Ciência da Computação

Programação Orientada a Objetos

Prof. Luciano Rodrigo Ferretto

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 é 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 - Dinâmico
Linguagem Fracamente Tipada

- E agora???

```
1  class Pessoa:
2      def __init__(self, ano_nascimento):
3          self.ano_nascimento = ano_nascimento
4
5  ano_nascimento = input('Digite o Ano de Nascimento: ')
6  p = Pessoa(ano_nascimento)
7
8  print("A idade é: ", 2024 - p.ano_nascimento)
```

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
//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”.

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

```java
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**.

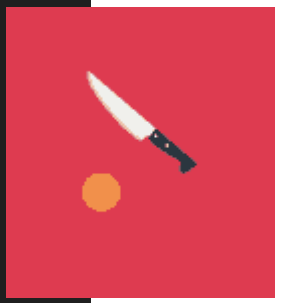
```
1   public class Cesta {
2       private Object[] itens;
3       private int tamanho;
4
5       // Construtor para inicializar a cesta com um tamanho fixo
6   >   public Cesta(int capacidade) {···
10
11      // Método para inserir um objeto na cesta
12  >   public void inserir(Object o) {···
19
20      // 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 exiteItens() {···
36
37  }
```

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

```java
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"));
```

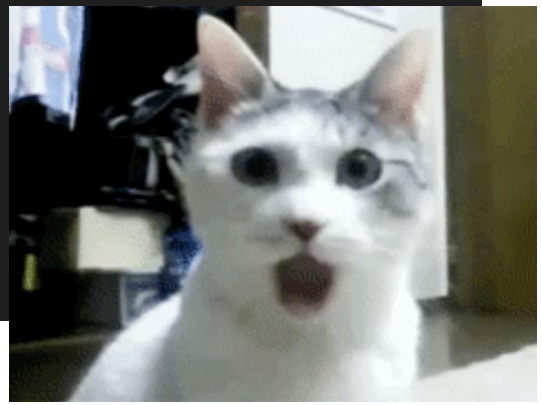
... ou então temos uma cesta cheia de "lindos gatinhos"...

```
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;
}
```

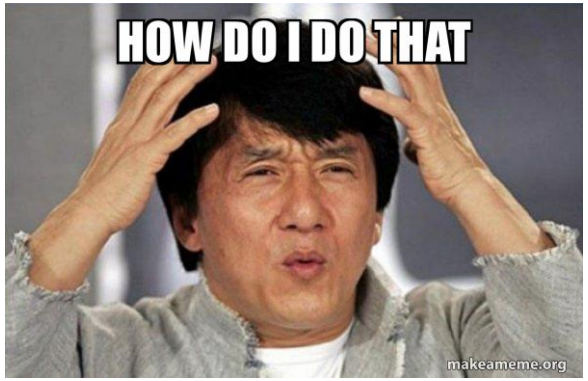


Para sorte dos “gatinhos” e decepção da nossa *“Rainha do Rebolado”*, a execução não chegará no “Cortar”

```
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 <>

```java
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.

```java
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”**

```java
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 Sintax

- 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.

```java
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();
    }
```

```java
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)

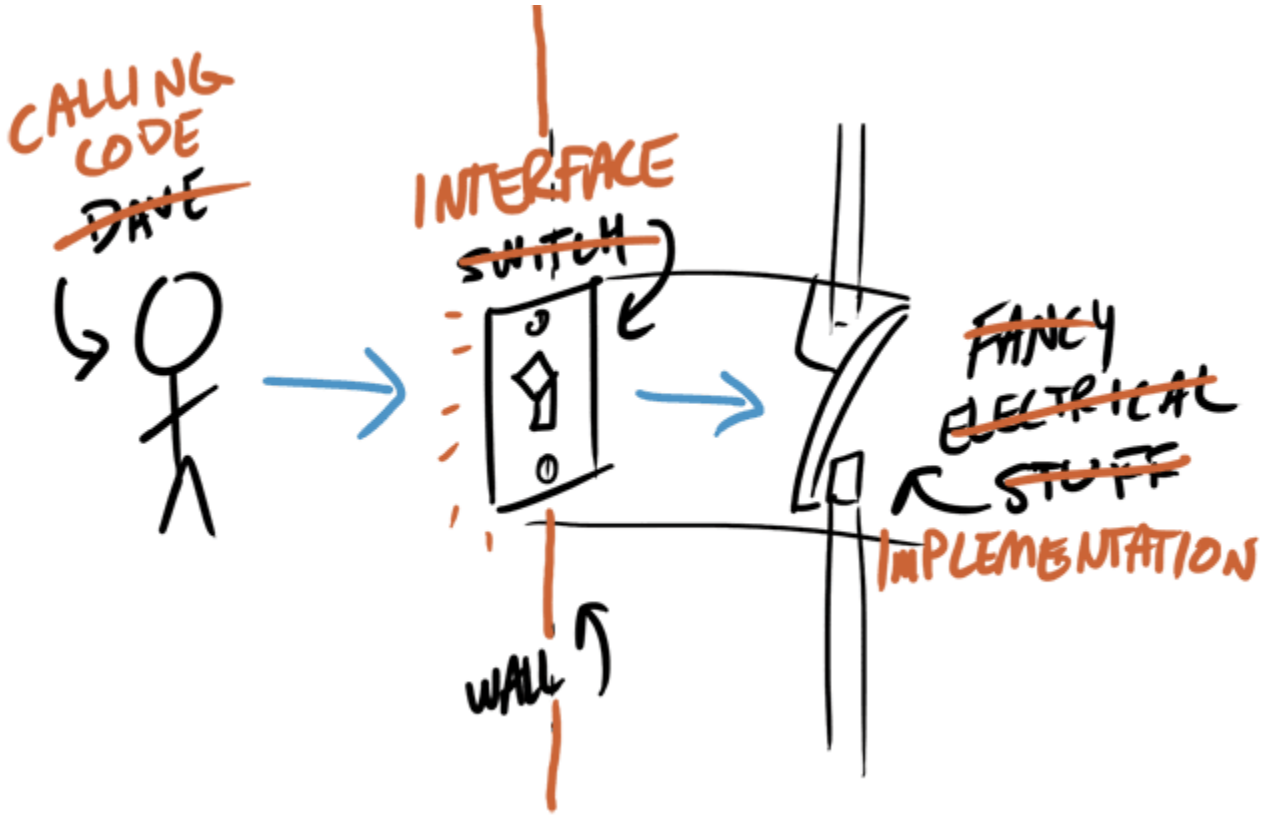
- Os coringas (wildcards) são representados pelo caractere **?** e são usados para flexibilizar os parâmetros de tipo em métodos, classes e interfaces.

```java
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:

  - **Itarable:** 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

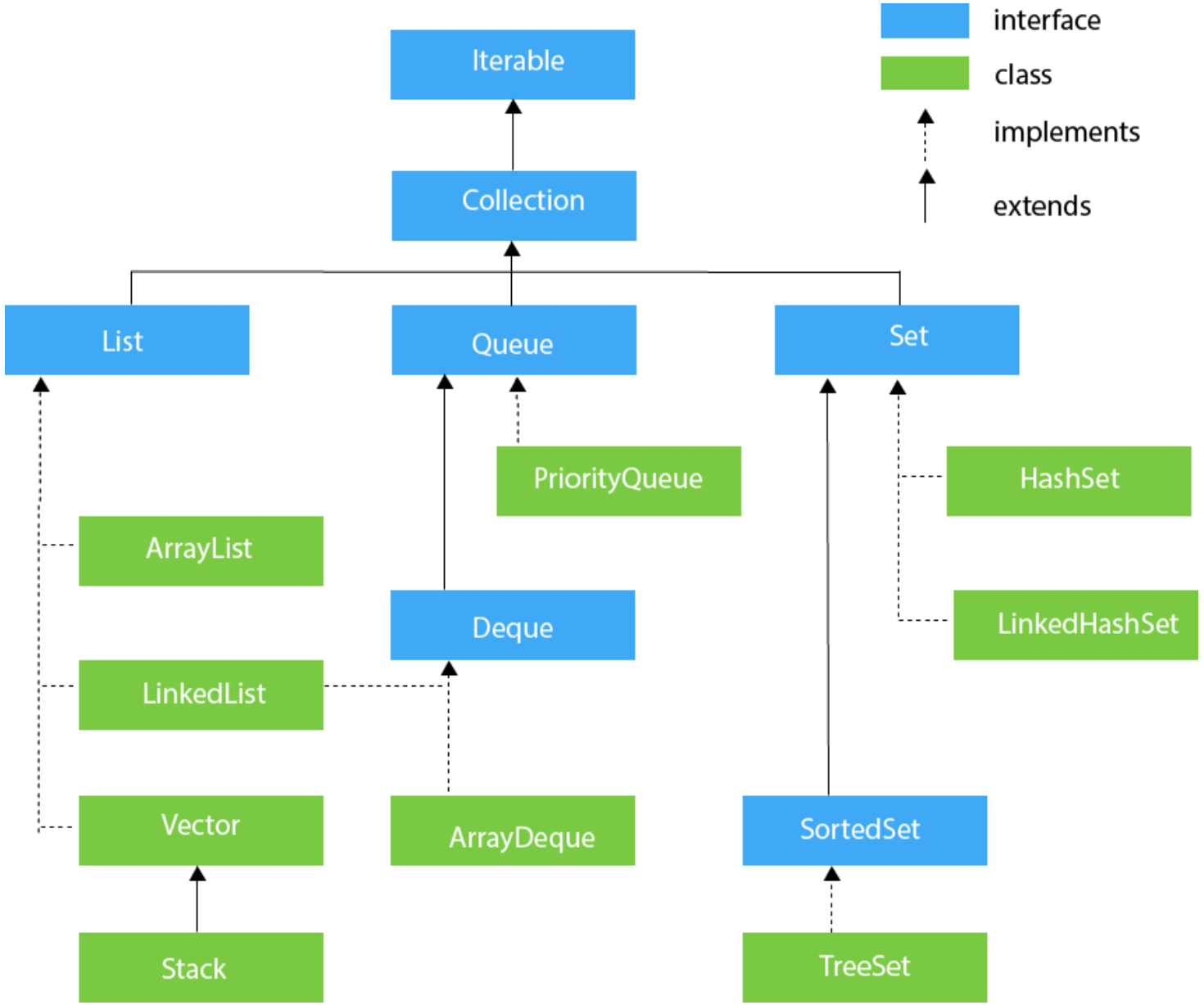
- Além das interfaces acima, o framework de coleções em Java também inclui várias classes úteis para trabalhar com coleções, como **ArrayList**, **LinkedList**, **HashSet**, **TreeSet** e **HashMap**. Essas classes fornecem implementações concretas das interfaces acima e adicionam recursos adicionais, 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.

```java
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);
}
```

```java
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.



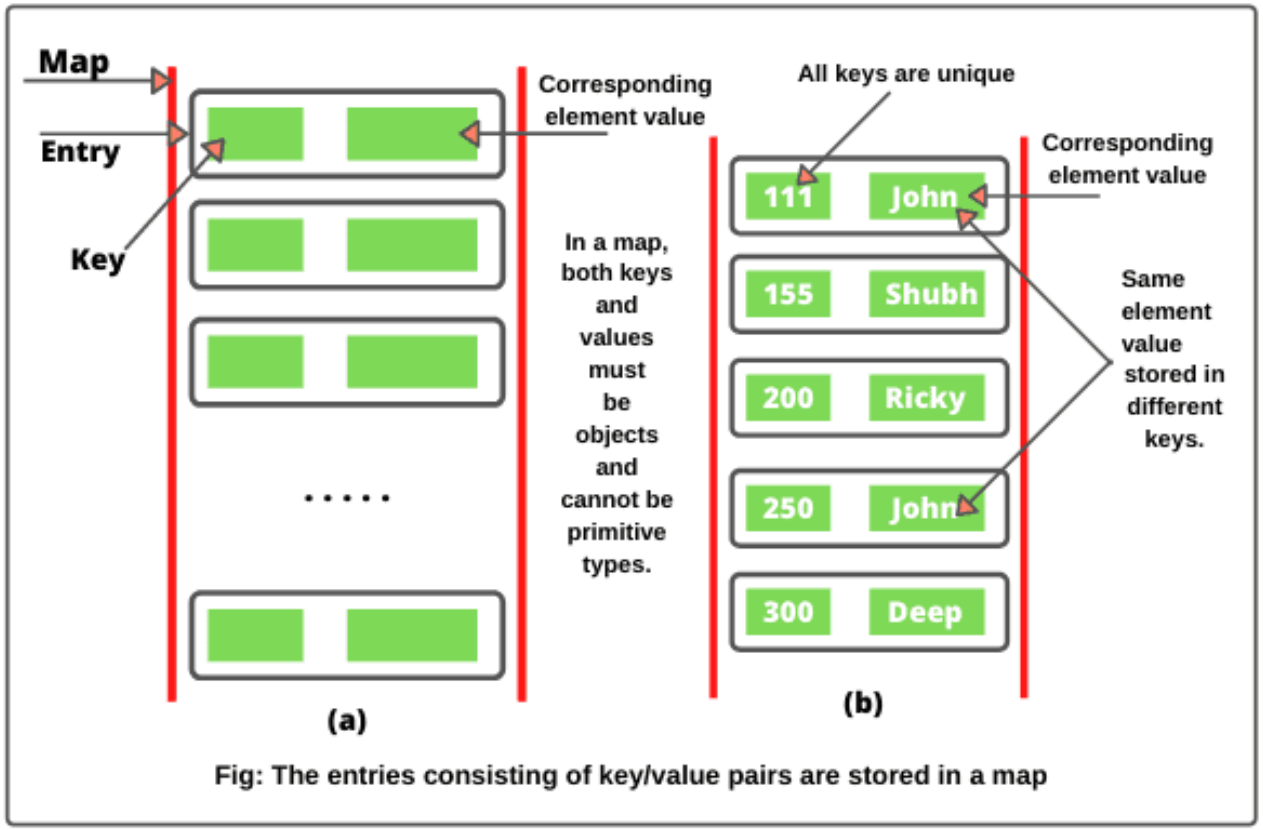
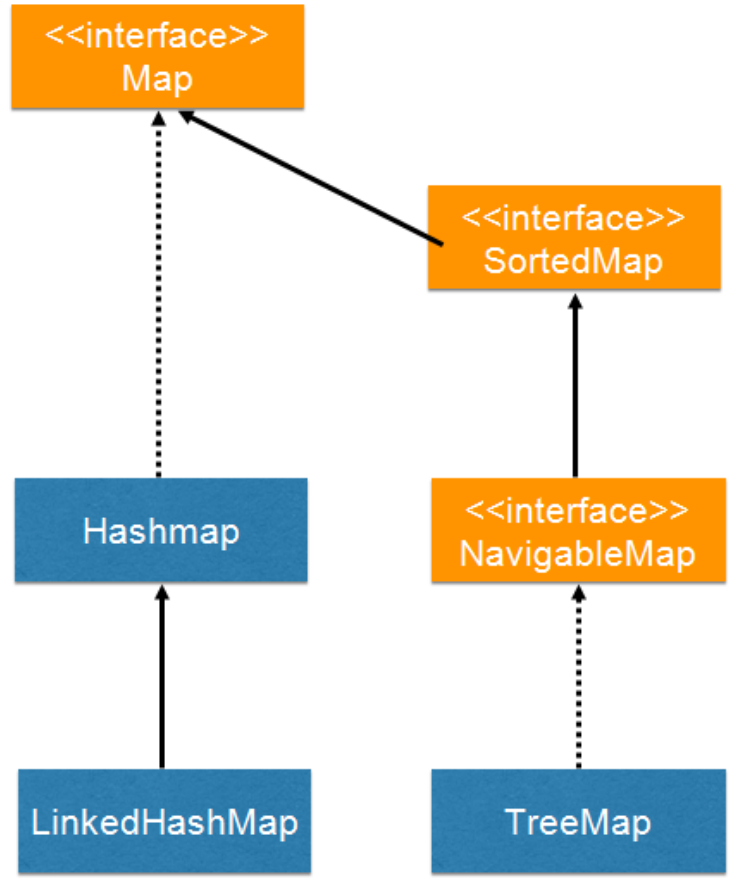


Fig: The entries consisting of key/value pairs are stored in a map

Map Interface



implements

extends