



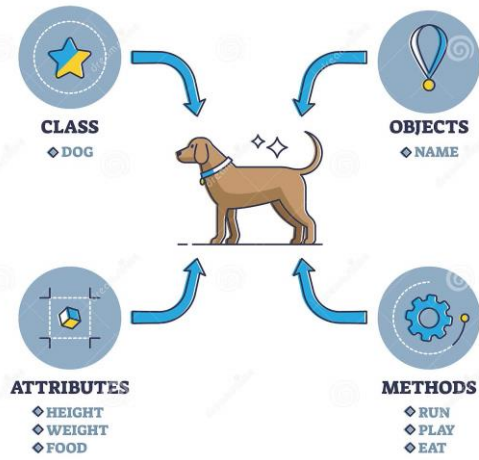
Ciência da **Computação**

Programação Orientada a Objetos
Prof. Luciano Rodrigo Ferretto



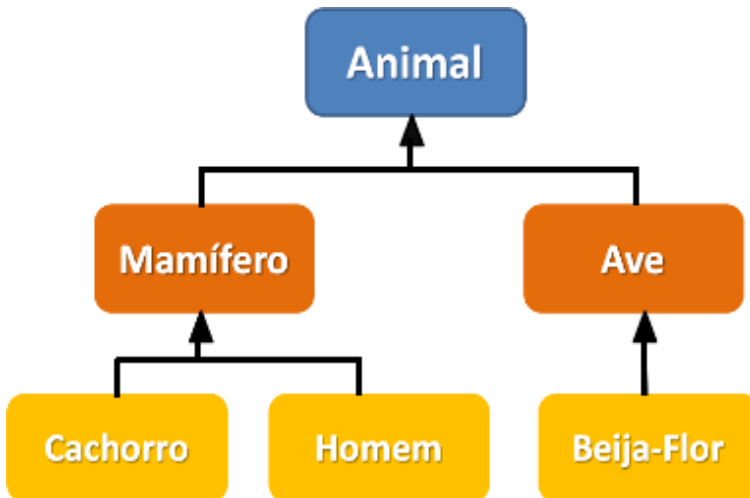
Vamos relembrar um pouco

OBJECT ORIENTED PROGRAMMING



dreamstime.com

ID 239724045 © VectorMine



arbabwaseer@gmail.com

Pilares da Programação Orientada a Objetos

- **Abstração:**

- É o processo de identificar e definir características essenciais de objetos do mundo real, representando essas características em forma de **classes** e **objetos**.

- **Encapsulamento:**

- É a prática de **ocultar os detalhes internos** de um objeto (classe), **expondo apenas as operações relevantes** para manipular esses detalhes. Isso promove a modularidade, a segurança e a manutenibilidade do código.

- **Herança:**

- É o princípio que permite que uma classe (**subclasse**) herde os atributos e métodos de outra classe (**superclasse**), facilitando a reutilização de código, a organização hierárquica e a promoção de relações entre objetos.

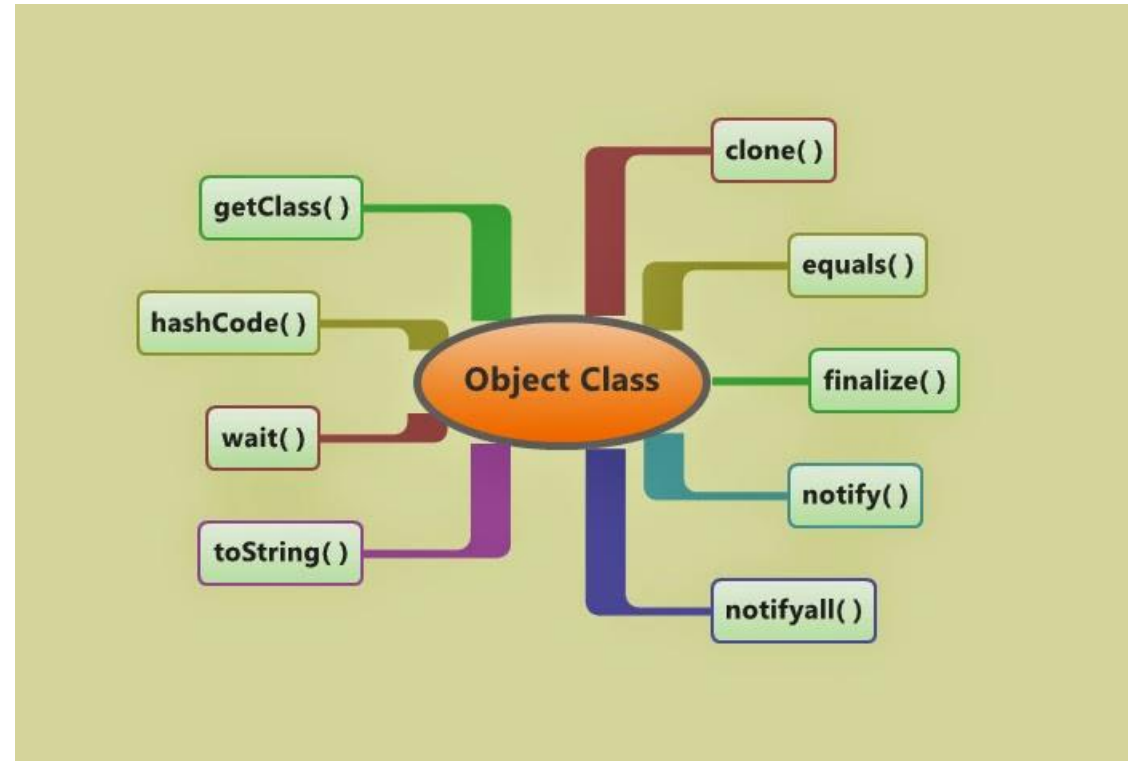
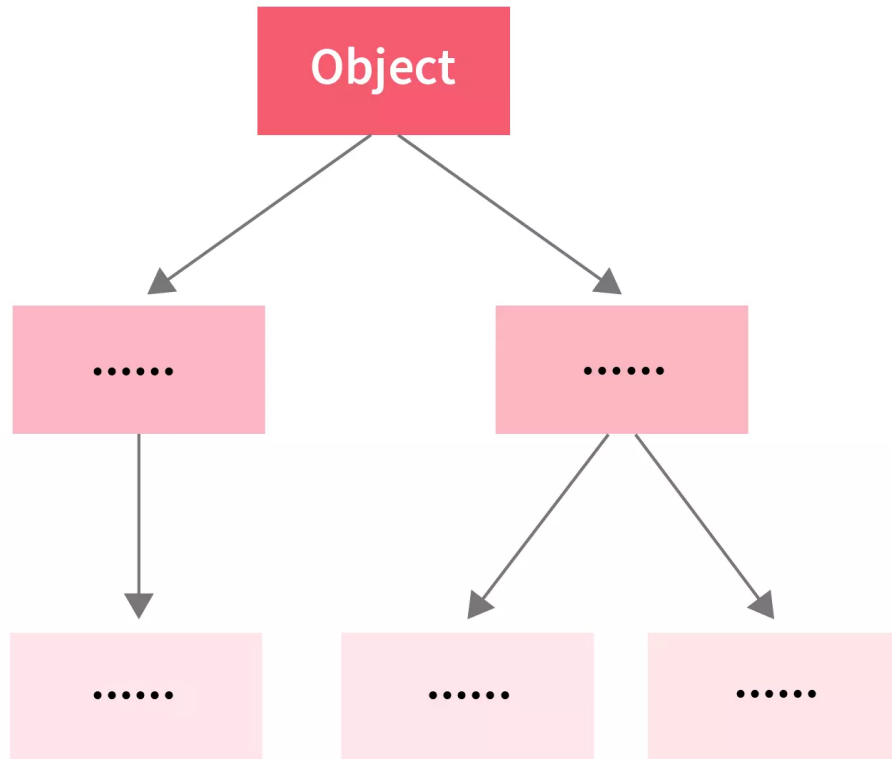
- **Polimorfismo:**

- Significa "muitas formas". O polimorfismo permite que objetos de diferentes classes sejam tratados como objetos de uma mesma classe porém com comportamentos diferentes. Isso facilita a criação de código mais flexível e extensível.



Ciência da
Computação

Object em Java



Abstract – Classes e métodos abstratos

- Uma classe abstrata **não pode ser instanciada, apenas herdada.**
- Sua principal função é servir como base para outras classes.
- Pode conter métodos abstratos e não abstratos.
- Métodos abstratos **não possuem implementação, e devem ser obrigatoriamente implementados (@override) nas subclasses.**
- Métodos abstratos são uma forma de garantir que todas as subclasses tenham um determinado comportamento definido

Abstract – Classes e métodos abstratos

```
1  ✓ public abstract class Animal {  
2      .... String nome;  
3  ✓  .... Animal(String nome) {  
4      .... | .... this.nome = nome;  
5      .... |  
6      .... }  
7      .... abstract void fazerSom();  
8  }
```

```
1  public class Cachorro extends Animal {  
2      .... Cachorro(String nome) {  
3          .... | .... super(nome);  
4          .... |  
5          .... @Override  
6          .... void fazerSom() {  
7              .... | .... System.out.println("Au au!");  
8              .... |  
9              .... }  
10     }
```

```
1  public class Gato extends Animal {  
2      .... Gato(String nome) {  
3          .... | .... super(nome);  
4          .... |  
5          .... @Override  
6          .... void fazerSom() {  
7              .... | .... System.out.println("Miauuuu!");  
8              .... |  
9              .... }  
10     }
```


Final – Classes, métodos e atributos finais

- Uma classe final **não pode ser herdada**.
- Um método final **não pode ser sobrescrito**.
- Um atributo (variável) final **não pode ter seu valor alterado**.

```
1  ✓ public final class ExemploFinalClasse {
2      | ... // Conteúdo da classe
3      | }
4
5      // A seguinte linha gera um erro, pois não é possível estender uma classe final
6      class SubClasse extends ExemploFinalClasse { }
```

```
1  public class ExemploFinalMetodo {
2      | ... public final void metodoFinal() {
3      |     | ... System.out.println("Este método é final e não pode ser sobrescrito.");
4      |     | }
5      | }
6      class subClasse extends ExemploFinalMetodo {
7      |     | ... //Essa sobrescrita gera um erro, pois o método da
8      |     | ... //super classe é final
9      |     | @Override
10     |     | public void metodoFinal() {
11     |     |     | ... System.out.println("Erro ao sobrescrever o método");
12     |     |     | }
13     |     | }
```

```
1  public class ExemploFinalVariavel {
2      | ... public static void main(String[] args) {
3      |     | ... final int constante = 10;
4      |     |     | constante = 20; // Isso gera um erro, pois a variável é final
5      |     |     | System.out.println(constante);
6      |     |     | }
7      | }
```

Casting em Java

- Java é uma linguagem **Fortemente Tipada**, ou seja, uma variável “nasce” com um tipo e permanece fiel a ele até o fim.
- Porém as vezes é preciso converter¹ o valor armazenado/referenciado por essa variável.
- Como dito, **não podemos alterar o tipo da variável**, mas podemos alterar o valor para um novo tipo, podendo armazenar em uma nova variável.
- Esse processo chamamos de **Casting**

1- Aqui leia-se “converter” ou “acessar” o valor de forma diferente

Casting em Tipos Primitivos

- **Cast Implícito (Widening Cast):** Esse tipo de casting ocorre de forma automática quando você está convertendo um tipo menor para um tipo maior.
- Neste casting **não** há perda de dados.

```
int numeroInteiro = 100;  
long numeroLong = numeroInteiro; // cast implícito de int para long  
double numeroDouble = numeroInteiro; // cast implícito de int para double  
double numeroDouble2 = numeroLong; // cast implícito de long para double
```

Casting em tipos primitivos

- **Cast Explícito (Narrowing Cast):** Esse tipo de casting ocorre quando você está convertendo um tipo maior para um tipo menor.
- Isso pode resultar em perda de dados, pois o tipo de destino pode não ser capaz de representar todos os valores do tipo de origem.
- Um casting explícito deve ser feito pelo programador usando parênteses e especificando o tipo desejado.

```
double numeroDoublePI = 3.14159265359;  
float numeroFloatPI = (float) numeroDoublePI; // cast explícito de double para float  
int numeroInteiroPI = (int) numeroDoublePI; // cast explícito de double para int  
System.out.println("PI em double: " + numeroDoublePI + "\nPI em float: " +  
    numeroFloatPI + "\nPI em int: " + numeroInteiroPI);
```

```
PI em double: 3.14159265359  
PI em float: 3.1415927  
PI em int: 3
```

Casting em Tipos Objeto

- **Cast Implícito (Upcast)**: Nesse caso, você pode fazer um casting de um objeto da subclasse para a superclasse (upcast) de forma implícita.
- Diferente dos tipos primitivos, neste caso **não** há perda de informações, porém, os métodos e atributos específicos da subclasse ficam inacessíveis.
- **Casting em Tipos Objeto não altera o Objeto em si, este continua sendo o mesmo, apenas cria uma nova variável que vai referenciá-lo.**

Casting em Tipos Objeto – Upcast-Implicito

```
Carro carro = new Carro("Chevrolet", "Opala 6 Caneco Diplomata", 1986, "AAA-9999", 2);  
//Upcast de Carro → Veiculo  
Veiculo veiculo = carro;  
//Upcast de um NOVO Objeto Carro → Veiculo  
Veiculo veiculo2 = new Carro("Ford", "Escort Xr3 Conversível", 1995, "CCC-7777", 2);
```

The method `getNumeroPortas()` is undefined for the type `Veiculo` Java(67108964)

[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

```
veiculo2.getNumeroPortas();  
carro.getNumeroPortas();
```

```
> carro: Carro@12 "Veículo: Chevrolet Opala 6 Caneco Diplomata 1986 - N. Portas: 2"  
> veiculo: Carro@12 "Veículo: Chevrolet Opala 6 Caneco Diplomata 1986 - N. Portas: 2"  
✓ veiculo2: Carro@24 "Veículo: Ford Escort Xr3 Conversível 1995 - N. Portas: 2"  
    MAX_TEMPO_USO: 30  
    ano: 1995  
> marca: "Ford"  
> modelo: "Escort Xr3 Conversível"  
    numeroPortas: 2  
> placa: "CCC-7777"
```

Casting em Tipos Objeto

- **Cast Explícito (Downcast):** Casting da superclasse para a subclasse (downcast) de forma explícita.
- Lembre-se: Casting em Tipos Objeto não altera o Objeto em si, este continua sendo o mesmo, apenas cria uma nova variável que vai referenciá-lo.
- O downcast deve ser feito com **cuidado** para evitar exceções em tempo de execução.
- Recomenda-se sempre fazer a verificação do tipo antes.

Casting em Tipos Objeto

```
List<Veiculo> veiculos = veiculoService.getVeiculosDB();
for (Veiculo veiculo : veiculos) {
    . . . veiculo.getNumeroPortas(); //Não posso, pois a variável é do tipo Veiculo (super classe),
    . . . // mesmo que o objeto referenciado possua este método
    . . . if (veiculo instanceof Carro) { //o operador "instanceof" verifica se o objeto referenciado
    . . . // pela variável é do tipo "Carro"
    . . .     Carro carro = (Carro) veiculo; //Downcast de Veiculo → Carro
    . . .     carro.getNumeroPortas(); //Aqui sim posso, pois a variável é do tipo Carro (sub classe)
    . . . }
}
```

Preparação para próxima Aula!!!

Antes de começarmos a falar de coisas novas, vamos falar sobre as novas ferramentas que iremos utilizar na próxima aula.

Java Avançado – Migrando de IDE



Eclipse IDE

ATENÇÃO: O aplicativo **NÃO** precisa ser instalado. Sugiro utilizar ele na modalidade Portable.

Eclipse é uma IDE para desenvolvimento Java, porém suporta várias outras linguagens a partir de plugins como C/C++, PHP, ColdFusion, Python, Scala e Kotlin. Ele foi feito em Java e segue o modelo open source de desenvolvimento de software.

Para nossas aulas vocês podem utilizar a última versão disponível (Neste momento: Eclipse-jee-2024-06-R).

Para download recomendo utilizarem o link abaixo, pois irá abrir a página conforme imagem a segui. Baixem a versão conforme seu sistema operacional do **Eclipse IDE for Enterprise Java and Web Developers**

Type Safety

Segurança de Tipos

Type Safety

- Type safety (ou segurança de tipos) é um conceito importante em programação orientada a objetos que se refere à garantia de que o tipo de um valor corresponde ao tipo esperado em um determinado contexto.
- Ele garante que as operações em variáveis, funções e objetos são feitas de maneira consistente com os tipos de dados definidos, prevenindo uma série de erros que podem ocorrer durante a execução do programa.
- A segurança de tipos é importante porque ajuda a prevenir erros de **tempo de execução** causados por operações inválidas ou conversões de tipo incorretas.
- Muitas linguagens de programação modernas, como Java, C# e Ruby, fornecem recursos avançados de segurança de tipos, como tipos genéricos, verificação de tipos em tempo de compilação e tratamento de exceções de tipo. Esses recursos ajudam a garantir que o código seja mais seguro, confiável e fácil de manter.

Type Safety - Dinâmico

Linguagem Fracamente Tipada

- Qual a função do operador “+” ???
- Qual o valor e o tipo de dados da variável “z” ????
- O código irá rodar sem erros ???

```
#PYTHON  
x = 10  
y = "20"  
z = x + y  
z2 = x / y
```

Type Safety – Estático

Linguagem Fortemente Tipada

- Em linguagens fortemente tipadas como o Java, fica muito claro para o desenvolvedor o que está acontecendo, facilitando assim a manutenibilidade do código, principalmente em grandes times de programadores.

```
//JAVA  
int x = 10;  
String y = "20";  
int z = x + y;  
int z2 = x / y;
```

Type mismatch: cannot convert from String to int Java(16777233)

[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

Type Safety – Estático x Dinâmico

Aspecto	Segurança de Tipos Estática	Segurança de Tipos Dinâmica
Verificação	Em tempo de compilação	Em tempo de execução
Detecção de Erros	Erros detectados antes da execução	Erros detectados durante a execução
Desempenho	Melhor desempenho em tempo de execução	Potencial sobrecarga em tempo de execução
Flexibilidade	Menos flexível	Mais flexível
Segurança	Maior segurança de tipos	Menor segurança de tipos

Type Safety - Estático

- **Vantagens da Segurança de Tipos Estática**

- 1. Prevenção de Erros:** A segurança de tipos ajuda a prevenir muitos erros comuns de programação. Por exemplo, se uma função espera um número inteiro, a linguagem não permitirá que você passe uma string para essa função.
- 2. Melhoria na Legibilidade e Manutenção do Código:** Quando o tipo de cada variável é conhecido e consistente, o código se torna mais fácil de ler e entender. Isso facilita a manutenção do código, pois os desenvolvedores podem prever o comportamento das variáveis.
- 3. Detecção Precoce de Erros:** Em linguagens que são verificadas estaticamente, como Java, muitos erros de tipo são detectados em tempo de compilação, antes que o código seja executado. Isso ajuda a reduzir bugs em tempo de execução.
- 4. Facilitação de Ferramentas de Desenvolvimento:** Ferramentas como IDEs (Ambientes de Desenvolvimento Integrado) podem fornecer melhores funcionalidades, como autocompletar, navegação de código e refatoração, quando têm informações claras sobre os tipos de dados.

Type Safety – Estático – Casting Seguro (explícito)

- Em linguagens fortemente tipadas como Java, é comum ser necessário utilizar o casting explícito para converter um tipo de objeto em outro.
- Essa prática serve como um lembrete importante de que devemos ser cuidadosos ao realizar tais conversões. Embora o compilador permita essas operações, é responsabilidade do desenvolvedor garantir que a conversão seja segura.

```
String str = "Texto";  
Object obj = str;  
String str2 = (String) obj;  
Integer inteiro = (Integer) obj;
```

```
java.lang.ClassCastException: class java.lang.String cannot be cast to class java.lang.Integer
```

Type Safety – Estático – Tipos Genéricos

- Os genéricos em Java fornecem segurança de tipos em coleções e outras estruturas de dados, permitindo que você defina o tipo de elementos que uma coleção pode conter.

Mas o que “diab...” são esses tipos genéricos?

Tipos Genéricos

Vamos imaginar um pouco ...

- Imagine que você possui várias classes, entre elas, você tem uma classe para “Frutas” e outra para “Gatinhos”.

```
public class Fruta {  
    private String nome;  
    private boolean cortada;
```

```
public class Gato {  
    private String nome;  
    private boolean dormindo;
```



Agora você quer criar uma classe que possa armazenar tanto “Frutas” quanto “Gatos”

Detalhes:

- Aqui não vamos usar o “List” nativo do Java.
- Queremos que uma mesma classe lide tanto com objetos do tipo “Fruta” quanto do tipo “Gato”.

Qual o tipo de variável que pode apontar para qualquer outro tipo de objetos no Java ????



Nesse caso você poderia criar uma classe para administrar um **Array** do tipo **Object**.

```
1  public class Cesta {
2      private Object[] itens;
3      private int tamanho;
4
5      // Construtor para inicializar a cesta com um tamanho fixo
6  > public Cesta(int capacidade) { ...
10
11     // Método para inserir um objeto na cesta
12  > public void inserir(Object o) { ...
19
20     // Método para obter o último objeto adicionado na cesta e removê-lo (LIFO)
21  > public Object getProximo() { ...
31
32     // Método para verificar se existem itens na cesta
33  > public boolean existeItens() { ...
36
37 }
```

Imagine agora que você encheu essa cesta com frutas ...



```
public static void main(String[] args) {  
    Cesta cestaFrutas = new Cesta(10);  
    cestaFrutas.inserir(new Fruta("Maçã"));  
    cestaFrutas.inserir(new Fruta("Pera"));  
    cestaFrutas.inserir(new Fruta("laranja"));  
    fazerSaladaDeFrutas(cestaFrutas);  
}
```


... e então enviou a cesta para nossa máquina para fazer uma “deliciosa” salada de frutas.

Para a salada é importante você “cortar” as frutas.



```
public static Fruta[] fazerSaladaDeFrutas(Cesta cestaFrutas) {  
    Fruta[] salada = new Fruta[20];  
    int i = 0;  
    while (cestaFrutas.exiteItens()) {  
        Fruta fruta = (Fruta) cestaFrutas.getProximo();  
        fruta.cortar();  
        salada[i++] = fruta;  
    }  
    return salada;  
}
```



Mas se nessa cesta você adicionar outro tipo de objeto, um gatinho, por exemplo ...

```
Cesta cestaFrutas = new Cesta(10);  
cestaFrutas.inserir(new Fruta("Maçã"));  
cestaFrutas.inserir(new Fruta("Pera"));  
cestaFrutas.inserir(new Fruta("laranja"));  
cestaFrutas.inserir(new Gato("Mimi"));
```



... ou então temos uma cesta cheia de “lindos gatinhos”...



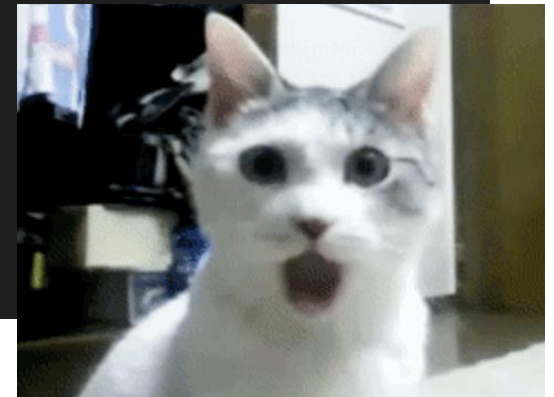
```
Cesta cestaGatinhos = new Cesta(10);  
cestaGatinhos.inserir(new Gato("Miuki"));  
cestaGatinhos.inserir(new Gato("Mimi"));
```

... e você, por engando ou não (kkkk), acabou enviando a cesta com esses “lindos gatinhos” para nossa máquina de fazer saladas de frutas

```
fazerSaladaDeFrutas(cestaGatinhos);
```



```
public static Fruta[] fazerSaladaDeFrutas(Cesta cestaFrutas) {  
    Fruta[] salada = new Fruta[20];  
    int i = 0;  
    while (cestaFrutas.exiteItens()) {  
        Fruta fruta = (Fruta) cestaFrutas.getProximo();  
        fruta.cortar();  
        salada[i++] = fruta;  
    }  
    return salada;  
}
```





Para sorte dos “gatinhos” e decepção da nossa “*Rainha do Rebolado*”, a execução não chegará no “Cortar”

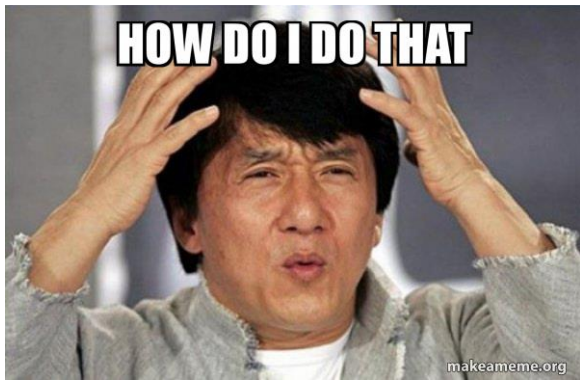


```
Fruta fruta = (Fruta) cestaFrutas.getProximo();
```

```
Exception in thread "main" java.lang.ClassCastException:  
class Gato cannot be cast to class Fruta (Gato and Fruta
```



Erro em tempo
de **Execução**



Certo, como evitamos esse erro ?

- Podemos partir para uma solução “caseira”, mas, como diria Sandra Annenberg, “... que deselegante ...”

```
Object item = cestaFrutas.getProximo();  
if (item instanceof Fruta) {  
    Fruta fruta = (Fruta) item;  
    fruta.cortar();  
    salada[i++] = fruta;  
}
```



Para isso temos os maravilhosos

Tipos Genéricos



- Tipos genéricos em POO (Programação Orientada a Objetos) é um recurso que permite escrever classes, interfaces e métodos que são **independentes do tipo de dado específico** que será usado como parâmetro.
- Em outras palavras, em vez de escrever classes e métodos para lidar com tipos de dados específicos (como uma classe para lidar com inteiros, outra para lidar com strings, etc.), os tipos genéricos permitem escrever classes e métodos que podem funcionar com qualquer tipo de dados que seja especificado no momento da utilização.



Tipos Genéricos

- O uso de tipos genéricos em POO traz vários benefícios, como o aumento da reutilização de código, a maior segurança e a facilidade de manutenção, uma vez que os erros de tipos são detectados em tempo de compilação.
- Um exemplo prático em Java de uso de tipos genéricos é a classe **ArrayList** que implementa a interface **List**, que é uma lista dinâmica que pode armazenar objetos de qualquer tipo.
- Por exemplo, podemos criar um **ArrayList** de inteiros usando **ArrayList<Integer>**, ou um **ArrayList** de strings usando **ArrayList<String>**. O tipo de dados é especificado entre os colchetes angulares (< >) que seguem o nome da classe.

O Tipo da classe é passado via parâmetro e é declarado utilizando os símbolos `<>`

```
public class Cesta<E> {  
    private E[] itens;  
    private int tamanho;  
    // Construtor para inicializar a cesta com um tamanho fixo  
    public Cesta(int capacidade) { ...  
    // Método para inserir um objeto na cesta  
    public void inserir(E o) { ...  
    // Método para obter o último objeto adicionado na cesta e removê-lo (LIFO)  
    public E getProximo() { ...  
    // Método para verificar se existem itens na cesta  
    public boolean existeItens() { ...  
}
```

O Tipo da classe pode ser usado para definir o tipo de parâmetros e de retorno.

Convenções e Significado das Letras Maiúsculas para parâmetros de tipo genéricos

- **Clareza e Brevidade:** Usar uma única letra maiúscula torna o código mais conciso e fácil de ler, especialmente em definições genéricas onde os nomes de tipos podem aparecer com frequência. Isso ajuda a distinguir rapidamente os parâmetros de tipo dos nomes de classes e variáveis regulares.

Significado Convencional:

- **T: Tipo genérico (Type).** É o parâmetro de tipo genérico mais comum e representa qualquer tipo de objeto.
- **E: Elemento (Element).** Usado comumente em coleções como listas e conjuntos (List<E>, Set<E>).
- **K: Chave (Key).** Usado em pares chave-valor, como em mapas (Map<K, V>).
- **V: Valor (Value).** Usado em pares chave-valor, como em mapas (Map<K, V>).
- **N: Número (Number).** Usado para indicar que o tipo é numérico.
- **R: Resultado (Result).** Usado em contextos onde um método retorna um resultado de um tipo genérico.



```
Cesta<Fruta> cestaFrutas = new Cesta<Fruta>(10);  
cestaFrutas.inserir(new Fruta("Maçã"));  
cestaFrutas.inserir(new Fruta("Pera"));  
cestaFrutas.inserir(new Fruta("laranja"));  
cestaFrutas.inserir(new Gato("Mimi"));
```

The method `inserir(Fruta)` in the type `Cesta<Fruta>` is not applicable for the arguments `(Gato)` Java(67108979)

```
void Cesta.inserir(Fruta o)
```

[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)



```
Cesta<Gato> cestaGatinhos = new Cesta<Gato>(10);  
cestaGatinhos.inserir(new Gato("Miuki"));  
cestaGatinhos.inserir(new Gato("Mimi"));
```

Agora podemos garantir que nossa máquina de fazer salada de frutas só seja alimentado com “frutas”

```
public static Fruta[] fazerSaladaDeFrutas(Cesta<Fruta> cestaFrutas) {  
    Fruta[] salada = new Fruta[20];  
    int i = 0;  
    while (cestaFrutas.exiteItens()) {  
        Fruta fruta = (Fruta) cestaFrutas.getProximo();  
        fruta.cortar();  
        salada[i++] = fruta;  
    }  
    return salada;  
}
```

E não há mais a necessidade de *Casting*, pois temos certeza que a cesta só virá com “frutas”

Desta forma temos Type Safety funcionando em tempo de compilação

```
Cesta<Gato> cestaGatinhos = new Cesta<Gato>(10);  
cestaGatinhos.inserir(new Gato("Miuki"));  
cestaGatinhos.inserir(new Gato("Mimi"));  
fazerSaladaDeFrutas(cestaGatinhos);
```

```
The method fazerSaladaDeFrutas(Cesta<Fruta>) in the type Main  
is not applicable for the arguments  
(Cesta<Gato>) Java(67108979)
```

```
Fruta[] Main.fazerSaladaDeFrutas(Cesta<Fruta> cestaFrutas)
```

[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)



Erro em tempo
de **Compilação**



Diamond Syntax

- A sintaxe diamond, também conhecida como operador diamante ou diamante vazio, é um recurso adicionado ao Java 7 que permite que o compilador infira o tipo de um objeto em uma declaração de um objeto genérico.
- Antes da introdução da sintaxe diamond, a declaração de um objeto genérico exigia que o tipo de objeto fosse especificado duas vezes - uma vez na declaração da variável e novamente na criação do objeto. Por exemplo, para criar um ArrayList que armazena objetos do tipo String, era necessário declará-lo assim:

```
Cesta<Fruta> cestaFrutas = new Cesta<Fruta>();
```

- Com a sintaxe diamond, no entanto, é possível omitir a especificação do tipo de objeto na criação do objeto, deixando que o compilador infira o tipo com base no tipo especificado na declaração da variável. O exemplo acima poderia ser reescrito assim:

```
Cesta<Fruta> cestaFrutas = new Cesta<>();
```

Bounded Type Parameters (Parâmetros de Tipo Limitados)

- Você pode restringir os tipos que podem ser utilizados como parâmetros de tipo usando a sintaxe **T extends**.
- Isso define uma "limitação superior" (upper bound) que especifica que o tipo deve ser uma subclasse de um determinado tipo ou implementar uma interface específica.

```
public class Caixa<T extends Number> {  
    private T valor;  
  
    public void setValor(T valor) {  
        this.valor = valor;  
    }  
  
    public T getValor() {  
        return valor;  
    }  
  
    public double doubleValue() {  
        return valor.doubleValue();  
    }  
}
```

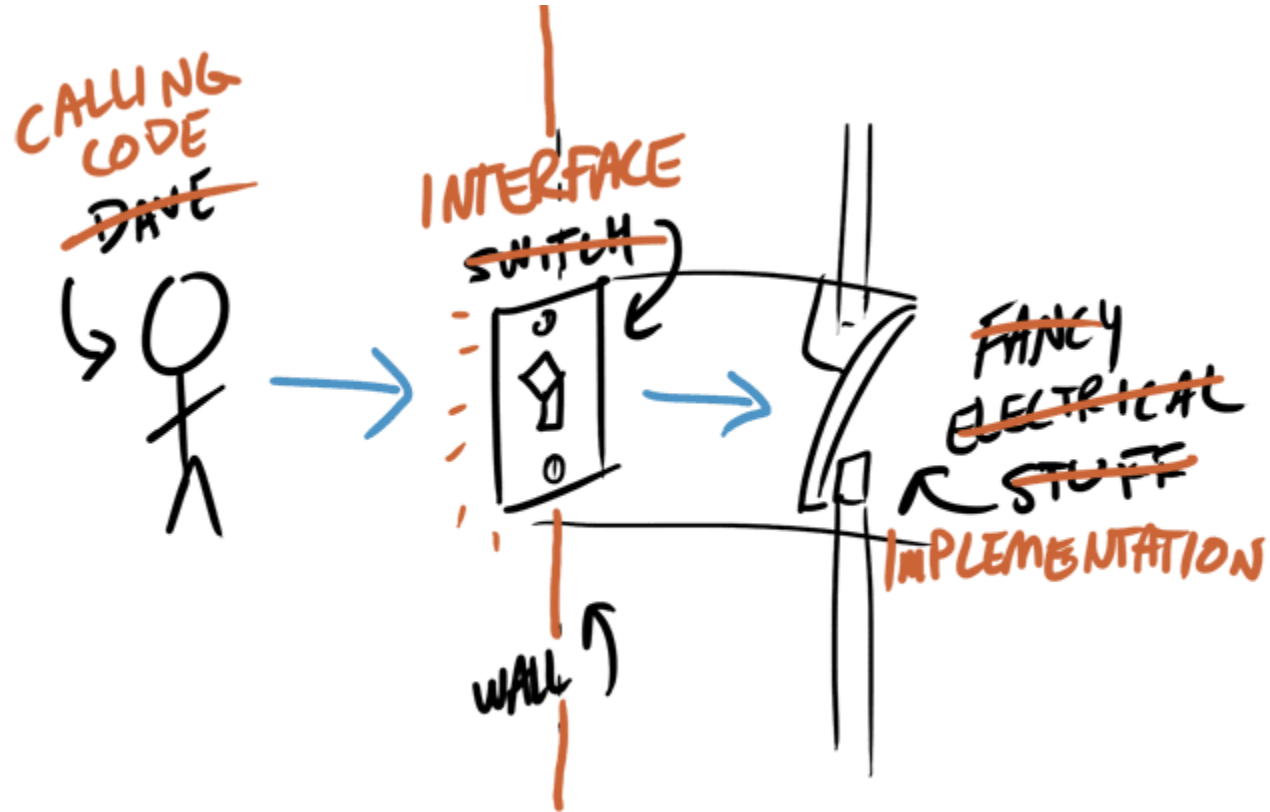
```
public static void main(String[] args) {  
    Caixa<Integer> caixaInt = new Caixa<>();  
    caixaInt.setValor(10);  
    System.out.println("Valor: " + caixaInt.getValor());  
    System.out.println("Valor em double: " + caixaInt.doubleValue());  
  
    Caixa<Double> caixaDouble = new Caixa<>();  
    caixaDouble.setValor(5.5);  
    System.out.println("Valor: " + caixaDouble.getValor());  
    System.out.println("Valor em double: " + caixaDouble.doubleValue());  
}
```


Wildcards (Coringas)

- Os coringas (wildcards) são representados pelo caractere ? e são usados para flexibilizar os parâmetros de tipo em métodos, classes e interfaces.

```
public static void imprimirNumeros(List<? extends Number> lista) {  
    for (Number num : lista) {  
        System.out.print(num + " ");  
    }  
    System.out.println();  
}  
  
public static void main(String[] args) {  
    List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);  
    List<Double> doubleList = Arrays.asList(1.1, 2.2, 3.3);  
  
    imprimirNumeros(intList); // Saída: 1 2 3 4 5  
    imprimirNumeros(doubleList); // Saída: 1.1 2.2 3.3  
}
```

Interfaces em POO



Uma visão geral de Interfaces

- Interfaces são um conceito da programação orientada a objetos que tem a ver com o **comportamento esperado** para uma ou um conjunto de classes.
- Interfaces definem o que uma classe **deve fazer e não como**. Assim, interfaces nem sempre possuem a implementação de métodos pois podem apenas declarar um conjunto de métodos, ou seja, o comportamento que uma ou um conjunto de classes deve ter.

Uma visão geral de Interfaces

- **Múltipla herança:** Em Java, uma classe só pode herdar de uma única classe pai (herança única), o que pode ser limitante em alguns casos. No entanto, uma classe pode implementar várias interfaces, o que permite que ela herde comportamentos de várias fontes diferentes. Isso é útil quando você deseja que uma classe tenha funcionalidades de várias fontes sem herdar toda a implementação de várias classes.
- **Contratos:** Interfaces definem contratos que as classes que as implementam devem cumprir. Isso é útil quando você deseja garantir que várias classes tenham um conjunto consistente de métodos, independentemente de sua hierarquia de herança. Isso promove a consistência e a padronização em seu código.

Uma visão geral de Interfaces

- **Separação de preocupações:** Interfaces permitem uma separação mais clara de preocupações no seu código. Enquanto uma classe abstrata geralmente fornece alguma implementação inicial e, possivelmente, compartilha estado entre subclasses, uma interface define apenas a estrutura dos métodos que uma classe deve implementar. Isso ajuda a evitar acoplamento excessivo e permite que as classes implementem apenas o que é necessário.
- **Composição sobre herança:** A preferência por composição em vez de herança direta é uma prática recomendada em design de software. Interfaces se alinham melhor com a composição, pois uma classe pode implementar várias interfaces e, assim, "compor" diferentes comportamentos, sem herdar uma grande quantidade de código de várias classes base.

Coleções em Java – API

Collections

Coleções em Java – API Collections

- Collections em Java se referem a um conjunto de classes e interfaces que fornecem uma estrutura para armazenar e manipular dados em grupo. As collections são parte do framework de coleções de Java e foram projetadas para serem flexíveis, eficientes e seguras para threads. O framework de coleções em Java inclui as seguintes interfaces principais:
 - **Iterable:** é a interface raiz de maioria das interfaces de coleções em Java.
 - **Collection:** Define métodos básicos para adicionar, remover e recuperar elementos de uma coleção.
 - **List:** uma coleção ordenada que permite elementos duplicados.
 - **Set:** uma coleção não ordenada que não permite elementos duplicados.
 - **Map:** uma coleção que armazena pares de chave-valor únicos. Cada elemento é acessado por meio de sua chave exclusiva.

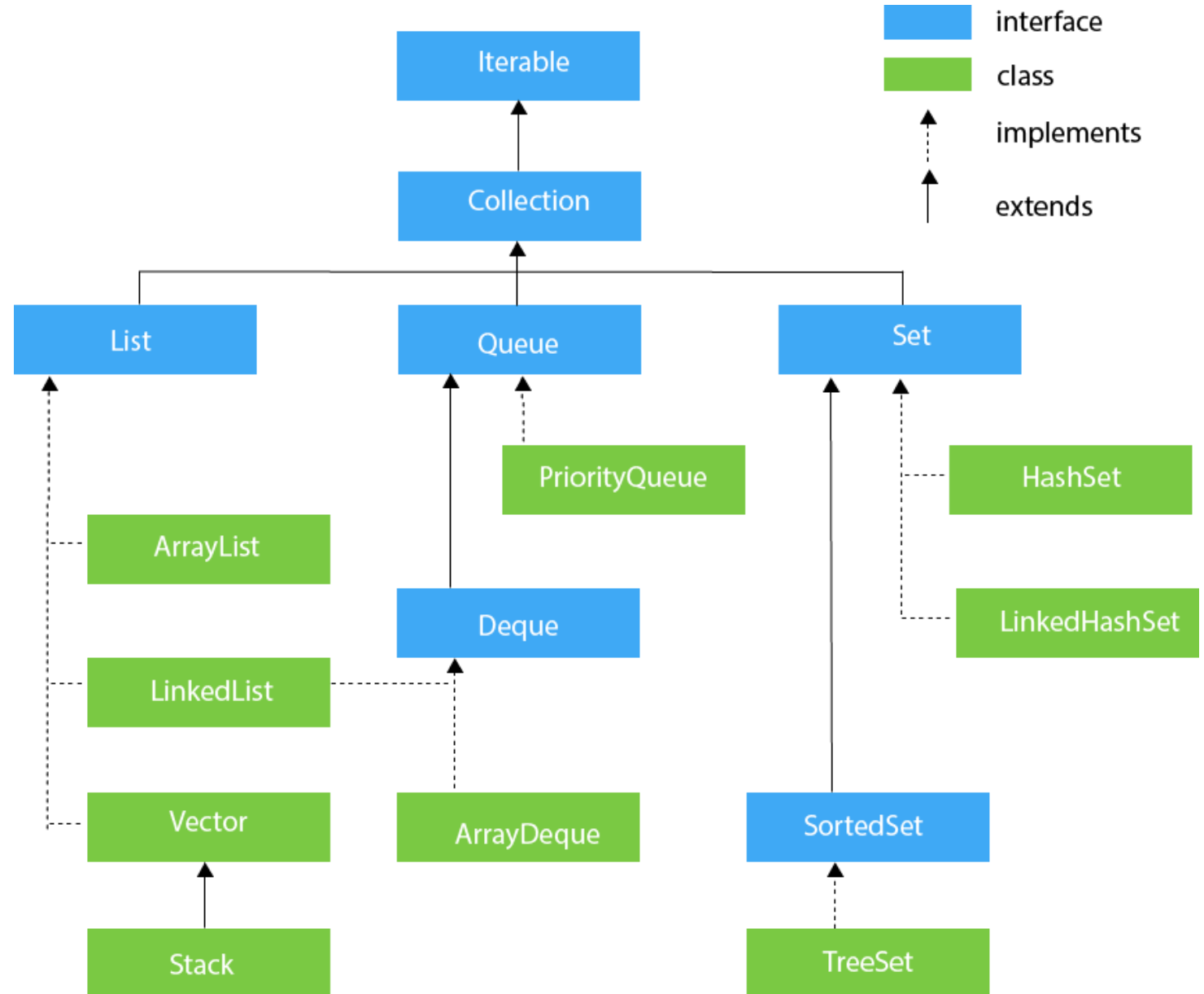
Coleções em Java – API Collections

- Além das interfaces acima, o framework de coleções em Java também inclui várias classes úteis para trabalhar com coleções, como **ArrayList**, **LinkedList**, **HashSet**, **TreeSet** e **HashMap**. Essas classes fornecem implementações concretas das interfaces acima e adicionam recursos adicionais, como iteração, ordenação e pesquisa.
- Com a introdução dos tipos genéricos em Java, o framework de coleções foi atualizado para permitir a criação de coleções com tipos de dados específicos, garantindo maior segurança em tempo de compilação e reduzindo o risco de erros em tempo de execução. O framework de coleções em Java é amplamente utilizado em muitos aplicativos Java, desde aplicativos de desktop até aplicativos da web e móveis.

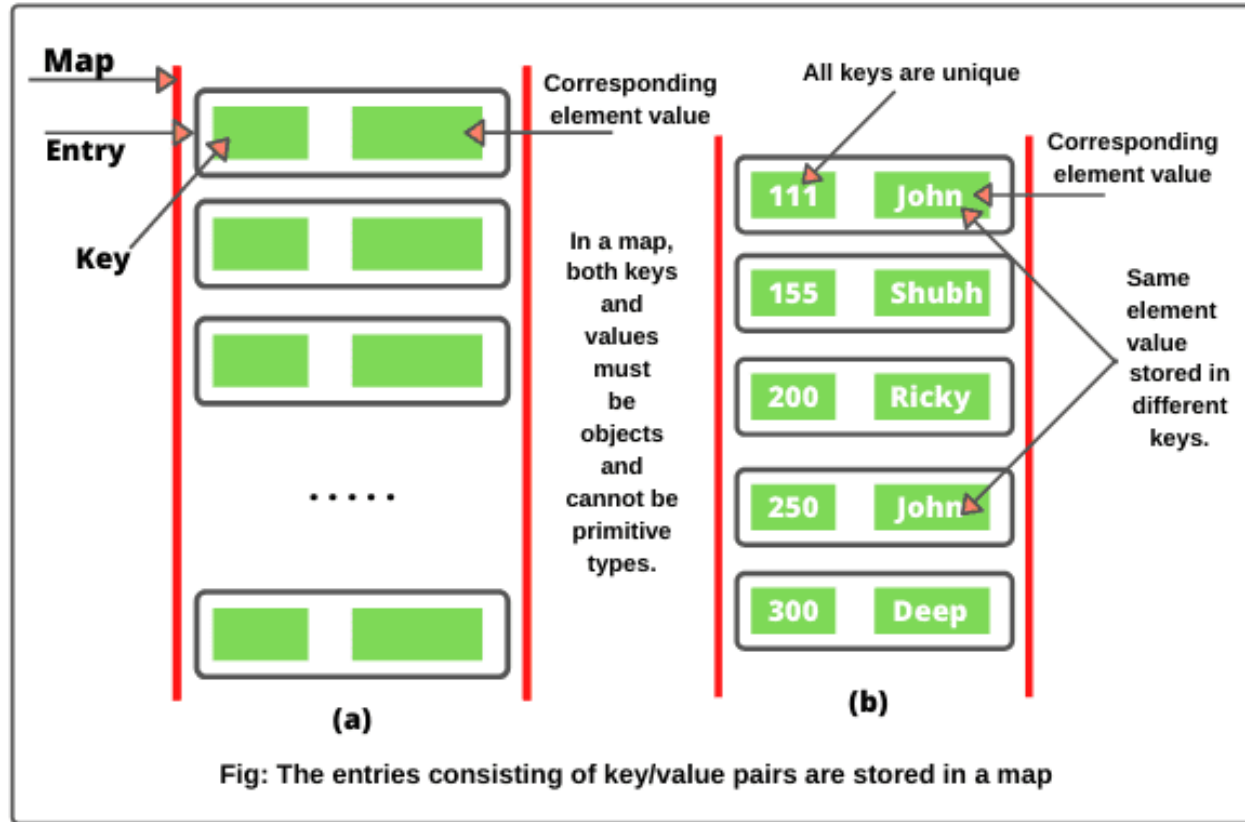
```
public interface List {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator iterator();
    Object[] toArray();
    boolean add(Object o);
    boolean remove(Object o);
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    Object get(int index);
    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator listIterator();
    ListIterator listIterator(int index);
    List subList(int fromIndex, int toIndex);
}
```

```
public interface List<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
}
```

- **List:** Sequência;
Ordenado
- **Set:** Não contém
elementos duplicados
- **Queue:** Provê
operações de fila.



Map Interface



.....→ implements
——→ extends

