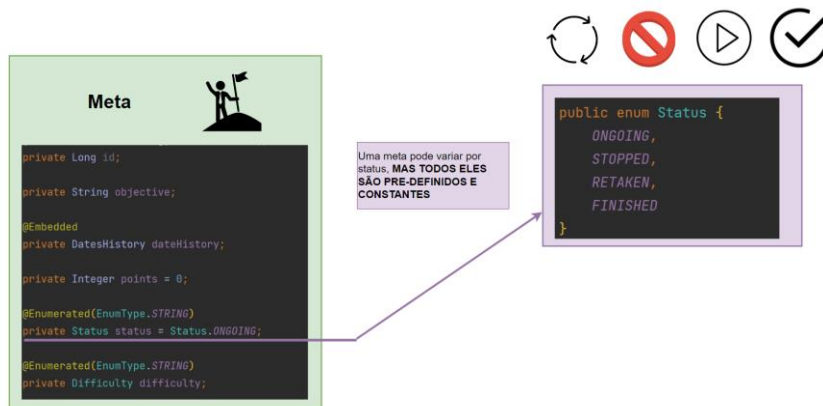




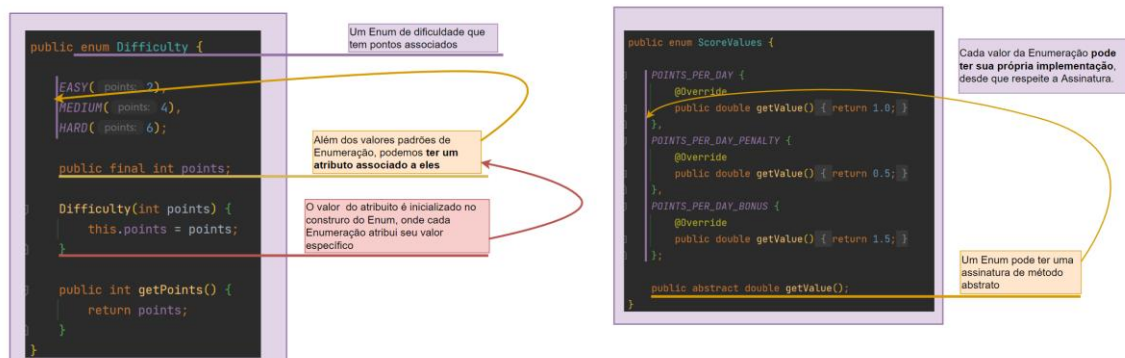
Enums – Java

Enums são um tipo de classe que representa **valores CONSTANTES** (que não mudam). São interessantes de se utilizar quando você tem uma quantidade de possibilidades pré-definidas para se trabalhar.

Um uso comum do Enum é **representar o estado** de um determinado Objeto:



Embora o **Enum** possa representar estados, ele também é capaz de retornar valores específicos e ter comportamentos:



Generics – Java

Generics trazem benefício de **type safety**. Por exemplo, se eu fizesse uma abstração **usando o tipo Object**, eu precisaria implicitamente **fazer o casting pro tipo específico** quando eu precisasse, isso

poderia lançar uma **RuntimeException**:

Uma caixa que pode guardar qualquer objeto

```

public class Caixa {
    private Object qualquerCoisa;

    public void setQualquerCoisa(Object qualquerCoisa) { this.qualquerCoisa = qualquerCoisa; }

    public Object getQualquerCoisa() { return qualquerCoisa; }
}

```

```

public static void main(String[] args) {
    Pessoa humano = new Pessoa( nome: "Lucas", idade: 23);
    Caixa caixa = new Caixa();
    caixa.setQualquerCoisa(humano);

    Pessoa pessoa = (Pessoa) caixa.getQualquerCoisa();
    Pessoa(nome="Lucas", idade=23)

    String coisaString = (String) caixa.getQualquerCoisa();
    Double coisaDouble = (Double) caixa.getQualquerCoisa();
}

```

Guardando o objeto de pessoa dentro da Caixa

O casting do objeto na Caixa para Pessoa funciona

Obviamente isso vai estourar uma exceção, mas em tempo de compilação nenhum erro é detectado

```

Exception in thread "main" java.lang.ClassCastException: Pessoa cannot be cast to class java.lang.String
at BRINCANDO.main(BRINCANDO.java:15)

```

Quando usamos Generics garantimos uma segurança de erro a nível de compilação, isso porque “resolvemos” o tipo específico do genérico quando instanciamos a classe:

Uma caixa que pode guardar qualquer objeto

```

public class Caixa<T> {
    private T qualquerCoisa;

    public void setQualquerCoisa(T qualquerCoisa) { this.qualquerCoisa = qualquerCoisa; }

    public T getQualquerCoisa() { return qualquerCoisa; }
}

```

o T poderia ser uma palavra, um, ou outra letra

A Caixa sabe o tipo de objeto que ela está lidando

```

public static void main(String[] args) {
    Pessoa humano = new Pessoa( nome: "Lucas", idade: 23);
    Caixa<Pessoa> caixa = new Caixa<>();
    caixa.setQualquerCoisa(humano);

    Pessoa pessoa = caixa.getQualquerCoisa();

    String coisaString = caixa.getQualquerCoisa();
    Double coisaDouble = (Double) caixa.getQualquerCoisa();
}

```

Guardando o objeto de pessoa dentro da Caixa

Em tempo de compilação a Caixa consegue resolver o tipo que está guardado nela, indicando erro mesmo com tentativa de casting

Generics Delimitados – Java

Os **Generics** tem bem mais poderes do que só facilitar a vida do desenvolvedor em tempo de compilação (o que já é muito). Você pode aliar o uso de genéricos com herança para garantir coesão

Eu posso dizer que minha caixa para numeros só vai aceitar Genéricos(N) que estendam de número

```

public class CaixaNumerica<N> extends Number<> extends Caixa<> {
    private final N qualquerNumero;

    public CaixaNumerica(N qualquerNumero) { this.qualquerNumero = qualquerNumero; }

    public N getQualquerNumero() { return qualquerNumero; }
}

```

```

public static void main(String[] args) {
    CaixaNumerica<BigDecimal> caixaNumerica = new CaixaNumerica<>(BigDecimal.TEN);
    CaixaNumerica<Integer> caixaNumerica2 = new CaixaNumerica<>(Integer.MAX_VALUE);
    CaixaNumerica<Double> caixaNumerica3 = new CaixaNumerica<>(Double.NEGATIVE_INFINITY);

    CaixaNumerica<String> caixaNumerica4 = new CaixaNumerica<>("qualquer nome: \"Joãozinho\"");
    CaixaNumerica<Pessoa> caixaNumerica5 = new CaixaNumerica<>();
}

```

Qualquer valor que estenda de Number funciona.

É impossível atribuir um valor que não seja filho de Number na minha caixa numérica

É uma opção considerável para garantir que uma classe trabalhe somente com uma hierarquia de herança específica

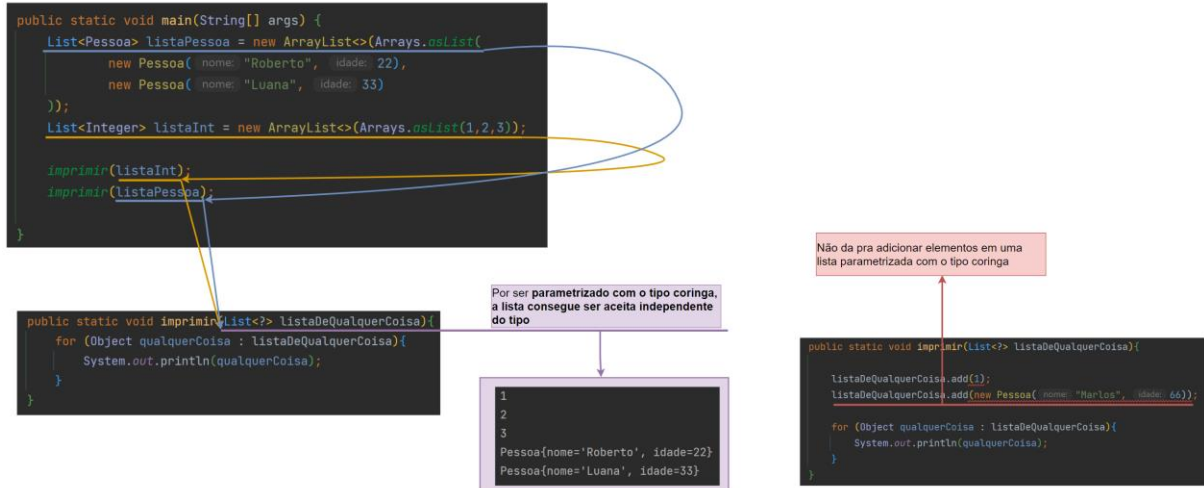
de uma classe:

Isso também é útil quando você itera sobre uma lista genérica e precisar usar comparação por exemplo, você pode delimitar que o Objeto genérico “extends” de Comparable.



Coringa (wildcard Types) – Java

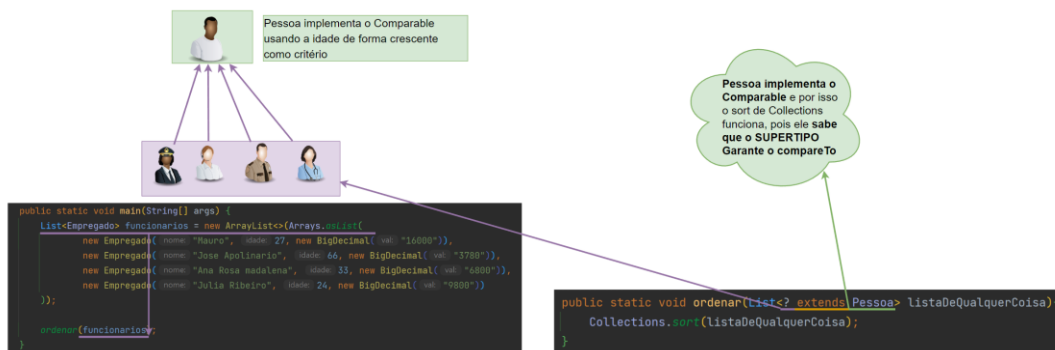
O tipo coringa é representado por “`<?>`”. Os tipos coringa são **usados como referência apenas**, por exemplo, eu **posso imprimir uma lista coringa**, mas **não posso adicionar nela**:



Coringas/Generics delimitados com **super** e **extend** – Java

Generico/Coringa que **Estende** um **super tipo**:

Quando especificado que o **coringa EXTEND** de **algo**, você está dizendo que pode acessar esse **coringa** e usar os “**métodos**” dele que são compatíveis com o “**algo**”:



Não dá para adicionar em uma lista coringa que EXTEND de um supertipo, **isso porque um subtipo pode ser diferente de outro**, então você não tem garantia que os subtipos sejam compatíveis.

Generico/Coringa que **tem um super** de **determinado Tipo**:

Aqui eu posso adicionar subtipos diferentes, isso porque tanto faz desde que o Super deles sejam o mesmo. Mas eu não posso recuperar esses elementos, porque eu não posso garantir que eles sejam

compatíveis a nível de instância:



Num macro a diferença básica é essa, um `<?>` que estende pode ser acessado, um `<?>` que tem um super pode ser adicionado. Ai dá pra combinar os dois em algum trabalho ultra genérico.



Coringas/Generics diferença for real – Java

O coringa `<?>` você não tem noção do que pode estar parametrizado ali, então você tem menos opções de manipulação, como adicionar em uma lista.

Usando **Generics** você está especificando um tipo quando for usar a classe, então você tem mais opções de manipulação, pois você pode garantir que o tipo genérico vai ser mais “controlável”.