CONTEXTS AND DEPENDENCY INJECTION (1/2)

33

33

Contexts and Dependency Injection

- Example of **inversion of control**:
  - Configuration is done by the container instead of the component
- Automated lifetime management
  - Explicit scope for components
- Black box customization
  - Interceptors and decorators
- Type-safe discovery
  - No more name-based lookup

34

34

Contextual Lifetime Management

- **Scopes:**
  `@ApplicationScoped`
  `@SessionScoped`
  `@RequestScoped`

- **Qualifers:** `@Named` (deprecated), others defined by app

- Dependency injection: `@Inject`

35

35

Example

- Declaration

```
@SessionScoped
public class ShoppingCartBean
   implements IShoppingCart {...}
```

- Use

```
@Inject
IShoppingCart cart;
```

36

36

Producer Method

- Declaration

```
@ApplicationScoped
public class RandGenerator {
  private Random random =
   new Random(System.currentTimeMillis());
  @Produces @Named @Random int getRand() {
   return random.nextInt(100);
  }
}
```

- Use

```
@Inject @Random int rand;
```

37

37

Example

- Declaration

```
@ApplicationScoped
public class RandGenerator {
  private Random random =
   new Random(System.currentTimeMillis());
  @Produces @Named @Random int getRand() {
   return random.nextInt(100);
  }
}
```

- Use

```
@Inject @Random int rand;
```

38

38

Example

- Declaration

```
@ApplicationScoped
public class RandGenerator {
  private Random random =
   new Random(System.currentTimeMillis());
  @Produces @Named @Random int getRand() {
   return random.nextInt(100);
  }
}
```

- Use

```
@Inject @Random int rand;
```

39

39

CONTEXTS AND DEPENDENCY INJECTION (2/2)

40

40

Producer Method

```
public class StrategyGenerator {
  @Produces @SessionScoped ICalcStrategy getStrategy (
                                                      ) {
      }
  }
}
@Inject ICalcStrategy strategy;
```

41

41

Producer Method

```
public class StrategyGenerator {
  @Produces @SessionScoped ICalcStrategy getStrategy (
                                                      ) {
      switch (chooseStrategy()) {
            case MONEY_MARKET: return mm;
            case FUTURE: return f;
            case SWAP: return s;
            default: return null;
      }
  }
}
@Inject ICalcStrategy strategy;
```

???

42

42

Producer Method

```
public class StrategyGenerator {
  @Produces @SessionScoped ICalcStrategy getStrategy (
            @New @MoneyMarketStrategy ICalcStrategy mm,
            @New @FutureStrategy ICalcStrategy f,
            @New @SwapStrategy ICalcStrategy s) {
      switch (chooseStrategy()) {
            case MONEY_MARKET: return mm;
            case FUTURE: return f;
            case SWAP: return s;
            default: return null;
      }
  }
}
@Inject ICalcStrategy strategy;
```

43

43

Producer Method

```
public class StrategyGenerator {
  @Produces @SessionScoped ICalcStrategy getStrategy (
            @New @MoneyMarketStrategy ICalcStrategy mm,
            @New @FutureStrategy ICalcStrategy f,
            @New @SwapStrategy ICalcStrategy s) {
      switch (chooseStrategy()) {
            case MONEY_MARKET: return mm;
            case FUTURE: return f;
            case SWAP: return s;
            default: return null;
      }
  }
}
```

44

44

Defining a New Qualifier

```
public enum Calc {
  MONEY_MARKET,FUTURE,SWAP
}

@Qualifier
@Retention(RUNTIME)
@Target({METHOD,FIELD,PARAMETER,TYPE})
public @interface CalcStrategy {
  Calc value();
}
```

45

45

Producer Method

```
public class StrategyGenerator {
  @Produces @SessionScoped ICalcStrategy getStrategy (
            @New @CalcStrategy(Calc.MONEY_MARKET)
                  ICalcStrategy mm,
            @New @CalcStrategy(Calc.FUTURE)
                  ICalcStrategy f,
            @New @CalcStrategy(Calc.SWAP)
                  ICalcStrategy s) {
      switch (chooseStrategy()) {
            …
      }
  }
}
```

46

46

Disposer Method

- For manual finalization:

```
@Produces @RequestScoped
Connection open(...) {...}

void close
   (@Disposes Connection connection)
   { connection.close(); }
```

47

47

Type-Safe Dependency Injection

- Declaration

```
public class DatabaseConnectionFactory {
  @PersistenceContext(unitName="...")
  private EntityManager em;

  @Produces
  EntityManager getEntityManager() { return em; }
}
```

- Use

```
public class DatabaseClient {
  @Inject EntityManager entityManager;
}
```

48

48

INTERCEPTORS AND TRANSACTIONS

49

49

Aspect-Oriented Programming

- Separation of concerns
- CDI Decorators:
  - Separate logically separate parts of business logic
- CDI Interceptors:
  - Separate specification of cross-cutting logic

50

50

Decorators

- Example: Additional logic for large purchases

```
@Decorator
public abstract class CartLogger
   implements IShoppingCart {

  @Inject @Delegate @Any
  IShoppingCart cart;

  public void checkout () {
   … cart.checkout(…) …
  }
}
```

51

51

Interceptors

- Application defines qualifier for those points where cross-cutting logic should be injected

```
@InterceptorBinding
@Target({METHOD,TYPE})
@Retention(RUNTIME)
public @interface Transactional { ... }
```

52

52

Interceptors

- Use the qualifier with the @Interceptor annotation to identify cross-cutting logic:

```
@Transactional @Interceptor
public class TransactionInterceptor {
   @Resource UserTransaction transaction;
   @AroundInvoke public Object
        manageTransaction
        (InvocationContext ctx)
        throws Exception { ...}
```

53

53

Interceptors

- Use the qualifier to associate transactional semantics with business methods, or an entire bean:

```
@Transactional
public class ShoppingCart { ... }
```

54

54

TRANSACTIONS: BEAN-MANAGED VS CONTAINER-MANAGED

55

55

Application-Managed

```
public class ShoppingCart {

   EntityManager em;

   CartDAO cartDAO;

   void init() {
      EntityManagerFactory factory = Persistence
         .createEntityManagerFactory("cartPU");
      em = factory.createEntityManager();
      cartDAO = new CartDAO(em);
   }

   public void addPurchase(Item item) {
      cartDAO.add(item);  // executes em.persist(item);
   }
}
```

FAILS!

56

56

Container-Managed

```
@Stateless
public class ShoppingCart {

   @PersistenceContext(unitName="cart")
   EntityManager em;

   CartDAO cartDAO;

   @PostConstruct
   void init() {
      cartDAO = new CartDAO(em);
   }

   public void addPurchase(Item item) {
      cartDAO.add(item);  // executes em.persist(item);
   }
}
```

57

57

Bean-Managed

```
@RequestScoped
public class ShoppingCart {

   @PersistenceContext(unitName="cart")
   EntityManager em;

   CartDAO cartDAO;

   @PostConstruct
   void init() {
      cartDAO = new CartDAO(em);
   }

   public void addPurchase(Item item) {
      cartDAO.add(item);  // executes em.persist(item);
   }
}
```

FAILS!

58

58

Bean-Managed

```
@RequestScoped
public class ShoppingCart {

   @PersistenceContext(unitName="cart")
   EntityManager em;

   CartDAO cartDAO;

   @PostConstruct
   void init() {
      cartDAO = new CartDAO(em);
   }

   @Transactional public void addPurchase(Item item) {
      cartDAO.add(item);  // executes em.persist(item);
   }
}
```

59

59

Bean-Managed

```
@SessionScoped
public class ShoppingCart {

   @Inject
   CartDAO cartDAO;

   @Transactional public void addPurchase(Item item) {
      cartDAO.add(item);  // executes em.persist(item);
   }
}
```

60

60

Bean-Managed

```
@RequestScoped
public class CartDAO {

    @PersistenceContext
    private EntityManager entityManager;

    public add(Item item) {
        entityManager.persist(item);
    }

}
```

61

61