6. GENÉRICOS

PROGRAMAÇÃO ORIENTADA AOS OBJETOS

Desenvolvido por:

Carlos Urbano
Catarina Reis
José Magno
Marco Ferreira
Ricardo Antunes

ÍNDICE

6.1. CONCEITO

6.2. CUIDADOS

6.3. FUNCIONALIDADES AVANÇADAS

6.4. OVERRIDING

6.5. SOLUÇÃO COM GENÉRICOS

6.1. CONCEITO

Os genéricos apenas apareceram na versão 1.5 do JAVA

Qual a razão da sua criação?

- **Permitem a abstração de tipos**
- **Tipos de parâmetros explícitos e casts implícitos**
- **Os exemplos mais comuns são os contentores da hierarquia das coleções (listas, pilhas, filas, tabelas de hash, árvores, etc)**

6.1. CONCEITO

Exemplo (antes dos genéricos)

```
List pedras;

public ExemploSemGenericos() {
    pedras = new LinkedList();
    pedras.add(new Pedra(RED));
    pedras.add(new Pedra(GREEN));
    pedras.add(new Pedra(RED));
    Pedra primeira = (Pedra) pedras.get(0);
}

public int getNumeroPedras(Color cor) {
    int cont = 0;
    for (Object pedra : pedras) {
        if (((Pedra) pedra).getColor() == cor) {
            cont++;
        }
    }
    return cont;
}
```

```
class Pedra {
    Color color;

    public Pedra(Color color) {
        this.color = color;
    }

    public Color getColor() {
        return color;
    }

    @Override
    public String toString() {
        final StringBuilder sb
         = new StringBuilder("Pedra{");
        sb.append("color=");
        sb.append(color);
        sb.append('}');
        return sb.toString();
    }
}
```

O *CAST* É UMA "CHATICE" MAS É, TAMBÉM, IMPRESCINDÍVEL!

6.1. CONCEITO

Exemplo (utilizando genéricos)

```
List<Pedra> pedras;

public ExemploComGenericos() {
    pedras = new LinkedList<>();
    pedras.add(new Pedra(RED));
    pedras.add(new Pedra(GREEN));
    pedras.add(new Pedra(RED));
    Pedra primeira = /*sem cast*/ pedras.get(0);
}

public int getNumeroPedras(Color cor) {
    int cont = 0;
    for (Pedra pedra : pedras) {
        /*sem cast*/
        if (pedra.getColor() == cor) {
            cont++;
        }
    }
    return cont;
}
```

List É UMA INTERFACE GENÉRICA QUE RECEBE UMA CLASSE, INTERFACE OU ENUMERAÇÃO COMO PARÂMETRO

INTERFACE PARAMETRIZADA

6.1. CONCEITO

Exemplo (antes dos genéricos)

- **Para assegurar que uma variável do tipo `Pedra` é *type safe* torna-se obrigatório utilizar um *cast***
- **Um *cast* introduz confusão além de aumentar o risco de erro, uma vez que o programador pode fazer uma conversão para um tipo incompatível**

6.1. CONCEITO

Exemplo (utilizando genéricos)

- O programador expressa uma lista como sendo restringida para conter um tipo particular de dados. Daí dizer-se que `List` é uma interface genérica que recebe a classe `Pedra` como parâmetro
- Assim, o compilador pode verificar a correção do programa em tempo de compilação. A declaração indica o tipo e o compilador garante que o que a lista contém é correto
- Em contraste, o *cast* indica o que o programador pensa ser verdade num ponto particular do código
- Os genéricos melhoram a leitura e a robustez sendo mesmo imprescindíveis em algumas situações

6.1. CONCEITO

Tempo de Compilação vs Tempo de Execução

SEM GENÉRICOS

```
List pedras;

public ExemploCompilacaovsExecucao() {
    pedras = new LinkedList<>();
    pedras.add("Isto não é uma pedra");
    Pedra pedra = (Pedra) pedras.get(0);
}
```

SEM VERIFICAÇÃO
INSEGURO

Runtime-error

```
Exception in thread "main" java.lang.ClassCastException: java.base/java.lang.String cannot be cast to Pedra
```

COM GENÉRICOS

```
List<Pedra> pedras;

public ExemploCompilacaovsExecucao() {
    pedras = new LinkedList<>();
    pedras.add("Isto não é uma pedra");
    Pedra pedra = pedras.get(0);
}
```

add (Pedra) in List cannot be applied
to (java.lang.String)

VERIFICAÇÃO EM
COMPILAÇÃO

Runtime EM
SEGURANÇA

6.1. CONCEITO

Criação de uma lista de inteiros

```
public ExemploListaInteiros() {
    List<Integer> inteiros = new LinkedList<>();
    inteiros.add(42); // auto boxing
    int soma = 0;
    for (int i : inteiros) { // auto unboxing
        soma += i;
    }
}
```

- Quando é adicionado 42, internamente é efetuado `new Integer(42)` – *auto boxing*
- O contrário acontece no ciclo *foreach* – *auto unboxing*

6.1. CONCEITO

Genéricos e subtipagem

```
public ExemploGenericosESubtipagem() {
    List<String> strings = new ArrayList<>();
    List<Object> objetos = strings;
    objetos.add(0, new Object()); // É legal?
    strings.get(0); // Não é uma string?!
}
```

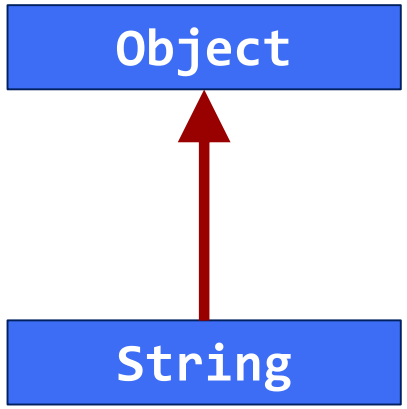
6.2. CUIDADOS

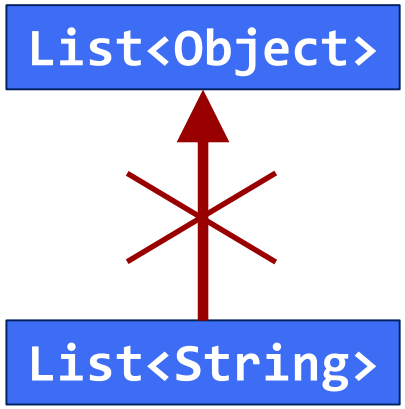
Genéricos e subtipagem

```
public ExemploGenericosESubtipagem() {
    List<String> strings = new ArrayList<>();
    List<Object> objetos = strings;
    objetos.add(0, new Object()); // É legal?
    strings.get(0); // Não é uma string?!
}
```

Incompatible types.
Required: List <java.lang.Object>
Found: List <java.lang.String>

Erro de compilação devido ao *type safe*!





6.3. FUNCIONALIDADES AVANÇADAS

Wildcards

- escrever todos os elementos de uma coleção

SEM GENÉRICOS

```
void escrever(Collection colecao) {
    for (Object objeto : colecao) {
        System.out.println(objeto);
    }
}
```

```
public ExemploSemGenericos() {
    …
    escrever(pedras);
}
```

Compila sem problemas

6.3. FUNCIONALIDADES AVANÇADAS

Wildcards

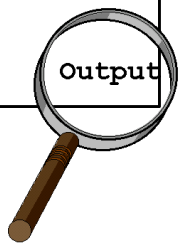
- **escrever todos os elementos de uma coleção**

```
Collection<?>
```

Uma coleção de desconhecidos é uma coleção cujo tipo de elementos corresponde a qualquer "coisa" – *wildcard type*

```
void escrever(Collection<?> colecao) {
    for (Object objeto : colecao) {
        System.out.println(objeto);
    }
}
```

```
public ExemploComGenericos() {
    …
    escrever(pedras);
}
```




```
Pedra{color=java.awt.Color[r=255,g=0,b=0]}
Pedra{color=java.awt.Color[r=0,g=255,b=0]}
Pedra{color=java.awt.Color[r=255,g=0,b=0]}
```

6.3. FUNCIONALIDADES AVANÇADAS

Armadilhas dos *wildcards*

```
public ExemploArmadilhasWildcards() {
    // armadilhas nos wildcards
    String string;
    Object objeto;
    List<?> strings = new ArrayList<String>();

    strings.add
}
```

```
add(capture of ? e)                    boolean
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>
```

Armadilhas dos *wildcards*

```
public ExemploArmadilhasWildcards() {
    // armadilhas nos wildcards
    String string;
    Object objeto;
    List<?> strings = new ArrayList<String>();

    // strings.add("hello world"); // erro de compilação
    // strings.add(new Object()); // erro de compilação
    ((List<String>) strings).add("hello world");
    ((List<Object>) strings).add(new Object()); // não dá erro de compilação!

    // String string = strings.get(0); // erro de compilação
    string = (String) strings.get(0);
    objeto = strings.get(0);
    string = (String) strings.get(1); // erro de execução!
}
```

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

- **Upper bound** COVARIÂNCIA

  `? extends E` – E ou qualquer subtipo de E

  $\{T : \forall\, t \in T \Rightarrow t \texttt{ instanceof } E\}$

- **Lower bound** CONTRAVARIÂNCIA

  `? super E` – E ou qualquer supertipo de E

  $\{T : \forall\, e \in E \Rightarrow e \texttt{ instanceof } T\}$

VARIÂNCIA

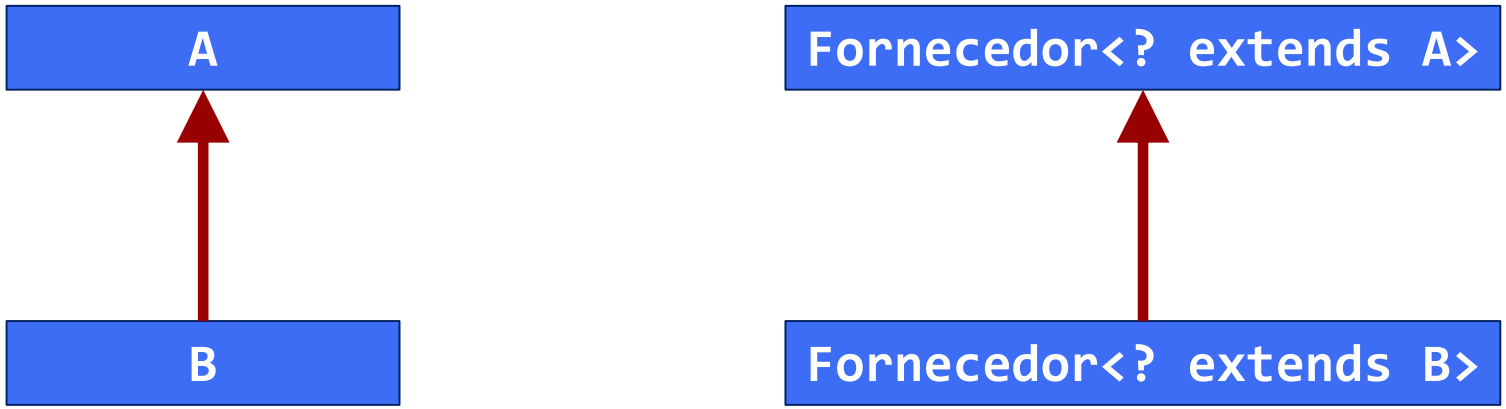
PECS: "Producer Extends, Consumer Super"

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

- `? extends E` – E ou qualquer subtipo de E

COVARIÂNCIA



6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

- `? extends E` – E ou qualquer subtipo de E

COVARIÂNCIA

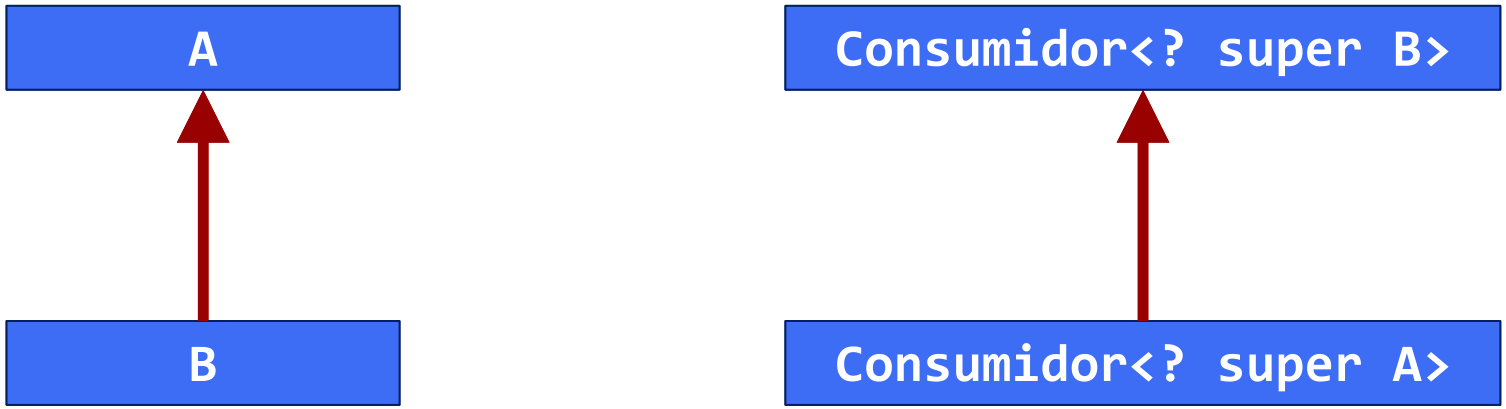
Exemplo

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

- `? super E` – E ou qualquer supertipo de E

CONTRAVARIÂNCIA



6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

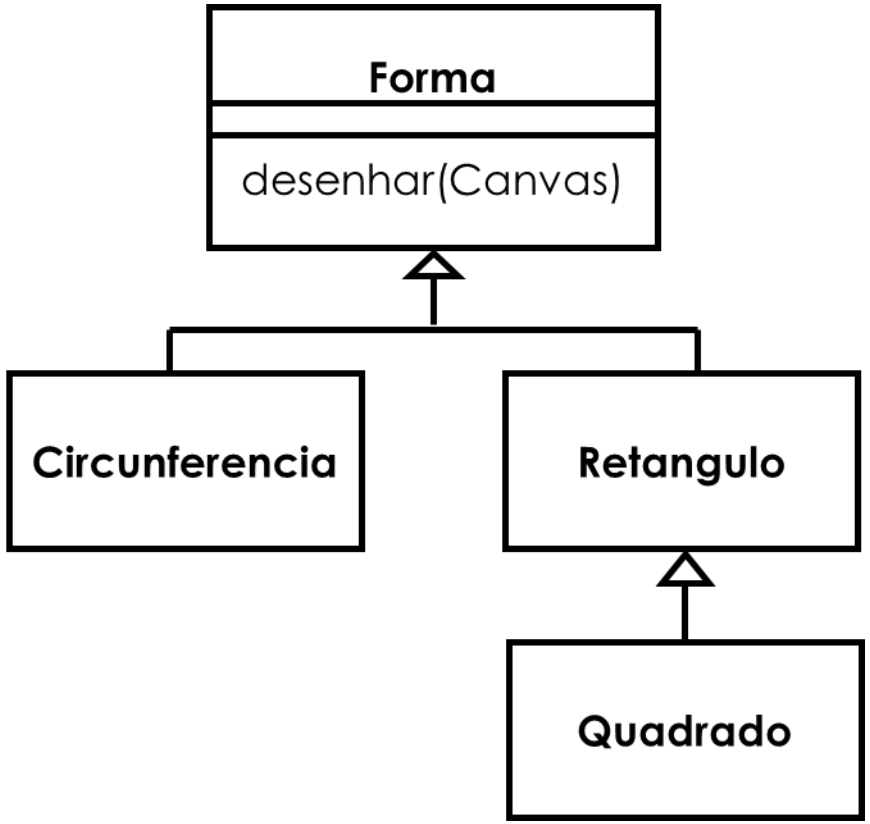
- `? super E` – E ou qualquer supertipo de E

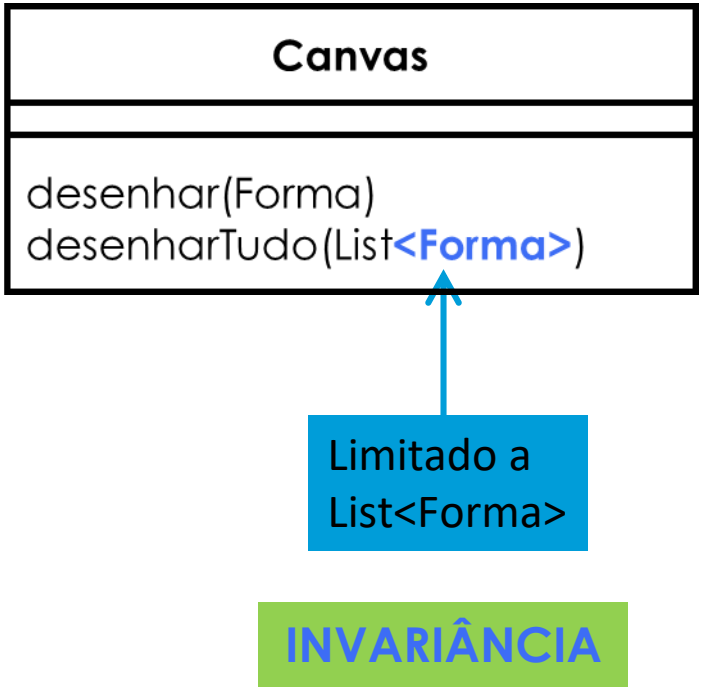
CONTRAVARIÂNCIA

Exemplo

Delimitações dos *wildcards*

- **Considere-se uma pequena aplicação de desenho que permite desenhar formas (circunferências, retângulos, …)**





INVARIÂNCIA

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

- **Então como se implementaria um método que aceitasse qualquer tipo de `Forma`?**

```
public void desenharTudo(List<? extends Forma> formas) {…}
```

wildcard delimitado

Diz-se, então, que Forma é o limite superior (*upper bound*) do *wildcard*

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    List<Forma> formas = new LinkedList<>();
    formas.add(new Retangulo());
    formas.add(new Quadrado());
    formas.add(new Circunferencia());

    List<Retangulo> retangulos = new LinkedList<>();
    retangulos.add(new Retangulo());
    retangulos.add(new Quadrado());

    List<Quadrado> quadrados = new LinkedList<>();
    quadrados.add(new Quadrado());

    List<Quadrado> quadrados2 = new LinkedList<>();
    List<Retangulo> retangulos2 = new LinkedList<>();
    List<Forma> formas2 = new LinkedList<>();

    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

COMO DEVE SER IMPLEMENTADO O MÉTODO **copiar** QUE PERMITA EFETUAR A COPIA DOS ELEMENTOS

DE UMA:

- LISTA DE QUADRADOS

PARA OUTRA:

- LISTA DE QUADRADOS
- LISTA DE RETÂNGULOS
- LISTA DE FORMAS

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

INVARIÂNCIA INVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<Quadrado> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

Wrong 2nd argument type. Found: 'java.util.List<Retangulo>', required: 'java.util.List<Quadrado>' more...

Wrong 2nd argument type. Found: 'java.util.List<Forma>', required: 'java.util.List<Quadrado>' more...

INVARIÂNCIA INVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<Quadrado> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

ESTA SOLUÇÃO APENAS PERMITE COPIAR OS ELEMENTOS:

- **DE UMA LISTA DE QUADRADOS**
- **PARA OUTRA LISTA DE QUADRADOS**

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

INVARIÂNCIA INVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<Retangulo> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);  Wrong 2nd argument type. Found: 'java.util.List<Quadrado>', required: 'java.util.List<Retangulo>' more...
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);  Wrong 2nd argument type. Found: 'java.util.List<Forma>', required: 'java.util.List<Retangulo>' more...
}
```

INVARIÂNCIA INVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<Retangulo> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

ESTA SOLUÇÃO APENAS PERMITE COPIAR OS ELEMENTOS:

- **DE UMA LISTA DE QUADRADOS**
- **PARA OUTRA LISTA DE RETÂNGULOS**

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

INVARIÂNCIA INVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<Forma> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);   Wrong 2nd argument type. Found: 'java.util.List<Quadrado>', required: 'java.util.List<Forma>' more...
    copiar(quadrados, retangulos2);  Wrong 2nd argument type. Found: 'java.util.List<Retangulo>', required: 'java.util.List<Forma>' more...
    copiar(quadrados, formas2);
}
```

INVARIÂNCIA INVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<Forma> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

ESTA SOLUÇÃO APENAS PERMITE COPIAR OS ELEMENTOS:

- **DE UMA LISTA DE QUADRADOS**
- **PARA OUTRA LISTA DE FORMAS**

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

INVARIÂNCIA

CONTRAVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<? super Quadrado> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, quadrados2);
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
}
```

INVARIÂNCIA

CONTRAVARIÂNCIA

```
public static void copiar(List<Quadrado> origem, List<? super Quadrado> destino) {
    for (Quadrado quadrado : origem) {
        destino.add(quadrado);
    }
}
```

ESTA SOLUÇÃO PERMITE COPIAR OS ELEMENTOS:

- DE UMA LISTA DE QUADRADOS
- PARA OUTRA LISTA DE QUADRADOS, LISTA DE RETÂNGULOS, LISTA DE FORMAS OU LISTA DE OBJECTS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    List<Forma> formas = new LinkedList<>();
    formas.add(new Retangulo());
    formas.add(new Quadrado());
    formas.add(new Circunferencia());

    List<Retangulo> retangulos = new LinkedList<>();
    retangulos.add(new Retangulo());
    retangulos.add(new Quadrado());

    List<Quadrado> quadrados = new LinkedList<>();
    quadrados.add(new Quadrado());

    List<Quadrado> quadrados2 = new LinkedList<>();
    List<Retangulo> retangulos2 = new LinkedList<>();
    List<Forma> formas2 = new LinkedList<>();

    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
    copiar(retangulos, retangulos2);
    copiar(retangulos, formas2);
}
```

COMO DEVE SER IMPLEMENTADO O MÉTODO **copiar** QUE PERMITA EFETUAR A COPIA DOS ELEMENTOS

DE UMA:

- LISTA DE QUADRADOS
- LISTA DE RETÂNGULOS

PARA OUTRA:

- LISTA DE RETÂNGULOS
- LISTA DE FORMAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
    copiar(retangulos, retangulos2);
    copiar(retangulos, formas2);
}
```

INVARIÂNCIA

CONTRAVARIÂNCIA

```
public static void copiar(List<Retangulo> origem, List<? super Retangulo> destino) {
    for (Retangulo retangulo : origem) {
        destino.add(retangulo);
    }
}
```

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, retangulos2);  Wrong 1st argument type. Found: 'java.util.List<Quadrado>', required: 'java.util.List<Retangulo>' more...
    copiar(quadrados, formas2);  Wrong 1st argument type. Found: 'java.util.List<Quadrado>', required: 'java.util.List<Retangulo>' more...
    copiar(retangulos, retangulos2);
    copiar(retangulos, formas2);
}
```

INVARIÂNCIA

CONTRAVARIÂNCIA

```
public static void copiar(List<Retangulo> origem, List<? super Retangulo> destino) {
    for (Retangulo retangulo : origem) {
        destino.add(retangulo);
    }
}
```

ESTA SOLUÇÃO APENAS PERMITE COPIAR OS ELEMENTOS:

- **DE UMA LISTA DE RETÂNGULOS**
- **PARA OUTRA LISTA DE RETÂNGULOS, LISTA DE FORMAS OU LISTA DE OBJECTS**

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
    copiar(retangulos, retangulos2);
    copiar(retangulos, formas2);
}
```

COVARIÂNCIA

```
public static void copiar(List<? extends Retangulo> origem,
                                                List<? super Retangulo> destino) {
    for (Retangulo retangulo : origem) {
        destino.add(retangulo);
    }
}
```

CONTRAVARIÂNCIA

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, retangulos2);
    copiar(quadrados, formas2);
    copiar(retangulos, retangulos2);
    copiar(retangulos, formas2);
}
```

COVARIÂNCIA

```
public static void copiar(List<? extends Retangulo> origem,
                                                List<? super Retangulo> destino) {
    for (Retangulo retangulo : origem) {
        destino.add(retangulo);
    }
}
```

CONTRAVARIÂNCIA

ESTA SOLUÇÃO PERMITE COPIAR OS ELEMENTOS:

- **DE UMA LISTA DE QUADRADOS OU LISTA DE RETÂNGULOS**
- **PARA OUTRA LISTA DE RETÂNGULOS, LISTA DE FORMAS OU LISTA DE OBJECTS**

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    List<Forma> formas = new LinkedList<>();
    formas.add(new Retangulo());
    formas.add(new Quadrado());
    formas.add(new Circunferencia());

    List<Retangulo> retangulos = new LinkedList<>();
    retangulos.add(new Retangulo());
    retangulos.add(new Quadrado());

    List<Quadrado> quadrados = new LinkedList<>();
    quadrados.add(new Quadrado());

    List<Quadrado> quadrados2 = new LinkedList<>();
    List<Retangulo> retangulos2 = new LinkedList<>();
    List<Forma> formas2 = new LinkedList<>();

    copiar(quadrados, formas2);
    copiar(retangulos, formas2);
    copiar(formas, formas2);
}
```

COMO DEVE SER IMPLEMENTADO O MÉTODO **copiar** QUE PERMITA EFETUAR A COPIA DOS ELEMENTOS

DE UMA:

- LISTA DE QUADRADOS
- LISTA DE RETÂNGULOS
- LISTA DE FORMAS

PARA OUTRA:

- LISTA DE FORMAS

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, formas2);
    copiar(retangulos, formas2);
    copiar(formas, formas2);
}
```

COVARIÂNCIA

```
public static void copiar(List<? extends Forma> origem,
                                                List<? super Forma> destino) {
    for (Forma forma : origem) {
        destino.add(forma);
    }
}
```

CONTRAVARIÂNCIA

6.3. FUNCIONALIDADES AVANÇADAS

Delimitações dos *wildcards*

```
public ExemploGenericosESubtipagem() {
    …
    copiar(quadrados, formas2);
    copiar(retangulos, formas2);
    copiar(formas, formas2);
}
```

COVARIÂNCIA

```
public static void copiar(List<? extends Forma> origem,
                                                List<? super Forma> destino) {
    for (Forma forma : origem) {
        destino.add(forma);
    }
}
```

CONTRAVARIÂNCIA

ESTA SOLUÇÃO PERMITE COPIAR OS ELEMENTOS:

- **DE UMA LISTA DE QUADRADOS, LISTA DE RETÂNGULOS OU LISTA DE FORMAS**
- **PARA OUTRA LISTA DE FORMAS OU LISTA DE OBJECTS**

6.3. FUNCIONALIDADES AVANÇADAS

Varargs – método com número variável de argumentos

```
public Poligono poligonoDe(Ponto p1, Ponto p2, Ponto p3) {
    …
}
```

```
public Poligono poligonoDe(Ponto p1, Ponto p2, …, Ponto pN) {
    …
}
```

```
public Poligono poligonoDe(Ponto... cantos) {
    for (Ponto p : cantos) {
        …
    }
    …
}
```

6.4. OVERRIDING

Uma das alterações do java 1.4 para o java 1.5 foi a introdução dos tipos de retorno covariantes na reescrita (*overriding*) de métodos

Esta funcionalidade permite criar métodos em subclasses que retornem um objeto cujo tipo é um subtipo do retornado pelo método que está a ser reescrito

6.4. OVERRIDING

Exemplo

```
class A {
}

class B extends A {
}

class Super {
    public A metodo() {
        return new A();
    }
}

class Sub extends Super {
    public B metodo() {
        return new B();
    }
}
```

MÉTODO NA SUPERCLASSE DEVOLVENDO UM OBJETO DO TIPO **A**

MÉTODO NA SUBCLASSE DEVOLVENDO UM OBJETO DO TIPO **B** QUE É UM SUBTIPO DE **A**

6.4. OVERRIDING

No entanto, o mesmo raciocínio não se pode aplicar aos parâmetros dos métodos

- **Mas recorrendo aos genéricos poderemos encontrar uma solução**

Vejamos o seguinte exemplo com base na relação entre animais e a sua respetiva comida

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal {
    void comer(Comida comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal {
    public void comer(Comida comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal {
    public void comer(Comida comida) {
    }
}
```

Um antílope é um animal que come comida e ao mesmo tempo serve de comida a outros animais – como o leão

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal {
    void comer(Comida comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal {
    public void comer(Comida comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal {
    public void comer(Comida comida) {
    }
}
```

Um antílope é um animal que come comida e ao mesmo tempo serve de comida a outros animais – como o leão

O problema com esta implementação é que este código permite que o antílope coma qualquer coisa do tipo **Comida** (o que não é verdade)

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal {
    void comer(Comida comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal {
    public void comer(Comida comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal {
    public void comer(Comida comida) {
    }
}
```

Um antílope é um animal que come comida e ao mesmo tempo serve de comida a outros animais – como o leão

O problema com esta implementação é que este código permite que o antílope coma qualquer coisa do tipo **Comida** (o que não é verdade)

Vejamos como ficaria o código se quiséssemos restringir a comida do antílope apenas a erva e a do leão apenas a antílope

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal {
    void comer(Comida comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal {
    public void comer(Erva comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal {
    public void comer(Antilope comida) {
    }
}
```

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal {
    void comer(Comida comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal {
    public void comer(Erva comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal {
    public void comer(Antilope comida) {
    }
}
```

Class 'Antilope' must either be declared abstract or implement abstract method 'comer(Comida)' in 'Animal'

Class 'Leao' must either be declared abstract or implement abstract method 'comer(Comida)' in 'Animal'

Infelizmente, este código já não compilará!

É possível adicionar o método com a assinatura `comer(Erva comida)` à classe `Antilope` mas então não se pode declarar que o `Antilope` implementa a interface `Animal`

Isto porque não estamos a implementar a assinatura declarada nessa interface

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal<F extends Comida> {
    void comer(F comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal<Erva> {
    public void comer(Erva comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal<Antilope> {
    public void comer(Antilope comida) {
    }
}
```

Recorrendo aos Genéricos poderemos representar a ideia de que um tipo de animal come um tipo particular de comida

6.4. OVERRIDING


```
interface Comida {
    int getIndiceGlicemico();
}

interface Animal<F extends Comida> {
    void comer(F comida);
}

class Erva implements Comida {
    public int getIndiceGlicemico() {
        return 20;
    }
}

class Antilope implements Comida, Animal<Erva> {
    public void comer(Erva comida) {
    }

    public int getIndiceGlicemico() {
        return 5;
    }
}

class Leao implements Animal<Antilope> {
    public void comer(Antilope comida) {
    }
}
```

Recorrendo aos Genéricos poderemos representar a ideia de que um tipo de animal come um tipo particular de comida

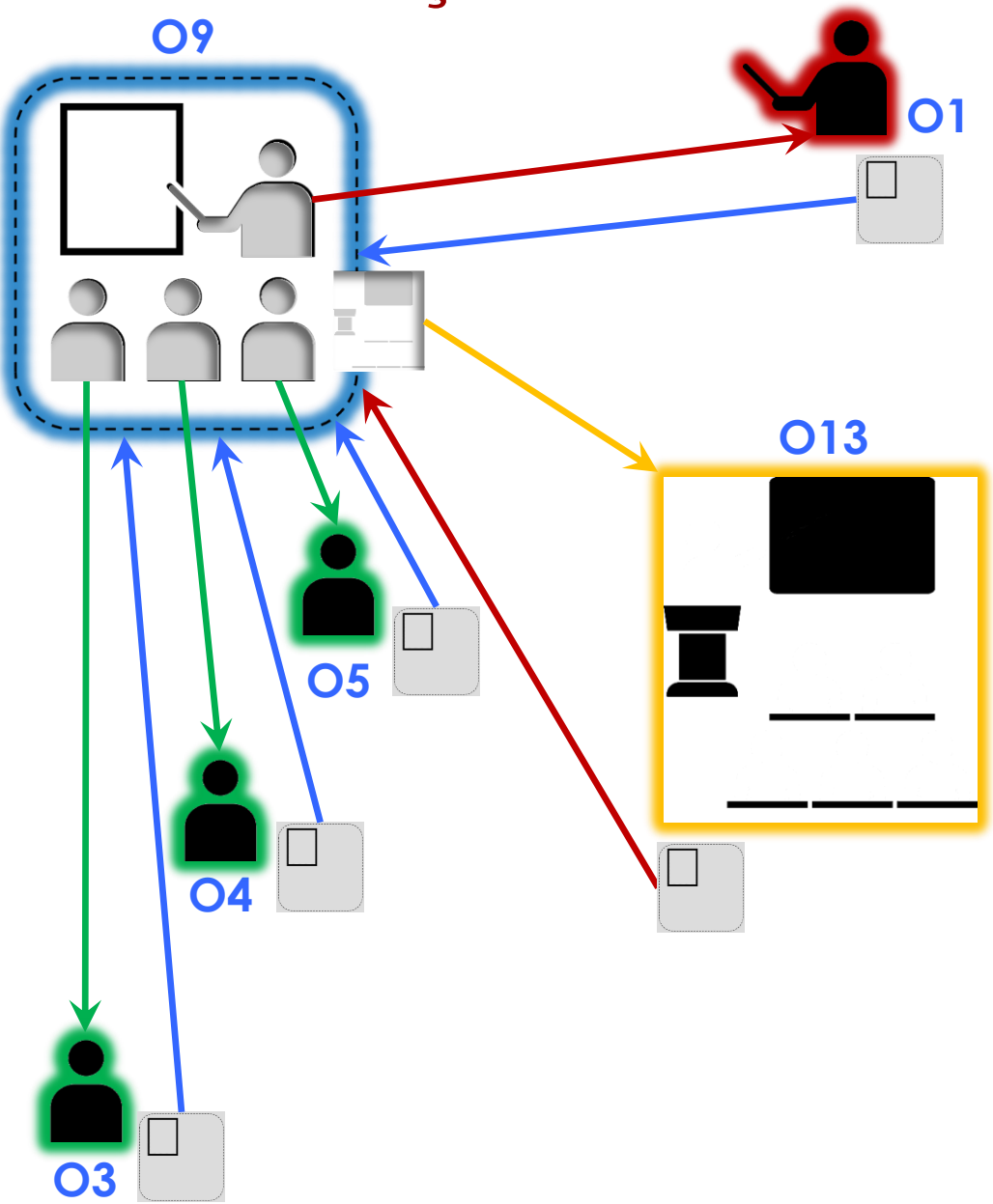
Conclusão: não há nada no Java que controle automaticamente tipos de parâmetros covariantes da mesma forma que faz com os tipos de retorno

Exemplo de utilização:

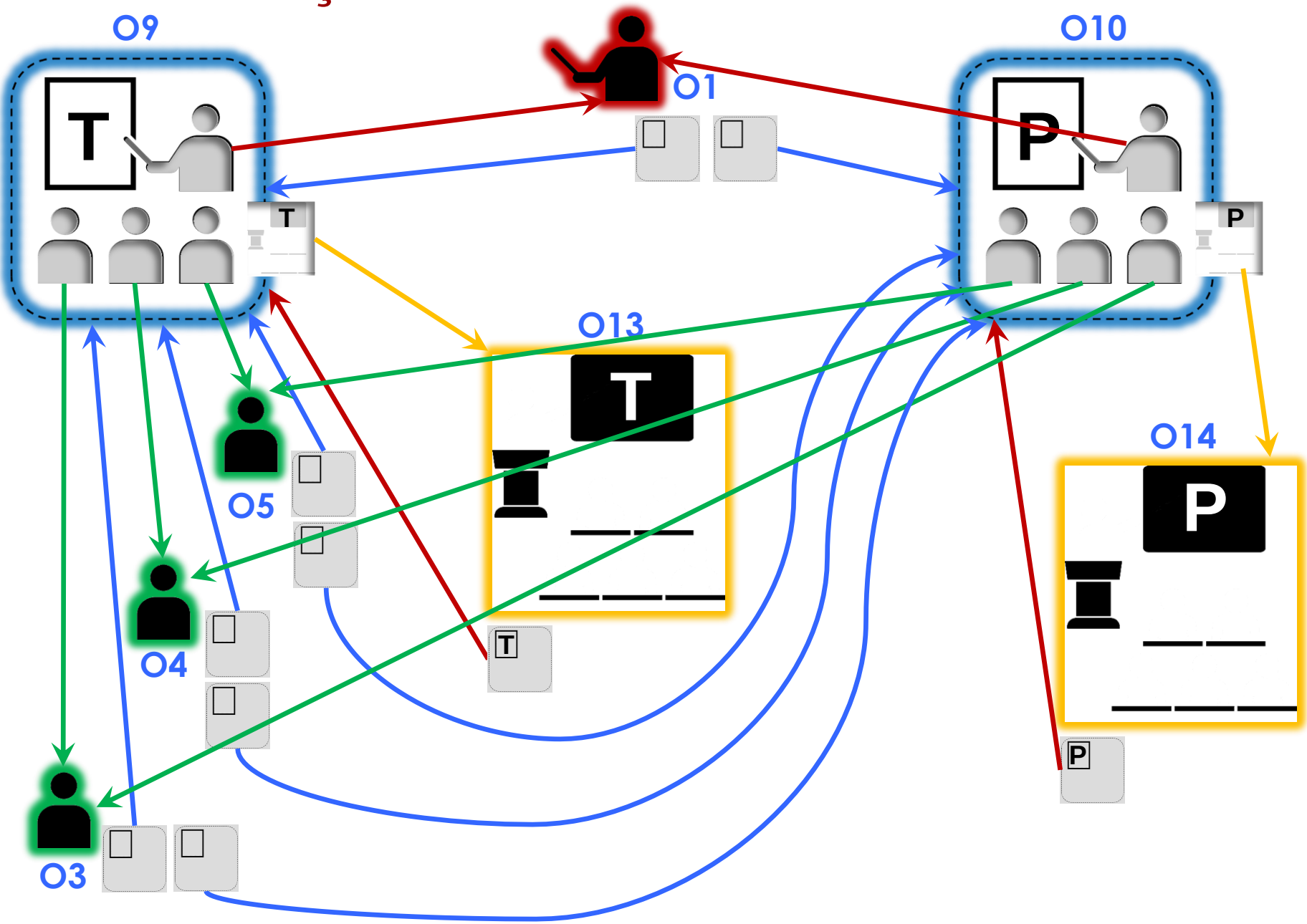
```
public ExemploAnimalComida() {
    comer(new Leao(), new Antilope());
}

static <F extends Comida> void
        comer(Animal<F> animal, F comida) {
    animal.comer(comida);
}

public static void main(String[] args) {
    new ExemploAnimalComida();
}
```

Vamos agora considerar que existem aulas práticas que decorrem apenas em salas práticas e aulas teóricas que decorrem apenas em salas teóricas

6.5. SOLUÇÃO COM GENÉRICOS



6.5. SOLUÇÃO COM GENÉRICOS



6.5. SOLUÇÃO COM GENÉRICOS

ANÁLISE PARCIAL SEM RECURSO A GENÉRICOS

	superclasses/interfaces	Pessoa	Pessoa	Identificador	Descritor	Identificador {RepositorioAulas} {AssociavelAulas}	Descritor {RepositorioAulas} {AssociavelAulas}					Aula	Aula	Sala	Sala	GestorAulas	GestorAulas
	classes	Professor	Aluno	Aula	Identificador	Pessoa	Sala	Descritor	{RepositorioAulas}	GestorAulas	{AssociavelAulas}	AulaTeorica	AulaPratica	SalaTeorica	SalaPratica	GestorAulasTeorica	GestorAulasTeorica
tipo	***atributo***																
String	nome							x									
long	numero				x												
LinkedList<Aula>	aulas									x							
String	sumario			x													
Professor	professor			x													
LinkedList<Aluno>	alunos			x													
boolean	aberta						x										
SalaTeorica	sala											x					
SalaPratica	sala												x				
GestorAulas	gestorAulas					x											
GestorAulasTeorica	gestorAulas													x			
GestorAulasPraticas	gestorAulas														x		
AssociavelAulas	associavelAulas									x							

6.5. SOLUÇÃO COM GENÉRICOS

ANÁLISE PARCIAL SEM RECURSO A GENÉRICOS

DUPLICAÇÃO DE CLASSES E INTERFACES DEPENDENTE APENAS DO TIPO DOS ATRIBUTOS

	superclasses/interfaces	*Pessoa*	*Pessoa*	*Identificador*	*Descritor*	*Identificador {RepositorioAulas} {AssociavelAulas}*	*Descritor {RepositorioAulas} {AssociavelAulas}*					*Aula*	*Aula*	*Sala*	*Sala*	GestorAulas	GestorAulas
	classes	Professor	Aluno	*Aula*	*Identificador*	*Pessoa*	*Sala*	*Descritor*	*{RepositorioAulas}*	GestorAulas	*{AssociavelAulas}*	AulaTeorica	AulaPratica	SalaTeorica	SalaPratica	*GestorAulasTeorica*	*GestorAulasTeorica*
tipo	***atributo***																
String	nome							x									
long	numero				x												
LinkedList<Aula>	aulas									x							
String	sumario			x													
Professor	professor			x													
LinkedList<Aluno>	alunos			x													
boolean	aberta						x										
SalaTeorica	sala											x					
SalaPratica	sala												x				
GestorAulas	gestorAulas					x											
GestorAulasTeorica	gestorAulas													x			
GestorAulasPraticas	gestorAulas														x		
AssociavelAulas	associavelAulas									x							

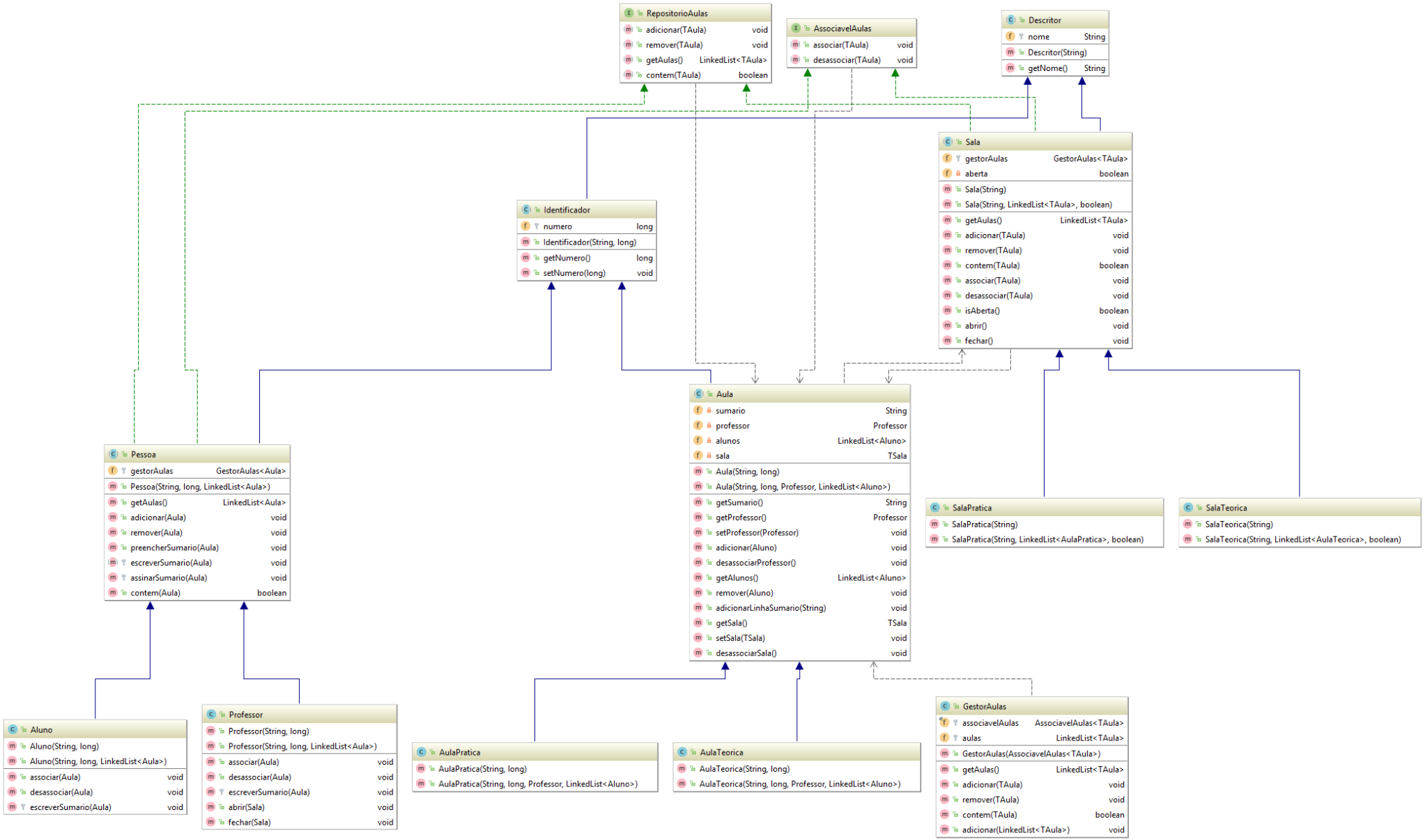
6.5. SOLUÇÃO COM GENÉRICOS

	superclasses/interfaces	Pessoa	Pessoa	Identificador	Descritor	Identificador {RepositorioAulas<Aula>} {AssociavelAulas<Aula>}	Descritor {RepositorioAulas<TAula>} {AssociavelAulas<TAula>}					Aula<SalaTeorica>	Aula<SalaPratica>	Sala<AulaTeorica>	Sala<AulaPratica>
	classes	Professor	Aluno	Aula<TSala>	Identificador	Pessoa	Sala<TAula>	Descritor	{RepositorioAulas<TAula>}	GestorAulas<TAula>	{AssociavelAulas<TAula>}	AulaTeorica	AulaPratica	SalaTeorica	SalaPratica
tipo	***atributo***														
String	nome							x							
long	numero				x										
LinkedList<TAula>	aulas									x					
String	sumario			x											
Professor	professor			x											
LinkedList<Aluno>	alunos			x											
boolean	aberta						x								
TSala	sala			x											
GestorAulas<Aula>	gestorAulas					x									
GestorAulas<TAula>	gestorAulas						x								
AssociavelAulas<TAula>	associavelAulas									x					

6.5. SOLUÇÃO COM GENÉRICOS

	superclasses/interfaces	Pessoa	Pessoa	Identificador	Descritor	Identificador {RepositorioAulas<Aula>} {AssociavelAulas<Aula>}	Descritor {RepositorioAulas<TAula>} {AssociavelAulas<TAula>}					Aula<SalaTeorica>	Aula<SalaPratica>	Sala<AulaTeorica>	Sala<AulaPratica>
	classes	Professor	Aluno	Aula<TSala>	Identificador	Pessoa	Sala<TAula>	Descritor	{RepositorioAulas<TAula>}	GestorAulas<TAula>	{AssociavelAulas<TAula>}	AulaTeorica	AulaPratica	SalaTeorica	SalaPratica
return	**método**														
void	setProfessor(Professor)			x											
void	adicionar(TAula)					ID	ID		x	x					
void	preencherSumario(Aula)					RD									
boolean	contem(TAula)					ID	ID		x	x					
void	adicionar(Aluno)			x											
void	adicionarLinhaSumario(String)			x											
String	getNome()							x							
long	getNumero()				x										
void	setNumero(long)				x										
String	getSumario()			x											
Professor	getProfessor()			x											
void	desassociarProfessor()			x											
void	remover(TAula)					ID	ID		x	x					
LinkedList<Aluno>	getAlunos()			x											
void	remover(Aluno)			x											
void	associar(TAula)	I	I				x				x				
void	desassociar(TAula)	I	I				x				x				
void	escreverSumario(Aula)	I	I			x									
void	assinarSumario(Aula)					x									
LinkedList<TAula>	getAulas()					ID	ID		x	x					
void	abrir(Sala)	x													
void	fechar(Sala)	x													
TSala	getSala()			x											
void	setSala(TSala)			x											
void	desassociarSala()			x											
boolean	isAberta()						x								
void	abrir()						x								
void	fechar()						x								
void	adicionar(LinkedList<Aula>)									x					

6.5. SOLUÇÃO COM GENÉRICOS



Vamos agora apresentar o código da solução com recurso a genéricos

6.5. SOLUÇÃO COM GENÉRICOS


```
public interface RepositorioAulas<TAula extends Aula> {
    void adicionar(TAula aula);

    void remover(TAula aula);

    LinkedList<TAula> getAulas();

    boolean contem(TAula aula);
}
```

```
public interface AssociavelAulas<TAula extends Aula> {
    void associar(TAula aula);

    void desassociar(TAula aula);
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```
public abstract class Aula<TSala extends Sala> extends Identificador {
    …
    private TSala sala;

    public TSala getSala() {
        return sala;
    }

    public void setSala(TSala sala) {
        if (sala == null || this.sala == sala) {
            return;
        }
        if (this.sala != null) {
            this.sala.desassociar(this);
        }
        this.sala = sala;
        sala.adicionar(this);
    }

    public void desassociarSala() {
        if (sala == null) {
            return;
        }
        TSala salaARemover = sala;
        sala = null;
        salaARemover.remover(this);
    }
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```
public class AulaTeorica extends Aula<SalaTeorica> {

    public AulaTeorica(String nome, long numero) {
        super(nome, numero);
    }

    public AulaTeorica(String nome, long numero, Professor professor,
                       LinkedList<Aluno> alunos) {
        super(nome, numero, professor, alunos);
    }
}
```

```
public class AulaPratica extends Aula<SalaPratica> {

    public AulaPratica(String nome, long numero) {
        super(nome, numero);
    }

    public AulaPratica(String nome, long numero, Professor professor,
                       LinkedList<Aluno> alunos) {
        super(nome, numero, professor, alunos);
    }
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```
public abstract class Sala<TAula extends Aula> extends Descritor
                        implements RepositorioAulas<TAula>, AssociavelAulas<TAula> {
    protected GestorAulas<TAula> gestorAulas;
    …
    public Sala(String nome, LinkedList<TAula> aulas, boolean aberta) {
        super(nome);
        gestorAulas = new GestorAulas<>(this);
        gestorAulas.adicionar(aulas);
        this.aberta = aberta;
    }

    @Override
    public LinkedList<TAula> getAulas() {
        return gestorAulas.getAulas();
    }

    @Override
    public void adicionar(TAula aula) {
        gestorAulas.adicionar(aula);
    }

    @Override
    public void remover(TAula aula) {
        gestorAulas.remover(aula);
    }

    @Override
    public void associar(TAula aula) {
        aula.setSala(this);
    }

    @Override
    public void desassociar(TAula aula) {
        aula.desassociarSala();
    }
    …
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```java
public class SalaTeorica extends Sala<AulaTeorica> {

    public SalaTeorica(String nome) {
        super(nome);
    }

    public SalaTeorica(String nome, LinkedList<AulaTeorica> aulaTeoricas, boolean aberta) {
        super(nome, aulaTeoricas, aberta);
    }
}
```

```java
public class SalaPratica extends Sala<AulaPratica> {

    public SalaPratica(String nome) {
        super(nome);
    }

    public SalaPratica(String nome, LinkedList<AulaPratica> aulaPraticas, boolean aberta) {
        super(nome, aulaPraticas, aberta);
    }
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```
public abstract class Pessoa extends Identificador
                            implements RepositorioAulas<Aula>, AssociavelAulas<Aula> {
    protected GestorAulas<Aula> gestorAulas;

    public Pessoa(String nome, long numero, LinkedList<Aula> aulas) {
        super(nome, numero);
        gestorAulas = new GestorAulas<>(this);
        gestorAulas.adicionar(aulas);
    }

    @Override
    public LinkedList<Aula> getAulas() {
        return gestorAulas.getAulas();
    }

    @Override
    public void adicionar(Aula aula) {
        gestorAulas.adicionar(aula);
    }

    @Override
    public void remover(Aula aula) {
        gestorAulas.remover(aula);
    }

    public void preencherSumario(Aula aula) {
        if (!gestorAulas.contem(aula)) {
            return;
        }
        escreverSumario(aula);
    }
    …
```

6.5. SOLUÇÃO COM GENÉRICOS


```
    …

    public abstract void associar(Aula aula);

    public abstract void desassociar(Aula aula);

    protected abstract void escreverSumario(Aula aula);

    protected void assinarSumario(Aula aula) {
        aula.adicionarLinhaSumario(nome);
    }

    @Override
    public boolean contem(Aula aula) {
        return gestorAulas.contem(aula);
    }
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```java
public class GestorAulas<TAula extends Aula> {
    protected final AssociavelAulas<TAula> associavelAulas;
    protected LinkedList<TAula> aulas;

    public GestorAulas(AssociavelAulas<TAula> associavelAulas) {
        this.associavelAulas = associavelAulas;
        this.aulas = new LinkedList<>();
    }

    public LinkedList<TAula> getAulas() {
        return new LinkedList<>(aulas);
    }

    public void adicionar(TAula aula) {
        if (aula == null || aulas.contains(aula)) {
            return;
        }
        aulas.add(aula);
        associavelAulas.associar(aula);
    }

    public void remover(TAula aula) {
        if (!aulas.contains(aula)) {
            return;
        }
        aulas.remove(aula);
        associavelAulas.desassociar(aula);
    }

    public boolean contem(TAula aula) {
        return aulas.contains(aula);
    }

    public void adicionar(LinkedList<TAula> aulas) {
        for (TAula aula : aulas) {
            adicionar(aula);
        }
    }
}
```

6.5. SOLUÇÃO COM GENÉRICOS


```
public class Professor extends Pessoa {
    …
    public Professor(String nome, long numero, LinkedList<Aula> aulas) {
        super(nome, numero, aulas);
    }

    @Override
    public void associar(Aula aula) {
        aula.setProfessor(this);
    }

    @Override
    public void desassociar(Aula aula) {
        aula.desassociarProfessor();
    }

    @Override
    protected void escreverSumario(Aula aula) {
        aula.adicionarLinhaSumario(aula.getNome());
        aula.adicionarLinhaSumario(String.valueOf(aula.getNumero()));
        assinarSumario(aula);
        for (Aluno aluno : aula.getAlunos()) {
            aluno.preencherSumario(aula);
        }
    }
    …
}
```

Incompatible types.
Required: Object
Found: Aluno

COMO DEVE SER IMPLEMENTADO O MÉTODO

6.5. SOLUÇÃO COM GENÉRICOS

Para implementar genéricos, o compilador Java aplica o *type erasure* do seguinte modo:

- **Substitui todos os tipos de parâmetros dos tipos genéricos pelos seus *wildcards* ou por `Object`. O bytecode produzido contém apenas classes, interfaces e métodos não parametrizados**
- **Usa *casts*, se necessário, para preservar o *type safety***
- **Gera métodos *bridge* para preservar o polimorfismo em tipos genéricos estendidos**

6.5. SOLUÇÃO COM GENÉRICOS

O *type erasure* assegura que nenhuma nova classe seja criada para tipos parametrizados. Assim, os genéricos não incorrem em acréscimo de tempo de execução

Devido ao *type erasure* não é permitido em java, por exemplo, que uma classe `A` implemente as interfaces `I<T1>` e `I<T2>`, isto é, uma mesma interface `I` com tipos diferentes `T1` e `T2`

6.5. SOLUÇÃO COM GENÉRICOS


```java
public class Professor extends Pessoa {
    …
    public Professor(String nome, long numero, LinkedList<Aula> aulas) {
        super(nome, numero, aulas);
    }

    @Override
    public void associar(Aula aula) {
        aula.setProfessor(this);
    }

    @Override
    public void desassociar(Aula aula) {
        aula.desassociarProfessor();
    }

    @Override
    protected void escreverSumario(Aula aula) {
        aula.adicionarLinhaSumario(aula.getNome());
        aula.adicionarLinhaSumario(String.valueOf(aula.getNumero()));
        assinarSumario(aula);
        for (Aluno aluno : ((Aula<? extends Sala>)aula).getAlunos()) {
            aluno.preencherSumario(aula);
        }
    }
    …
}
```

DOWNCAST OBRIGATÓRIO

BIBLIOGRAFIA

- **Oscar Nierstrasz**
http://scg.unibe.ch/download/p2/08Generics.pdf

- **Java Tips**
http://www.java-tips.org/java-se-tips/java.lang/covariant-parameter-types-2.html

- **The Java™ Tutorials**
https://docs.oracle.com/javase/tutorial/java/generics/erasure.html