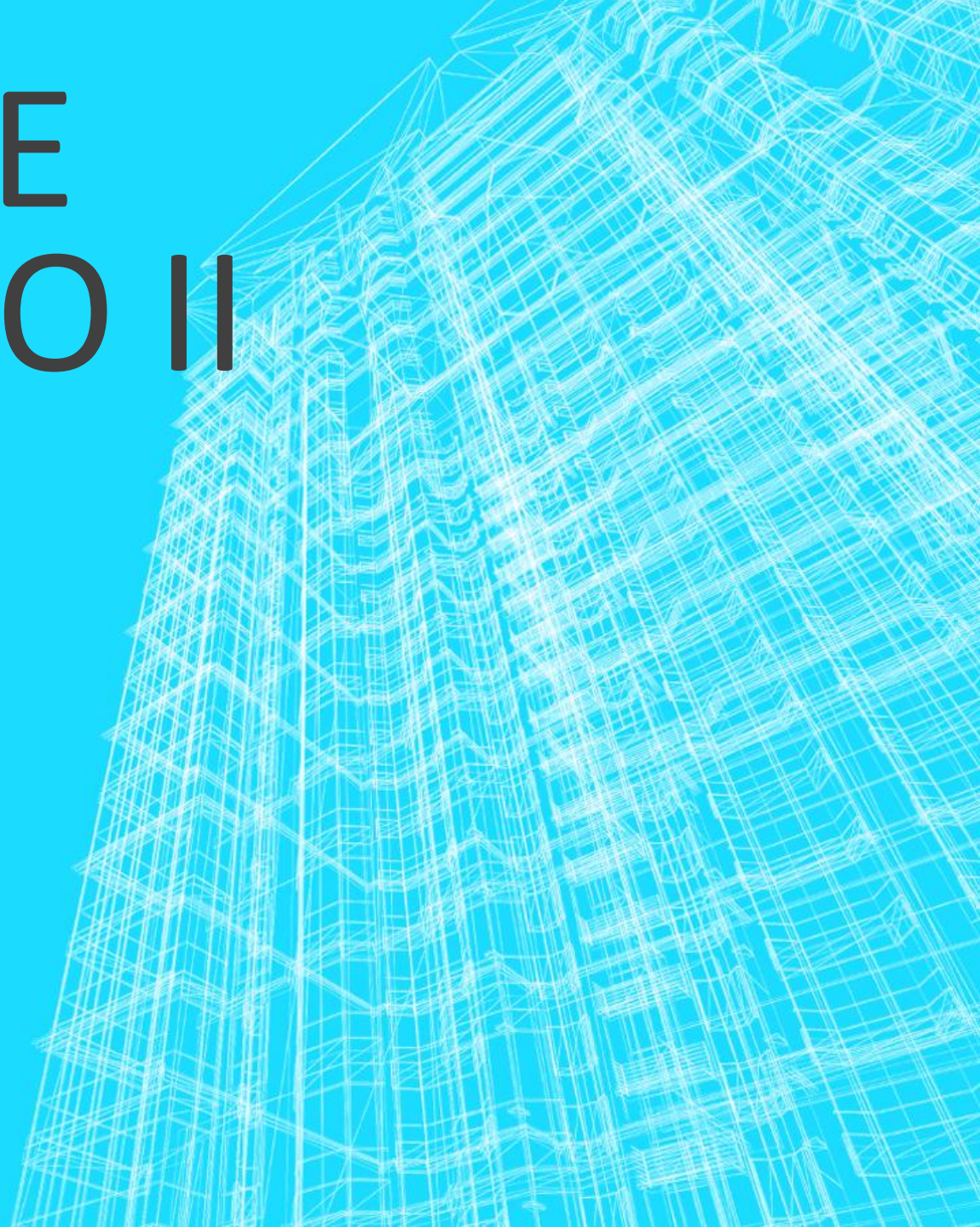
LINGUAGEM DE PROGRAMAÇÃO II
AULA 12



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

OBJETIVOS

- Compreender o conceito de tipagem genérica e sua importância em Java.
- Criar e utilizar classes, interfaces e métodos genéricos.
- Entender como os Generics melhoram a segurança de tipos em tempo de compilação.
- Diferenciar o uso de tipos genéricos e tipos brutos (raw types).
- Aplicar Generics em coleções (List, Set, Map) e em classes criadas pelo próprio aluno.
- Reconhecer boas práticas e limitações no uso de Generics.

INTRODUÇÃO

- Antes da introdução dos Generics (no Java 5), era comum usar coleções que armazenavam qualquer tipo de objeto.
- Isso tornava o código propenso a erros, já que era preciso realizar conversões (casts) manuais.

INTRODUÇÃO

- Com os Generics, Java passou a permitir que classes, interfaces e métodos trabalhassem de forma parametrizada, ou seja, com tipos genéricos, mantendo a flexibilidade e, ao mesmo tempo, garantindo segurança de tipos em tempo de compilação.
- Em resumo, Generics permitem que escrevamos uma classe ou método que funcione para vários tipos, sem duplicar código.

EXEMPLO DE PROBLEMA SEM GENERICS

- Código

```
import java.util.ArrayList;
import java.util.List;

public class ExemploSemGenerics {
    public static void main(String[] args) {
        List lista = new ArrayList(); // lista sem tipo (raw type)
        lista.add("Ana");
        lista.add(25); // compila, mas gera erro em tempo de execução

        for (Object obj : lista) {
            String nome = (String) obj; // ClassCastException!
            System.out.println(nome);
        }
    }
}
```

EXEMPLO DE PROBLEMA SEM GENERICS

- O código anterior compila, mas falha em tempo de execução, pois um número (25) é forçado a ser String.

EXEMPLO COM GENERICS

- Código

```
import java.util.ArrayList;
import java.util.List;

public class ExemploComGenerics {
    public static void main(String[] args) {
        List<String> nomes = new ArrayList<>();
        nomes.add("Ana");
        nomes.add("Carlos");
        // nomes.add(25); // Erro em tempo de compilação!

        for (String nome : nomes) {
            System.out.println(nome.toUpperCase());
        }
    }
}
```

EXEMPLO COM GENERICS

- Aqui, o compilador impede a inserção de um tipo errado.
- O uso de List<String> indica que a lista só aceitará objetos do tipo String.

CRIANDO UMA CLASSE GENÉRICA

- Código

CRIANDO UMA CLASSE GENÉRICA

- Uso da classe genérica:

```
public class Caixa<T> {
    private T conteudo;

    public void guardar(T item) {
        this.conteudo = item;
    }

    public T abrir() {
        return conteudo;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Caixa<String> caixa1 = new Caixa<>();
        caixa1.guardar("Olá Mundo");
        System.out.println(caixa1.abrir());

        Caixa<Integer> caixa2 = new Caixa<>();
        caixa2.guardar(123);
        System.out.println(caixa2.abrir());
    }
}
```

CRIANDO UMA CLASSE GENÉRICA

- T é um tipo genérico.
- Pode ser String, Integer, Livro, etc.
- Cada instância da classe pode ter um tipo diferente.

CLASSES GENÉRICAS COM MÚLTIPLOS TIPOS

- Podemos usar mais de um tipo genérico na mesma classe:

```
public class Par<K, V> {
    private K chave;
    private V valor;

    public Par(K chave, V valor) {
        this.chave = chave;
        this.valor = valor;
    }

    public K getChave() { return chave; }
    public V getValor() { return valor; }
}
```

```
public class Main {
    public static void main(String[] args) {
        Par<String, Integer> idadePessoa = new Par<>("Ana", 30);
        System.out.println(idadePessoa.getChave() + " tem " + idadePessoa.getValor() + "
                                                    anos.");
    }
}
```

LIMITAÇÃO DE TIPO (BOUNDED TYPE PARAMETERS)

- É possível restringir o tipo genérico a uma superclasse específica (ou interface).
- Isso é útil quando o tipo genérico precisa ter certos comportamentos.

LIMITAÇÃO DE TIPO (BOUNDED TYPE PARAMETERS)

```
public class Calculadora<T extends Number> {
    public double somar(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }
}

public class Main {
    public static void main(String[] args) {
        Calculadora<Integer> c1 = new Calculadora<>();
        System.out.println(c1.somar(10, 20)); // 30.0

        // Calculadora<String> c2 = new Calculadora<>(); // Erro: String não é Number
    }
}
```

LIMITAÇÃO DE TIPO (BOUNDED TYPE PARAMETERS)

- No código anterior, apenas tipos numéricos (Integer, Double, Float, etc.) são permitidos.

MÉTODOS GENÉRICOS

- Um método pode ser genérico mesmo dentro de uma classe não genérica.
- Basta declarar o tipo antes do retorno:

```
public class Util {
    public static <T> void exibirElemento(T elemento) {
        System.out.println("Elemento: " + elemento);
    }

    public static void main(String[] args) {
        exibirElemento("Texto");
        exibirElemento(42);
        exibirElemento(3.14);
    }
}
```

MÉTODOS GENÉRICOS

- O método exibirElemento é flexível e pode receber qualquer tipo de dado.
- “Mesmo código anterior”

```
public class Util {
    public static <T> void exibirElemento(T elemento) {
        System.out.println("Elemento: " + elemento);
    }

    public static void main(String[] args) {
        exibirElemento("Texto");
        exibirElemento(42);
        exibirElemento(3.14);
    }
}
```

TIPOS BRUTOS (RAW TYPES)

- Um tipo bruto é quando se usa uma classe genérica sem especificar o tipo.
- Isso deve ser evitado, pois ignora as verificações de segurança.

```
Caixa caixa = new Caixa(); // tipo bruto
caixa.guardar("Teste");
Integer valor = (Integer) caixa.abrir(); // Erro em tempo de execução!
```

APROFUNDANDO

- Antes da introdução dos Generics no Java (na versão 5), todas as coleções e classes da biblioteca padrão aceitavam qualquer tipo de objeto, pois não havia a possibilidade de declarar o tipo esperado.
- Isso fazia com que, por exemplo, uma lista pudesse misturar valores de tipos completamente diferentes: String, Integer, Double, etc.

APROFUNDANDO

- Esse comportamento ainda existe por retrocompatibilidade, mas é altamente desencorajado.
- Chamamos isso de tipo bruto (raw type), ou seja, o uso de uma classe genérica sem especificar o parâmetro de tipo entre os sinais < >.

EXEMPLO

- Código

```
import java.util.ArrayList;
import java.util.List;

public class ExemploRawType {
    public static void main(String[] args) {
        List lista = new ArrayList(); // Tipo bruto (sem <String> ou <Integer>)
        lista.add("Alan");
        lista.add(10);
        lista.add(3.14);

        for (Object obj : lista) {
            System.out.println(obj);
        }
    }
}
```

EXEMPLO

- Esse código compila, mas o compilador gera um aviso (“unchecked or unsafe operation”).
- O problema é que, como a lista aceita qualquer tipo de objeto, perdemos a verificação de tipos em tempo de compilação.

EXEMPLO

- Isso significa que, se tentarmos converter os elementos, podemos causar erros em tempo de execução:

```
String nome = (String) lista.get(1); // ClassCastException!
```

- O tipo bruto ignora completamente as verificações que os Generics proporcionam.
- Quando o programador omite o parâmetro de tipo (<T>), o compilador entende que a coleção está operando em “modo antigo”, aceitando e retornando apenas Object.

POR QUE OS TIPOS BRUTOS AINDA EXISTEM?

- O Java mantém o suporte aos raw types por uma questão de retrocompatibilidade.
- Isso significa que programas antigos (escritos antes de 2004) ainda podem ser compilados nas versões modernas da linguagem.

POR QUE OS TIPOS BRUTOS AINDA EXISTEM?

- Na prática, porém, usar tipos brutos hoje é uma má prática, porque:
  - Remove toda a segurança de tipos.
  - Obriga o uso de casting, o que é perigoso.
  - Prejudica a legibilidade e a manutenção do código.

POR QUE OS TIPOS BRUTOS AINDA EXISTEM?

- O uso correto é sempre especificar o tipo de dado, como no exemplo a seguir:

```
List<String> nomes = new ArrayList<>();
nomes.add("Ana");
nomes.add("Carlos");
```

- Dessa forma, o compilador impede qualquer inserção incorreta e garante que o método get() retorne sempre uma String, sem necessidade de conversão.

RELAÇÃO COM AS CLASSES WRAPPER

- Para entender completamente os Generics e as coleções em Java, é necessário compreender também o conceito de classes wrapper.
- As coleções (como List, Set, Map, etc.) só aceitam objetos, não tipos primitivos.
- Ou seja, não podemos criar uma List<int>, uma List<double> ou uma List<char>.

RELAÇÃO COM AS CLASSES WRAPPER

- Para resolver isso, o Java oferece uma série de classes especiais que “embrulham” os tipos primitivos em objetos equivalentes.
- Daí o nome wrapper classes (classes “invólucro”).

RELAÇÃO COM AS CLASSES WRAPPER

- A tabela a seguir mostra a correspondência:

Tipo Primitivo	Classe Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

RELAÇÃO COM AS CLASSES WRAPPER

- Essas classes pertencem ao pacote java.lang, portanto não precisam ser importadas.

RELAÇÃO COM AS CLASSES WRAPPER

```
import java.util.ArrayList;
import java.util.List;

public class ExemploWrapper {
    public static void main(String[] args) {
        List<Integer> numeros = new ArrayList<>();
        numeros.add(10);
        numeros.add(20);
        numeros.add(30);

        for (Integer n : numeros) {
            System.out.println("Número: " + n);
        }
    }
}
```

RELAÇÃO COM AS CLASSES WRAPPER

- Observe que, embora tenhamos escrito numeros.add(10), o valor 10 é um int, e o Java faz automaticamente a conversão para um Integer.
- Esse processo é chamado de autoboxing (conversão automática de tipo primitivo para wrapper).

RELAÇÃO COM AS CLASSES WRAPPER

- Quando fazemos o inverso:
  - pegamos um Integer e o atribuimos a um int ocorre o unboxing.
  - Exemplo:

```
Integer x = 100;   // autoboxing (int → Integer)
int y = x;         // unboxing (Integer → int)
```

POR QUE AS CLASSES WRAPPER SÃO IMPORTANTES NOS GENERICS

- Os Generics e as coleções em Java só trabalham com objetos.
- Isso significa que, ao usar listas, conjuntos ou mapas com valores numéricos, é obrigatório usar wrappers.

POR QUE AS CLASSES WRAPPER SÃO IMPORTANTES NOS GENERICS

- Por exemplo, o código abaixo não compila:

```
List<int> valores = new ArrayList<>(); // Erro: tipo primitivo não permitido
```

- A forma correta é:

```
List<Integer> valores = new ArrayList<>();
valores.add(10);
valores.add(20);
```

POR QUE AS CLASSES WRAPPER SÃO IMPORTANTES NOS GENERICS

- Por exemplo, o código abaixo não compila:

```
List<int> valores = new ArrayList<>(); // Erro: tipo primitivo não permitido
```

- A forma correta é:

```
List<Integer> valores = new ArrayList<>();
valores.add(10);
valores.add(20);
```

- Aqui, cada número é automaticamente convertido para um Integer por meio do autoboxing.
- Na hora de recuperar, o Java faz o unboxing automático, devolvendo o valor primitivo.

EXEMPLO UNINDO GENERICS, RAW TYPES E WRAPPER CLASSES

- O código a seguir mostra o contraste entre o uso antigo (raw type) e o moderno (com generics e wrapper):

EXEMPLO UNINDO GENERICS, RAW TYPES E WRAPPER CLASSES

```
import java.util.ArrayList;
import java.util.List;

public class Comparacao {
    public static void main(String[] args) {
        // Tipo bruto (sem generics)
        List listaAntiga = new ArrayList();
        listaAntiga.add(10);
        listaAntiga.add("Texto"); // aceita qualquer tipo!

        // Tipo seguro com Generics e wrapper
        List<Integer> listaNova = new ArrayList<>();
        listaNova.add(10);
        listaNova.add(20);
        // listaNova.add("Texto"); // erro em tempo de compilação

        System.out.println("Lista antiga: " + listaAntiga);
        System.out.println("Lista nova: " + listaNova);
    }
}
```

EXEMPLO UNINDO GENERICS, RAW TYPES E WRAPPER CLASSES

- Saída:

```
Lista antiga: [10, Texto]
Lista nova: [10, 20]
```

- A lista antiga mistura tipos, o que pode causar falhas de conversão posteriormente.
- A lista nova é tipada, segura e evita erros.

OBRIGADO!