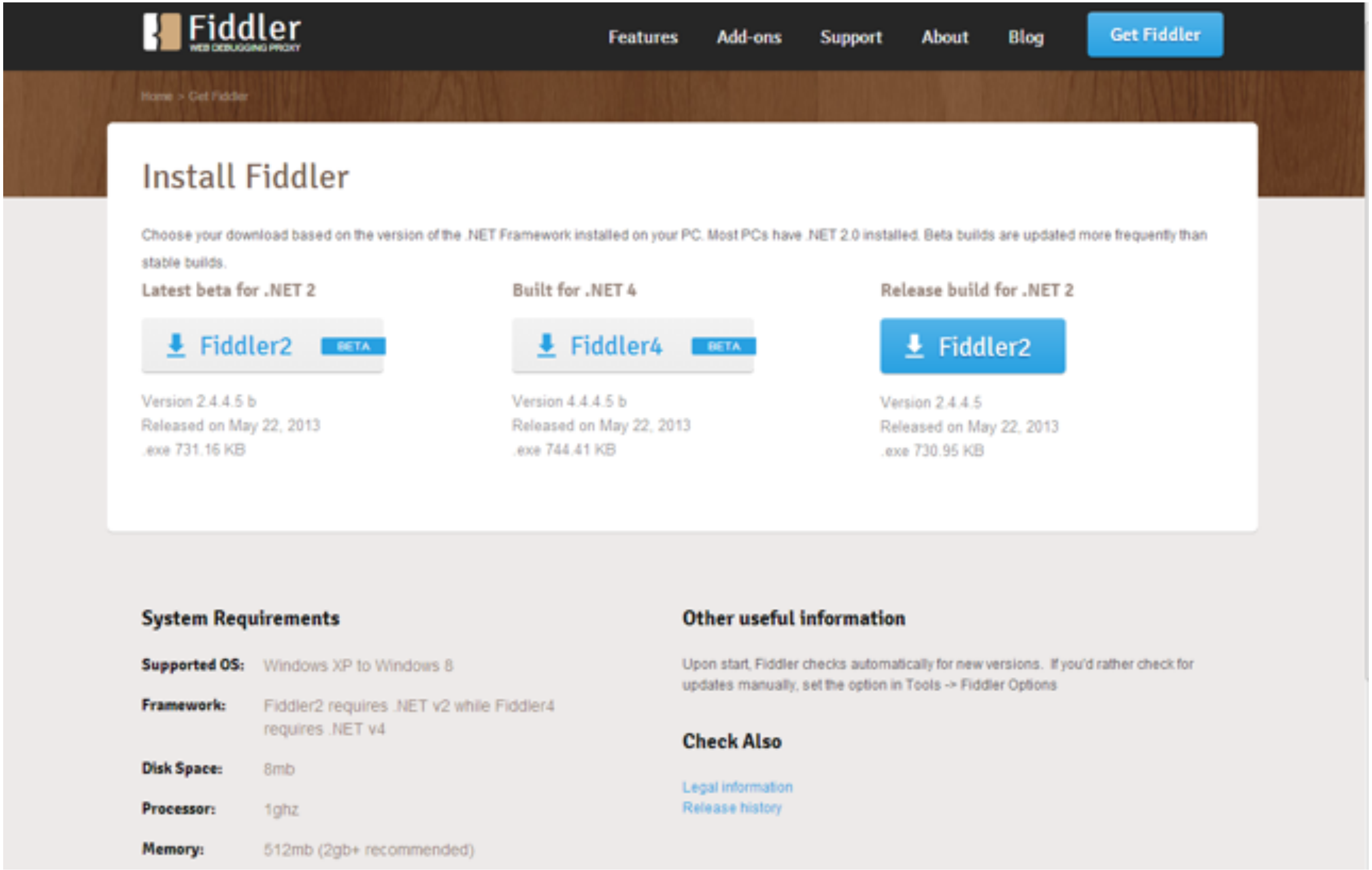
  http://curl.haxx.se/download.html
- Allows control of URL, HTTP method, Headers, body, and much more. E.g.:

curl –v	— verbose
–X PUT	— make a PUT request
–H “Accept: application/json”	— Accept JSON reply
–H “Content-Type: application/json”	— Content JSON
–d ‘{“name”:”Fred”}	— Body “entity”
http://localhost:8080/names/1243	— Target URL

Browser Tools

- Fiddler
  - Windows only, plugin for IE.
  - Built with .NET, available for .NET 2 and .NET 4 (beta)
  - Create requests
  - Capture traffic
  - Filter traffic
  - View response
  - Manipulate streams & more
- Download at:
  `http://fiddler2.com/get-fiddler`
- Download page has overview video

Fiddler



Install Fiddler

Choose your download based on the version of the .NET Framework installed on your PC. Most PCs have .NET 2.0 installed. Beta builds are updated more frequently than stable builds.

Latest beta for .NET 2

Version 2.4.4.5 b
Released on May 22, 2013
.exe 731.16 KB

Built for .NET 4

Version 4.4.4.5 b
Released on May 22, 2013
.exe 744.41 KB

Release build for .NET 2

Version 2.4.4.5
Released on May 22, 2013
.exe 730.95 KB

System Requirements

Supported OS:	Windows XP to Windows 8
Framework:	Fiddler2 requires .NET v2 while Fiddler4 requires .NET v4
Disk Space:	8mb
Processor:	1ghz
Memory:	512mb (2gb+ recommended)

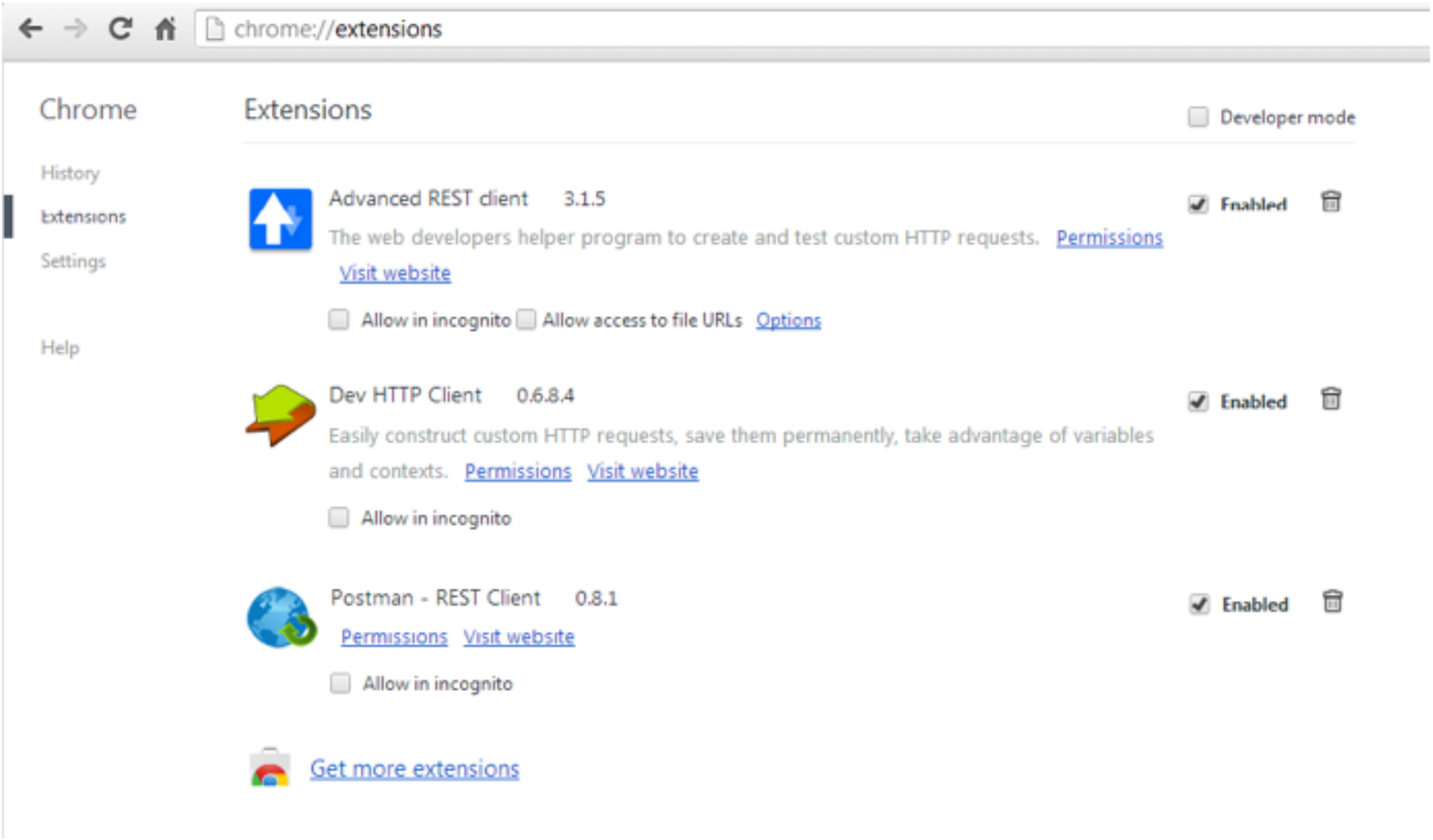
Other useful information

Upon start, Fiddler checks automatically for new versions. If you'd rather check for updates manually, set the option in Tools -> Fiddler Options

Check Also

Legal information
Release history

Browser Extensions for Chrome



Extensions

☐ Developer mode

Advanced REST client 3.1.5 ☑ Enabled

The web developers helper program to create and test custom HTTP requests. Permissions Visit website

☐ Allow in incognito ☐ Allow access to file URLs Options

Dev HTTP Client 0.6.8.4 ☑ Enabled

Easily construct custom HTTP requests, save them permanently, take advantage of variables and contexts. Permissions Visit website

☐ Allow in incognito

Postman - REST Client 0.8.1 ☑ Enabled

Permissions Visit website

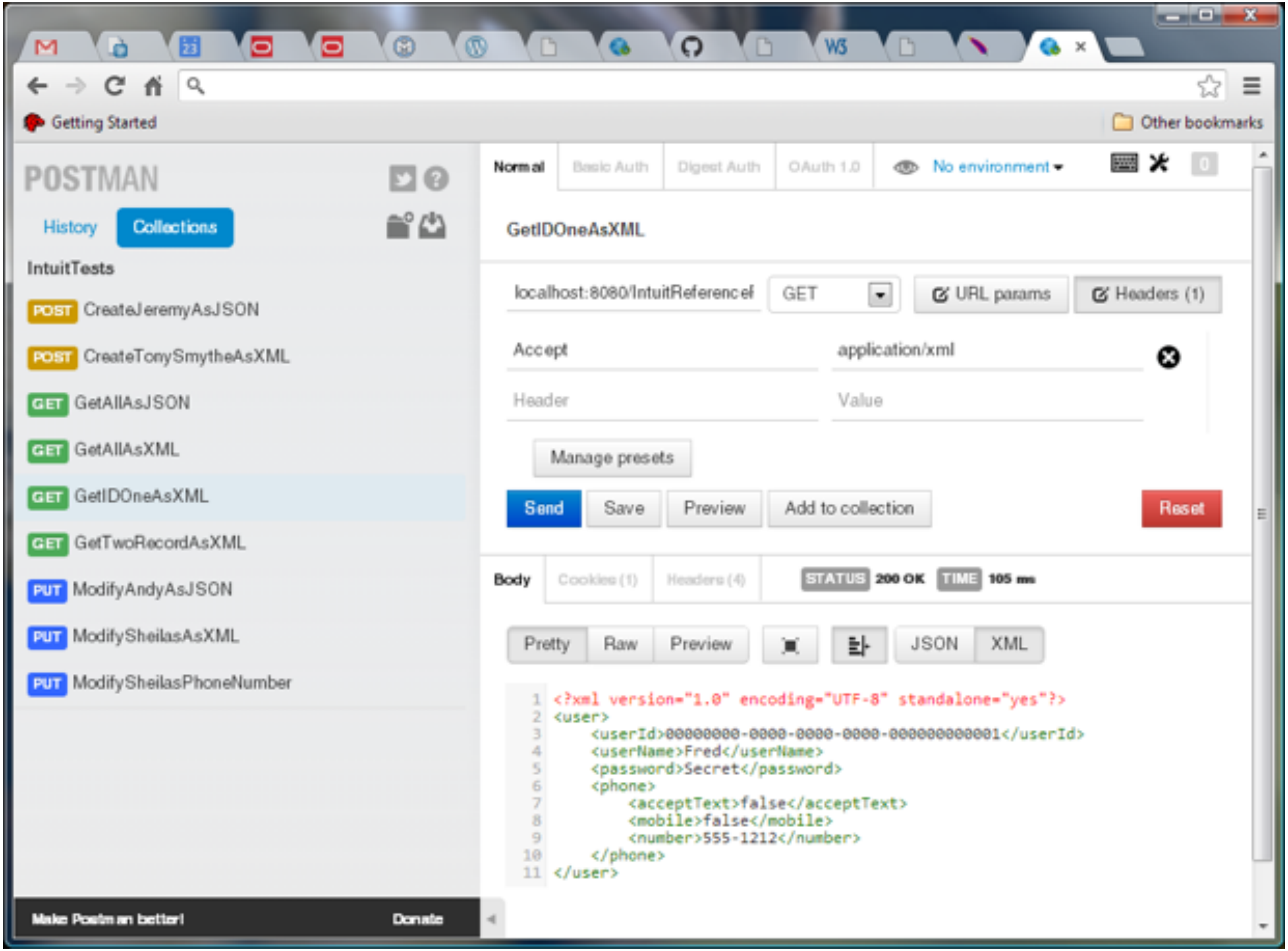
☐ Allow in incognito

Get more extensions

Postman REST Client

- Postman permits:
  - Creating and executing HTTP requests
  - Control of headers
  - Control of request entity as form key/value, url-encoded, XML, JSON, plain text, HTML
  - Display of results
  - Basic, Digest, OAuth security credentials
  - Saving requests in collections for reuse
  - Display of results

Postman REST Client



GetIDOneAsXML

localhost:8080/IntuitReferenceI GET

Accept: application/xml

STATUS 200 OK TIME 105 ms

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <userId>00000000-0000-0000-0000-000000000001</userId>
    <userName>Fred</userName>
    <password>Secret</password>
    <phone>
        <acceptText>false</acceptText>
        <mobile>false</mobile>
        <number>555-1212</number>
    </phone>
</user>
```

TestNG/JUnit, HTTPUnit

- TestNG and JUnit can interact directly with the Resource objects
  - Call the methods, receive Response objects, verify contents of Response object
- HTTP Unit can run the JAX-RS implementation servlet, allowing a very light weight or “containerless” environment
  - The @Path annotations are complex and form part of the logic of the resources, so they should probably be tested as part of “unit” testing, rather than being delayed to integration test

Summary

You should now be able to:

- Build a simple REST solution
  - Create a REST Web Service using JAX-RS
  - Write a client for a REST Web Service

XML Handling

Objectives

On completion you should be able to:

Manipulate XML data using:

- SAX
- DOM
- JAX-B

Major XML APIs in Java

- javax.xml.parsers.SAXParser
    - Input only (cannot write XML documents out)
    - Lowest memory footprint option (document not stored)
    - Error handling may be managed by client
- javax.xml.parsers.DocumentBuilder
    - Entire document represented in memory
        - Traverse nodes
        - Insert, delete, modify nodes
    - Output supported using Transformer

Major XML APIs in Java

- Build a Java data object from the input
  - JAX-B Java API for XML Binding
  - Converts XML data into Java Objects
  - Creates Java classes to suit XSD
  - Converts Java objects into XML data
  - Generates/consumes XSD

Stream Processing With SAX

- Callback/event oriented processing
- SAXParserFactory creates new parser instance
- Connect the parser to the input document
- Get a callback for each token parsed
  - Many callbacks can be generated, use an adaptor class to simplify listener implementation

SAXParser Example

- Start the parser

```
public static void main(String[] args)
   throws Throwable {

   FileInputStream fis =
      new FileInputStream("something.xml");
   InputSource xis = new InputSource(fis);

   SAXParser parser = SAXParserFactory.newInstance()
      .newSAXParser();

   parser.parse(xis, new MySaxHandler());
}
```

SAXParser Example

- Key callbacks
  - `startElement`, `endElement`—indicate <element> and </element>
  - `characters`—indicates text in the body of an element
  - `ignorableWhitespace`—might not care about this
  - `warning`, `error`, `fatalError`—report problems with parsing
- Parameters provided with the callbacks vary based on what’s being described

SAXParser Example

```
public class MySaxHandler extends DefaultHandler {
  @Override public void startElement(String uri,
    String localName, String qName, Attributes atts)
    throws SAXException {
    System.out.println("startElement " + uri + " "
      + localName + " " + qName + " " + atts);
  }
  @Override public void endElement(String uri,
    String localName, String qName)
    throws SAXException {
    System.out.println("endElement " + uri + " "
      + localName + " " + qName);
  }
[...]
```

DOM Parser Example

```
FileInputStream fis = new
FileInputStream("something.xml");

DocumentBuilder db =
  DocumentBuilderFactory.newInstance()
  .newDocumentBuilder();

Document d = db.parse(fis); // d is root Node

processNode(d, 0); // investigate the document tree
```

DOM Parser Example

```
public static void processNode(Node n, int level) {
  System.out.println(indent(level) + ""
    + n.getNodeName());
  NodeList nList = n.getChildNodes();
  int count = nList.getLength();
  for (int i = 0; i < count; i++) {
    processNode(nList.item(i), level + 1);
  }
}
public static String indent(int level) {
  String[] spaces = {"", "  ", "    ", "      ",
    "        ", "          ", "            "};
  if (level < spaces.length) { return spaces[level]; }
  else {
    return spaces[spaces.length - 1] + indent(level - spaces.length + 1);
  }
}
```

Updating A DOM Tree

```
// ask the Document object to create new element
// for that document (nodes may not be freely
// interchanged between Documents
Element c1 = d.createElement("Something-New");
// Set the text content of the element
c1.setTextContent("Something new in the document");
// add this new node to the end of an existing node
existingNode.appendChild(c1);
// Another new element
Element c2 = d.createElement("Something-Borrowed");
// put an attribute into this node
c1.setAttribute("item-color", "Blue");
existingNode.appendChild(c2);
```

Writing An XML Document

```
TransformerFactory transformerFactory =
  TransformerFactory.newInstance();

Transformer transformer =
  transformerFactory.newTransformer();

// d is our Document
DOMSource source = new DOMSource(d);

// Send XML representation to the console
StreamResult result = new StreamResult(System.out);
transformer.transform(source, result);
```

JAX-B

- JAX-B API and tools provide for:
  - Reading an XML Schema definition (XSD) file and creating Java classfiles
  - Reading classfiles and creating an XSD file
  - Parsing an XML document to create a Java object
  - Creating an XML document from a Java object
- JAX-B is largely automatic with annotation based overrides
- JAX-B is the foundation for both JAX-WS and JAX-RS handling of XML data

Creating Java Classes From XSD

- Given a file `data.xsd` defining an XML specification of a data type:
  - `xjc -p packageName data.xsd`
  - Creates Java files in the package `packageName`
  - Creates Java classes and supporting types, such as `enum` and `List` types for sequences
  - Creates an object factory that might be used for creating instances of the Java data types
- `schemagen` tool creates xsd from Java class or source files
  - Needs `-classpath` configured to resolve compilation of those Java classes

Example XSD Schema — 1

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="2.0">

<xsd:element name="Greetings" type="GreetingListType"/>

<xsd:complexType name="GreetingListType">
  <xsd:sequence>
    <xsd:element name="Greeting" type="GreetingType"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Example XSD Schema — 2

```
<xsd:complexType name="GreetingType">
  <xsd:sequence>
    <xsd:element name="Text" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="language" type="xsd:language"/>
</xsd:complexType>

</xsd:schema>
```

- To generate Java output, execute:

```
xjc -p greetingpkg theXsdFile.xsd
```

Generated Java (Skeletons)

```
public class GreetingListType {
  protected List<GreetingType> greeting;
  public List<GreetingType> getGreeting() ...
}

public class GreetingType {
  protected String text;
  protected String language;
  public String getText() ...
  public void setText(String value) ...
  public String getLanguage() ...
  public void setLanguage(String value) ...
  public String toString() ...
}
```

Generated Java (Skeletons)

```
public class ObjectFactory {
  public ObjectFactory() ...
  public GreetingListType createGreetingListType() ...
  public GreetingType createGreetingType() ...
  public JAXBElement<GreetingListType>
    createGreetings(GreetingListType value) ...
}
```

- These generated files are shown as skeletons, the files also include:
  - Imports
  - Method implementations
  - ***Annotations*** that tie these classes to the XML elements they represent

Using the Java Classes For Output

```
ObjectFactory of = new ObjectFactory();
GreetingListType grList =
  of.createGreetingListType();
GreetingType g = of.createGreetingType();
g.setText("Bonjour"); g.setLanguage("fr");
grList.getGreeting().add( g );
g = of.createGreetingType(); // create a second entry
g.setText("Gday"); g.setLanguage("en_AU");
grList.getGreeting().add( g );
JAXBElement<GreetingListType> gl =
  of.createGreetings(grList);
JAXBContext jc =
  JAXBContext.newInstance("greetingpkg");
Marshaller m = jc.createMarshaller();
m.marshal(gl, System.out);
```

Using the Java Classes For Input

```
ObjectFactory of = new ObjectFactory();
JAXBContext jc = JAXBContext.newInstance("customers");
Unmarshaller um = jc.createUnmarshaller();
File f = new File("input.xml");
JAXBElement jaxbe = (JAXBElement)(um.unmarshal(f));
CustomerListType customers =
  (CustomerListType) jaxbe.getValue();
List<CustomerDefType> custs = customers.getCustomer();
```

```
for (CustomerDefType cust : custs) {
  System.out.println("customer is:   " + cust.getName()
    + "\n at:           " + cust.getAddress1()
    + "\n joined:       " + cust.getJoined()
    + "\n credit limit: " + cust.getCredit());
}
```

Alternate Route to JAXBContext

- If you don’t have the `ObjectFactory`, you can create a `JAXBContext` for specific classes

```
JAXBContext.newInstance(This.class,
That.class, TheOther.class)
```

Two Approaches

- JAX-B can start with XSD and create suitable Java classes
- Or, can start with Java classes, and create XSD
- In either case, some translations and/or adjustments might need to be made

XML To Java Type Conversions

- XML numerics should indicate type:

`<xsd:element name=“val” type=“xsd:`***`type-info`***`”/>`

- Type representation in Java:
  - xsd:decimal → `BigDecimal`
  - xsd:integer → `BigInteger`
  - xsd:long → `long`
  - xsd:int → `int`
  - xsd:short → `short`
  - xsd:byte → `byte`

XML To Java Type Conversions

- Unsigned types have larger maximum values, so need larger holders
  - xsd:nonNegativeInteger → `BigInteger`
  - xsd:unsignedLong → `BigInteger`
  - xsd:unsignedInt → `long`
  - xsd:unsignedByte → `short`
- xsd:date, xsd:time and xsd:dateTime → `XMLGregorianCalendar`
  - XMLGregorianCalendar is abstract:

```
df = DatatypeFactory.newInstance()
df.newXMLGregorianCalendarDate([fields]);
```

XML To Java Type Conversions

- xsd:list → List<sometype>

```
<xsd:simpleType name="NumberListType">
  <xsd:list itemType="xsd:int"/>
</xsd:simpleType>
```

- Yields:

```
public class ListsType {
  protected List<Integer> numbers;
  public List<Integer> getNumbers
...
```

- Avoid `<xsd:list itemType="xsd:string"/>` as this will generate ambiguous XML
  - Space-separated list looks like spaces in single string element

XML To Java Type Conversions

- Range significant-digits, string-length, and pattern-matching limits are supported in XSD, but are not enforced in JAXB generated Java code
- Nillable types are converted to wrappers
  - <xsd:element name=“s” type=“short” nillable=“true” → Short
- Fields with Java keyword names get leading underscore
  - <xsd:element name=“long” …> → long **_long**
- xsd:union does not map well, JAXB gives String

Java to XML/XSD Conversions

- If a class is annotated
  - `@XmlRootElement`
- JAX-B can convert objects of that type to and from XML
  - (In fact, even this annotation is not absolutely *required*)
- Many default assumptions are made
  - These work well if you have ownership of the XSD
  - Annotations can override these assumptions where necessary

Java to XML Defaults

- Provided a class has a zero-arg constructor:
  - Any field that is `public`, non-`static`, non-`transient` will become an XML field with the same name
  - Any JavaBeans method pair (`setXxx`, `getXxx`) that are public will become an XML field with the same name (“`xxx`”)
  - Note that a `public` field with `public` get/set methods causes an error “XML field defined twice”
- Objects of other types referred to by these fields will also be part of the XML, even if not annotated.

Omitting the Zero-Arg Constructor

- JAX-B generally mandates a zero argument constructor for any type it will convert
  - Note that this constructor is typically public, but can actually be `private`
- Can omit constructor if a factory is provided
  - Specified using the `@XmlType` annotation

```
@XmlType(
      factoryClass=MyFactory.class,
      factoryMethod="makeAnObject")
```

- Note, the factory method must be `static`, and take no arguments

Controlling Java to XML Defaults

- `@XmlAccessorType` determines what is automatically transferred to XML
- This takes four parameters:
  - `XmlAccessType.PUBLIC_MEMBER`
    - Default, as previously described
  - `XmlAccessType.FIELD`
    - public, non-transient, fields are exported
  - `XmlAccessType.PROPERTY`
    - public get/set method pairs are exported
  - `XmlAccessType.NONE`
    - Nothing is exported

Explicitly Exporting Elements

- `@XmlElement` causes elements that are not exported by the default rules may be exported explicitly
- Note that JAX-B uses privileged reflection, even private aspects can be exported

Omitting Java Fields From XML

- If the default rules would cause something unintended or undesirable to be included in the XML, label the field or method `@XmlTransient`

Cycles In Object Graph

- Unlike Java objects in memory, XML data is constrained to a tree structure, cycles cannot be expressed
  - So, if `a` has a reference to `b`, and `b` has a reference to `c`, then neither `b` nor `c` can have a reference to `a`
- One way to solve this problem is to mark one of the references as `@XmlTransient` so that the cycle is broken on output
  - Of course, the link will also be missing when the structure is reconstructed on unmarshalling

Creating XSD From Java Types

- Create Java class representing desired XML data
- Annotate class using

```
javax.xml.bind.annotation.XmlType
javax.xml.bind.annotation.XmlRootElement
```

- Annotate fields using

```
javax.xml.bind.annotation.XmlElement
```

- Generate the schema using:

```
schemagen package.AccountInfo
```

Sample JAX-B Annotations

```
@XmlRootElement(name="complex-type")
@XmlType(
    propOrder={"number", "name", "otherCT", "greetings"})
public class ComplexType {
  private String name;
  private int number;
  private ComplexType otherCT;
  private String [] greetings;

  @XmlElement public String getName() { return name; }
  public void setName(String name) {this.name = name;}

  @XmlElement public int getNumber() { return number; }
  public void setNumber(int num) {number = num;}
```

Sample JAX-B Annotations

```
  @XmlElement(name=“greeting-list”)
  public String[] getGreetings() {
    return greetings;
  }

  // Circular references will cause errors when generating
  // the XML, but references to other instances of
  // “own type” are ok
  @XmlElement public ComplexType getOtherCT() {
    return otherCT;
  }

  // default constructor…
}
```

Taking Control of Generated XML

- `xjc` converts from XSD to Java, creating workable Java
- `schemagen` creates XSD describing the XML that will result from a Java class
  - Default conversion is often just fine
  - Sometimes need more control, e.g. field names, attributes, field ordering, and more
- JAX-B annotations allow comprehensive control of the Java → XML conversion

Controlling Names Of XML Fields

- By default, JAX-B creates XML tags using the property names of the Java class
- To override this, or to allow for XML that contains names that would not be legal if converted to Java, use this annotation:

```
@XmlElement(name=“special-name”)
```

Controlling Field Order

- XML Schemas often specify order of elements
- `@XmlType( propOrder={“name”, “address1”})`
- Notes:
  - Names must be the Java field names, not a modified name given in an `@XmlElement(name=“xml-name”)` annotation
  - If a field specified in `propOrder` is not found, this is a fatal error

Specifying XML Attributes

- In Java data exists as fields or as set/get methods, but in XML, tags can have attributes with values
- `@XmlAttribute` supports this
- Attribute is part of the tag that represents the class, not any of the enclosed fields

Specifying Tag Body

- By default, every field in a Java object gets its own sub-tag within the tag that represents the overall object.
- For a single field, alone in the object, the field can be made into the body of the class tag using: `@XmlValue`

Null Handling

- By default a field with a `null` value will disappear from the generated XML
- XSD is capable of indicating that a field may be omitted, which is then treated differently
- XSD specifies `nillable=true`
- `@XmlElement(nillable=true)` indicates a nillable field in Java source
- Output XML includes the field, which takes a special value in the case of the null value:

```
<myfield […] xsi:nil=“true” />
```

Marshalling/Unmarshalling Non-XmlRootElement Types

- Normally, annotate a class that will be represented as an XML document using `@XmlRootElement`
- JAX-B can convert non-annotated types too, but needs a little help
  - It wants to know the “qualified name” of the root document tag
- `JAXBElement<Type>` allows this

Using `JAXBElement<Type>`

- Wrap the POJO instance in `JAXBElement`:

```
JAXBElement<MyType> je =
  new JAXBElement<MyType>(
    new QName(“my-type”,
    MyType.class, myTypeObject);
```

- Then marshal the `JAXBElement`:

```
marshaller.marshal(je,
outputStream);
```

- `QName` defines the “qualified name” of the generated XML element

JAX-B Compatible Types

- Term JAX-B compatible shows up in documentation, but is not explicitly defined, and is really a misnomer
- XSD places some restrictions on objects that may be represented in XML. For Java:
  - No RMI Remote objects
  - No cyclic object graphs (no circular references)
  - Be careful with Collections or other classes you do not know the runtime structure of.
- Notice the restriction of cyclic graphs is a runtime issue, an object *may* have a member of its own type

Lab 5

Summary

You should now be able to:

Manipulate XML data using:

- SAX
- DOM
- JAX-B