Modulo II
Técnicas para desenvolvimento de Software Ágil – Anotações

Prof. Ismael H F Santos

Bibliografia

- *JUnit in Action*

Agenda:

- Anotações
- JavaDoc

MA-JUNIT

Anotações

POJO x JavaBeans

- Um POJO (Plain Old Java Object) é a denominação que se dá a uma classe que em geral não deriva de nenhuma interface ou classe, contém atributos privados, um construtor padrão sem argumentos, e métodos publicos get/set para acessar os seus atributos.
- A denominação POJO foi criada com a intenção de defender a criação de designs mais simples em contraposição aos diferentes frameworks (em especial Enterprise Java Beans, antes de EJB 3.0) que vinham criando soluções cada vez mais complexas dificultando em muito a programação e tornando o reaproveitamento cada vez mais difícil.

POJO x JavaBeans

- O padrão JavaBeans foi criado com a intenção de se permitir a criação de objetos de interface reutilizáveis, isto é, componentes (sw reusable components) que possam ser manipulados visualmente por IDEs (Eclipse, Netbeans, etc) para a composição de novas aplicações.
- A especificação de um JavaBeans estabelece algumas convenções:
  - construtor padrão sem argumentos, para permitir a sua instanciação de forma simples pelas aplicações que os utilizem;

POJO x JavaBeans

- conjunto de propriedades implementados através de atributos privados, e métodos de acesso públicos get/set para acessar os seus atributos. Os métodos de acesso devem seguir a uma convenção padronizada para os seus nomes, getXxxx e setXxxx quando os métodos de acesso referentes propriedade xxxx;

- deve ser serializada, isto é, implementar a interface Serializable do pacote java.io. Isto permite que as aplicações e frameworks salvem e recuperem o seu estado de forma independente da JVM

POJO x JavaBeans

- Declarando um JavaBeans

```
// PersonBean.java
public class PersonBean implements java.io.Serializable{
  private String  name;
  private boolean deceased;
  // No-arg constructor (takes no arguments).
  public PersonBean() { }
  // Property "name" (capitalization) readable/writable
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  // Property "deceased"
  // Different syntax for a boolean field (is vs. get)
  public boolean isDeceased() { return this.deceased; }
  public void    setDeceased(boolean deceased) {
                              this.deceased = deceased; }
}
```

POJO x JavaBeans

- Usando um JavaBeans

```
// TestPersonBean.java
public class TestPersonBean {
  public static void main(String[] args) {
    PersonBean person = new PersonBean();
    person.setName("Bob");
    person.setDeceased(false);
    System.out.print(person.getName());
    System.out.println(person.isDeceased() ?
                       " [deceased]" : " [alive]"); }
}
=> Output: "Bob [alive]“
```

- Como as especificações de JavaBeans são baseadas em convenções e não implicam na criação de interfaces muitas vezes se diz que os JavaBeans são POJOs que seguem um conjunto de convenções específicas.

Anotações (*Java Annotations*)

- Anotações são uma forma especial de declaração de metadados que podem ser adicionadas ao código-fonte pelo programador. Provêem informações sobre o comportamento de um programa.
- São aplicáveis à classes, métodos, atributos e outros elementos de um programa, além disso, não tem nenhum efeito sobre a execução do código onde estão inseridas.
- Diferentemente de comentários javadoc, anotações são reflexivas no sentido de que elas podem ser embutidas nos arquivos de definição classes (byte-codes) gerados pelo compilador podendo ser “consultadas” durante a sua execução pela JVM.

Anotações (*Java Annotations*)

- Anotações tem diferentes utilidades, entre elas:
  - Instruções para o compilador – detecção de erros e supressão de warnings;
  - Instruções para uso em tempo de compilação e em tempo de “deployment” – usadas para geração de código, arquivos XML;
  - Instruções para tempo de execucao (runtime) – criação de informações que podem ser consultadas durante a execução;

Anotações (*Java Annotations*)

- Exemplos:

```
@Author( name = "Benjamin Franklin", date = "3/27/2003" )
class MyClass() { }                          or
@SuppressWarnings(value = "unchecked")
void myMethod() { }                          or
@SuppressWarnings("unchecked") // caso só exista um elemento “value”
void myMethod() { }                          or
@Override                      // caso não existam elementos na anotação
void mySuperMethod() { }
```

- As anotações podem possuir nenhum, um ou mais de um elemento em sua definição. Se um tipo de anotação possui elementos, o valor para cada atributo deve ser passo entre parênteses.

```
@interface TesteAnotacao {    // definição da anotação
  String nomeUser(); int idade();
}
......
@TesteAnotacao(nomeUser= "Fulano de tal", idade= 25 ) // uso da anotação
public void metodoTeste{
  System.out.println(“o usuario” + nomeUser + “idade: ” + idade + ...);
}
```

Anotações (*Java Annotations*)

- Exemplo anotações para documentação – declaração :

```
import java.lang.annotation.*; // import this to use @Documented
@Documented                    // anotação de documentação
@interface ClassPreamble {
  String author();
  String date();
  int currentRevision() default 1;
  String lastModified() default "N/A";
  String lastModifiedBy() default "N/A";
  String[] reviewers(); // Note use of array
}
```

- uso da anotação no código fonte:

```
@ClassPreamble {
  author = "John Doe", date = "3/17/2002",
  currentRevision = 6, lastModified = "4/12/2004",
  lastModifiedBy = "Jane Doe“,
  reviewers = {"Alice", "Bob", "Cindy"} // Note array notation
}
public class Generation3List extends Generation2List { //class code... }
```


Anotações (*Java Annotations*)

- A anotação anterior irá gerar a seguinte saída do javadoc:

```
public class Generation3List extends Generation2List {
  // Author: John Doe
  // Date: 3/17/2002
  // Current revision: 6
  // Last modified: 4/12/2004
  // By: Jane Doe
  // Reviewers: Alice, Bill, Cindy
  // class code goes here
}
```

- Exemplo anotações para compilação (J2SE built-in)
  - A anotação @Override informa ao compilador que o elemento anotado tem o objetivo de sobrescrever o elemento definido na superclasse.

```
@Target(ElementType.METHOD)
public @interface Overrides {}
```

  - Usando a anotação ....

```
class A extends B {
  @Overrides
  void myMethod() { }   // marca o método como um método que esta
  .....                 // sobrescrevendo um método da superclasse
}
```

Anotações (*Java Annotations*)

- Exemplo anotações para compilação (cont.)
  - Embora esta anotação não seja necessária ela ajuda ao compilador informar ao programador caso exista alguma inconsistência entre a declaração do método na classe derivada e a da superclasse
- Outros exemplos de anotações (J2SE built-in)
  - @Documented informa que a anotação será utilizada pelo javadoc ou tools similares.

```
@Documented
@Target(ElementType.ANNOTATION_TYPE)
@public @interface Documented { }
```

  - @Deprecated informa ao compilador para avisar os usuários que se utilizem da classe, método ou atributo anotado que o uso do objeto não é mais recomendado.

```
@Documented
@Retention(RetentionPolicy.SOURCE)
@public @interface Deprecated { }
```

  - @Inherited informa que a anotação será herdada por todas as subclasses da classe anotada herdarão a mesma anotação automaticamente.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
@public @interface Inherited { }
```


Anotações (*Java Annotations*)

- Outros exemplos de anotações (J2SE built-in)
  - @Retention informa a vida útil da anotação. Podendo ser: SOURCE, CLASS ou RUNTIME. O default é SOURCE.

```
@Documented
@Retention(RetentionPolcy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
@public @interface Retention { RetentionPolicy value(); }
```

  - @Target usada para informar o tipo do elemento sobre o qual a anotacao pode ser associada (classe, método, ou campo). Quando não estiver presente significa que a anotacao pode ser aplicada a qualquer elemento do programa.

```
@Documented
@Retention(RetentionPolcy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
@public @interface Target { ElementType[] value(); }
```

- O JSE 5.0 inclui a ferramenta APT (annotation processing tool) que pode ser usada para ler um programa Java e tomar ações baseadas nas anotações declaradas.

Anotações (*Java Annotations*)

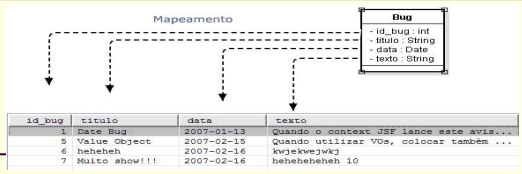
- Para disponibilizar uma anotação que pode ser aplicada a classes e métodos para implementar uma política de verificação de segurança em tempo de execução (runtime) devemos declará-la da seguinte forma:

```
import java.lang.annotation.*; // import this to use @Retention
@Documented
@Target({METHOD, TYPE})
@Retention(RetentionPolicy.RUNTIME)  // available in RUNTIME
public @interface SecurityPermission {
  // Elements that give information for runtime processing
  String[] value();
}
```

  - Usando a anotação ....

```
@SecurityPermission("ALL") // marca a classe sem restrição de acesso
public class AnnotationTest {
 public AnnotationTest() { SecurityChecker.checkPermission(); ... }
 @SecurityPermission("None") // marca o método com restrição de acesso
 void Method1() { SecurityChecker.checkPermission(); ... }
 @SecurityPermission("ALL") // marca o método sem restrição de acesso
 void Method2() { SecurityChecker.checkPermission(); ... }
}
```


Exemplo JPA



- Exemplo 2 - JPA

Listagem 01. Bug.java

```
package exemploJPA;
import javax.persistence.*;
@Entity
@Table(name="BUGS")
public class Bug {
  private Integer idBug;
  private String titulo;
  private java.util.Date data;
  private String texto;
  public Bug() {}
    @Id
    @GeneratedValue(strategy=
                    GenerationType.SEQUENCE)
    // informa que o id será gerado pelo DB.
    @Column(name="idBug")
    public Integer getIdBug(){ return idBug;}
    public void setIdBug(Integer iBug) {
        this.idBug = idBug;
    }
    @Column(name="titulo")
    public String getTitulo() { return titulo;}
    public void setTitulo(String titulo){
      this.titulo = titulo;
    }
    @Temporal(TemporalType.DATE)
    @Column(name="data")
    public Date getData(){return data; }
    public void setData(Date data) {
        this.data = data; }
    @Column(name="texto")
    public String getTexto(){ return
    texto; }
    public void setTexto(String texto) {
        this.texto = texto; }
    @Override
    public String toString(){
        return "ID: "+this.id_bug; }
}
```

Listagem 02. Recuperar objeto.

```
public Object findByPk( int pKey ) {
       EntityManager em = getEntityManager();
       return em.find(Bug.class, pKey);
}
```

Exemplo **JPA Annotations**

- JPA define as anotações: @Entity, que torna uma classe persistente. A anotação @Table informa o nome da tabela, através do atributo name, para a qual a classe será mapeada. Quando o nome da tabela é igual ao nome da classe, está anotação não precisa ser informada, basta a anotação @Entity.

```
@Entity
@Table(name = "BUGS")
public class Bug implements Serializable { ... }
```

- A anotação @Id define o atributo a que ela se refere como o identificador único para os objetos desta classe, ie, a chave primária da tabela. Já a anotação @GeneretedValue faz com que o framework de persistência gere valores para a chave primária na tabela de forma automática e com valores únicos. Estratégias: AUTO, SEQUENCE, IDENTITY e TABLE.

```
public class Bug implements Serializable {
    @Id @GeneratedValue(strategy= GenerationType.SEQUENCE)
    private Long idBug;  ...
}
```


Exemplo **JPA Annotations**

- Mapeamento de atributos primitivos não são necessários. Opcionalmente todo atributo pode ser anotado com @Column que tem os seguintes elementos: name, lenght, nullable, unique, etc.

```
public class Bug implements Serializable {
  @Column (name="comNome", length=30, nullable=false,
           unique=true)
  private String nome;
     ...
}
```

- Em atributos para datas (Date ou Calendar), deve ser usada a anotaçao @Temporal para definir o tipo de informacao desejada, podendo ser um dos tipos DATE, TIME, TIMESTAMP

```
public class Bug implements Serializable {
  @Temporal(TemporalType.DATE)
  private Date data;
     ...
}
```

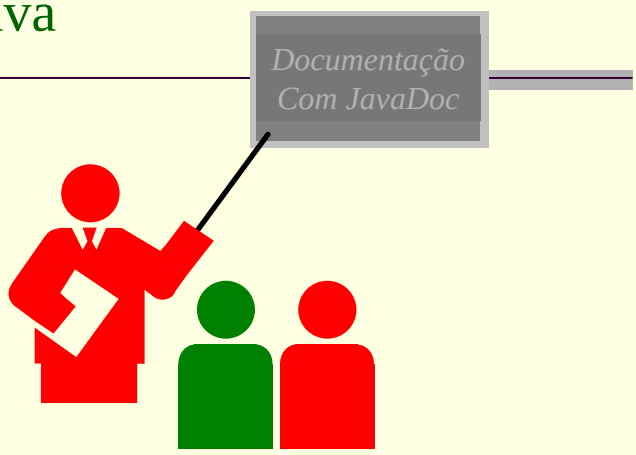
Exemplo **JPA Annotations**

- Os atributos da classe marcados com @Transient não serão salvos no BD e os que não possuem anotações associadas são mapeados com anotações padrões que mapeiam os atributos para colunas na tabela com tipos padrões, por exemplo, o atributo nome do tipo String é mapeado para uma coluna nome da tabela do tipo VARCHAR.
- Este tipo de comportamento caracteriza o que chamamos de Configuração por Exceção (Configuration by Exception), isto permite ao mecanismo de persistência aplicar defaults que se aplicam a maioria das aplicações fazendo que o usuário tenha que informar apenas os casos em que o valor default não se aplica.

Exemplo **JPA Annotations**

- Como vimos, anotações em JPA seguem o pattern Decoratior e podem ser divididas em 2 tipos básicos:
  - Physical annotations - @Table
  - Logical anotations - @ManyToOne, etc
- Configuração por decoração permite:
  - Refinar o modelo fisico do BD
    - @Table, @SecondaryTable, @Column
  - Definir o mapeamento lógico objeto-relacional
    - @Version, @Transient, @Embeddable
  - Definir a estratégia de geração dos identificadores
    - @GeneratedValue
    - @SequenceGenerator / @TableGenerator

POO-Java



Documentação

- Ferramenta **javadoc**
- Documentação a partir de comentários, colocados no código fonte
- Formato HTML: permite visualização via *browser*
- Manual do usuário × Guia de referência
- A saída (conteúdo e formato) gerada pelo Javadoc pode ser customizada através do uso de doclets.

Uso do **javadoc**

- Comentário especiais /** **...** ***/***
  - se referem ao próximo identificador definido
  - permitem o uso de *tags* HTML
- Parágrafos especiais
  - documentam assinaturas de métodos
  - fazem referências cruzadas
  - sinalizam código depreciado
  - identificam autoria

Exemplo

```
package org.enterprise.myapp;
/**
 * Classe exemplo. Documentada com javadoc.
 * <p>
 * Exemplo de uso tags HTML (see the {@link Exemplo2} class)
 * @since 1.0
 * @see org.enterprise.myapp
 */
public class Exemplo {
  /**
   * Divide um número por dois. Esse método retorna a
   * divisão inteira por 2 do número fornecido.
   * @param i número a ser dividido.
   * @return resultado da divisão inteira por 2.
   */
  public int div2(int i) { return i/2; }
}
```

Parágrafo **@param**

- Documenta um parâmetro de um método
- Recebe o nome do parâmetro e sua descrição
- Exemplo:

```
@param id Identificador a ser buscado
```


Parágrafo **@return**

- Documenta o valor de retorno de um método
- Recebe a descrição do valor
- Exemplo:

```
@return Nome do elemento encontrado
```

Parágrafo **@exception**

- Documenta uma exceção gerada por um método
- Recebe o tipo da exceção e sua descrição
- Exemplo:

```
@exception IdNotFound Identificador não encontrado
```


Parágrafo **@see**

- Cria de uma referência cruzada
- Recebe o nome de um identificador
- Exemplo:

```
@see estruturas.Coleção#insere
```

Parágrafo **@deprecated**

- Marca um identificador como depreciado
- Qualquer código que utilize o identificador receberá um aviso em tempo de compilação
- Recebe uma descrição
- Exemplo:

```
@deprecated Esse método foi descontinuado. Use o método
   xxx.
```


Parágrafo **@author**

- Identifica o autor do código
- Recebe um nome
- Exemplo:

```
@author João José
```

Parágrafo **@version**

- Identifica a versão do código
- Recebe o identificador da versão
- Exemplo:

```
@version 1.0β
```


Parágrafo **@since**

- Especifica a versão onde o identificador foi introduzido
- Recebe o identificador da versão
- Exemplo:

```
@since 1.0β
```