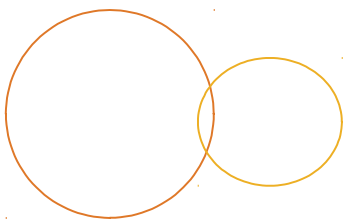
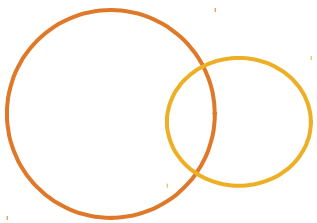
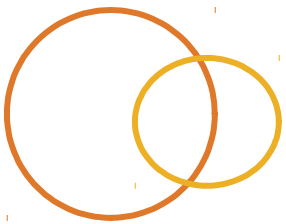
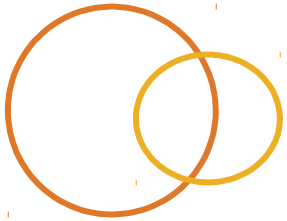
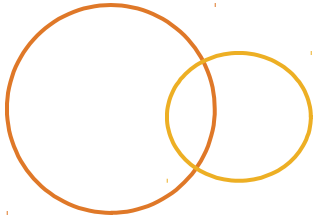
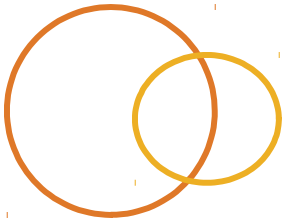
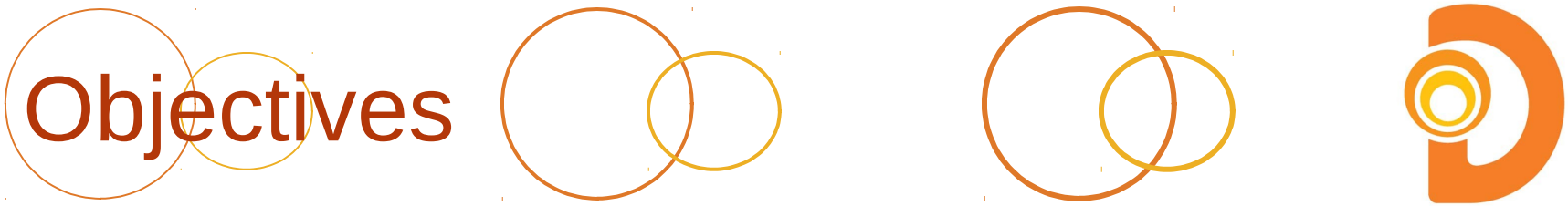





# Error Handling


# Objectives



- Apply standard Java language tools to identify/catch problems and use the Response object to send an appropriate entity body and HTTP status code to the client.
- Use WebApplicationException to control how a problem is converted to a response in a method that does not return a Response object
- Use an ExceptionMapper object to provide catch-all handling of a category of exceptions thrown by the application
- Use an ExceptionMapper object to take control of how problems arising in the JAX-RS navigation/service method selection are reported to the client

# Three Approaches To Problems

- Report non successful status directly through Response object
  - Applicable to service methods, particularly for unique error situations
- Throw a WebApplicationException
  - Applicable to non-service methods for unique error situations
- Throw an application domain exception, handle with ExceptionMapper<DomainException>
  - Applicable to service and non-service methods for recurring error situations

# Using Response To Report Status

- Use Java code to identify the error, use the ResponseBuilder status method to set the status representing the problem

```
public Response doStuff() {
  ResponseBuilder rb = Response.ok();
  // try something..
  if (unsuccessful) {
    rb.status(404).entity(“Not Found!”);
  }
  return rb.build();
```

# Using Response to Report Status

```
@GET @Path("/vehicles/{id}")
public Response getVeh(@PathParam("id") int id) {
  ResponseBuilder rb = Response.ok();
  try {
    String vehicle = dbLookup(id);
    rb.entity(vehicle);
  } catch (SQLException sqle) {
    rb.status(Response.Status.NOT_FOUND);
    rb.entity("Broken!");
  }
  return rb.build();
}
```

# Using Response To Report Status

- Easy to use
- Only workable when the method returns a Response
  - Can pass exceptions up call stack to the service method and then use this technique
- Tends to cause code duplication with errors that occur in many places
- Tends to clutter service code with unhappy path code
- Embeds “presentation” in service logic

# Using WebApplicationException To Report Status

- javax.ws.rs.WebApplicationException is a RuntimeException
- If thrown from a service method, can specify aspects of the response to the caller

```
throw new WebApplicationException(
  Response.status(404)
    .entity(“Not Found!”)
    .build());
```

- The Response embedded in WebApplicationException is sent to caller

# Using WebApplicationException To Report Status

- Easy to use
- Can be used from methods, e.g. “subroutines”, that don’t directly return Response

- Tends to cause code duplication with errors that occur in many places
- Tends to clutter service code with unhappy path code
- Embeds “presentation” in service logic

# Using WebApplicationException To Report Status

```
String dbLookup(int id) {
  // perform lookup
  if (failed) {
    throw new WebApplicationException(
      Response
        .status(Response.Status.NOT_FOUND)
        .entity("WAE Not found!")
        .build()
    );
  }
}
```

- Stylistically, this example is bad because it clutters dbLookup with JAX-RS specifics

# Using ExceptionMapper<T> To Report Status

- ExceptionMapper<E extends Throwable> may be embedded as a Provider class in JAX-RS system
- If a T is thrown from service code into the JAX-RS system, JAX-RS will look for an ExceptionMapper<T>
  - If found, JAX-RS passes the exception to the method
  - Response toResponse(T exception)
    - in that ExceptionMapper, and uses the Response to send to the caller

# Using ExceptionMapper<T> To Report Status

- Mappers are type specific and generalized, just like catch blocks
- The most specific, applicable, mapper is used
- You can install many mappers, and based on the type they are declared to handle, they will be called, just like having many catch blocks

- ExceptionMappers will also be checked when the JAX-RS system has a problem
  - E.g. no method found to handle a given request
  - Allows standard error format, even for system problems

# Using ExceptionMapper<T> To Report Status

```
public class WidgetException extends
  RuntimeException {
  // various standard constructors
}
```

---

```
private String findWidget() {
  throw new WidgetException("Widgets used up");
}

@GET @Path("/widgets")
public Response getWidget() {
  return Response
    .ok(findWidget())
    .build();
}
```

# Using ExceptionMapper<T> To Report Status

```
@Provider
public class WidgetExceptionMapper
  implements ExceptionMapper<WidgetException> {
  @Override
  public Response toResponse(WidgetException ex) {
    return Response
      .status(Response.Status.GONE)
      .entity("No widgets here! "
          + exception.getMessage())
   .build();
  }
}
```

# Using ExceptionMapper<T> To Report Status

- Can be used from methods, e.g. “subroutines”, that don’t directly return Response
- Handles system errors
- Handles domain-specific errors
- Avoids duplication of error handling
- Separates “presentation” and service logic

- More complex to configure

# Lab Exercise

- Create a new service endpoint that responds to /problems/<id>
- The service method should be declared to return a Response object
- The service should respond to a GET request as follows:
  - If <id> is negative, use the Response to set the status to 404, with an textual entity body
  - If <id> is between 0 and 99, throw a WebApplicationException to report a status of 400, with an entity message of your choice

# Lab Exercise

- Declare a new exception class MyException, as a subclass of RuntimeException
- Define. and install into your service, an ExceptionMapper<MyException> that returns a 500 error with an entity body of your choice
- Arrange that the service endpoint throws this exception if the value of <id> is 100 or more