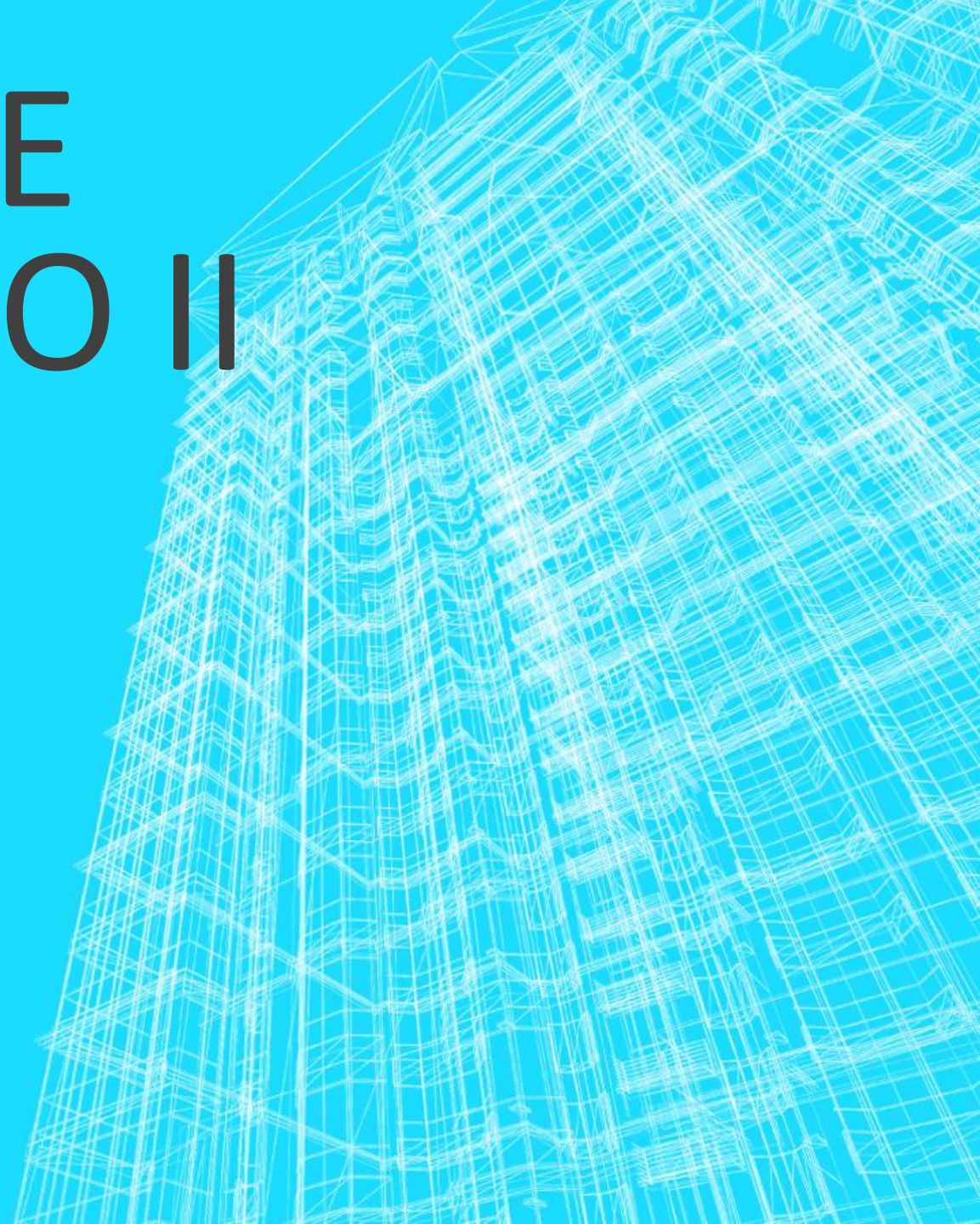


LINGUAGEM DE PROGRAMAÇÃO II

AULA 14

Prof. Dr. Alan de Oliveira Santana
alandeoliveirasantana@gmail.com





OBJETIVOS

- Compreender o que são anotações (annotations) em Java e por que elas existem.
- Entender como o compilador e frameworks usam anotações para meta-informação.
- Aprender a usar as anotações padrão da linguagem (@Override, @Deprecated, @SuppressWarnings, etc.).
- Criar anotações personalizadas, com parâmetros e níveis de retenção.
- Explorar como reflexão (reflection) pode ler anotações em tempo de execução.



O QUE SÃO ANOTAÇÕES

- As anotações (ou annotations) são metadados, isto é, informações sobre o código que não alteram diretamente sua execução, mas podem ser interpretadas por compiladores, ferramentas, IDEs ou frameworks.

O QUE SÃO ANOTAÇÕES

- Elas são indicadas pelo símbolo @ antes de seu nome:

```
@Override  
public String toString() {  
    return "Aluno";  
}
```

- Neste exemplo, @Override informa ao compilador que o método sobrescreve outro da superclasse.
- Se o nome ou a assinatura estiver errada, o compilador gera erro.



PARA QUE SERVEM AS ANOTAÇÕES?

- As anotações permitem:
 - Dar instruções ao compilador (como avisos, validações e otimizações).
 - Configurar comportamento de frameworks, sem precisar de XML ou código repetitivo.
 - Gerar código automaticamente, via processadores de anotação.
 - Controlar execução de programas (por exemplo, frameworks de testes ou injeção).
- Elas não têm efeito direto no código, seu valor está em quem as interpreta.



TIPOS DE USO DE ANOTAÇÕES

- Em código-fonte, para indicar comportamentos.
 - Ex: @Deprecated, @SuppressWarnings
- Em bibliotecas e frameworks, como forma de configuração.
 - Ex: @Entity, @Id no JPA, @Autowired no Spring.
- Em ferramentas e build systems, como Maven, Lombok, etc.
 - Ex: @Getter, @Builder.

PRINCIPAIS ANOTAÇÕES PADRÃO DO JAVA

Anotação	Função	Onde usar
@Override	Indica sobrescrita de método da superclasse	Métodos
@Deprecated	Marca elemento como obsoleto	Classes, métodos, campos
@SuppressWarnings	Evita avisos específicos do compilador	Qualquer elemento
@SafeVarargs	Suprime alertas de varargs com generics	Métodos com varargs
@FunctionalInterface	Indica que a interface deve ter um único método abstrato	Interfaces
@Retention	Define até quando a anotação existe (fonte, classe ou execução)	Declaração da anotação
@Target	Define onde a anotação pode ser usada (método, campo, tipo...)	Declaração da anotação



EXEMPLOS SIMPLES

- Sem @Override, o compilador não reclamaria se você errasse o nome do método.
- Com ele, a checagem é feita automaticamente.

```
class Animal {  
    void emitirSom() {}  
}  
  
class Cachorro extends Animal {  
    @Override  
    void emitirSom() {  
        System.out.println("Au au!");  
    }  
}
```




@DEPRECATED

- O compilador mostrará um aviso ao usar `conectarAntigo()`, indicando que há uma versão mais recente.

```
@Deprecated  
void conectarAntigo() {  
    // código antigo  
}  
  
void conectarNovo() {  
    // novo método  
}
```



@SuppressWarnings

- Essa anotação suprime o aviso de tipo unchecked, útil em código legado.

```
@SuppressWarnings("unchecked")  
void exemplo() {  
    List lista = new ArrayList();  
    lista.add("Texto");  
}
```



CRIANDO SUAS PRÓPRIAS ANOTAÇÕES

- Em Java, você pode definir novas anotações com @interface:

```
public @interface Autor {  
    String nome();  
    String data();  
}
```

```
@Autor(nome="Alan Santana", data="29/02/2028")  
public class Relatorio {  
    // ...  
}
```



PARÂMETROS EM ANOTAÇÕES

- Os parâmetros (também chamados elementos) são definidos como métodos abstratos dentro da anotação.



PARÂMETROS EM ANOTAÇÕES

```
public @interface Tarefa {  
    String descricao();  
    int prioridade() default 1;  
}
```

- Uso

```
@Tarefa(descricao="Implementar login", prioridade=2)  
public class Sistema {}
```

- Se você omitir prioridade, o valor default é usado



CONTROLANDO ONDE E QUANDO A ANOTAÇÃO EXISTE

- Duas anotações meta (ou “meta-anotações”) definem comportamento da sua anotação:
- `@TargetDefine` onde a anotação pode ser aplicada.

```
@Target({ElementType.METHOD, ElementType.TYPE})  
public @interface Auditado {}
```

- (outras opções: FIELD, PARAMETER, CONSTRUCTOR, etc.)



CONTROLANDO ONDE E QUANDO A ANOTAÇÃO EXISTE

- Define até quando a anotação é mantida.

```
@Retention(RetentionPolicy.RUNTIME)  
public @interface Auditado {}
```



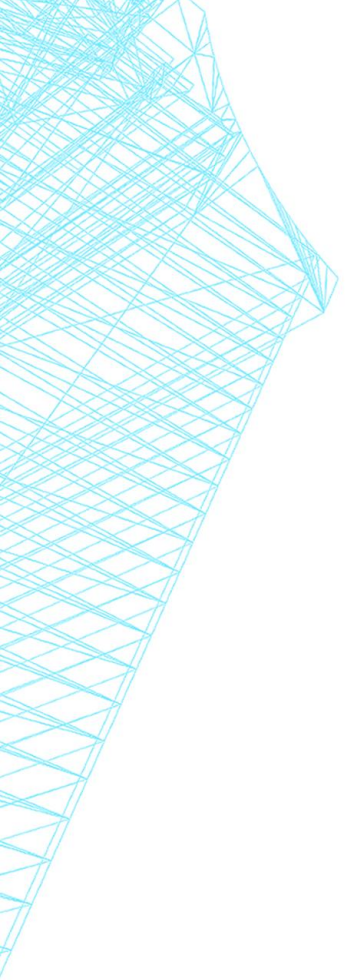
CONTROLANDO ONDE E QUANDO A ANOTAÇÃO EXISTE

Tipo de Retenção	Descrição
SOURCE	Somente no código fonte, removida na compilação.
CLASS	Mantida no .class, mas não acessível em tempo de execução.
RUNTIME	Disponível em tempo de execução (usada por frameworks/reflection).



LENDO ANOTAÇÕES EM TEMPO DE EXECUÇÃO

- Você pode usar reflection para inspecionar anotações dinamicamente.



```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Info {
    String autor();
    double versao();
}

@Info(autor="Alan", versao=1.1)
class Sistema {}

public class Teste {
    public static void main(String[] args) {
        Class<Sistema> clazz = Sistema.class;
        Info anot = clazz.getAnnotation(Info.class);
        System.out.println("Autor: " + anot.autor());
        System.out.println("Versão: " + anot.versao());
    }
}
```




USO PRÁTICO EM FRAMEWORKS

- Muitos frameworks Java usam anotações para automatizar tarefas:

Framework	Exemplo	Função
Spring	@Autowired	Injeta dependências automaticamente.
Hibernate/JPA	@Entity, @Id	Define mapeamento objeto-relacional.
JUnit	@Test, @BeforeEach, @AfterEach	Marca métodos de teste automatizado.
Jakarta REST (JAX-RS)	@GET, @POST, @Path	Define endpoints RESTful.
Lombok	@Getter, @Setter, @Builder	Gera código automaticamente em tempo de compilação.



EXEMPLO REAL: JUNIT

```
import org.junit.jupiter.api.Test;

public class ExemploTeste {
    @Test
    void somaDeveSerCorreta() {
        int resultado = 2 + 3;
        assert(resultado == 5);
    }
}
```

- Aqui, `@Test` indica ao JUnit que este método deve ser executado como teste unitário, sem precisar de um main.



META-ANOTAÇÕES PRINCIPAIS

Meta-anotação	Descrição
@Retention	Define o ciclo de vida (SOURCE, CLASS, RUNTIME).
@Target	Define onde pode ser aplicada (classe, método, campo, etc.).
@Documented	Inclui a anotação no Javadoc.
@Inherited	Permite herança da anotação por subclasses.
@Repeatable	Permite aplicar a mesma anotação várias vezes.



EXEMPLO COM @REPEATABLE

- Exemplo da versão básica e reduzida:

```
@Times({  
    @Time("08:00"),  
    @Time("14:00"),  
    @Time("20:00")  
})  
public class Servico {}
```

```
@Time("08:00")  
@Time("14:00")  
@Time("20:00")  
public class Servico {}
```



ANOTAÇÕES E PROCESSAMENTO AUTOMÁTICO

- O Java permite criar Annotation Processors, que inspecionam e transformam o código durante a compilação.
- Esses processadores são usados, por exemplo, pelo Lombok e MapStruct, para gerar código automaticamente (como getters, setters ou mapeadores).
- Ferramentas como APT (Annotation Processing Tool) ou bibliotecas modernas de build (Gradle/Maven) registram esses processadores.



REFLECTION EM JAVA

- Reflection é o mecanismo da linguagem Java que permite a um programa inspecionar e manipular a própria estrutura do código em tempo de execução.
- Ou seja: o programa consegue descobrir informações sobre classes, métodos, atributos, construtores e anotações, e até executá-los dinamicamente, mesmo que não saiba seus nomes em tempo de compilação.



POR QUE ISSO É ÚTIL?

- Sem Reflection, o Java precisa saber tudo em tempo de compilação:

```
Aluno a = new Aluno();  
a.getNome();
```

- Com Reflection, você pode descobrir e chamar métodos de forma dinâmica, mesmo sem saber o tipo:

```
Class<?> clazz = Class.forName("Aluno");  
Object obj = clazz.getDeclaredConstructor().newInstance();  
Method m = clazz.getMethod("getNome");  
System.out.println(m.invoke(obj));
```



POR QUE ISSO É ÚTIL?

- Esse código:
 - Descobre a classe Aluno pelo nome.
 - Cria um objeto dela, sem usar new diretamente.
 - Invoca o método `getNome()` mesmo sem conhecer a classe em tempo de compilação.
- É isso que os frameworks fazem para injeção de dependência, testes automáticos, mapeamento de entidades, etc.



A CLASSE CLASS

- Toda classe Java tem associada uma instância da classe especial `java.lang.Class`.

```
String s = "Teste";  
Class<?> c = s.getClass();  
System.out.println(c.getName());
```

- Saída

```
java.lang.String
```



A CLASSE CLASS

- Com o objeto Class, você pode descobrir:
 - Nome da classe (getName())
 - Pacote (getPackage())
 - Métodos (getMethods())
 - Atributos (getFields())
 - Construtores (getConstructors())



PRINCIPAIS CLASSES DO PACOTE JAVA.LANG.REFLECT

- Essas classes permitem explorar e manipular tudo o que está dentro de um .class.

Classe	Função
Class	Representa uma classe carregada na JVM.
Field	Representa um atributo (variável de instância).
Method	Representa um método.
Constructor	Representa um construtor da classe.
Parameter	Representa um parâmetro de método.
Modifier	Verifica modificadores (public, private, etc.).

EXEMPLO PRÁTICO

```
import java.lang.reflect.*;

class Pessoa {
    private String nome = "Alan";
    public void ola() {
        System.out.println("Olá, " + nome);
    }
}
```

```
public class TesteReflection {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Pessoa.class;

        System.out.println("Classe: " + clazz.getName());
        for (Field f : clazz.getDeclaredFields()) {
            System.out.println("Atributo: " + f.getName());
        }
        for (Method m : clazz.getDeclaredMethods()) {
            System.out.println("Método: " + m.getName());
        }

        Object obj = clazz.getDeclaredConstructor().newInstance();
        Method m = clazz.getMethod("ola");
        m.invoke(obj); // executa o método dinamicamente
    }
}
```



REFLECTION + ANNOTATIONS

- O Reflection é o que permite ler anotações em tempo de execução.



REFLECTION + ANNOTATIONS

```
@Retention(RetentionPolicy.RUNTIME)
@interface Autor {
    String nome();
}

@Autor(nome = "Alan")
class Relatorio {}

public class TesteAnot {
    public static void main(String[] args) {
        Class<Relatorio> clazz = Relatorio.class;
        Autor a = clazz.getAnnotation(Autor.class);
        System.out.println("Autor: " + a.nome());
    }
}
```



REFLECTION + ANNOTATIONS

- No código mostrado:
 - O Reflection lê a anotação `@Autor` da classe `Relatorio`.
 - E imprime o valor definido dentro dela.
- Sem Reflection, isso seria impossível, o Java não teria como “ler” uma anotação após a compilação.



MODIFICANDO VALORES COM REFLECTION

- Reflection também permite alterar atributos privados, o que normalmente não é permitido.



MODIFICANDO VALORES COM REFLECTION

```
import java.lang.reflect.Field;

class Conta {
    private double saldo = 100.0;
}

public class Teste {
    public static void main(String[] args) throws Exception {
        Conta c = new Conta();
        Field f = c.getClass().getDeclaredField("saldo");
        f.setAccessible(true); // quebra o encapsulamento
        f.set(c, 500.0);
        System.out.println("Saldo alterado!");
    }
}
```



MODIFICANDO VALORES COM REFLECTION

- Atenção:
 - Isso deve ser usado com cuidado.
 - Reflection quebra o encapsulamento e pode afetar segurança e desempenho.



PERFORMANCE E SEGURANÇA

Questão	Impacto
Desempenho	Reflection é mais lento que acesso direto, pois exige verificações e permissões extras.
Segurança	Pode violar encapsulamento (setAccessible(true)). Em ambientes restritos, pode ser bloqueado.
Manutenibilidade	Código reflexivo é mais difícil de depurar.
Uso ideal	Somente quando for realmente necessário — ex: frameworks, serialização, ORM, testes.



APLICAÇÕES PRÁTICAS DE REFLECTION

Contexto	Uso
JUnit	Executa automaticamente métodos anotados com <code>@Test</code> .
Spring	Injeta dependências (<code>@Autowired</code>) e configura beans.
Hibernate	Lê anotações de mapeamento (<code>@Entity</code> , <code>@Column</code>) para gerar SQL.
Lombok	Gera métodos em tempo de compilação, lendo anotações como <code>@Getter</code> .
Serialização	Converte objetos em JSON/XML sem precisar de código manual.

OBRIGADO!