



LINGUAGEM DE PROGRAMAÇÃO JAVA

Cefet – Maracanã -RJ

BCC/TSI

Prof. Gustavo Guedes

E-mail: gustavo.guedes@cefet-rj.br

A PALAVRA RESERVADA FINAL

- O Java permite que você aplique a palavra reservada final em classes. Se você fizer isso, a classe não pode ter subclasses. Por exemplo, a classe `java.lang.String` é uma classe final. Isso é feito por questões de segurança, porque ela garante que se um método referenciar uma string, ele é definitivamente uma string de classe String e não uma string de uma classe que é uma subclasse modificada de String, que poderia ter sido alterada.
- Você também pode marcar métodos individuais como final. Os métodos marcados como final não pode ser sobrepostos. Por segurança, você deve tornar um método final se esse método tiver uma implementação que não deve ser alterada e é crítica para o estado de consistência do objeto.



CLASSES ABSTRATAS

- Problema: imagine a classe Veiculo. Imagine 2 subclasses (Caminhonete e Barca) que tenham que possuir cálculos para determinar o consumo de combustível. O consumo de combustíveis da barca e do caminhonete são diferentes. A classe Veiculo não pode fornecer esses dois métodos, mas suas subclasses Caminhonete e Barca os fornecem.
- O java permite que o projetista de uma classe especifique que uma superclasse declara um método que não fornece uma implementação. Isso se chama método abstrado. A implementação desse método é fornecida pelas subclasses. Qualquer classe que possua um ou mais métodos abstratos se chama classe abstrata.
- O compilador impede que você crie uma instância de uma classe abstrata.
- Entretanto, as classes abstratas podem ter atributos de dados, métodos concretos e construtores.



CLASSES ABSTRATAS

```
○ public abstract class Abstrato {  
○     public abstract double calculaConsumoCombustivel();  
○ }  
○ -----  
○ public class Barca extends Abstrato{  
○     private final double PRECO_CARVAO = 10.1;  
○     public double calculaConsumoCombustivel(){  
○         return PRECO_CARVAO * 100; //100 KM  
○     }  
○ }  
○ -----  
○ public class Caminhonete extends Abstrato{  
○     private final double PRECO_DIESEL = 14.3;  
○     public double calculaConsumoCombustivel(){  
○         return PRECO_DIESEL * 100; //100 KM  
○     }  
○ }
```



INTERFACES

- A interface de uma classe é um contrato entre a classe e determinada interface. As classes concretas fornecem a implementação de cada método.
- Entretanto, uma classe abstrata pode retardar a implementação declarando que o método deve ser abstrato; uma interface Java declara apenas o contrato e nenhuma implementação.
- Uma classe concreta implementa uma interface definindo todos os métodos declarados pela interface. Muitas classes podem implementar a mesma interface. Essas classes não precisam compartilhar a mesma hierarquia de classe. Uma classe também pode implementar mais de uma interface.



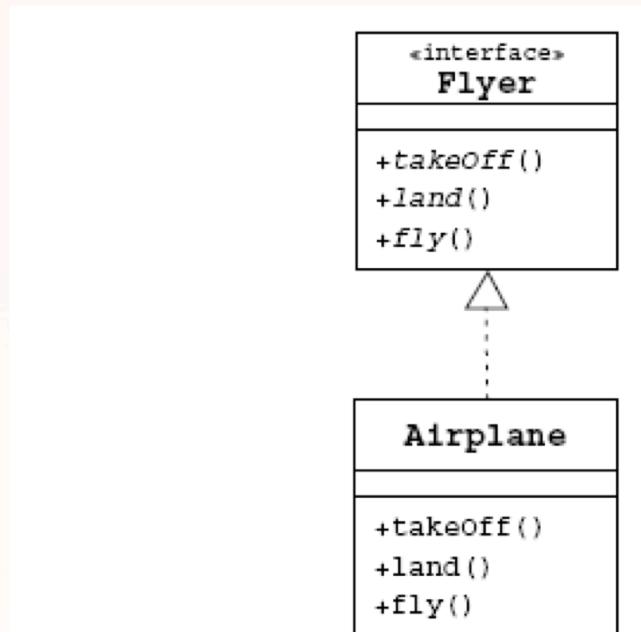
INTERFACES

- Assim como acontece com as classes abstratas, você pode usar o nome de uma interface como tipo da variável de referência. A vinculação dinâmica normal entra em efeito. As referências são convertidas para os tipos de interface, e você usa o operador instanceof para determinar se uma classe implementa uma interface.
- Uma interface também pode declarar constantes public, static e final.



INTERFACES

- Imagine um grupo de objetos que compartilha a mesma capacidade: **eles voam**. Você pode construir uma interface pública chamada **Flyer** que suporta três operações: **takeOff**, **land** e **fly**;



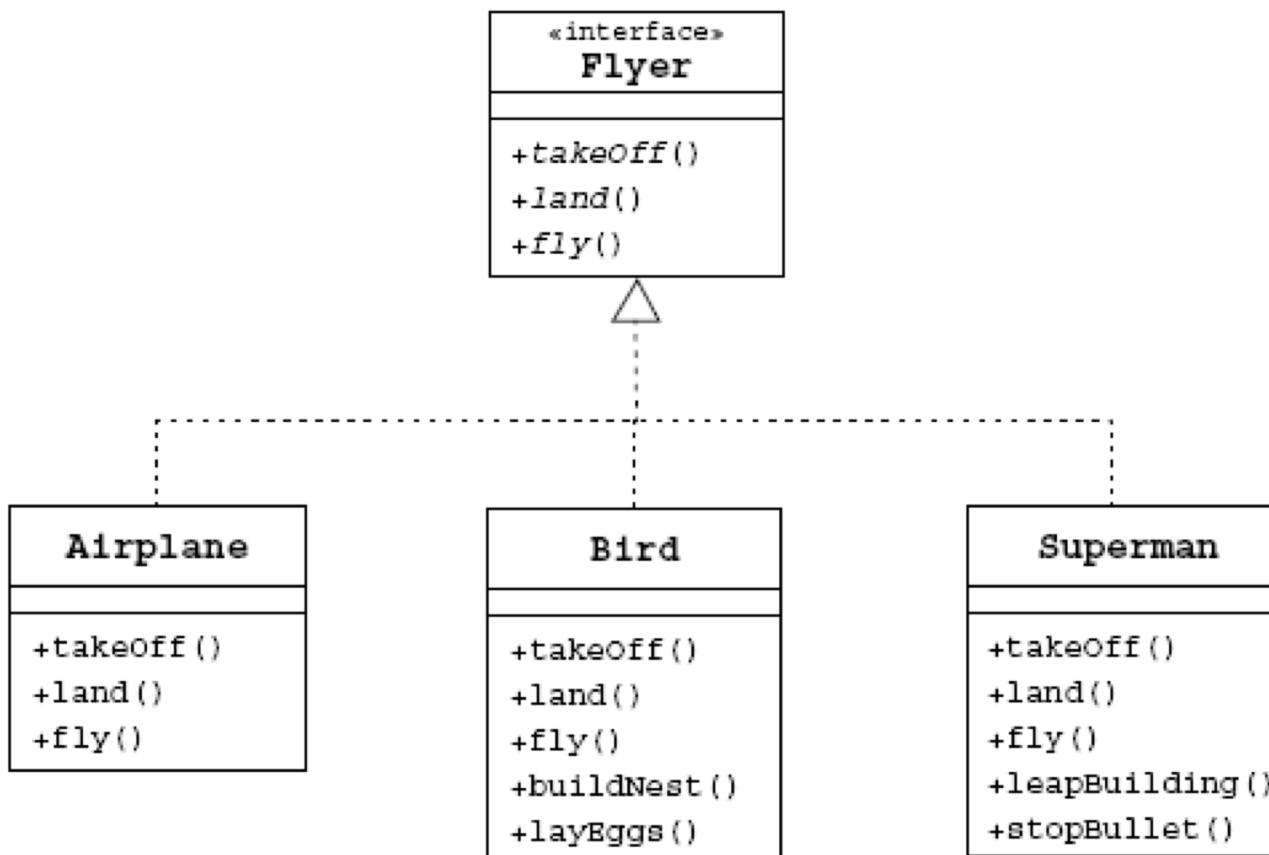
```
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}
```

INTERFACES

```
public class Airplane implements Flyer {
    public void takeOff() {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land() {
        // lower landing gear
        // decelerate and lower flaps until touch-down
        // apply brakes
    }
    public void fly() {
        // keep those engines running
    }
}
```



INTERFACES



Declaração de Interfaces

- Uma interface é como uma classe 100% abstrata;
- As interfaces são implicitamente abstract, colocar esse modificador é opcional;
- Uma interface só pode ser public ou default;
- Uma interface NÃO pode ter o modificador private ou protected;
- Uma interface pode estender várias interfaces (isso mesmo, a palavra usada é extends);
- Uma interface NÃO pode implementar outra interface ou classe (parece estranho, mas uma interface estende outras interfaces);



INTERFACES (REGRAS DE CLASSES)

- Uma classe **concreta** (não abstract) que implementa uma interface deve fornecer implementações para todos os seus métodos;
- Uma classe **abstrata** quem implementa uma interface não é obrigada a implementar seus métodos, mas a primeira classe concreta da árvore de herança deve implementá-los;



Interfaces (Regras de Métodos)

- Todos os métodos de uma interface são implicitamente **public** e **abstract**;
- E já que os métodos são abstract eles devem terminar com ponto-e-vírgula (');
- Os métodos declarados em uma interface **não podem ser static**;
- **Regras das Variáveis (Constantes)**
- Todas as variáveis de uma interface são public, static e final, sendo assim,
- todas as variáveis são constantes (toda constante deve ser inicializada);
- As declarações public, static e final são opcionais;



Interfaces (Regras de Variáveis)

- Todas as variáveis de uma interface são **public**, **static** e **final**, sendo assim, todas as variáveis são constantes (toda constante deve ser inicializada);
- As declarações **public**, **static** e **final** são opcionais;

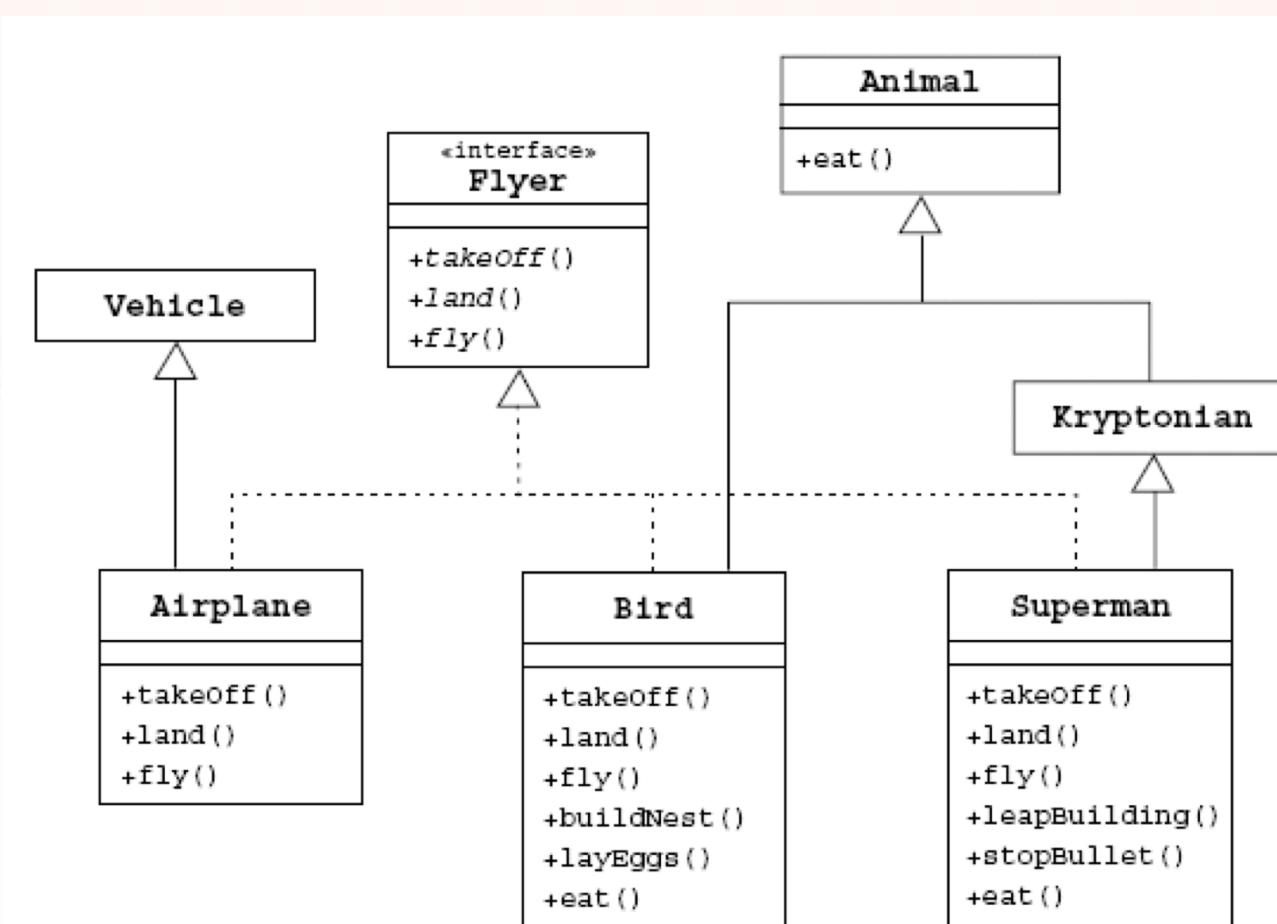


INTERFACES

- É possível haver várias classes que implementam a interface Flyer. Um avião pode voar, um pássaro também pode voar, o Superman pode voar e assim por diante.

INTERFACES

- A seguir, a descrição da classe Bird:



INTERFACES

- É possível haver várias classes que implementam a interface Flyer. Um avião pode voar, um pássaro também pode voar, o Superman pode voar e assim por diante. Um Airplane é um Vehicle e pode voar. Um Bird é um Animal e pode voar. Esses exemplos mostram que uma classe pode herdar de uma classe, mas também pode implementar alguma outra interface.
- Isso lembra uma herança múltipla, mas não é exatamente a mesma coisa. O perigo da herança múltipla é que uma classe pode herdar duas implementações distintas do mesmo método. Isso não é possível no caso das interfaces, porque uma declaração de método de interface não fornece nenhuma implementação.

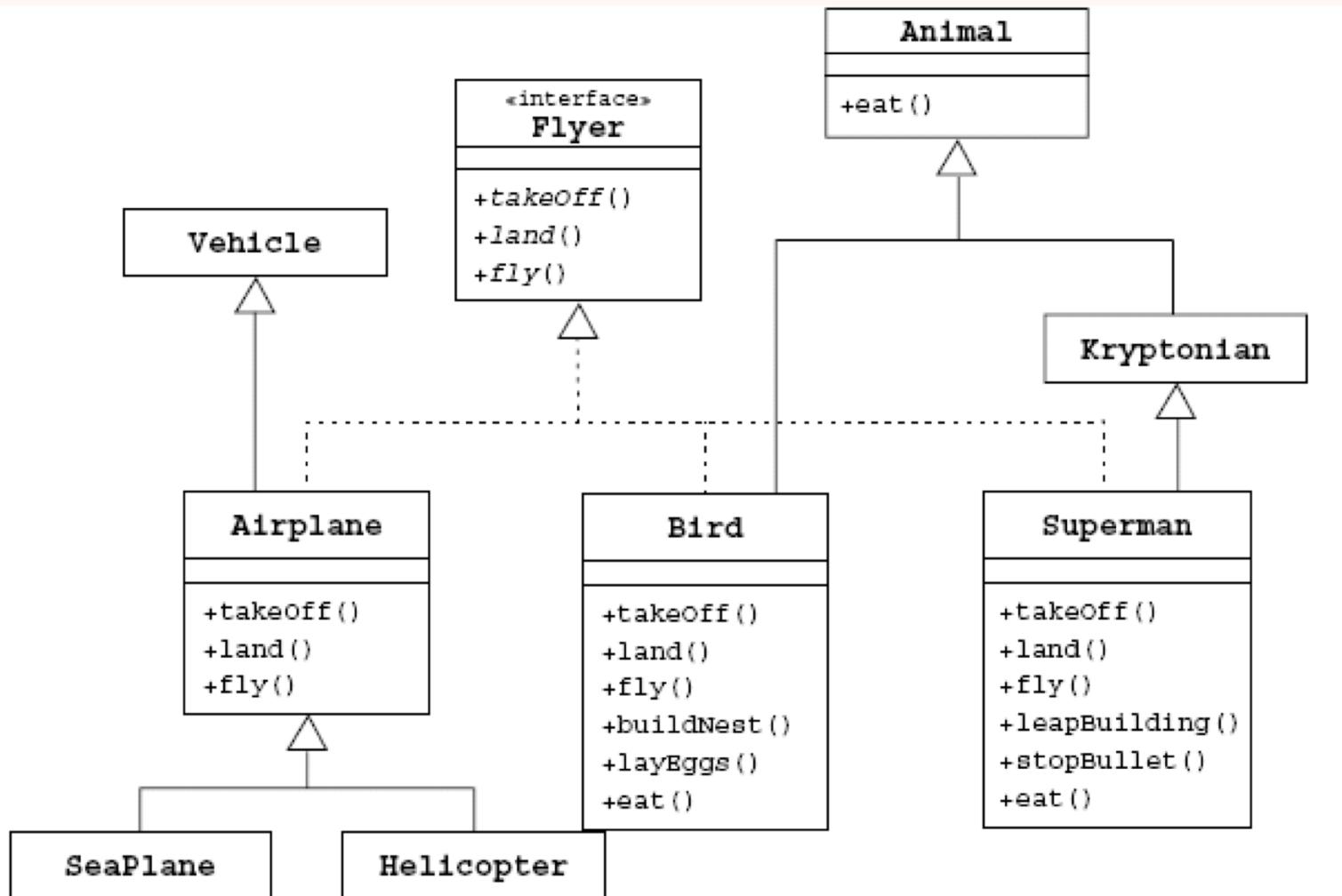


INTERFACES

```
public class Bird extends Animal implements Flyer {  
    public void takeOff()    { /* take-off implementation */ }  
    public void land()       { /* landing implementation */ }  
    public void fly()        { /* fly implementation */ }  
    public void buildNest() { /* nest building behavior */ }  
    public void layEggs()   { /* egg laying behavior */ }  
    public void eat()        { /* override eating behavior */ }  
}
```

- A cláusula `extends` deve vir antes da cláusula `implements`. A classe `Bird` pode fornecer seus próprios métodos (`buildNest` e `layEggs`) e também pode sobrepor os métodos da classe `Animal` (`eat`).
- Suponhamos que você esteja construindo um sistema de software para controle de aviões. Ele precisa conceder permissão de aterrissagem e decolagem para objetos voadores de todos os tipos.

INTERFACES



INTERFACES

```
public class Airport {  
    public static void main(String[] args) {  
        Airport metropolisAirport = new Airport();  
        Helicopter copter = new Helicopter();  
        SeaPlane sPlane = new SeaPlane();  
  
        metropolisAirport.givePermissionToLand(copter);  
        metropolisAirport.givePermissionToLand(sPlane);  
    }  
  
    private void givePermissionToLand(Flyer f) {  
        f.land();  
    }  
}
```

Exercício

- Crie um atributo de instância “nome” na classe Airport juntamente com os getters e setters.
- Ao criar um aeroporto novo, atribua um nome.
 - Ex:“Galeão”.
- Dentro do método givePermissionToLand, incluir a linha:
 - Aeroporto [Galeão] vai dar permissão para pouso.
 - XXX pousando (Já está feito).
 - XXX pousando (Já está feito).



Exercício

- Criar em Seaplane o método ligarTurbinas.
- Criar em Helicopter o método ligarHelice
- Criar uma classe principal com o método main.
- Criar um outro método estático em principal com a seguinte assinatura:
 - public static Airplane [] retornaAleatorio()
- Esse método deve criar um array com o tamanho sendo calculado por Math.random de 0 a 100. Esse método deve ser preenchido com instâncias de Helicopter e Seaplane. Cada posição do array deverá conter um Helicopter ou Seaplane. Para saber qual instância será criada, crie um terceiro método estático chamado alea() que deverá retornar aleatoriamente um número. Utilize esse método para preencher o array. Se o retorno do metodo ale() for ímpar, crie um Helicopter, se for par, um Seaplane.
- Ao final de tudo, crie um novo método estático que mande todos os

Exercício

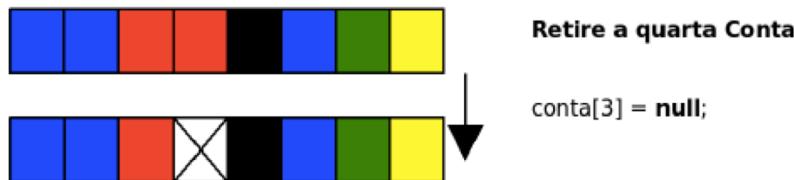
- Ao final de tudo, crie um novo método estático que mande todos os objetos decolarem, entretanto, se for um Helicopter, chame ligar hélice antes. Se for um Seaplane, ligar turbinas.



Coleções

Como vimos no capítulo de arrays, manipulá-las é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:

- não podemos redimensionar um array em Java;
- é impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
- não conseguimos saber quantas posições do array já foram populadas sem criar, para isso, métodos auxiliares.



Na figura acima, você pode ver um array que antes estava sendo completamente utilizado e que, depois, teve um de seus elementos removidos.

Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco? Precisaremos procurar por um espaço vazio? Guardaremos em alguma estrutura de dados externa, as posições vazias? E se não houver espaço vazio? Teríamos de criar um array maior e copiar os dados do antigo para ele?

Há mais questões: como posso saber quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array inteiro para conseguir essa informação?

Coleções

- Para resolver estes problemas, a Sun criou um conjunto de classes e interfaces conhecido como Collections Framework, que reside no pacote `java.util`.
- O primeiro recurso que veremos será o recurso das Listas.
- Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.
- Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos.



Coleções

- Ela resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho “infinito”,...). [Esse código já está pronto!](#)
- A API de Collections traz a interface `java.util.List`, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.
- A implementação mais utilizada da interface `List` é a `ArrayList`, que trabalha com um array interno para gerar uma lista. Ela é mais rápida na pesquisa do que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.



Coleções

- É comum confundirem uma ArrayList com um array, porém ela não é um array. O que ocorre é que, internamente, ela usa um array como estrutura para armazenar os dados, porém este atributo está propriamente encapsulado e você não tem como acessá-lo. Repare, também, que você não pode usar [] com uma ArrayList, nem acessar atributo length. **Não há relação!**



ARRAYLIST

- Para criar um ArrayList, basta chamar o construtor:
 - `ArrayList lista = new ArrayList();`
- É sempre possível abstrair a lista a partir da interface List:
 - `List lista = new ArrayList();`
- Para criar uma lista de nomes (String), podemos fazer:

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
```



ARRAYLIST

- Note que, em momento algum, dizemos qual é o tamanho da lista;
- podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário. Toda lista (na verdade, toda Collection) trabalha do modo mais genérico possível. Isto é, não há uma ArrayList específica para Strings, outra para Números, outra para Datas etc. Todos os métodos trabalham com Object. Assim, é possível criar, por exemplo, uma lista de Contas Correntes:
 - `ContaCorrente c1 = new ContaCorrente();`
 - `c1.deposita(100);`
 - `ContaCorrente c2 = new ContaCorrente();`
 - `c2.deposita(200);`
 - `List contas = new ArrayList();`
 - `contas.add(c1);`
 - `contas.add(c2);`
- Para saber quantos elementos há na lista, podemos usar o método `size()`:
 - `System.out.println(contas.size());`



ARRAYLIST

- Note que, em momento algum, dizemos qual é o tamanho da lista;
- podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário. Toda lista (na verdade, toda Collection) trabalha do modo mais genérico possível. Isto é, não há uma ArrayList específica para Strings, outra para Números, outra para Datas etc. Todos os métodos trabalham com Object. Assim, é possível criar, por exemplo, uma lista de Contas Correntes:
 - `ContaCorrente c1 = new ContaCorrente();`
 - `c1.deposita(100);`
 - `ContaCorrente c2 = new ContaCorrente();`
 - `c2.deposita(200);`
 - `List contas = new ArrayList();`
 - `contas.add(c1);`
 - `contas.add(c2);`
- Para saber quantos elementos há na lista, podemos usar o método `size()`:
 - `System.out.println(contas.size());`



ARRAYLIST

- Há ainda um método `get(int)` que recebe como argumento o índice do elemento que se quer recuperar. Através dele, podemos fazer um `for` para iterar na lista de contas:

- ```
for (int i = 0; i < contas.size(); i++) {
 • contas.get(i); // código não muito útil....
}
}
```
  
- Mas como fazer para imprimir o saldo dessas contas? Podemos acessar o `getSaldo()` diretamente após fazer `contas.get(i)`? Não podemos; lembre-se que toda lista trabalha sempre com `Object`. Assim, a referência devolvida pelo `get(i)` é do tipo `Object`, sendo necessário o cast para `ContaCorrente` se quisermos acessar o `getSaldo()`:
  
- ```
for (int i = 0; i < contas.size(); i++) {  
    • ContaCorrente cc = (ContaCorrente) contas.get(i);  
    • System.out.println(cc.getSaldo());  
}  
}
```



ARRAYLIST

- Há ainda outros métodos como remove() que recebe um objeto que se deseja remover da lista; e contains(), que recebe um objeto como argumento e devolve true ou false, indicando se o elemento está ou não na lista.
- Uma lista é uma excelente alternativa a um array comum, já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.



ARRAYLIST

- Vale ressaltar a importância do uso da interface List: quando desenvolvemos, procuramos sempre nos referir a ela, e não às implementações específicas. Por exemplo, se temos um método que vai buscar uma série de contas no banco de dados, poderíamos fazer assim:
- ```
class Agencia {
 • public ArrayList buscaTodasContas() {
 • ArrayList contas = new ArrayList();
 • return contas;
 • }
○ }
○ Porém, para que precisamos retornar a referência específica a uma ArrayList? Para que ser tão específico? Dessa maneira, o dia que optarmos por devolver uma
LinkedList em vez de ArrayList, as pessoas que estão usando o método
buscaTodasContas poderão ter problemas, pois estavam fazendo referência a uma
ArrayList. O ideal é sempre trabalhar com a interface mais genérica possível.
```



# ARRAYLIST

---

- `class Agencia {`
  - // modificacao apenas no retorno:
  - `public List buscaTodasContas() {`
  - `ArrayList contas = new ArrayList();`
  - `return contas;`
  - `}`
- `}`
- Assim como no retorno, é boa prática trabalhar com a interface em todos os lugares possíveis: métodos que precisam receber uma lista de objetos têm List como parâmetro em vez de uma implementação em específico como ArrayList, deixando o método mais flexível:
- `class Agencia {`
  - `public void atualizaContas(List contas) {`
  - `// ...`
  - `}`
- `}`



# Exercício

---

- Criar uma interface chamada “Corredor” que possui dois métodos: correr() e parar().
- Criar uma classe abstrata chamada Animal. [Nome e idade]
- Criar uma classe concreta chamada Urso que estende Animal.
- Criar uma classe abstrata chamada Veiculo [marca]
- Criar uma classe concreta chamada Carro que estende Veiculo e implementa Corredor.
- Criar uma classe chamada Doce [nome].
- Crie uma classe Principal. Nessa classe, crie um ArrayList. Insira na lista 1 urso, 1 carro e um doce.
- Crie um outro método estático que fará todos os objetos da lista correrem se contiverem esse método (correr()).
- Depois de tudo, colocar Urso como implementando Corredor.

