

LP Orientada a Objetos II

Introdução

Diego Addan

DS142 - UFPR - 2023

Para hoje

Apresentação

Introdução a DS142

Referências

Introdução

Ementa e Objetivos

Site

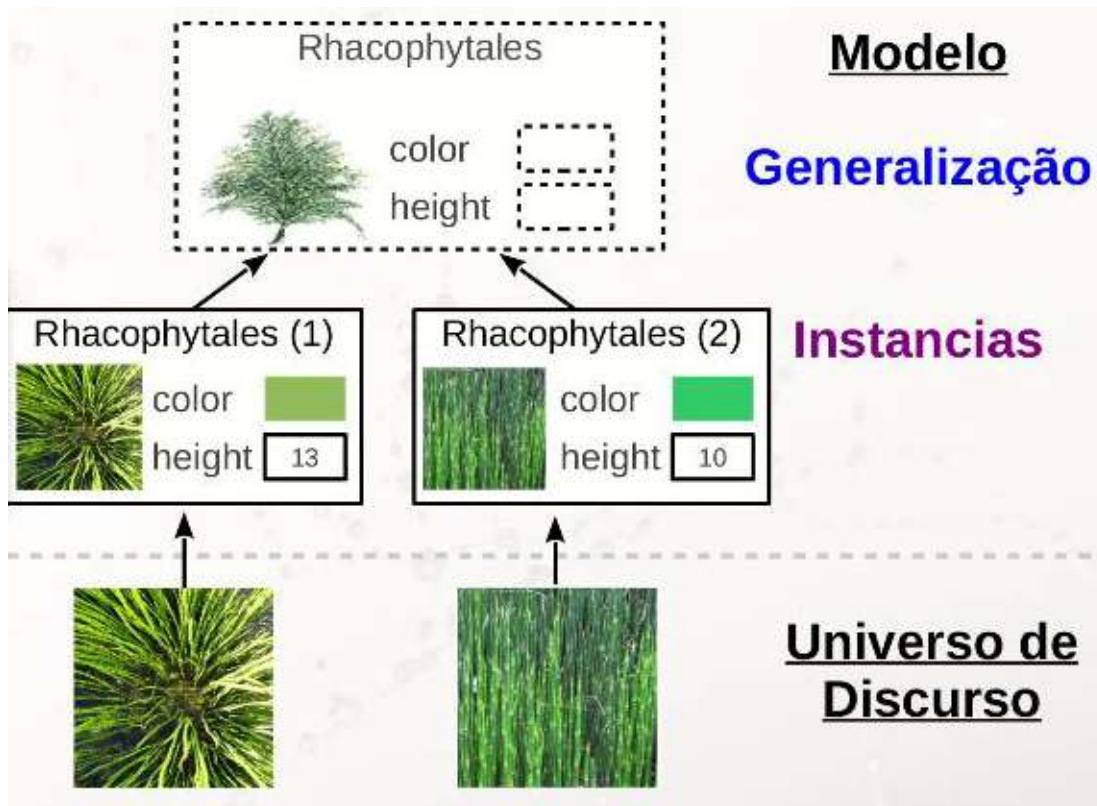
https://www.inf.ufpr.br/dagoncalves/_disciplinas.html

Introdução

POO é uma programação que permite uma abstração.

Poderosa forma de se estruturar os dados

É preciso formalização



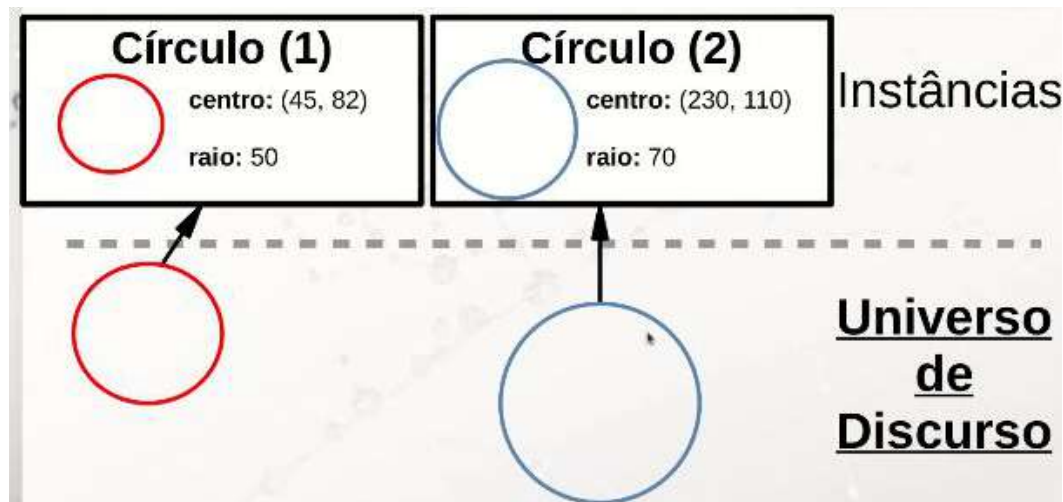
Introdução

Baseada em Objetos:

- Identidade
- Atributos
- Comportamento

Exemplo: Como representar um
Círculo.

Identidade, Valores dos Atr

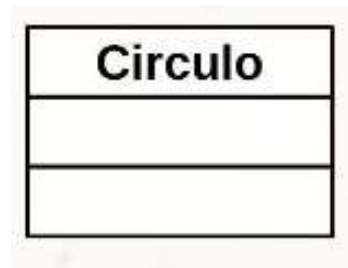
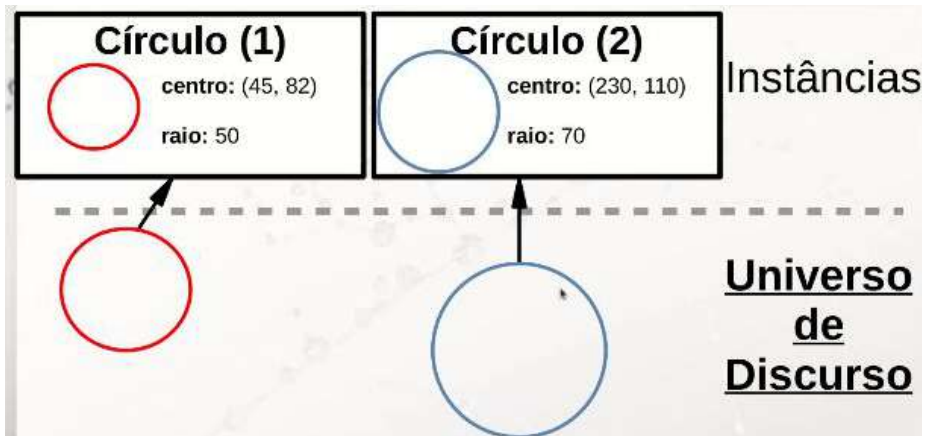


Introdução

UML (Unified Modeling language)

uml.org

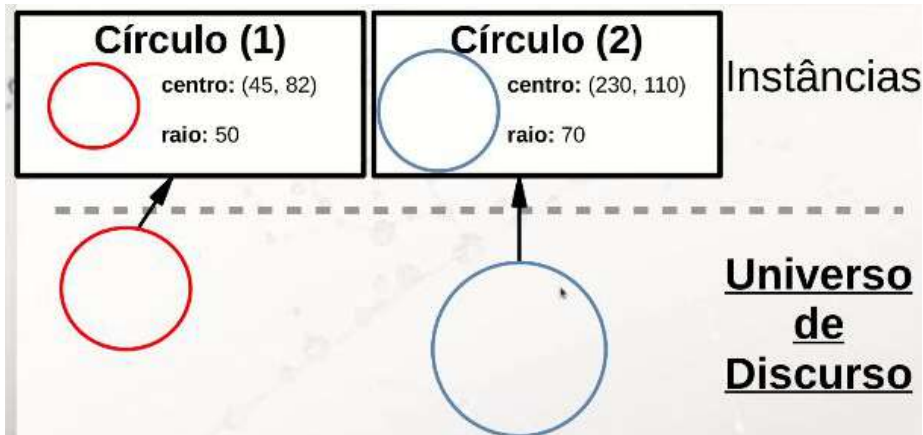
Padrão OMG (Object Management Group) no fim dos anos 90



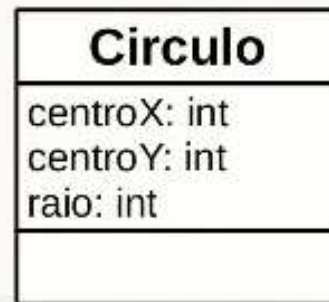
Introdução

UML (Unified Modeling language)

Classe



```
public class Circulo {  
    int centroX, centroY;  
    int raio;  
}
```

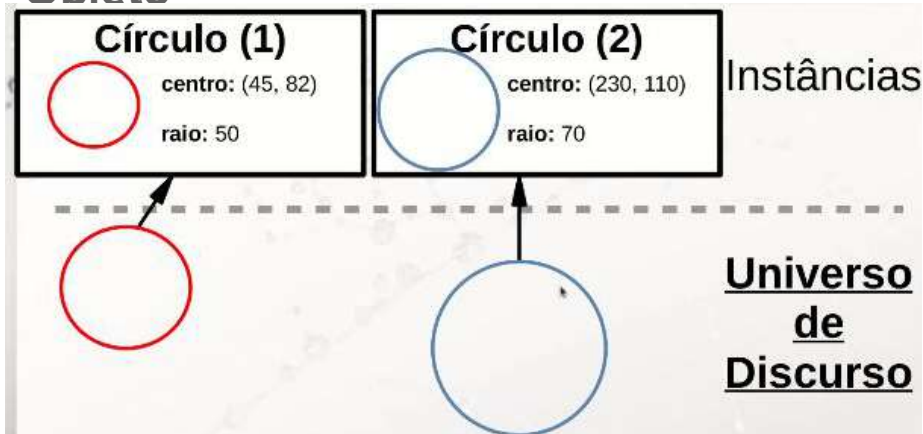


Introdução

Variável com Ponteiro

Circulo circ = new Circulo();

Objeto



```
public class Circulo {  
    int centroX, centroY;  
    int raio;  
}
```

Declaração da Referência

```
Circulo circ;
```

Instanciação do Objeto (chamada do construtor)

```
circ = new Circulo();
```


Introdução

■ Construtor (mesmo nome da classe)

- Todo o objeto deve ser instanciado (criado) através da ativação do método construtor.

■ Destrutor (finalize)

- O destrutor é o inverso do construtor, ele é ativado automaticamente quando o objeto está sendo destruído a fim de liberar a memória ocupada pelo mesmo.

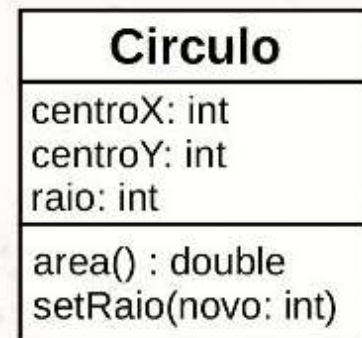
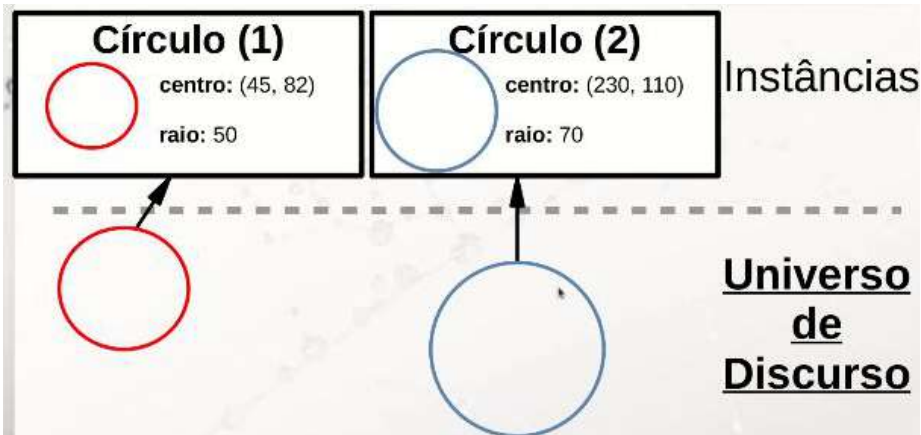
■ *Garbage Collection* (Coleta de Lixo)

- O mecanismo de gerência automática de memória que destrói o objeto quando ele não está mais sendo usado.

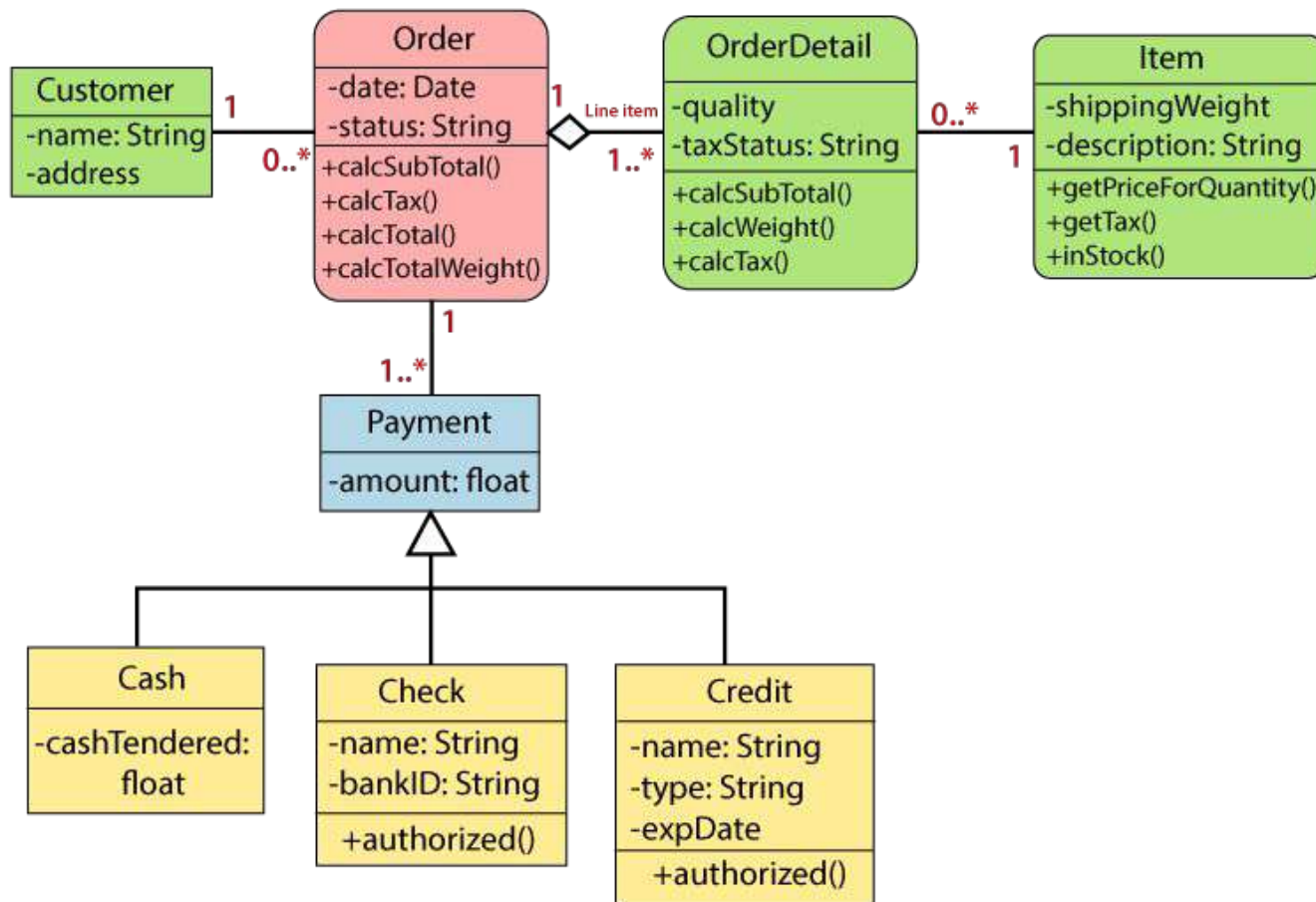
Introdução

UML (Unified Modeling language)

```
package pt.c02oo.s02classe.s01circulo02;  
  
public class Circulo {  
    int centroX, centroY;  
    int raio;  
  
    void mostraArea() {  
        System.out.println(Math.PI * raio * raio);  
    }  
}
```



Introdução

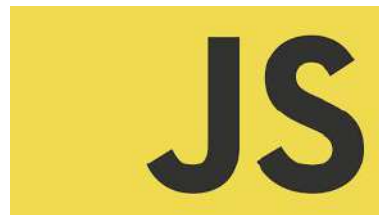
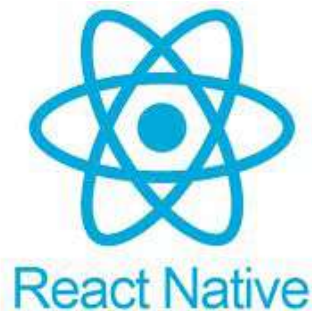
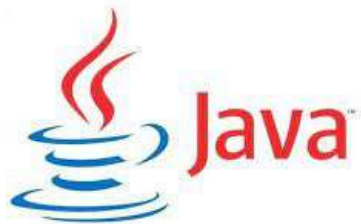


Introdução

Pensamos nas classes sendo objetos suas instâncias.

Prog. Funcional ~ Prog. OO

Ferramentas



Um pouco de prática

Vamos ver o conceito de:

- **Classes Genéricas**
- **Classes Aninhadas**
- **Funções Lambda**

Um pouco de prática

Classes Genéricas

O Java usualmente permite criar listas de objetos

```
public class Generic {  
    public static void main(){  
        List lista = new ArrayList();  
        lista.add(e: "Ana");  
        lista.add(e: 12);  
        lista.add(new Amigo());  
    }  
}
```

Isso gera problemas caso precise manipular os dados

```
for(Object o : lista){  
    if(o instanceof String){System.out.println(":" o);}
```

Um pouco de prática

Classes Genéricas

Podemos então utilizar o operador generics <> em tempo de compilação:

```
public static void main() {  
    List<String> lista = new ArrayList<>();  
    lista.add(e: "Ana");  
    lista.add(e: "Pedro");  
    lista.add(e: "Zagalo");  
    for(String o : lista) {  
        System.out.println(o);  
    }  
}
```

Problema: **Type Erasure**... Imagine tentar adicionar em uma lista tipada um objeto!

Um pouco de prática

Classes Genéricas

Problema: **Type Erasure**... Imagine tentar adicionar em uma lista tipada um objeto!

```
Public static void add(List lista, Pessoa pessoa){  
    lista.add(pessoa);}
```

Neste caso poderíamos adicionar a tipagem

```
Public static void add(List<Pessoa> lista, Pessoa pessoa){  
    lista.add(pessoa);}
```


Um pouco de prática

Classes Genéricas

Imagine o cenário:

Se eu adicionar

`printAnimais(passaros);` ou

`printAnumais(cachorros);`

Vai funcionar.

O Java sabe o tipo do objeto

```
abstract class Animal{
    public abstract void movimento();
}

class Cachorro extends Animal{
    @Override
    public void movimento(){System.out.println("andar");}
}

class Passaro extends Animal{
    @Override
    public void movimento(){System.out.println("voar");}
}

public class Generic {
    public static void main(){
        Passaro[] passaros={new Passaro(), new Passaro()};
        Cachorro[] cachorros={new Cachorro(), new Cachorro()};
    }

    private static void printAnimais(Animal[] animais){
        for (Animal animal : animais ) {animal.movimento();}
    }
}
```

Um pouco de prática

Classes Genéricas

Imagine o cenário:

O que acontece se eu tentar
adicionar na função
printAnimais o comando:
animais[1] = new Cachorro();

Da um erro de **Store Exception**

O mesmo vale para **Lista<>**

```
abstract class Animal{
    public abstract void movimento();
}

class Cachorro extends Animal{
    @Override
    public void movimento(){System.out.println("andar");}
}

class Passaro extends Animal{
    @Override
    public void movimento(){System.out.println("voar");}
}

public class Generic {
    public static void main(){
        Passaro[] passaros={new Passaro(), new Passaro()};
        Cachorro[] cachorros={new Cachorro(), new Cachorro()};
    }

    private static void printAnimais(Animal[] animais){
        for (Animal animal : animais ){animal.movimento();}
    }
}
```

Um pouco de prática

Classes Genéricas

Para resolver deixamos o objeto genérico!

Utilizamos o operador `?` no parâmetro

```
public class Generic {  
    public static void main(String[] args){  
        List<Passaro> passaros= List.of(new Passaro(), new Passaro());  
        List<Cachorro> cachorro= List.of(new Cachorro(), new Cachorro());  
        printAnimais( animais: passaros);  
    }  
    private static void printAnimais(List<? extends Animal> animais){  
        for (Animal animal : animais ){animal.movimento();}  
    }  
}
```

Um pouco de prática

Classes Genéricas: Vamos imaginar uma aplicação para alugar veículos. Temos duas classes

```
class Carro{
    private String nome;
    public Carro(String nome){this.nome = nome;}
    @Override
    public String toString(){
        return "Carro{" + "nome=" + nome + "}";
    }
}

class Barco{
    private String nome;
    public Barco(String nome){this.nome = nome;}
    @Override
    public String toString(){
        return "Barco{" + "nome=" + nome + "}";
    }
}
```

Um pouco de prática

Classes Genéricas: Agora criamos uma classe para a “base de dados”

```
class CarroRent{  
    private List<Carro> carrosDisponveis= new ArrayList<>(  
        Collections.of(new Carro( nome: "BMW"), new Carro( nome: "Renault")));  
    public Carro buscarCarroDisp() {  
        System.out.println("Buscando carro disponivel...");  
        Carro carro = carrosDisponveis.remove( index: 0);  
        System.out.println("Alugando: " + carro);  
        System.out.println("Disponiveis agora: ");  
        System.out.println("Disponiveis agora: " + carrosDisponveis);  
        return carro;  
    }  
    public void retornaCarro(Carro carro) {  
        System.out.println("Devolvendo: " + carro);  
        carrosDisponveis.add( carro);  
    }  
}
```

Um pouco de prática

Classes Genéricas: Então podemos simular uma locação de um carro (**simples**)

```
public class Generic {  
    public static void main(String[] args) {  
        CarroRent carrorent = new CarroRent();  
        Carro carro = carrorent.buscarCarroDisp();  
        System.out.println("Utilizando o carro por 10 dias...");  
        System.out.println("..."); System.out.println("...");  
        carrorent.retornaCarro(carro);  
    }  
}
```

Um pouco de prática

Classes Genéricas: Então podemos simular uma locação de um carro (**simples**)

```
public class Generic {  
    public static void main(String[] args){  
        CarroRent carrorent = new CarroRent  
        Carro carro = carrorent.buscarCarro  
        System.out.println("Utilizando o  
        System.out.println("..."); System  
        carrorent.retornaCarro(carro);  
    }  
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli)  
Buscando carro disponivel...  
Alugando: Carro{nome=BMW}  
Disponiveis agora:  
[Carro{nome=Renault}]  
Utilizando o carro por 10 dias...  
...  
...  
Devolvendo: Carro{nome=BMW}
```

Um pouco de prática

Classes Genéricas:

Mas e se quiser implementar o sistema para o Barco? Replica as classes e métodos

Nesse caso trocamos toda a referencia de carro para barco (método, objetos, base)

Bastante trabalhoso se tivermos muitas situações e modelos

Um pouco de prática

Classes Genéricas:

Mas e se quiser implementar o sistema para o Barco? Replica as classes e métodos
Ou criamos uma **Classe genérica** **<T, X, i>**

```
public class RentServ<T> {  
    private List<T> objDisponiveis;  
    public RentServ(List<T> objDisponiveis){this.objDisponiveis = objDisponiveis;}  
  
    public T buscarObjDisp() {  
        System.out.println("Buscando veiculo disponivel...");  
        T t = objDisponiveis.remove(index: 0);  
        System.out.println("Alugando: " + t);  
        System.out.println("Disponiveis agora: ");  
        System.out.println(objDisponiveis);  
        return t;  
    }  
    public void retornaObj(T t){  
        System.out.println("Devolvendo: " + t);  
        objDisponiveis.add(e: t);  
    }  
}
```

Um pouco de prática

Classes Genéricas:

E agora na aplicação basta passar um objeto como referência

```
public class Generic {  
    public static void main(String[] args){  
        List<Carro> carrosDisponveis= new ArrayList<>(  
            c::List.of(new Carro( nome: "BMW"), new Carro( nome: "Renault")));  
        List<Barco> barcosDisponveis= new ArrayList<>(  
            c::List.of(new Barco( nome: "Profissional"), new Barco( nome: "Passeio")));  
        RentServ<Barco> rental = new RentServ<>( objDisponiveis: barcosDisponveis);  
        Barco barco = rental.buscarObjDisp();  
    }  
}
```

Um pouco de prática

Métodos Genéricos:

Podemos também criar métodos genéricos (quando não podemos definir o tipo diretamente na declaração da classe).

```
private static <Tipo> void nomeMetodo(Tipo tipo);
```

```
private static <T> List<T> criaArrayObj(T t){  
    List<T> list = List.of(e1: t);  
    return(list);  
}
```

Desta forma podemos chamar o método com qualquer tipo:

```
List<Barco> b = criaArrayObj(new Barco("Veleiro"));
```

```
System.out.println(b);
```

Um pouco de prática

Métodos Genéricos:

Métodos e classes genéricas podem ajudar quando sua base contém tipos complexos ou diversidade de exemplos.

Exemplo (Sensores).

O tipo genérico (T) pode ser especificado com **extends** e **super**

Um pouco de prática

Classes Internas (Inner Class) anônimas e aninhadas

Classes que tem uma relação hierárquica


```
public class InnerC {  
    private String name = "Beef";  
  
    class ClasseFilha{  
        public void printAttOuterClass() {  
            System.out.println(" : " + name);  
        }  
    }  
  
    public static void main(String[] args) {  
        InnerC classe = new InnerC();  
        ClasseFilha cf = classe.new ClasseFilha();  
        cf.printAttOuterClass();  
    }  
}
```

Um pouco de prática

Classes Internas (Inner Class) anônimas e aninhadas

Se utilizar o “**this**” na impressão, vai referenciar a classe **filha** ou **pai**?

```
public class InnerC {  
    private String name = "Beef";  
  
    class ClasseFilha{  
        public void printAttOuterClass(){  
            System.out.println(":" + name);  
        }  
    }  
  
    public static void main(String[] args){  
        InnerC classe = new InnerC();  
        ClasseFilha cf = classe.new ClasseFilha();  
        cf.printAttOuterClass();  
    }  
}
```



Um pouco de prática

Classes Internas (Inner Class) anônimas

Se utilizar o “[this](#)” na impressão, vai refer

```
public class InnerC {  
    private String name = "Beef";  
  
    class ClasseFilha{  
        public void printAttOuterClass(){  
            System.out.println(":" + name);  
        }  
    }  
  
    public static void main(String[] args){  
        InnerC classe = new InnerC();  
        ClasseFilha cf = classe.new ClasseFilha();  
        cf.printAttOuterClass();  
    }  
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ tad:  
Beef  
com.mycompany.tads001.InnerC$ClasseFilha@4517d9a3  
-----  
BUILD SUCCESS  
-----  
Total time: 0.927 s  
Finished at: 2023-07-28T18:29:55-03:00  
-----
```

System.out.println(this);

Um pouco de prática

Classes Internas (Inner Class) anônimas

Se utilizar o “[this](#)” na impressão, vai refer

```
public class InnerC {  
    private String name = "Beef";  
  
    class ClasseFilha{  
        public void printAttOuterClass(){  
            System.out.println(":" + name);  
        }  
    }  
  
    public static void main(String[] args){  
        InnerC classe = new InnerC();  
        ClasseFilha cf = classe.new ClasseFilha();  
        cf.printAttOuterClass();  
    }  
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ tad:  
Beef  
com.mycompany.tads001.InnerC$ClasseFilha@4517d9a3  
-----  
BUILD SUCCESS  
-----  
Total time: 0.927 s  
Finished at: 2023-07-28T18:29:55-03:00  
-----
```

System.out.println(this);

System.out.println(InnerC.this);

Um pouco de prática

Classes Internas (Inner Class) anônimas e aninhadas

Classe Local - Fica restrito ao domínio em que está criada. Aninhamento método>classe>método

```
void imprime() {  
    class ClasseLocal {  
        public void imprimeL() {  
            System.out.println("name");  
        }  
    }  
    ClasseLocal lc = new ClasseLocal(); lc.imprimeL();  
}
```

Um pouco de prática

Classes anônimas

Existem por um período de tempo de execução e não podem ser reutilizadas fora de seu escopo

```
class Animal{  
    public void andar(){  
        System.out.println("...andando");  
    }  
}
```




Agora imagine que queira alterar o método andar. Poderíamos criar uma outra classe que estende Animal alterando o comportamento,

Um pouco de prática

Classes anônimas

Existem por um período de tempo de execução e não podem ser reutilizadas fora de seu escopo

```
class Animal{  
    public void andar(){  
        System.out.println("...andando");  
    }  
}  
  
class Cachorro extends Animal{  
    @Override  
    public void andar(){System.out.println("cachorro andando...");}  
}
```



polimorfismo

Um pouco de prática

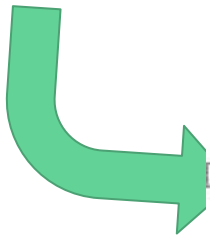
Classes anônimas

Não precisa criar uma classe com a variação de saída.

Podemos alterar somente em um determinado momento de execução

```
public static void main(String[] args){  
    Animal animal = new Animal();  
    animal.andar();|
```

Polimorfismo



```
class Cachorro extends Animal{  
    @Override  
    public void andar(){System.out.println("cachorro andando...");}  
}
```

Um pouco de prática

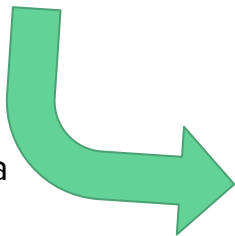
Classes anônimas

Não precisa criar uma classe com a variação de saída.

Podemos alterar somente em um determinado momento de execução

```
public static void main(String[] args){  
    Animal animal = new Animal();  
    animal.andar();  
}
```

Classe Anonima



```
public static void main(String[] args){  
    Animal animal = new Animal(){  
        @Override  
        public void andar(){System.out.println("andando muito");}  
    };  
    animal.andar();  
}
```

Um pouco de prática

Classes anônimas

Não consigo criar métodos que não existem na classe original.

Consigo sobrescrever os métodos da classe super (ou a classe instanciada) mas seu uso fica limitado ao escopo do @Override

Útil em ocasiões de exceção ou código dinâmico!

Um pouco de prática

Classes Internas Estáticas

Parecida com as classes aninhadas que vimos

```
▶ public class OuterClassesTest03 {  
    ⚡ static class Nested {  
  
    }  
  
    ▶ public static void main(String[] args) {  
  
    }  
}
```

← Classe Top Level

← Também é vista como Top Level (static)

É inserido internamente a outra classe por questão de empacotamento e hierarquia

Um pouco de prática

Classes Internas Estáticas

Dentro da classe filha não seria possível acessar diretamente, por exemplo, um atributo da classe mãe, pois não é static. A forma de referenciar o atributo neste caso seria instanciar como abaixo.

```
public class OuterClassesTest03 {  
    private String name;  
    static class Nested {  
        void print(){  
            System.out.println(new OuterClassesTest03().name);  
        }  
    }  
    public static void main(String[] args) {  
  
    }  
}
```

Para acessar o método na função main seria:
Nested n = new Nested();
n .print();

Um pouco de prática

Classes Internas Estáticas

Parametrizando os comportamentos: Vamos criar uma classe carro

```
class Car{  
    private String name = "Renault";  
    private String color;  
    private int year;  
  
    public Car(String color, int year){  
        this.color = color;  
        this.year = year;  
    }  
    public String getName() {return name;}  
    public String getColor() {return color;}  
    public int getYear() {return year;}  
}
```

Um pouco de prática

Classes Internas Estáticas

Agora criamos uma lista de objetos e um método, por exemplo, para filtrar os carros por uma cor:

```
private List<Car> cars = List.of(  
    new Car( color: "silver", year: 2021),  
    new Car( color: "black", year: 2019),  
    new Car( color: "red", year: 2023));  
  
private static List<Car> filterCar(List<Car> cars){  
    List<Car> colorCars = new ArrayList<>();  
    for(Car car : cars){  
        if(car.getColor().equals( anObject: "red")){ colorCars.add( e: car);}  
    } return colorCars;  
}  
  
public static void main(String[] args) {  
  
}
```

Um pouco de prática

Classes Internas Estáticas

E se quisermos fazer um filtro para cada cor? Replicariamos o método de filtragem?

Ou poderíamos passar a cor como parâmetro:

```
private static List<Car> filterCar(List<Car> cars, String color)
```

Ou ainda pode surgir a necessidade de filtrar com outro atributo, como o ano.

Vamos otimizar o processo:

```
interface CarPredicate{  
    boolean test(Car car);  
}
```

Um pouco de prática

Classes Internas Estáticas

A nova interface é quem cuidará da regra de negócio, através do parâmetro booleano

```
private static List<Car> filter(List<Car> cars, CarPredicate carpredicate){  
    List<Car> filteredCar = new ArrayList();  
    for (Car car : cars){  
        if (carpredicate.test(car)) {filteredCar.add(car);}  
    }  
    return filteredCar;  
}
```

Este método é genérico e utilizaremos uma classe anônima para definir o comportamento do parâmetro (regra)

Um pouco de prática

Classes Internas Estáticas

O método principal, com a classe anônima é que implementa a regra conforme a necessidade

```
public static void main(String[] args) {  
    //System.out.println(filterCar(cars));  
    List<Car> redCars = filter(cars, new CarPredicate() {  
        @Override  
        public boolean test(Car car) {  
            return car.getColor().equals("red");  
        }  
    });  
    System.out.println("redCars");  
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ tads001 ---  
[Car{name=Renault, color=red, year=2023}]
```

BUILD SUCCESS

Total time: 0.901 s

Um pouco de prática

Classes Internas Estáticas

Podemos deixar muito mais genérico

```
private static <T> List<T> filter(List<T> list, Predicate<T> predicate) {  
    List<T> filteredList = new ArrayList<>();  
    for (T e : list) {filteredList.add(e);}   
    return filteredList;  
}
```

```
private static List<Car> filterCar(List<Car> cars) {  
    List<Car> colorCars = new ArrayList<>();  
    for(Car car : cars) {  
        if(car.getColor().equals(anObject: "red")) { colorCars.add(e: car);}   
    } return colorCars;  
}
```

Nosso método agora recebe qualquer tipo de objeto com qualquer condição

Um pouco de prática

Funções Lambda

São funções de primeira ordem

Podem ser criadas, armazenadas e passadas como parâmetro (como outras variáveis).

Tem visibilidade diferente das funções normais

Utilizadas em padrões de projeto e interfaces gráficas

Um pouco de prática

Funções Lambda

Muito comum em arquiteturas mobile e web

```
## Criando como variável
```

```
Javascript
```

```
```js
```

```
let fn = (x) => x * 3 + 1;
```

```
console.log(fn(2)) //7
```

```
```
```

```
Java
```

```
```java
```

```
Function<Integer, Integer> function = (x) -> x * 3 + 1;
```

```
System.out.print(function.apply(2));
```

```
```
```


Um pouco de prática

Funções Lambda

Funções Lambda são anônimas, e sua sintaxe tem sempre:

(parametro) -> <expressão>

A expressão pode ou não retornar algo.

Ex

(Car car) -> car.getColor().equals("red");

Um pouco de prática

Funções Lambda

Vamos ver um exemplo para imprimir qualquer lista usando (f)lambda:

```
public class Lambda01 {  
  
    private static <T> void fEach(List<T> list, Consumer<T> consumer) {  
        for (T e : list) { consumer.accept(e); }  
    }  
  
    public static void main(String[] args) {  
        List<String> str = List.of("Ana", "Pedro", "Carlos", "Dante");  
        fEach(str, (String s) -> System.out.println(s));  
    }  
}
```

Para imprimir uma outra lista, de inteiros por exemplo, bastaria alterar o tipo na chamada

Um pouco de prática

Funções Lambda

Uma função Lambda tem sempre um parâmetro recebido e um de retorno.

Ex: Recebe uma lista de Strings e retorna o tamanho de cada uma.

Primeiro vamos criar o método genérico!

```
private static <T, R> List<R> map(List<T> list, Function<T, R> function){  
    List<R> result = new ArrayList<>();  
    for(T e : list){  
        R r = function.apply(e);  
        result.add(r);  
    }  
    return result;  
}
```

Um pouco de prática

Funções Lambda

```
private static <T, R> List<R> map(List<T> list, Function<T, R> function){
    List<R> result = new ArrayList<>();
    for(T e : list){
        R r = function.apply(e);
        result.add(r);
    }
    return result;
}

public static void main(String[] args) {
    List<String> str = List.of("Ana", "Pedro", "Carlos", "Dante");
    List<Integer> inteiros = map(str, (String s)->s.length());
    System.out.println(inteiros);
}
```

Um pouco de prática

Funções Lambda

Neste caso poderia passar qualquer função como parâmetro:

```
List<String> fn = map(str, s -> s.toUpperCase( ));
```

```
List<Integer> inteiros = map(str, (String s)->s.length());
```

Um pouco de prática

Classes Anônimas e funções Lambda

Como uma função Lambda encaixa no conceito de uma interface anônima?

SAM (Single Abstract Method)

Se a classe so possui um método abstrato podemos utilizar uma função Lambda para implementar este recurso.

`Trabalho.realizarJornada()`

Precisamos implementar o método assalariado

Um pouco de prática

Classes Anônimas e funções Lambda

Não recebe um parâmetro e gera como saída uma interface que implementa uma String, como função de valor:

```
Trabalho.realizarJornada( () -> ( “ Trabalhando de forma ágil“ ) );
```

Mais compacto e limpo

Útil em caso de uma operação única, como comparação de valores em uma busca

Um pouco de prática

Classes Anônimas e funções Lambda

```
(Integer idade) → idade > 10; //Entrada é um inteiro e devolve um booleano
```

```
(Integer idade) → idade > 10? "Criança" : "Não é Criança"; //Entrada é um inteiro e devolve uma String
```

```
(Integer idade) → {System.out.println(idade);}; //Entrega é Inteiro, saída é um void
```

```
() → {return Math.random() + "Number";}; //Sem Entrada, e devolve String
```


Um pouco de prática

Classes Anônimas e funções Lambda

Isso pode otimizar, por exemplo, uma busca:

```
(String nome, List<Pessoa> list) → {  
    return list.stream()  
        .filter(e → e.getNome().startsWith(nome))  
        .map(Pessoa::getIdade)  
        .findFirst();  
};  
// Entrada é uma string e uma lista de Pessoa, e o retorno é um  
Optional<Integer>
```

Um pouco de prática

Classes Anônimas e funções Lambda

Também é possível
definir (f)Lambda sem
passar um tipo

Lambda Implícitos

```
(idade) → idade > 10;  
  
(idade) → idade > 10? "Criança" : "Não é Criança";  
  
idade → {System.out.println();};  
() → {return Math.random() + "Number";};  
  
(nome, list) → {  
    return (  
        list.stream()  
            .filter(e → e.getNome().startsWith(nome))  
            .map(Pessoa::getIdade)  
            .findFirst()  
    );  
};
```

Conclusão

Introdução e overview

Classes aninhadas, anônimas e genéricas

Funções Lambdas

Referências

1. DEITEL. JAVA Como Programar. 8a. ed. São Paulo: Pearson Prentice Hall, 2010.
2. JANDL JUNIOR, Peter. Java Guia do Programador. São Paulo: Novatec, 2014.
3. FREEMAN, Eric. Use a cabeça: padrões e projetos. 2. ed. rev. Rio de Janeiro: Alta Books, 2009