Validation

Cognizant
Passion for building stronger businesses

C3: Protected

About the Author

Created By:	Ramesh C.P.(161646)
Credential Information:	SCJP, 8+ years of experience in technical training
Version and Date:	Spring MVC/PPT/1011/3.0

Cognizant Certified Official Curriculum

Icons Used

Questions

Tools

Hands on Exercise

Coding Standards

Test Your Understanding

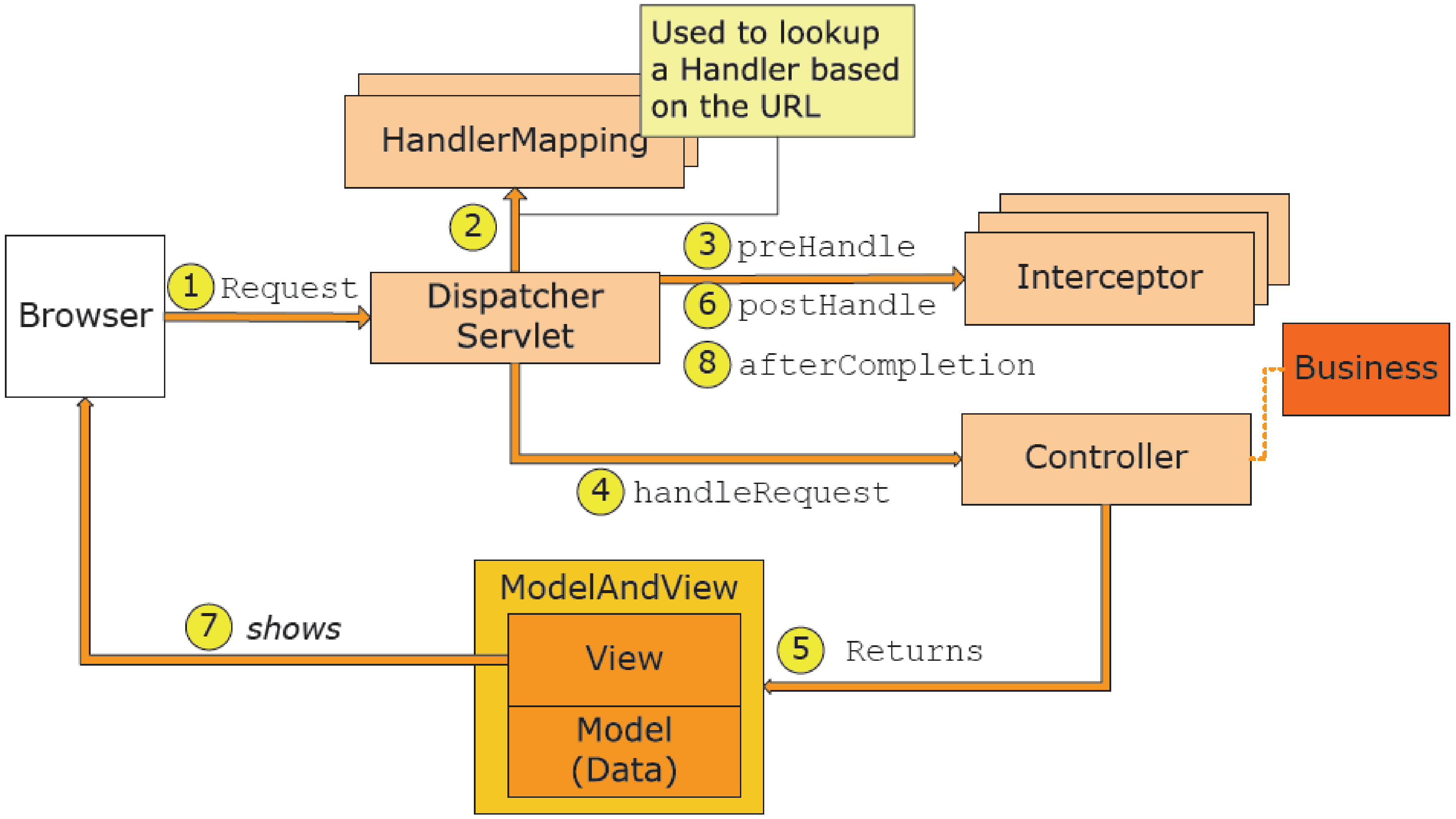
Reference

Demonstration

A Welcome Break

Contacts

Review of Architecture



Validation: Overview

- Introduction:
  - Command Objects that have been populated can be validated and report back errors when validation fails.
  - Spring offers its own validation infrastructure as an add-on to core Spring.
  - Spring 3 introduces several enhancements to its validation support.
    - First, the JSR-303 Bean Validation API is now fully supported.
    - Second, when used programmatically, Spring's DataBinder can now validate objects as well as bind to them.
    - Third, Spring MVC now has support for declaratively validating @Controller inputs.

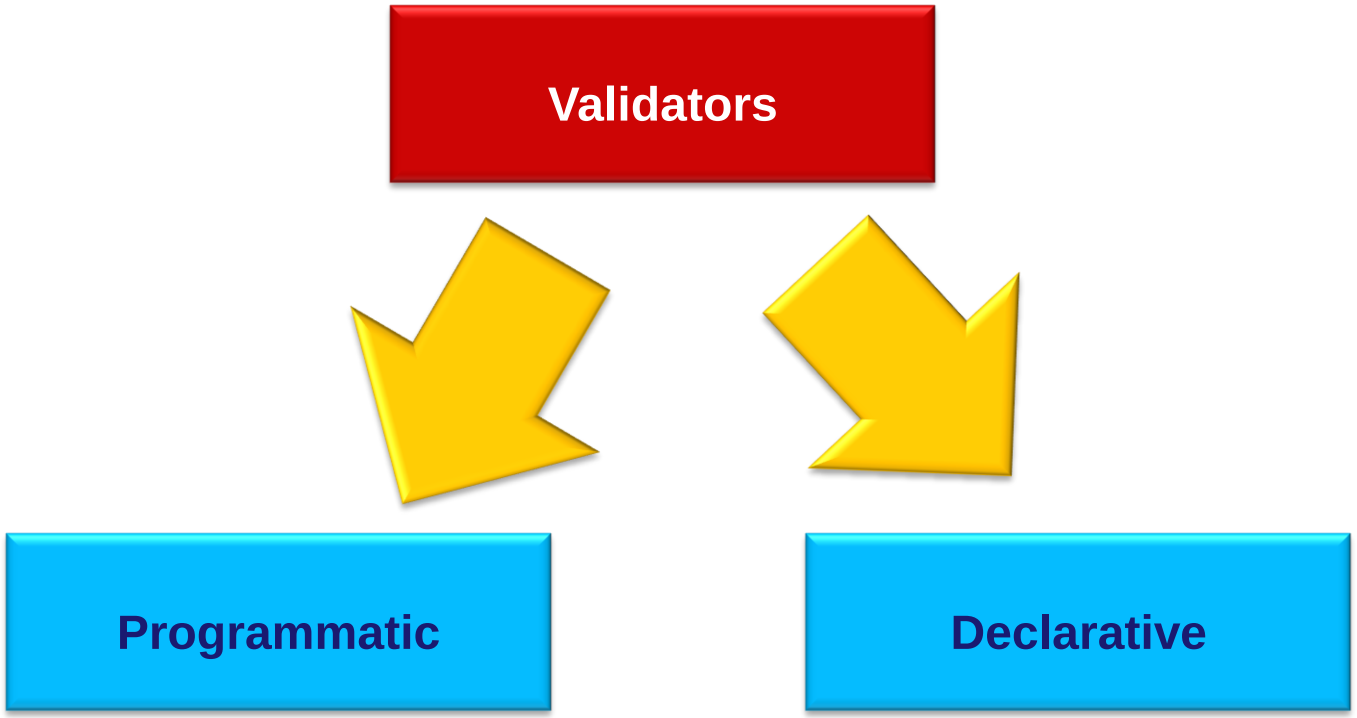
Validation: Objectives

- **Objective:**

After completing this chapter you will be able to:

- Write programmatic validators
- Write declarative validators

Types of Validators



Programmatic Validators

```
public interface Validator{
        public boolean supports(Class cl);
        public void validate(Object target, Errors errors);
}
```

Validates the given object

Can this validator validate instances of the supplied class?

Validator Interface

- One (the primary) or more validator objects can be registered
  - Has two methods that need to be implemented

```
public interface Validator {
  /**
  * Return whether or not this object can validate objects
  * of the given class.
  */
  boolean supports(Class clazz);
  /**
  * Validate an object, which must be of a class for which
  * the supports() method returned true.
  * @param obj Populated object to validate
  * @param errors Errors object we're building. May contain
  * errors for this field relating to types.
  */
  void validate(Object obj, Errors errors);
}
```

Validator- A Sample

```
public class PersonValidator implements Validator {
/** * This Validator validates just Person instances */
      public boolean supports(Class clazz) {
      return Person.class.equals(clazz);
}
public void validate(Object obj, Errors e) {
      ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
      Person p = (Person) obj;
       if (p.getAge() < 0) {
                e.rejectValue("age", "negativevalue");
      }
      else
      if (p.getAge() > 110) {
                 e.rejectValue("age", "too.darn.old");
     }
}
}
```

Reusing Validators

```
public class CustomerValidator implements Validator {
    private final Validator addressValidator;

    public CustomerValidator(Validator addressValidator) {
    if (addressValidator == null) {
            throw new IllegalArgumentException("The supplied [Validator] is required
    and must not be null.");
    }
    If (!addressValidator.supports(Address.class)){
             throw new IllegalArgumentException( "The supplied [Validator] must
    support the validation of [Address] instances.");
    }
            this.addressValidator = addressValidator;
 }
```

Reusing Validators (Contd.)

```
/** * This Validator validates Customer instances, and any subclasses of Customer too */
public boolean supports(Class clazz) {
    return Customer.class.isAssignableFrom(clazz);
}
public void validate(Object target, Errors errors) {
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
    Customer customer = (Customer) target;
     try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidator,
                                                   customer.getAddress(), errors);
}
finally {
    errors.popNestedPath(); } } }
```

Controller with Validator

```
<bean id=“logonValidator” c lass=“test.web.validation.LogonValidation”/>
<bean id="logonValidator" class="test.web.LogonValidator"/>
<bean id="logonForm" class="test.web.LogonFormController">
   <property name="sessionForm"><value>true</value></property>
   <property name="commandName"> <value>credentials</value> </property>
   <property name="commandClass"> <value>Credentials</value> </property>
   <property name="validator"><ref bean="logonValidator"/></property>
   <property name="formView"><value>logon.jsp</value></property>
   <property name="successView"><value>sucess.jsp</value></property>
</bean>
```

Adding Validation - Example

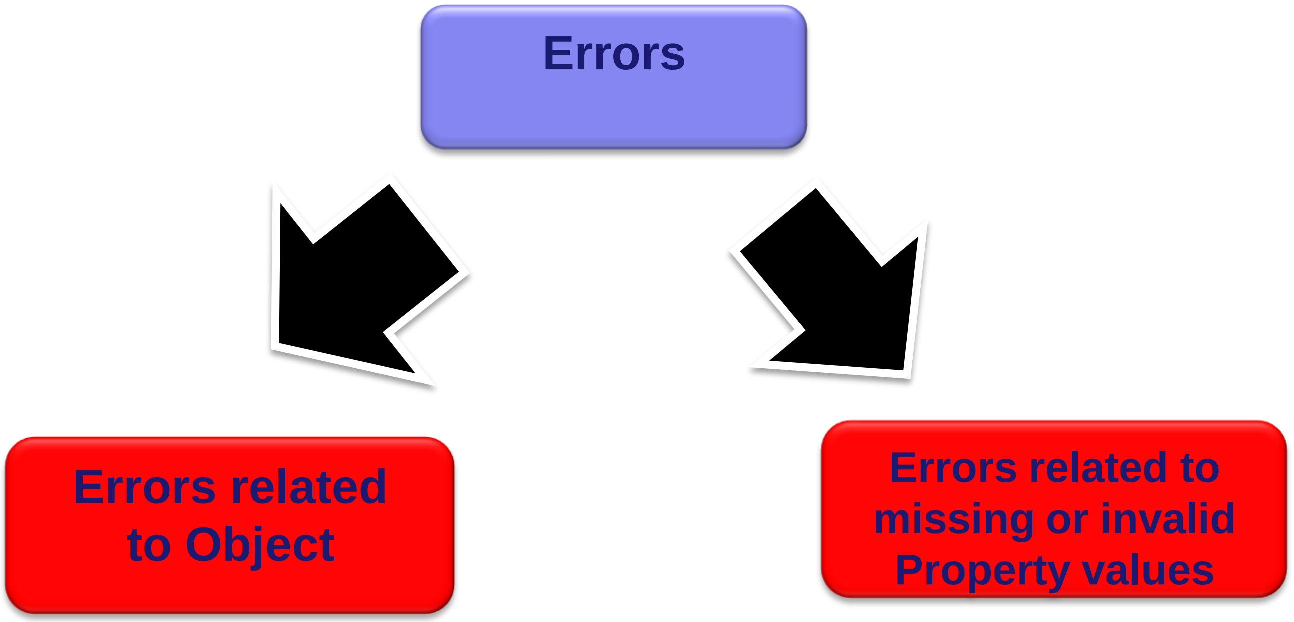
```
public class RegistrationValidator implements Validator {
  private final static Date MINBIRTHDATE;
  static {
    Calendar c = Calendar.getInstance();
    c.add(Calendar.YEAR,-14);
   MINBIRTHDATE = c.getTime();
  }
  public boolean supports(Class aClass) {
    return aClass.isAssignableFrom(RegistrationDetails.class);
  }
  public void validate(Object object, Errors errors) {
    RegistrationDetails details = (RegistrationDetails) object;
    Date date = details.getBirthdate();
    if (date!=null && date.after(MINBIRTHDATE)){
      errors.rejectValue("birthdate","tooyoung",
      "You have to be 14 years or older to register");
    }
    String username = details.getUsername();
    if ((username == null) || (username.length()==0)){
      errors.rejectValue("username","required","username is required");
    }
  }
}
```

Adding Validation - Example

❖ Registering the `Validator`

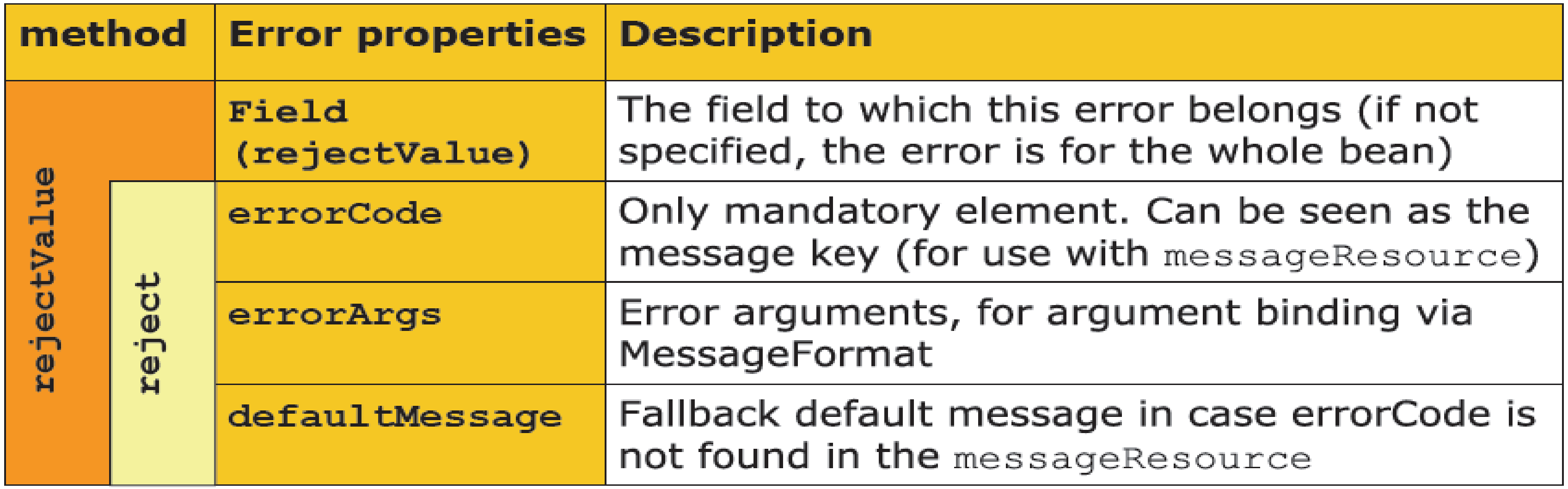
```
<bean id="registrationController"
      class="demos..SimpleRegistrationController">
  ...
  <property name="validators">
    <list>
      <bean class="demos..RegistrationValidator"/>
    </list>
  </property>
</bean>
```

Errors Interface



Adding Validation (Contd.)

❖ The Errors:



method		Error properties	Description
rejectValue		**Field (rejectValue)**	The field to which this error belongs (if not specified, the error is for the whole bean)
	reject	**errorCode**	Only mandatory element. Can be seen as the message key (for use with `messageResource`)
		errorArgs	Error arguments, for argument binding via MessageFormat
		defaultMessage	Fallback default message in case errorCode is not found in the `messageResource`

reject() method

Public void reject(String errorCode);

Public void reject(String errorCode, String defaultMessage);

Public void reject(String errorCode, Object[] errorArugments, String defaultMessage);

Note: Rejecting an object as a whole is called a global error, because though no specific property value is invalid, the form values cannot be processed. An example could be a customer who is underage.

rejectValue() method

```
public void rejectValue(String propertyName, String errorCode);
public void rejectValue(String propertyName, String errorCode,
                String defaultMessage);
public void rejectValue(String propertyName, String errorCode,
                                 Object[] errorArguments,
                                 String  defaultMessage);
```

Note: Rejecting a property value is called field error

Global errors typically appear on the top of a form in the view, while field errors typically appear next to the input fields they are related to.

Declarative Validator

- Support for declarative validation with JSR-303 (Bean Validation) annotations.
- Fortunately, you only need to add a single line of configuration to Spring xml configuration to flip on all of the annotation-driven features you'll need from Spring MVC:

```
<mvc:annotation-driven/>
```

- Along with many other Spring 3 features, The <mvc:annotation-driven> also registers JSR-303 validation support.

JSR-303 Bean Validation API

- JSR-303 standardizes validation constraint declaration and metadata for the Java platform.
- Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them.
- There are a number of built-in constraints you can take advantage of such as @NotNull, @Size(Min=,Max=), @Pattern
- You may also define your own custom constraints.

Declarative Validation example

- JSR-303 allows you to define declarative validation constraints against such properties.

```
public class PersonForm {
@NotNull
@Size(max=64)
private String name;

@Min(0)
private int age;
}
```

When an instance of this class is validated by a JSR-303 Validator, these constraints will be enforced.

Configure Spring as JSR-303 Validator

- The hibernate Validator is the default reference implementation for JSR -303.
- Spring provides full support for the JSR-303 Bean Validation API. This includes convenient support for bootstrapping a JSR-303 implementation as a Spring bean.
- Use the LocalValidatorFactoryBean to configure a default JSR-303 Validator as a Spring bean:

```
<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

Injecting a Validator

- LocalValidatorFactoryBean implements both javax.validation.ValidatorFactory and javax.validation.Validator, as well as Spring's org.springframework.validation.Validator.
- Inject a reference to javax.validation.Validator if you prefer to work with the JSR-303 API directly:

```
import javax.validation.Validator;
@Service
public class MyService  {
@Autowired
private Validator validator;
```

- Inject a reference to org.springframework.validation.Validator if your bean requires the Spring Validation API:

```
import org.springframework.validation.Validator;
@Service
public class MyService  {
@Autowired
private Validator validator;
}
```

Configuring Custom Constraints

- Each JSR-303 validation constraint consists of two parts.
- First, a @Constraint annotation that declares the constraint and its configurable properties.
- Second, an implementation of the javax.validation.ConstraintValidator interface that implements the constraint's behavior.
- To associate a declaration with an implementation, each @Constraint annotation references a corresponding ValidationConstraint implementation class.
- At runtime, a ConstraintValidatorFactory instantiates the referenced implementation when the constraint annotation is encountered in your domain model.

Configuring Custom Constraints (Contd.)

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=MyConstraintValidator.class)
public @interface MyConstraint {
}
```

declares the constraint and properties

```
import javax.validation.ConstraintValidator;
public class MyConstraintValidator implements ConstraintValidator  {
    @Autowired;
    private Foo aDependency;
    ...
}
```

Implements the ConstraintValidator interface that implements the constraint's behavior

Configuring a DataBinder

- Since Spring 3, a DataBinder instance can be configured with a Validator.
- Once configured, the Validator may be invoked by calling binder.validate(). Any validation Errors are automatically added to the binder's BindingResult.
- Binding and validation errors can be trapped and introspected by declaring a BindingResult parameter (see the example later).
- BindingResult’s getFieldError() method can be used to access those field errors in UI Form. Or JSP tag <sf:errors> can render field validation errors.

@Controller Input Validation

- To trigger validation of a @Controller input, simply annotate the input argument as @Valid.
- Spring MVC will validate a @Valid object after binding so-long as an appropriate Validator has been configured.

```
@Controller
public class MyController  {

      @RequestMapping("/foo", method=RequestMethod.POST)
      public String processFoo(@Valid Foo foo, BindingResult bindingResult)  {
            if(bindingResult.hasErrors()) {
                        return “error_edit";
            }
            //logic for processing Foo
      }
}
```

@Controller Input Validation (Contd.)

- The @Valid annotation is the first line of defense against faulty form input.
- Should anything go wrong while validating the Foo object, the validation error will be carried to the processFoo() method via the BindingResult that's passed in on the second parameter.
- If the BindingResult's hasErrors() method returns true, then that means that validation failed.
- In that case, the method will return error_edit as the view name to display the form again so that the user can correct any validation errors.

Advantages of Validators

- Validators are pluggable
- They can be injected into Controllers that call the business logic. The Spring MVC has the ability to automatically validate @Controller inputs
- Validators handle the first-level validation that more fine-grained, supports i18n, and fully integrated with the presentation layer through Errors interface.

Validation

Time for a Break !

Validation

- Questions from participants

Test Your Understanding

1. Spring supports only Programmatic validations. Say try or false.
2. What is the minimum Spring configuration required to configure a JSR-303-backed Validator with Spring MVC.

Validation: Summary

- Validator and Errors interfaces form the backbone for validation.
- Spring provides full Support for declarative validation with JSR-303 (Bean Validation) annotations.

Validation: Source

- http://docs.jboss.org/hibernate/validator/4.2/reference/en-US/html_single/#example-constraint-validator
- http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/validation.html#validation-beanvalidation

Disclaimer: Parts of the content of this course is based on the materials available from the Web sites and books listed above. The materials that can be accessed from linked sites are not maintained by Cognizant Academy and we are not responsible for the contents thereof. All trademarks, service marks, and trade names in this course are the marks of the respective owner(s).

You have successfully completed Validation

Click here to proceed

Cognizant
Passion for building stronger businesses