Java Technologies

Contexts and Dependency Injection (CDI)

The Context

- Do you remember **AOP, IoC, DI** ?
- Implicit Middleware seems like a good idea.

  Using transactions is easy in EJBs...

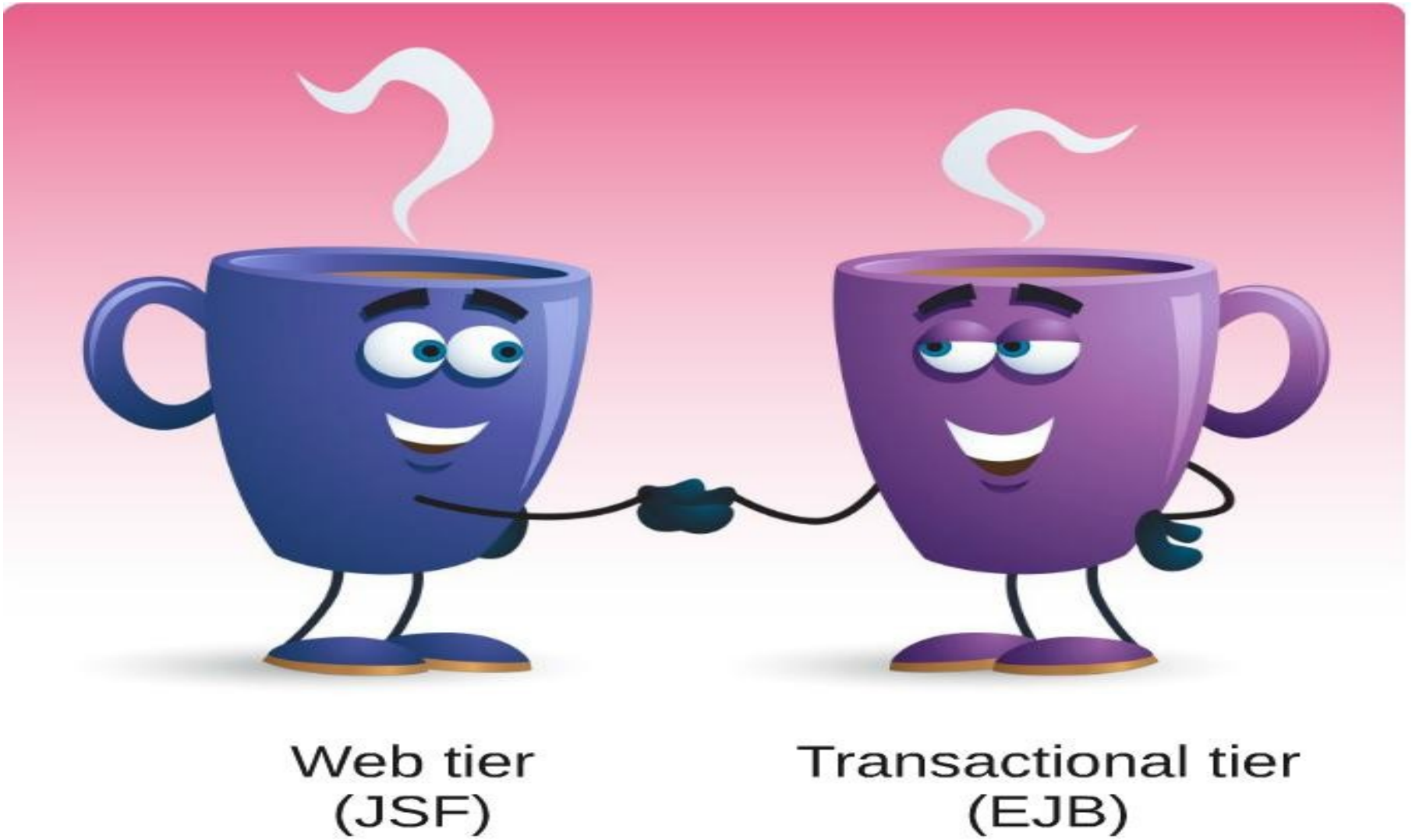
  Is it possible to extend this mechanism?
- We need flexibility to integrate various kinds of components in a **loosely coupled but typesafe way.**

  Using resources and EJBs is easy...

  Is it possible to extend this mechanism?
- Java Bean, Enterprise Java Bean, JSF Managed Bean, Web Bean, Spring Bean, Guice Bean,…

  What exactly is (should be) a bean?

Unified Bean Definition



Enterprise beans may act as JSF managed beans

Aspect Oriented Programming

- An **aspect** is a common feature that's typically **scattered across methods**, classes, object hierarchies, or even entire object models.
- It is a behavior that “looks and smells” like it should have structure, but you can't find a way to express this structure with traditional object-oriented techniques.
  - examples: logging, tracing, transactions, security,...
- Aspect-oriented programming gives you a way to encapsulate this type of functionality. It allows you to add behavior such as logging "around" your code.

Example: The *Product* Class

Concern 1: The product itself

```
public class Product {
  private String name;
  private BigDecimal price;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(BigDecimal price) {
    if (price.signum() == -1) throw new IllegalArgumentException();
    this.price = price;
  }
  ...
}
```

This code is nice and clear.

Changing Specifications

"Whenever the price of a product changes, this information must be written to a log file."

Concern 2: The Logger

```
public class Logger {

  public Logger() {

    // Init the logging mechanism

  }

  public void write(String text) {

    // Write something to the log

  }

}
```

The Crosscutting

```
public class Product {
  private BigDecimal price;
  ...
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(BigDecimal price) {
    Logger logger = new Logger(); //?
    logger.write("Price changed: " + this.price + " -> " + price);
    this.price = price;
  }
  ...
}
```

This code is now ugly.

Logging becomes a **crosscutting concern**, as it is a behavior that "cuts" across multiple points in your object models, yet is distinctly different.

Crosscutting Disadvantages

- Scattered Code

```
class Product {
 Logger logger;
 BigDecimal price;
 ...
 }
```

```
public class Commodity {
  Logger logger;
  BigDecimal purchasePrice;
  BigDecimal sellPrice;
  ...
}
```

- Tangled Code

```
 class Product {
 ...
 public String setPrice(BigDecimal price) {
   // Security
   User user = Application.getCurrentUser(); //?
   if (!user.hasPermission("price.change") {
     throw new AuthorizationException(user, "price.change");
   }
   // Logging
   Logger logger = new Logger(); //?
   logger.write("Price changed: " + this.price + " -> " + price);
   this.price = price;
 }
 ...
}
```

- Code duplication
- Inter-dependencies
  - hard to reuse
  - hard to test/change

Inversion of Control (IoC)

- **Implementations** should not depend upon other implementations, but instead

  → they should depend upon **abstractions**.

- The objects of an application should be (if possible) **loosely coupled**.
- IoC addresses resolving dependencies between components (objects) by reversing the direction of access to resources, services, etc.

The Hollywood Principle: "Don't call us, we'll call you"

Example

- Depending on implementations

```
public class PersonManagedBean {
    private PersonService service; //DAO

    public PersonManagedBean() {
        this.service = new JdbcPersonServiceImpl();
        //or maybe
        this.service = ServiceManager.createPersImpl("jdbc");
    }
}
```

- Decoupling implementations

```
public class PersonManagedBean {
    private PersonService service;

    public PersonManagedBean() { }

    public void setPersonService(PersonService service) {
        this.service = service;
    }
}
```

Dependency Injection (DI)

- Design pattern that **implements IoC**.
- **IoC containers** provide generic factories that create objects of specific abstract types.
- **Injection** is the passing of an actual object (the implementation) described as an abstract *dependency* (a generic service) to a *dependent* object (a client).
  - @EJB
  - @PersistenceContext, @PersistentUnit
  - @Resource (DataSource, UserTransaction, etc)

What is CDI?

- CDI is the Java **standard** for dependency injection(DI) and interception (AOP).
  - Like JPA did for ORM, CDI simplifies and sanitizes the API for DI and AOP.
- CDI is a set of services that **simplify** the use of enterprise beans along with JSF technology.
- Designed for use with **stateful objects**.
- It redefines the concept of **bean**.
- **JBoss Weld** - CDI Reference Implementation

  (other impl. Apache OpenWebBeans, used by Apache TomEE server)

What exactly is a “context”?

- ***Context*** = the circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood and assessed.
- Synonyms: frame of reference, conditions, factors, state of affairs, situation, background
- *“The problem is to decide what this means in the context in which the words are used.”*
- **A context is a container for objects which have been defined with a certain scope, offering them various services.**
- AppletContext, ServletContext, FacesContext, EJBContext, JMSContext, SOAPMessageContext, etc.

CDI Services

- **Contexts:** The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts.
- **Dependency injection:** The ability to inject components into an application in a typesafe way, including the ability to choose at deployment time *which implementation of a particular interface to inject.*
- Integration with the Expression Language (EL).
- The ability to **decorate** injected components.
- The ability to associate **interceptors**.
- An **event-notification** model.
- A **web conversation** scope.

Loose Coupling

Interconnecting components so they depend on each other as little as posibile.

- Implementation level: CDI **decouples the server and the client** by means of abstract types and **qualifiers**, so that the server implementation may vary
- Communication level:
  - **Decouples the lifecycles of collaborating components** by making components contextual, with automatic lifecycle management
  - **Decouples message producers from consumers**, by means of events
- **Decouples orthogonal concerns** by means of Java EE interceptors

Strong Typing

Knowing the type of a value before runtime.

CDI provides strong typing by:

- **Eliminating lookup using string-based names** for wiring and correlations, so that the compiler will detect typing errors.
- **Allowing the use of declarative Java annotations** to specify everything, making it easy to provide tools that introspect the code and understand the dependency structure at development time:

  → No XML-hell

The Concept of “Bean”

- JavaBeans: “reusable software components, conforming to a particular convention, used to encapsulate some data into a single object”.
- In CDI, a bean is **a source of contextual objects that define application state and/or logic**.
  - The lifecycle of its instances is managed by the container according to the lifecycle context model.
- Attributes of a bean:
  - A (nonempty) set of **bean types**
  - A (nonempty) set of **qualifiers**
  - A **scope**
  - Optionally, a bean EL **name**
  - A set of **interceptor bindings**
  - A bean **implementation**

Bean Types

- A bean type defines **a client-visible type of the bean**.
- Almost any Java type may be a bean type:

  an *interface*, a *concrete class*, an *abstract class*, an *array type* or a *primitive type*.
- **A bean may have multiple bean types**.

```
public class BookShop extends Business
                      implements Shop<Book> {
    ...
}
```

This bean has the following types:

- *BookShop*
- *Business*
- *Shop<Book>*
- *Object*

Beans as *Injectable Objects*

- (Almost) any Java class can be injected.

```
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

- **@Inject**

```
import greetings.Greeting;
import javax.inject.Inject;

public class Hello {
    @Inject Greeting greeting;

    public void sayHello(String name) {
        greeting.greet(name);
    }
}
```

@Inject
Identifies injectable constructors, methods, and fields. May apply to static as well as instance members. An injectable member may have any access modifier (private, package-private, protected, public).

Decoupling Implementations

- The *interface* of the service

```
public interface Greeting {
    String greet(String name);
}
```

- The *implementation* of the service

```
public class GreetingImpl implements Greeting {
     public String greet(String name) {
         return "Hello, " + name + ".";
     }
}
```

- The client

```
public class Hello {
    @Inject Greeting greeting;

    public void sayHello(String name) {
        greeting.greet(name);
    }
}
```

Injection Points

- Bean **constructor** parameter injection

```
public class Hello {
    private Greeting greeting;
    @Inject
    public Hello(Greeting greeting) {
        this.greeting = greeting;
    }
}
```

- **Initializer method** parameter injection

```
public class Hello {
    private Greeting greeting;
    @Inject
    public void setGreeting(Greeting greeting) {
        this.greeting = greeting;
    }
}
```

- **Direct field** injection

```
    @Inject Greeting greeting;
```

Constructors are injected first, followed by fields, and then methods. Fields and methods in superclasses are injected before those in subclasses. Ordering of injection among fields and among methods in the same class is not specified.

API Support for DI

- *javax.inject* Package

  This package specifies a means for obtaining objects in such a way as to maximize reusability, testability and maintainability compared to traditional approaches such as constructors, factories, and service locators (e.g., JNDI). This process, known as dependency injection, is beneficial to most nontrivial applications.

  - *Inject, Named, Qualifier, Scope* classes

- *javax.enterprise* Packages

  - *.context, .inject, .event*, etc.
  - Annotations and interfaces relating to scopes and contexts, bean and stereotype definition, events, etc.

beans.xml

CDI deployment descriptor (optional)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
            http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"

    bean-discovery-mode="annotated">
```

→ the container will scan for beans with annotated *scope types*.

```
</beans>
```

```
@ApplicationScoped
public class Greeting {
    ...
}
```

```
    bean-discovery-mode="all"
```

→ the container will scan all classes.

Type Identification

Just remember: "There can be only one."

CDI defines a simple **typesafe resolution algorithm** that helps the container decide what to do if there is more than one bean that satisfies a particular contract. However, problems may occur:

- Consider the following type definition:

```
public interface Greeting { ... }
```

- *Unsatisfied dependency - No bean matches the injection point*

```
@Inject Greeting greeting;
```

- *Ambiguous dependency*

```
public class Greeting1 implements Greeting { ... }

public class Greeting2 implements Greeting { ... }

@Inject Greeting greeting;
```

Circular Dependencies

- Circular dependencies between two constructors is an obvious problem, but you can also have a circular dependency between injectable fields or methods:

```
class A {
  @Inject B b;
}
class B {
  @Inject A a;
}
```

org.jboss.weld.exceptions.DeploymentException: WELD-001443: Pseudo scoped bean has circular dependencies.
Dependency path:
- Managed Bean [class B] with qualifiers [@Default @Any],
- [BackedAnnotatedField] @Inject B.b,
- Managed Bean [class A] with qualifiers [@Default @Any],
- [BackedAnnotatedField] @Inject A.a,

- When constructing an instance of A, a naive injector implementation might go into an infinite loop constructing an instance of B to set on A, a second instance of A to set on B, a second instance of B to set on the second instance of A, etc.
- A conservative injector might detect the circular dependency at build time and generate an error.
- **Scoping one of the dependencies** (using singleton scope, for example) may enable a valid circular relationship.

Qualifiers

A qualifier is an **annotation that you apply to a bean** in order to provide various implementations of a particular bean type. A qualifier may annotate an injectable field or parameter and, combined with the type, identify the implementation to inject.

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

```
@Default
public class Greeting1
             implements Greeting {
 public String greet(String name){
  return "Hello, " + name + "!";}
}
```

```
@Informal
public class Greeting2
             implements Greeting {
 public String greet(String name){
  return "Howdy, " + name + "!";}
}
```

```
public class Hello {
   @Inject @Informal Greeting greeting;
   ...
}
```

The Scope of a Bean

- The scope of a bean defines the lifecycle and visibility of its instances.
  - **@RequestScoped:** A user's interaction with a web application in a single HTTP request.
  - **@SessionScoped:** A user's interaction with a web application across multiple HTTP requests.
  - **@ApplicationScoped:** Shared state across all users' interactions with a web application.
  - **@Dependent**: The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).
  - **@ConversationScoped**

Example

```
@ApplicationScoped
public class Resources {
    private String message = "Hello";

    @PostConstruct
    public void afterCreate() {
        System.out.println("Message created");
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

```
@Stateless
public class MyComponent {

  @Inject
  public Resources res;

  public void myMethod(String str){
    res.setMessage(str);
  }
}
```

```
@EJB MyComponent myComp1;
@EJB MyComponent myComp2;
...
assertEquals("Hello",
    myComp1.res.getMessage());

myComp2.myMethod("Ciao");
assertEquals("Ciao",
    myComp1.res.getMessage());
```

@ConversationScoped

A **conversation** is a user's interaction within explicit developer-controlled boundaries that extend the scope across multiple invocations. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

```
@ConversationScoped @Stateful
public class OrderBuilder {
  private Order order;
  private @Inject Conversation conversation;
  private @PersistenceContext(type = EXTENDED) EntityManager em;
  public Order createOrder() {
    order = new Order();
    conversation.begin();
    return order;
  }
  public void addItem(Product product, int quantity) {
    order.add(new OrderItem(product, quantity));
  }
  @Remove
  public void saveOrder(Order order) {
    em.persist(order);
    conversation.end();
  }
}
```

Extended Persistence Context

Extended persistence context is a feature of the JPA specification that allows one to declare that **the injected EntityManager lives beyond the scope of the JTA transaction**, that is, to extend it to the lifetime of the stateful session bean itself.

First of all, within the extended persistence context, managed entities remain managed even after the transaction commits (or is rolled back). This behavior is very different than that of a transaction scoped context, were entities get detached at the end of the transaction.

Second implication important to note is that changes done to the managed entities from outside of a transaction will not be propagated to the database immediately, and will be buffered instead. In order to get the changes reflected in the database, the EntitiyManager needs to be accessed from a transaction context and flushed (by invoking it's flush() method). For obvious reasons, extended persistence context is only available within stateful Session Beans.

The most frequently quoted example of use is a multi-step conversation with a client, where the data that is to be persisted in a single transaction is acquired form a client in a series of steps with arbitrary periods of time in between. In this scenario, changes are sequentially applied to the managed entity in question, and the EntityManager is flushed at the very end of the conversation. In conclusion, it's worth to note that the same effect could have been achieved by wrapping the whole conversation in a single user-managed transaction, but that it still makes more sense to exploit the power of an extended persistence context as it provides both a simpler and cleaner solution.

Giving Beans EL Names

```
@Named("hello")
@RequestScoped
public class Hello {
  @Inject @Informal Greeting greeting;
  private String name;
  private String salutation;
  public void createSalutation() {
    this.salutation = greeting.greet(name);
  }
  public String getSalutation() { return salutation; }
  public void setName(String name) { this.name =name; }
  public String getName() { return name; }
}
```

JSF *@ManagedBean* are deprecated.

CDI annotations are used instead.

Using a Managed Bean in a Facelets Page

```
<h:form id="greetme">
  <p>
  <h:outputLabel value="Enter your name: " for="name"/>
  <h:inputText id="name" value="#{hello.name}"/> </p>
  <p>
  <h:commandButton value="Say Hello"
                   action="#{hello.createSalutation}"/> </p>
  <p><h:outputText value="#{hello.salutation}"/></p>
</h:form>
```

Using Alternatives

- When you have more than one version of a bean that you use for different purposes, you can choose between them during the development *phase* by injecting one **qualifier** or another.
- Instead of having to change the source code of your application, however, you can make the choice at deployment time by using **alternatives**.
  - Handle client-specific business logic that is determined at runtime.
  - Specify beans that are valid for a particular deployment scenario.
  - Create dummy (mock) versions of beans to be used for testing.

@Alternative

```
@Alternative //-> Specifies that this bean is an alternative.
public class SpecialDiscountImpl implements Discount {
  public void apply(ShoppingCart cart) { ... }
}
...
public class NormalDiscountImpl implements Discount {
  public void apply(ShoppingCart cart) { ... }
}
```

An alternative is not available for injection, lookup or EL resolution to classes or JSP/JSF pages in a module unless the module is a *bean archive* and the alternative is explicitly selected in that bean archive.

beans.xml (placed in META-INF or WEB-INF/classes)

```
<beans>
    <alternatives>
        <class>somepackage.SpecialDiscountImpl</class>
    </alternatives>
</beans>
```

Using Producer Methods

A *producer method* is a method that acts as a source of bean instances. Useful when you want to inject an object that is not itself a bean, when the concrete type of the object to be injected may vary at runtime or when the object requires some custom initialization that the bean constructor does not perform.

```
@ApplicationScoped
public class NumberGenerator {
  private static final int MAX = 100;
  @Produces @MaxNumber int getMaxNumber() {
    return MAX;
  }
  @Produces @Named @Random int getRandomNumber() {
    return new Random().nextInt(100);
  }
}
@Inject @MaxNumber int maxNumber; //Get the constant maxNumber

@Inject @Random Instance<Integer> randomInt;
this.number = randomInt.get(); //Get a random value (can vary at runtime)

//Even in a Unified EL expression:
<p>Your random number is #{randomNumber}.</p>
```

Producer Fields

A **producer field** is a simpler alternative to a producer method, a shortcut that lets us avoid writing a useless getter method. Producer fields are particularly useful for declaring Java EE resources.

```
@Produces
@UserDatabase
@PersistenceContext
private EntityManager em;
```

```
@Inject
@UserDatabase
EntityManager em;
```

Example

```
@ApplicationScoped
public class Resources implements Serializable {

    @Inject
    AuthService authService;

    @PersistenceContext(unitName = "MyDatabasePU_1")
    private EntityManager em1;

    @PersistenceContext(unitName = "MyDatabasePU_2")
    private EntityManager em2;

    @Produces
    @SessionScoped
    public EntityManager getEntityManager() {
        switch (authService.getTenant().getId()) {
            case 1:
                return em1;
            case 2:
                return em2;
            default:
                return null;
        }
    }
}
```

```
@Inject
EntityManager em;
```

Using Events in CDI

Dependency injection enables loose-coupling by allowing the implementation of the injected bean type to vary, either a deployment time or runtime. Events go one step further, **allowing beans to interact with no compile time dependency at all**.

Event producers raise events that are delivered to *event observers* by the container.

The event object carries state from producer to consumer. The event object is nothing more than an instance of a concrete Java class. An event may be assigned qualifiers, which allows observers to distinguish it from other events of the same type. The qualifiers function like topic selectors, allowing an observer to narrow the set of events it observes.

Event Producers / Observers

Event **producers** fire events using an instance of the parameterized **Event** interface. An instance of this interface is obtained by injection.

```
@Inject @Any Event<Document> documentEvent;
Document doc = new Document();
documentEvent.fire(document);

@Inject @Updated Event<Document> documentUpdatedEvent;
Document doc = new Document();
documentUpdatedEvent.fire(document);
```

An **observer** method is a method of a bean with a parameter annotated **@Observes**.

```
public void onAnyDocumentEvent(@Observes Document doc) {
   ...
}
public void afterDocumentUpdate(@Observes @Updated Document doc){
   ...
}
```

Example

```
@Named
@SessionScoped
public class PersonView extends DataView<Person, Integer> {
  @PostConstruct
  public void init() {
    loadData();
  }
  …
  public void onPersonEvent(@Observes Person person) {
     loadData();
  }
}
```

```
@Named
@SessionScoped
public class PersonEdit extends DataEdit<Person, Integer> {
  @Any
  Event<Person> dataEvent;
  …
  public void save() {
     …
     dataEvent.fire(person);
  }
}
```

Interceptors

- An interceptor is a class that is used to **interpose in method invocations** or lifecycle events that occur in an associated target class.
- The interceptor performs tasks, such as logging or auditing, that are **separate from the business logic** of the application and that are repeated often within an application. Such tasks are often called cross-cutting tasks. Interceptors allow you to specify the code for these tasks in one place for easy maintenance.
- When interceptors were first introduced to the Java EE platform, they were specific to enterprise beans. *You can now use them with Java EE managed objects of all kinds, including managed beans.*

Creating an Interceptor

Interceptor bindings are intermediate annotations that may be used **to associate interceptors with target beans.** An interceptor may specify multiple interceptor bindings.

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {}
```

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Valid {}
```

```
@Logged
@Interceptor
public class LoggedInterceptor implements Serializable {
  @AroundInvoke
  public Object logMethodEntry(InvocationContext invocationContext)
                throws Exception {
    System.out.println("Entering method: "
        + invocationContext.getMethod().getName() + " in class "
        + invocationContext.getMethod().getDeclaringClass().getName());
    return invocationContext.proceed();
  }
}
```

Using Interceptors

- At the class level

```
@Logged
public class Hello implements Serializable {…}
```

- At the method level

```
@Logged @Valid
public String sayHello() {…}
```

- Interceptors must be declared in *beans.xml*

```
<interceptors>
   <class>demo.interceptors.LoggedInterceptor</class>
   <class>demo.interceptors.ValidInterceptor</class>
</interceptors>
```

Decorators

Interceptors are a powerful way to capture and separate concerns which are orthogonal to the application (and type system). Any interceptor is able to intercept invocations of any Java type. This makes them perfect for solving technical concerns such as transaction management, security and call logging. However, by nature, interceptors are unaware of the actual semantics of the events they intercept. Thus, *interceptors aren't an appropriate tool for separating business-related concerns.*

The reverse is true of **decorators**. A decorator intercepts invocations only for a certain Java interface, and is therefore aware of all the semantics attached to that interface. Since decorators directly implement operations with business semantics, it makes them *the perfect tool for modelling some kinds of business concerns*. It also means that a decorator doesn't have the generality of an interceptor. Decorators aren't able to solve technical concerns that cut across many disparate types. Interceptors and decorators, though similar in many ways, are complementary.

Creating a Decorator

```
public interface Account {
  public void withdraw(BigDecimal amount);
  public void deposit(BigDecimal amount);
}
```

```
public class AccountImpl
       implements Account {
  ...
}
```

```
@Decorator
public abstract class LargeTransactionDecorator implements Account {
  @Inject @Delegate @Any Account account;

  public void withdraw(BigDecimal amount) {
    account.withdraw(amount);
    if (amount.compareTo(10_000) > 0) {
      System.out.println("Large transaction detected!");
    }
  }
  public abstract void deposit(BigDecimal amount); //not interested
}
```

We need to enable our decorator in the beans.xml descriptor of a bean archive.

```
<beans>
  <decorators>
    <class>demo.decorators.LargeTransactionDecorator</class>
  </decorators>
</beans>
```

@Transactional

- Provides the application the ability to declaratively control transaction boundaries on CDI managed beans, at both the class and method level.
- The TxType element of the annotation indicates whether a bean method is to be executed within a transaction context:
(MANDATORY, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW, SUPPORTS)

```
@RequestScoped
public class MyCDIBean {
  @PersistenceContext
  EntityManager em;

  @Transactional(TxType.REQUIRED)
  public void create(Person p) {
    em.persist(p);
  }
}
```

Transactional.TxType Enum

- **MANDATORY**
  If called outside a transaction context, a TransactionalException with a nested TransactionRequiredException must be thrown.
- **NEVER**
  If called outside a transaction context, managed bean method execution must then continue outside a transaction context.
- **NOT_SUPPORTED**
  If called outside a transaction context, managed bean method execution must then continue outside a transaction context.
- **REQUIRED**
  If called outside a transaction context, the interceptor must begin a new JTA transaction, the managed bean method execution must then continue inside this transaction context, and the transaction must be completed by the interceptor.
- **REQUIRES_NEW**
  If called outside a transaction context, the interceptor must begin a new JTA transaction, the managed bean method execution must then continue inside this transaction context, and the transaction must be completed by the interceptor.
- **SUPPORTS**
  If called outside a transaction context, managed bean method execution must then continue outside a transaction context.

Transactional Exception Handling

- **RuntimeException**s that are raised during the execution of a transactional business method cause the transaction to roll back by default.
- **Checked exceptions** do not result in the transactional interceptor marking the transaction for rollback.
  - The **rollbackOn** element can be set to indicate exceptions that must cause the interceptor to mark the transaction for rollback.
  - The **dontRollbackOn** element can be set to indicate exceptions that must not cause the interceptor to mark the transaction for rollback.

```
@Transactional(rollbackOn = InvalidPersonException.class,
               dontRollbackOn = LoggingException.class)
public void create(Person p) {
    entityManager.persist(p);
    log(p);
}
```

Bean Validation

Validating input received from the user to maintain data integrity is an important part of application logic. Validation of data can take place at different layers in even the simplest of applications.

An application may validate the user input (in the *h:inputText* tag) for numerical data, at the *presentation layer* and for a valid range of numbers at the *business layer*.

```
public class Person {
    @NotNull
    @Size(min=1, max=50)
    private String name;

    @Min(value = 18, message = "Age should not be less than 18")
    private int age;

    @Past
    private LocalDate birthday;
}
```

Method Constraints

```
public class Person {
    ...
    public void setName(@Size(min = 5) String name) {
        this.name = name;
    }


}
```

```
public class SomeOtherBean {
    @Inject Person pers;

    @PostConstruct
    public void init() {
        pers.setName("Test");
    }

}
```

javax.validation.ConstraintViolationException

Return Value Constraints

```
public class Person {
    …
    @NotNull
    public String createFullName() {
        return null;
    }

}
```

```
public class SomeOtherBean {
    @Inject Person pers;

    @PostConstruct
    public void init() {
        String fullName = pers.createFullName();
    }

}
```

javax.validation.ConstraintViolationException

Stereotypes

In many systems, use of architectural patterns produces a set of recurring bean roles. A *stereotype* allows a framework developer to identify such a role and declare some common metadata for beans with that role in a central place. A stereotype encapsulates any combination of: a default scope, and a set of interceptor bindings.

```
@RequestScoped
@Secure
@Transactional
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

Predefined: `@Model = @Named @RequestScoped`

Conclusion

- Contexts and Dependency Injection for the Java EE Platform (CDI) introduces a standard set of component management services to the Java EE platform.
- CDI manages the lifecycle and interactions of stateful components bound to well-defined contexts.
- CDI provides typesafe dependency injection.
- CDI provides interceptors and decorators to extend the behavior of components, an event model for loosely coupled components, and an SPI allowing portable extensions to integrate cleanly with the Java EE environment.
- EJBs offer more features (component pool, async operation, remote call, timer services, etc), however they require a “heavy” JavaEE server, while CDI beans have also support in micro-servers, suited for microservices architectures.