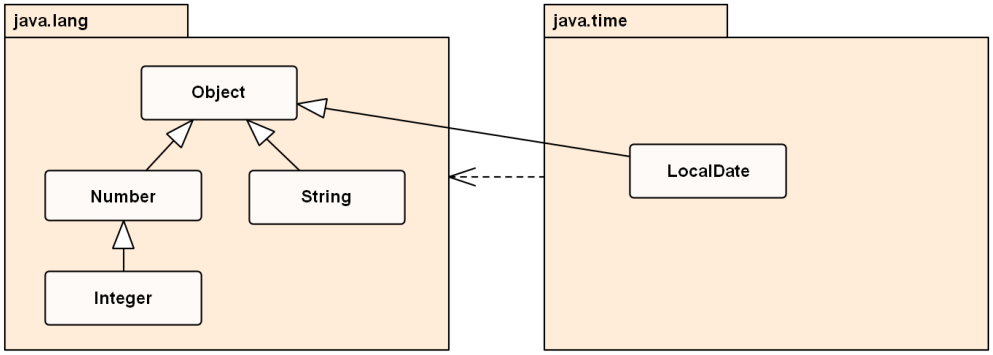
Roteiro 06 – Atividade 01/05

1. Observe o diagrama UML abaixo com as classes da API Java: `Integer`, `String` e `LocalDate`.



Já sabemos que qualquer classe Java é uma subclasse (direta ou indireta) de `java.lang.Object`. Também já conhecemos as classes `java.lang.String` e `java.time.LocalDate`. Elas instanciam objetos imutáveis que representam, respectivamente cadeias de caracteres e datas. A classe `java.lang.Integer` também não deveria ser novidade (considerando que você já leu os slides de referência da linguagem Java disponibilizados com o primeiro roteiro desta disciplina). `Integer` é considerada uma classe ***wrapper*** (empacotadora) porque ela encapsula (empacota) o tipo primitivo `int` (ela tem um único atributo, o qual é deste tipo). Todas as classes *wrapper* oferecem constantes, operações de conversão e de manipulação relacionadas com o tipo primitivo que elas empacotam. Veja também `Byte`, `Short`, `Long`, `Float`, `Double`, `Boolean` e `Character`.

Observe o trecho de código abaixo:

```
Integer numInt = 10;

System.out.println(numInt); // 10

System.out.println(numInt.intValue()); // 10

System.out.println(numInt == 10); // true
```

Autoboxing

Unboxing

Veja que a linguagem permite que `Integer` e `int` sejam utilizados com bastante flexibilidade. ***Autoboxing*** é a conversão automática que o compilador Java faz entre tipos primitivos e objetos de suas classes *wrapper* correspondentes (ex: `int` para `Integer`, `double` para `Double`). O processo inverso é chamado ***unboxing***. Classes *wrapper* permitem que tipos primitivos sejam utilizados como objetos. Entretanto, se você estiver realizando algum processamento matemático crítico, evite utilizar classes *wrapper*. *Autoboxing* e *unboxing* impactam negativamente no desempenho. Veja outro cuidado que devemos ter com classes wrapper:

```
Long numLong = 0L;
System.out.println(numLong == 0L);      // true
System.out.println(numLong == 0);       // true
System.out.println(numLong.equals(0L)); // true
System.out.println(numLong.equals(0));  // false (what??)
```

A constante 0 é *autoboxed* para `Integer`, não para `Long`. Como as classes são diferentes, `equals` retorna `false`. Pense na depuração de um problema como esse.

Na prática, utilize classes *wrapper* somente quando elas forem realmente necessárias.

2. Mas, voltando ao nosso diagrama inicial, percebemos que as classes `Integer`, `String` e `LocalDate` pertencem a hierarquias de generalização (herança) diferentes. Mas, por outro lado, veja se as afirmações abaixo fazem sentido (não considere sintaxe, mas apenas a lógica de raciocínio):

- O **numeroA** (`Integer`) é maior do que o **numeroB** (`Integer`).
- O **numeroA** (`Integer`) é menor do que o **numeroB** (`Integer`).
- A **stringA** (`String`) é maior do que a **stringB** (`String`)
- A **stringA** (`String`) é igual a **stringB** (`String`)
- A **dataA** (`LocalDate`) é menor do que a **dataB** (`LocalDate`)
- A **dataA** (`LocalDate`) é igual a **dataB** (`LocalDate`)

Roteiro 06 – Atividade 01/05

3. Se sua intepretação foi positiva em relação ao questionamento anterior, poderíamos ter uma operação com a seguinte assinatura (UML): **+ compareTo(o: Object): int**. O retorno poderia ser algo como:

   - Valor negativo para quando o objeto chamador for menor do que o objeto passado como parâmetro
   - Zero para quando o objeto chamador for igual ao objeto passado como parâmetro
   - Valor positivo para quando o objeto chamador for maior do que o objeto passado como parâmetro

   No código, o uso seria algo como:

```
System.out.println(new Integer(10).compareTo(15)); // -1
System.out.println(new Integer(15).compareTo(10)); // +1
System.out.println(new Integer(10).compareTo(10)); //  0
//
String s1 = "abc";
String s2 = "bac";
String s3 = "abc";
System.out.println(s1.compareTo(s2)); // -1
System.out.println(s2.compareTo(s1)); // +1
System.out.println(s3.compareTo(s1)); //  0
//
LocalDate a = LocalDate.parse("01/01/2017", AppConfig.DATE_FORMAT);
LocalDate b = LocalDate.parse("01/01/2018", AppConfig.DATE_FORMAT);
LocalDate c = LocalDate.parse("01/01/2017", AppConfig.DATE_FORMAT);
System.out.println(a.compareTo(b)); // -1
System.out.println(b.compareTo(a)); // +1
System.out.println(a.compareTo(c)); //  0
```

   Na verdade, o código acima funciona perfeitamente em Java. Todas estas três classes implementam a operação `compareTo`. Vale ressaltar que suas implementações são distintas.

   Mas e se quiséssemos obrigar estas classes a implementarem a operação `compareTo`, uma vez que seus valores são comparáveis?

   Se consideramos o que vimos na disciplina até agora, uma solução seria definir uma classe abstrata com esta operação também como abstrata. Deste modo qualquer subclasse direta desta classe seria obrigada a ter um método para a operação `compareTo`. O problema é que, em termos de negócio, não faz sentido algum afirmar que uma cadeia de caracteres é uma data ou que uma data é um inteiro ou vice-versa (a própria API Java colocou estas duas classes em hierarquias diferentes).

   Devemos utilizar classes abstratas quando queremos oferecer comportamento default para suas subclasses, ou seja, quando a reutilização de código é evidente. Classes abstratas também permitem que suas subclasses complementem o comportamento oferecido (por exemplo, utilizando o padrão de projeto Método Template). Se forem previstas alterações na classe abstrata, estas serão propagadas automaticamente para as subclasses. Um exemplo deste cenário é a nossa classe `Person`.

   Mas um critério fundamental continua sendo a **relação unidirecional "É UM" inerente a uma generalização** (herança). Podemos afirmar que um `Student` **É UM** `Person`, mas o contrário não é verdadeiro. Da mesma forma, podemos afirmar que um `Employee` **É UM** `Person` e não vice-versa. Por outro lado, como explicado anteriormente, `LocalDate` **NÃO É UM** `Integer` e `Integer` **NÃO É UM** `LocalDate`. Logo, utilizar classe abstrata neste caso seria um erro grosseiro de projeto. Lembre-se que a reutilização não é alcançada somente pela generalização.

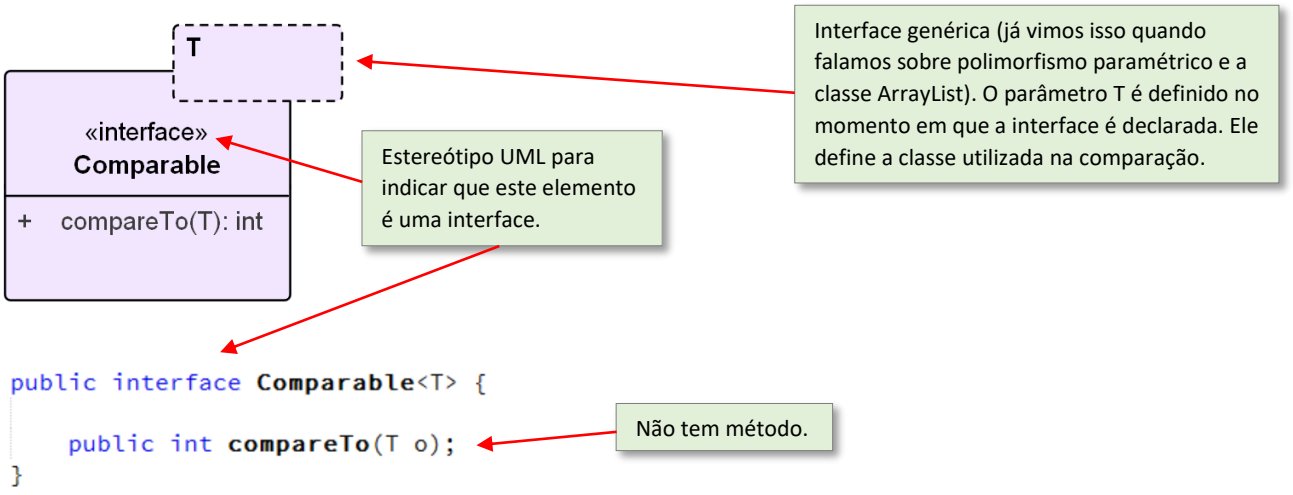
   Quando queremos que uma determinada classe seja capaz de realizar alguma operação (independentemente de sua hierarquia de generalização), utilizamos o conceito de interface[1]. Notem a diferença semântica entre "**ser um**" (generalização) e "**ser capaz de**" (interface). Uma **interface** é uma especificação de comportamento (ou contrato) que programadores concordam em implementar. Uma interface é similar a uma classe abstrata que tenha todas as suas operações também abstratas. Logo, ela não pode ser instanciada, pois não possui

[1] O termo interface no contexto apresentado aqui tem semântica diferente de quando utilizado no contexto de interação (entrada e saída) com o usuário.

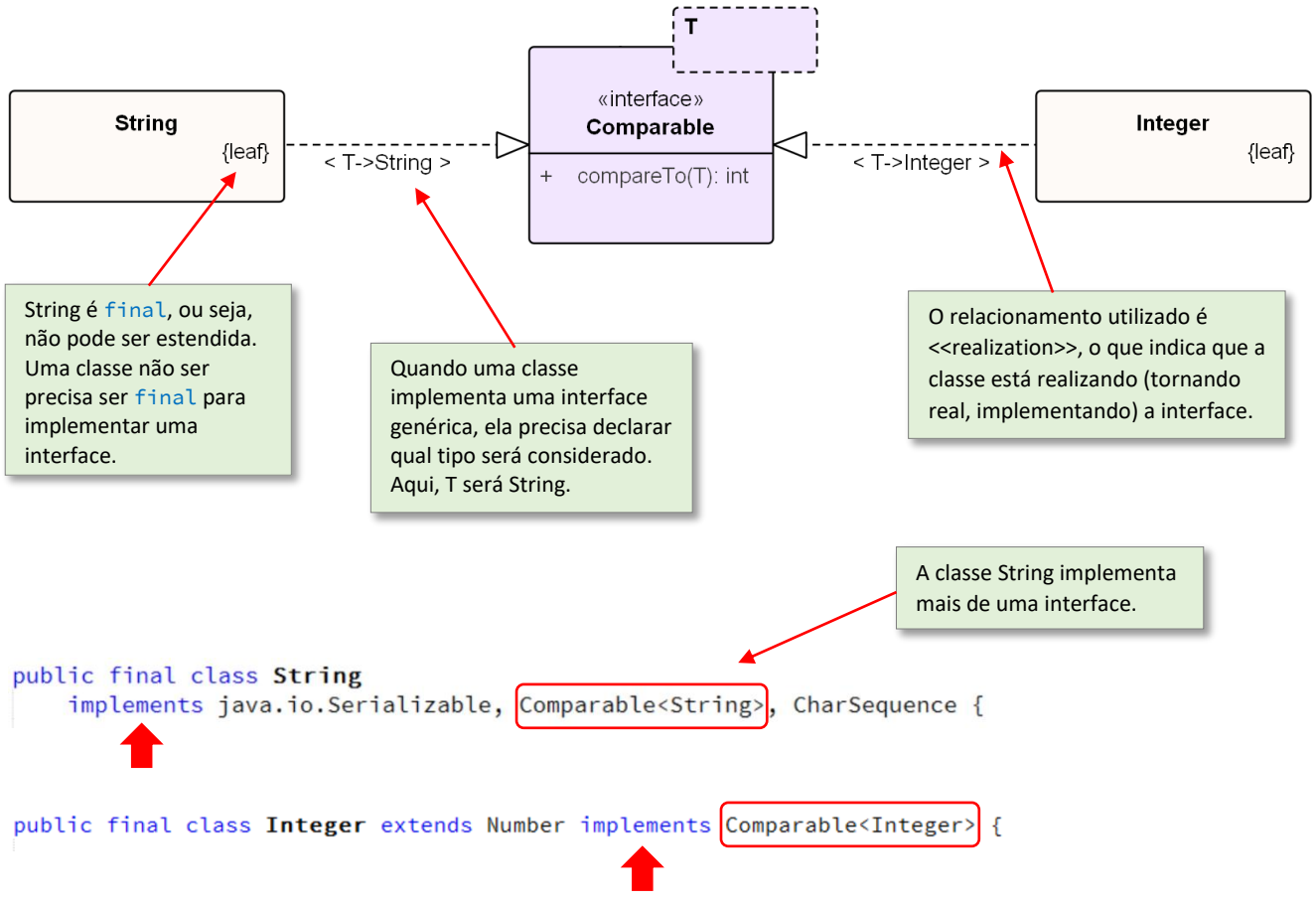
Roteiro 06 – Atividade 01/05

implementação própria. Qualquer classe que implementar uma interface deve garantir que todas as operações declaradas pela interface tenham um método (o contrato deve ser respeitado).

Uma classe não herda uma interface, ela a implementa. A classe é obrigada a definir um método para cada uma das operações definidas pela interface, caso contrário o compilador acusará um erro. A decisão de como cada método será desenvolvido é totalmente de responsabilidade de cada classe que implementa a interface. Considere por exemplo, a interface `Comparable` (Comparável) definida na API Java. Quando uma classe decide implementar esta interface, ela assume a responsabilidade de ser capaz de se comparar com outro objeto. Ela é obrigada a implementar a operação `compareTo`.

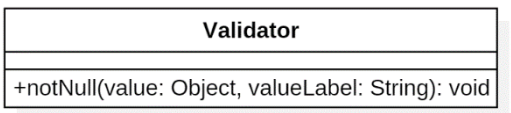


Agora, veja como representar que uma determinada classe implementa uma interface.



Roteiro 06 – Atividade 01/05

4. “Mas qual é a vantagem de garantir que uma classe implemente uma determinada operação se não temos reutilização de código?” A melhor forma de responder a este questionamento é demonstrar na prática. Vamos desenvolver uma pequena hierarquia para validações genéricas de dados. Ou seja, validações que podem ser utilizadas no momento em que valores estão sendo atribuídos aos atributos (construtores e operações modificadoras). É provável que você já tenha implementado a maioria destas validações (roteiros anteriores), mas nosso objetivo será separar o código de validação básica da lógica de negócio da classe. Vamos iniciar com a implementação da classe `Validator` (imagem abaixo). Esta classe será a nossa classe base da hierarquia de classes de validação.



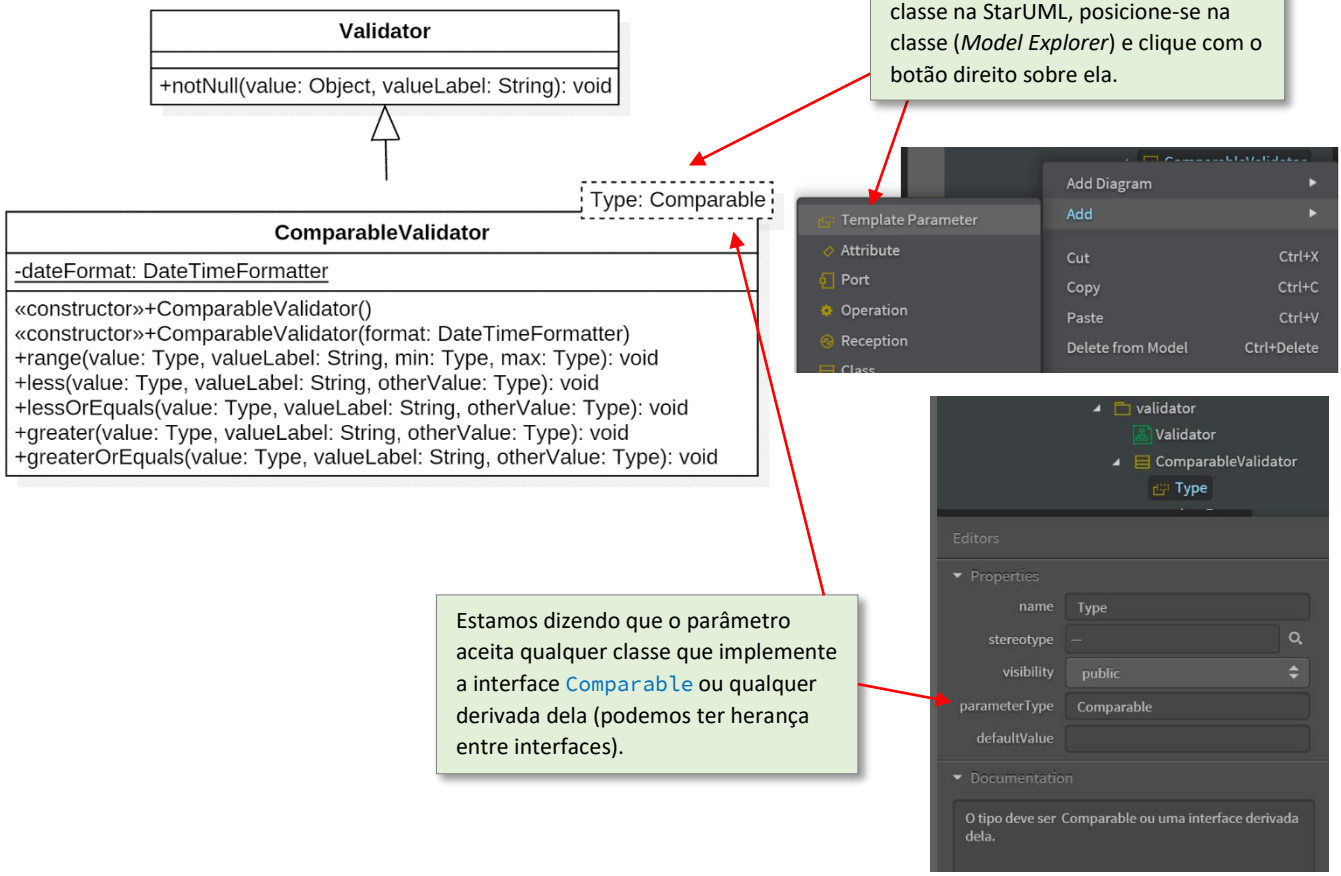

```java
/**
 * Base para a hieraquia de validadores. Nesta versão, foi implementado
 * apenas o comportamento default para verificar se um valor é null.
 *
 * @author Marcello Thiry
 */
public class Validator {

    /**
     * Verifica se um valor é null.
     *
     * @param value o valor a ser validado
     * @param valueLabel o texto referente ao valor que será utilizado na exceção
     *
     * @throws NullPointerException se value for null
     */
    public final void notNull(Object value, String valueLabel) {
        if (value == null) {
            throw new NullPointerException(valueLabel + " cannot be null");
        }
    }

}
```

Implementação trivial, mas note que eu optei por definir este método como `final`. É uma decisão de projeto, mas uma boa prática também, pois não existe motivo para completar futuramente o comportamento

5. O próximo passo é modelarmos e criarmos nossa primeira classe genérica (polimorfismo paramétrico). Vamos considerar uma classe `ComparableValidator<Type extends Comparable>`, a qual é responsável por realizar validações em qualquer classe (declarada no lugar do parâmetro de tipo `Type`) que implemente a interface `Comparable` ou uma derivada dela.



Roteiro 06 – Atividade 01/05

Veja que estamos tratando a interface `Comparable` como se ela fosse uma classe. Esta flexibilidade permite que tenhamos uma interface comum entre classes que pertencem a hierarquias diferentes.

```
ComparableValidator<Integer> valInt = new ComparableValidator<>();
ComparableValidator<String> valStr = new ComparableValidator<>();
ComparableValidator<LocalDate> valDate = new ComparableValidator<>();
ComparableValidator<BigDecimal> valBigDec = new ComparableValidator<>();
ComparableValidator<Double> valDouble = new ComparableValidator<>();
```

Todas estas classes implementam a interface `Compable<T>`.

Implemente esta parte inicial conforme a imagem abaixo.

```
/**
 * Classe genérica para validação de dados comparáveis
 *
 * @author Marcello Thiry
 *
 * @param <Type> qualquer classe que implemente Comparable
 */
public class ComparableValidator<Type extends Comparable> extends Validator {

    private static DateTimeFormatter dateFormat;

    public ComparableValidator() {
        dateFormat = DateTimeFormatter.ofPattern("dd/MM/yyyy");
    }

    public ComparableValidator(DateTimeFormatter format) {
        dateFormat = format;
    }

    /**
```

Type: Comparable

Somente para deixar datas mais legíveis. Podemos rever isso depois.

6. Agora, vamos considerar uma operação que valida se um determinado valor está dentro de uma determinada faixa [mín..máx]: `void range(Type value, String valueLabel, Type min, Type max)`. Veja que esta semântica se aplica a qualquer valor numérico (descendentes de Number, BigDecimal, etc.), mas também a datas (em determinadas situações, é necessário verificar se uma data está compreendida em um determinado período). Note que o tipo do parâmetro `value` foi definido como o mesmo nome do tipo passado como parâmetro pela classe: `Type`. Este tipo será substituído em tempo de compilação, conforme a declaração do nosso validador[2]. O parâmetro `valueLabel` (rótulo do valor) é utilizado apenas na formatação da mensagem passada para a exceção, caso a validação falhe. Por uma decisão de projeto, consideramos que esta operação não poderá ser redefinida em subclasses de `ComparableValidator`. Logo, ela foi declarada como `final`.

```
/**
 * Verifica se um valor Comparable está dentro de uma faixa específica.
 *
 * @param value o valor a ser validado
 * @param valueLabel o texto referente ao valor que será utilizado na exceção
 * @param min o valor mínimo para a faixa (inclusive)
 * @param max o valor máximo para a faixa (inclusive)
 *
 * @throws NullPointerException se o valor for null
 * @throws IllegalArgumentException se valor estiver fora da faixa especificada
 */
public final void range(Type value, String valueLabel, Type min, Type max) {
    notNull(value, valueLabel);
    if (value.compareTo(min) == -1 || value.compareTo(max) == 1) {
        throw new OutOfRangeException(value.toString(),
                value.getClass().getSimpleName() + "." + valueLabel, min.toString(), max.toString());
    }
}
```

[2] Na prática, por questões de otimização e compatibilidade, o compilador Java substituirá `Type` por `Object` e gerará o código com os *downcasts* apropriados. Este conceito é chamado **type erasure**. Voltaremos neste conceito mais tarde.

Roteiro 06 – Atividade 01/05

Note que a primeira linha do método verifica se o valor é `null`, utilizando o método herdado de `Validator`. Desta forma, o programador que utilizar esta validação não precisará se preocupar em verificar isso antes. Na condição, invocamos `compareTo` a partir do parâmetro passado `value`, primeiro para comparar com o valor mínimo e depois com o valor máximo. Lembre-se que compareTo pode retornar -1 (o objeto atual é menor que o objeto comparado), 0 (quando os objetos são iguais) ou -1 (o objeto atual é maior que o objeto comparado). Logo, a lógica usada é: **se (value < min OR value > max) dispara exceção "fora da faixa"**. Note também que foi necessária uma modificação na nossa classe de exceção `OutOfRangeException`, pois passamos `min` e `max` como strings. Isso é necessário, pois o Java não permite que exceções sejam genéricas. O método `getSimpleName()` da classe `Class`, retornada em `getClass()`, retorna o nome final da classe, sem o *namespace*.

7. Não sei se você percebeu, mas há um furo na lógica anterior. Não testamos se o valor mínimo é menor do que o valor máximo ou se eles são `null`. Ajuste sua implementação. Caso você não tenha alguma exceção implementada que se enquadre nesta situação (como `InvalidRangeBoundValueException`), utilize a exceção não checada (*runtime*) `IllegalArgumentException`. Outra melhoria é que estamos utilizando valores fixos -1 e 1 para verificar se o objeto é, respectiva menor que ou maior que o outro. O problema desta abordagem é que a especificação de `Comparable` afirma que o retorno de `compareTo` pode ser um valor negativo, zero ou um valor positivo. Desta forma, outras implementações de `compareTo` poderiam retornar qualquer valor negativo ou positivo e ainda atender a especificação. Faça os ajustes necessários na classe, substituindo o teste com -1 e 1 por quaisquer valores negativos ou positivos.

8. Para testar sua primeira classe genérica, você pode tentar o código abaixo. Inicialmente, ele deve passar sem exceções. Faça alterações para simular exceções em cada tipo e analise os resultados. Veja com o uso de interfaces com classes genéricas deixou nossa classe de validação flexível, mas sem perder sua simplicidade e semântica.

```
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate myDate  = LocalDate.parse("01/01/2017", dateFormat);
LocalDate minDate = LocalDate.parse("01/01/2014", dateFormat);
LocalDate maxDate = LocalDate.parse("01/01/2018", dateFormat);
new ComparableValidator<LocalDate>().range(myDate, "My Date", minDate, maxDate);

BigDecimal myNumber  = new BigDecimal("100.45");
BigDecimal minNumber = new BigDecimal("100.43");
BigDecimal maxNumber = new BigDecimal("100.47");
new ComparableValidator<BigDecimal>().range(myNumber, "My Date", minNumber, maxNumber);

Integer myInt  = 10;
Integer minInt = 1;
Integer maxInt = 10;
new ComparableValidator<Integer>().range(myInt, "My Integer", minInt, maxInt);

String myStr  = "DDDD";
String minStr = "CCCC";
String maxStr = "EEEE";
new ComparableValidator<String>().range(myStr, "My Integer", minStr, maxStr);

System.out.println("Validações feitas com sucesso. Todo mundo passou!");
```

9. Para fixar o conceito, implemente também a operação `lessOrEquals` abaixo. Como atividade extraclasse, complete a implementação da classe `ComparableValidator` e atualize seu pra para utilizá-la.

```
/**
 * Verifica se um valor Comparable é menor ou igual ao outro valor.
 *
 * @param value o valor a ser validado
 * @param valueLabel o texto referente ao valor que será utilizado na exceção
 * @param otherValue o valor a ser comparado
 *
 * @throws NullPointerException se um dos valores for null
 * @throws IllegalArgumentException se value não for menor ou igual ao outro valor
 */
public final void lessOrEquals(Type value, String valueLabel, Type otherValue) {
    notNull(value, valueLabel);
    notNull(otherValue, "Other Value");
    if (value.compareTo(otherValue) > 0) {
        throw new IllegalArgumentException(valueLabel + " (" + value.toString()
                + ") must be less than or equals to " + otherValue);
    }
}
```

Roteiro 06 – Atividade 02/05

Com base no que vimos até agora...

1. Quais as diferenças entre uma classe abstrata e uma interface?

2. O que é uma classe wrapper em Java?

3. O que é *autoboxing* e *unboxing* em Java?

4. Quando devemos utilizar uma classe abstrata e quando devemos utilizar uma interface?

5. Quais as vantagens de utilizar interfaces?

6. O que é uma interface genérica?

7. Por que precisamos utilizar classes ou interfaces genéricas?

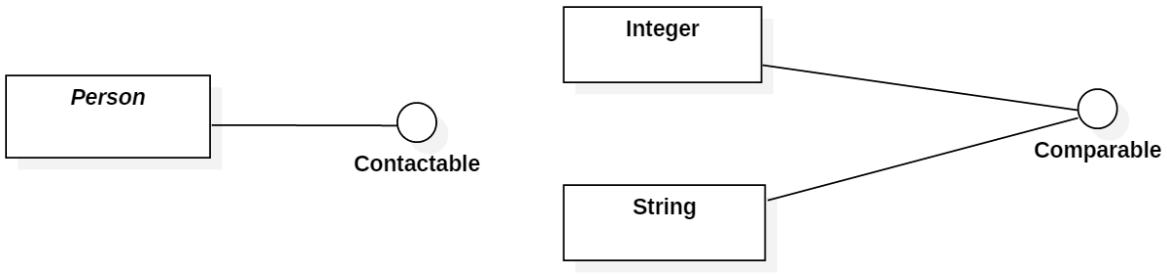
8. Quando um método deve ser `final`?

9. Explique o que significa a parte genérica na declaração da nossa classe `ComparableValidator`.

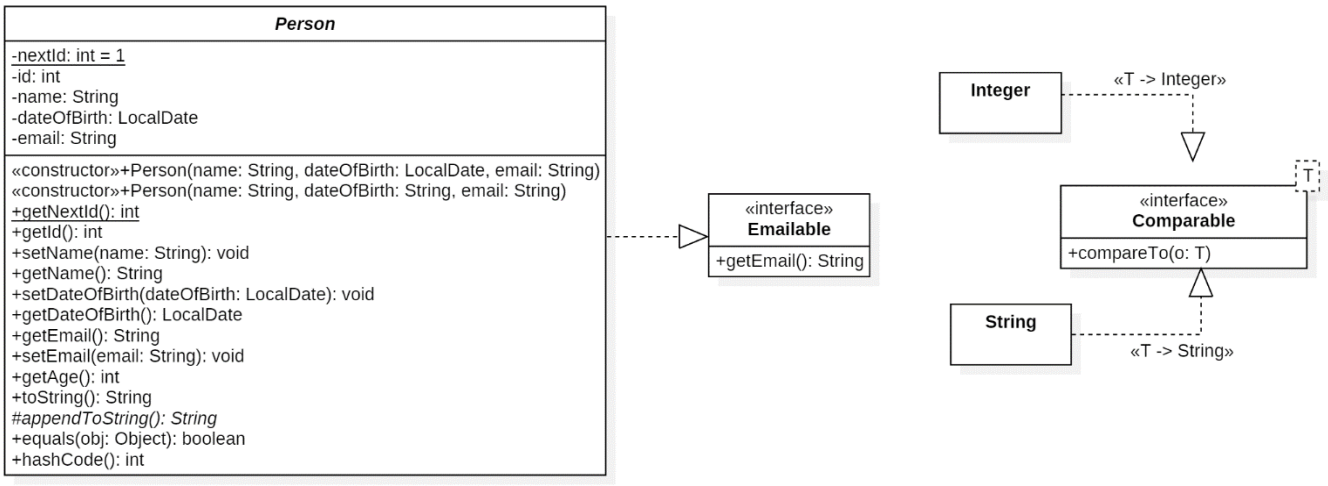
10. O que é um parâmetro de tipo em classes genéricas? Qual o seu propósito?

Roteiro 06 – Atividade 03/05

1. Observe o diagrama de classe na imagem abaixo. Alguma ideia?

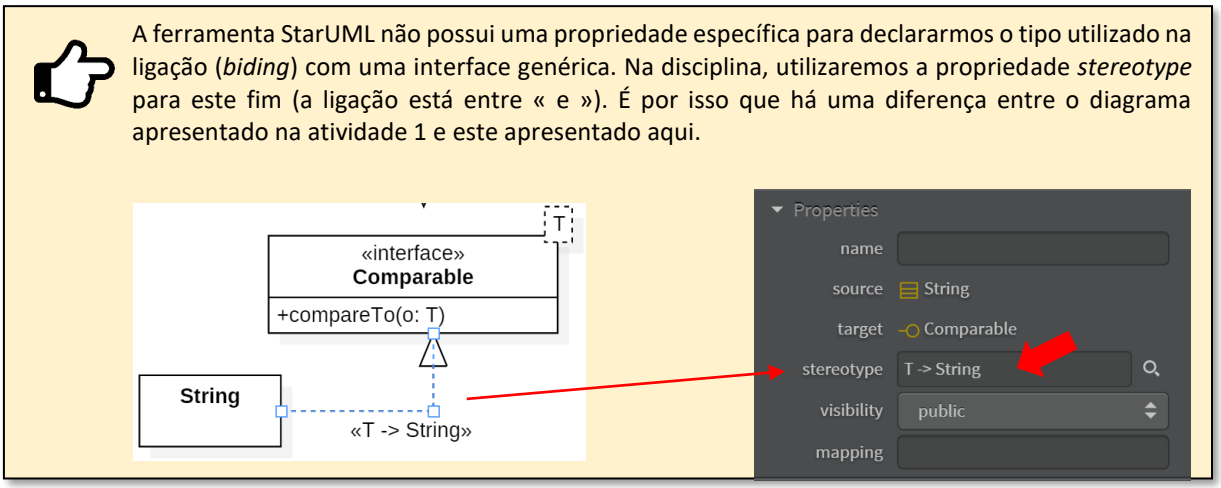


Agora, observe o mesmo diagrama, mas com uma notação alternativa (mais detalhada):



As duas notações acima mostram a mesma coisa, ou seja, quais classes implementam quais interfaces. A primeira notação de uma interface é chamada ***ball*** ou ***lollipop*** (por ser similar a um círculo/bola ou pirulito). A segunda notação já havia sido vista na atividade anterior e mostra as operações declaradas na interface. Na prática, a primeira notação tem sido mais utilizada por deixar o diagrama mais legível. Entretanto, nesta disciplina, utilizaremos ambos, pois em alguns casos, será mais didático mostrar os detalhes da interface.

Outro aspecto a ser observado é que a primeira notação não permite identificar se a interface é genérica ou não. Veja que, com a segunda notação, é possível afirmar que `Contactable` não é genérica (note que ela não possui um parâmetro de tipo). O relacionamento nas duas notações é o mesmo (realização no sentido da classe para a interface – uma classe realiza uma interface). Porém, a primeira notação é mais simples. Na segunda, podemos ver ainda como a ligação (*binding*) com o parâmetro de tipo é feita. Por exemplo, a classe `Integer` utiliza `Comparable<Integer>`, pois ela indica que `T -> Integer`.



A ferramenta StarUML não possui uma propriedade específica para declararmos o tipo utilizado na ligação (*biding*) com uma interface genérica. Na disciplina, utilizaremos a propriedade *stereotype* para este fim (a ligação está entre « e »). É por isso que há uma diferença entre o diagrama apresentado na atividade 1 e este apresentado aqui.

Roteiro 06 – Atividade 03/05

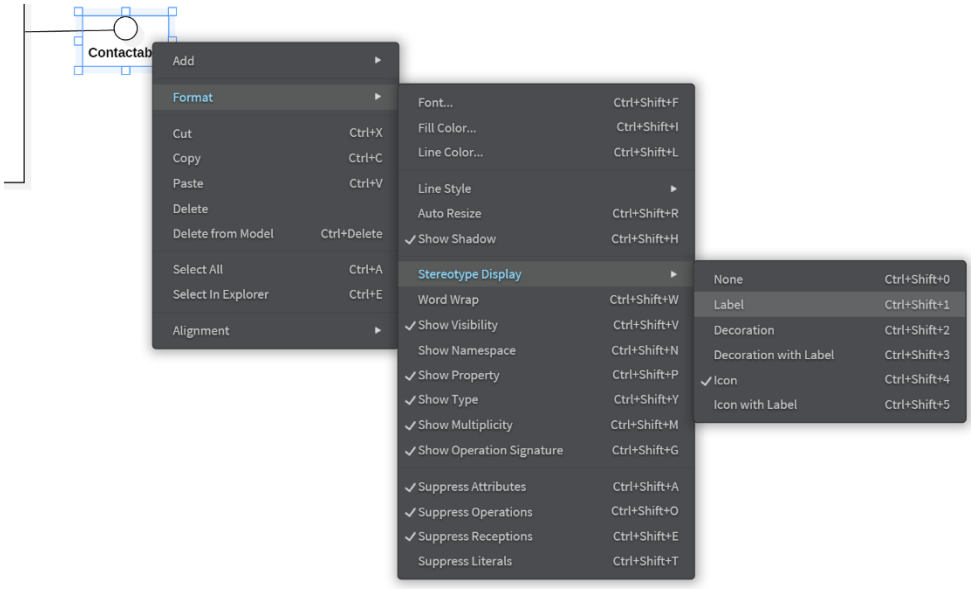
2. A linguagem Java tem como prática nomear suas interfaces com o sufixo “-able”, o qual significa “-ável” na língua portuguesa. O objetivo é demonstrar a capacidade de fazer alguma coisa. Por exemplo, quando uma classe implementa `Comparable` (Comparável) é assumido que ela é capaz de ser comparada. Entretanto, não há uma regra oficial de nomeação. Por exemplo, `Throwable`[3] (pode ser lançado) é uma classe e não uma interface. Por outro lado, `java.time.chrono.ChronoLocalDate` é uma subinterface de `Comparable`. A classe `LocalDate` não implementa diretamente `Comparable`, mas `ChronoLocalDate`. Esta interface é a representação abstrata de uma data independente do sistema de calendário. Na prática, podemos considerar que uma classe, ao implementá-la, é capaz de representar datas em qualquer sistema de calendário.

   Uma prática também bastante comum, mas não seguida pela API Java, é iniciar o nome de uma interface com a letra “I”, por exemplo, `IComparable`. Entretanto, existem muitas críticas sobre utilizar mnemônicos que representam tipos na nomeação (notação húngara[4]). Existem vantagens em manter nomes de classes e interfaces indistinguíveis, principalmente quando há necessidade de *refactoring*. Você pode transformar uma classe em uma interface (para aumentar a flexibilidade de seu programa, por exemplo) e o código legado não será afetado (tanto faz se invocação de um método é através de uma interface ou não, desde que o programa se comporte como esperado). Outro argumento é que interface é um tipo, logo ela deveria seguir a mesma convenção de qualquer abstração (o nome deve representar exatamente esta abstração).

   Nesta disciplina, não faremos distinção entre nomear uma classe ou uma interface. O nome deve ser de acordo com a abstração utilizada para sua definição. Quando você identificar uma situação onde há uma interface que tenha uma única classe a implementando, é muito provável que você não precise da interface. Com a orientação anterior, caso a necessidade apareça no futuro, quebrar em interface e classe concreta não será um problema. Evite criar complexidade desnecessária no seu código, mas também observe boas práticas que facilitarão as mudanças (e elas virão com certeza).

3. Após conhecermos um pouco mais sobre interfaces, podemos iniciar as alterações em nossa modelagem. Vamos modelar a interface `Contactable` e fazer com que a classe `Person` a realize (implemente). Na ferramenta StarUML, abra a toolbox [Classes (Basics)] e coloque uma interface no seu diagrama, exatamente como você já vem fazendo com classes e altere seu nome para `Contactable`).

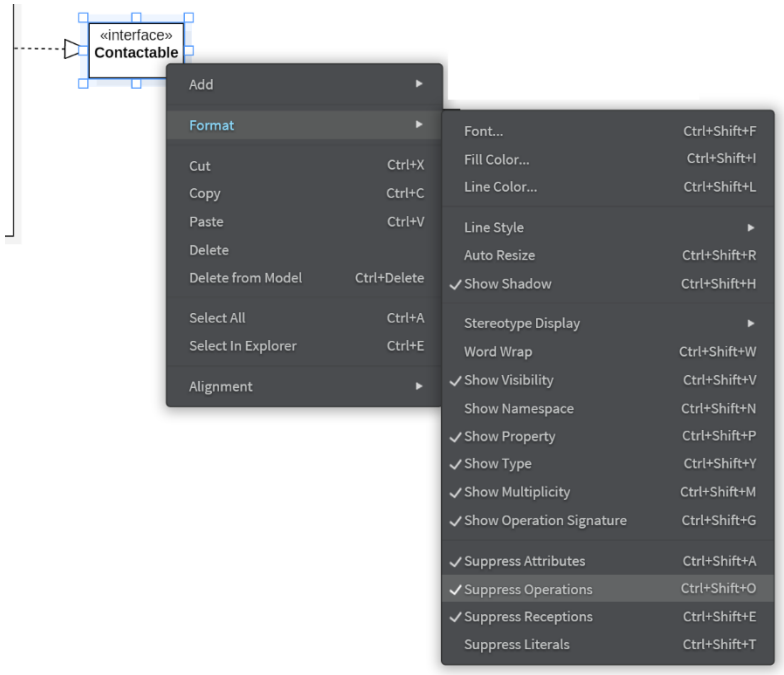
4. Para que Person realize `Contactable`, na toolbox, selecione o relacionamento com o sugestivo nome “Interface Realization”. Clique em `Person`, arraste até `Contactable` e solte (mesmo procedimento para qualquer relacionamento). Para alternar a forma de visualização da interface, você clicar com o botão direito sobre ela e selecionar [Format | Stereotype Display | Label]. Veja abaixo que existem outras opções de visualização.



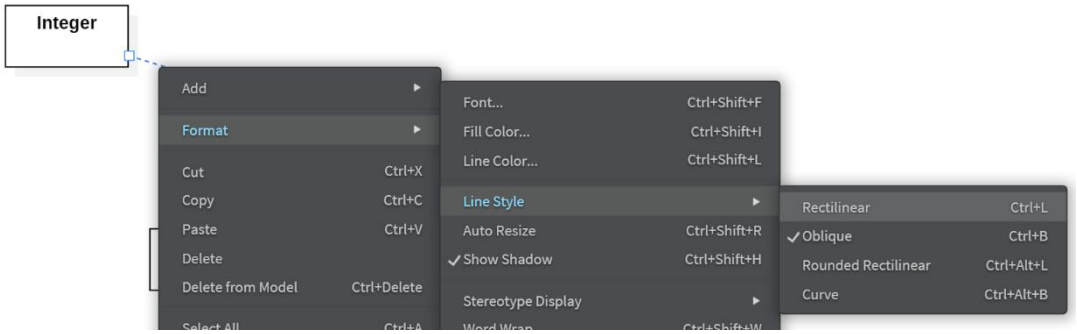
[3] Se você não lembra de `Throwable`, revise o roteiro sobre exceções.
[4] https://en.wikipedia.org/wiki/Hungarian_notation

Roteiro 06 – Atividade 03/05

5. Adicione a operação na interface, conforme o diagrama inicial desta atividade. O procedimento é o mesmo adotado para adicionar operações em uma classe. Veja que a operação não está sendo mostrada no diagrama. Para alterar isso, novamente abra o menu de contexto da interface (botão direito), selecione [Format] e desmarque a opção [Supress Operations].

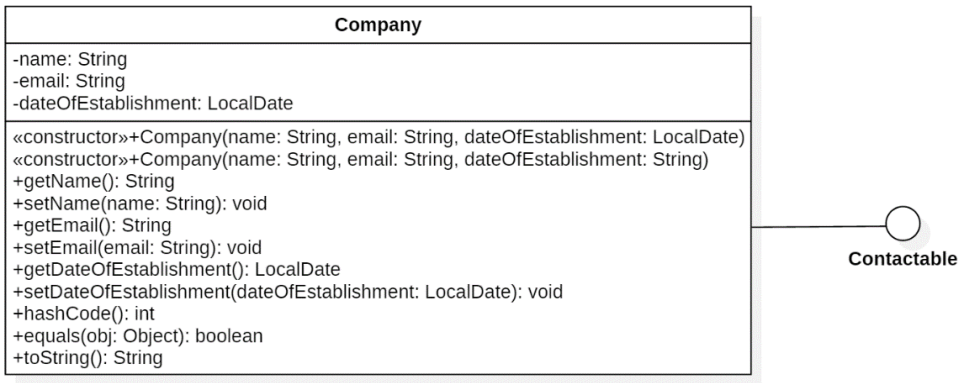


6. Se você quiser modificar a forma que a linha é desenhada (veja nosso primeiro diagrama), você pode abrir o menu de contexto do relacionamento de realização da interface, selecionar [Format | Line Style] e escolher entre as diferentes opções de visualização.



7. Programe a interface `Contactable` e atualize a classe `Person`, a qual deve agora realizar `Contactable`.

8. Modele e implemente a classe abaixo:



Roteiro 06 – Atividade 03/05

9. Importe para seu projeto a classe `ContactBook`, disponibilizada como parte deste roteiro. Coloque esta classe também no seu diagrama de classe.

10. Ao abrir o arquivo desta classe, você notará que há uma segunda classe de teste (este código pode ser excluído ou colocado em outro local depois).

```
class ContactBookLocalTest {

    private static ContactBook contactBook = new ContactBook();

    public static void addContacts() {
        contactBook.add(new Employee("Orin Curry", "15/01/1976", "orin@com", "01/03/2000", 40, "22.80"));
        contactBook.add(new Student("Bruce Wayne", "02/05/1996", "wayne@com", "01/03/2017"));
        contactBook.add(new Professor("Stephen Vincent Strange", "23/06/1971", "strange@com", "05/08/1996", 40, "22.80", AcademicDegree.DOCTORATE));
        contactBook.add(new Student("Emma Grace Frost", "23/09/1994", "frost@com", "31/07/2016"));
        contactBook.add(new Employee("Susan Kent-Barr", "06/10/1969", "susan@com", "09/08/1995", 40, "25.23"));
        contactBook.add(new Company("ACME", "acme@com", "01/01/1960"));
        contactBook.add(new Company("Alibi Bar", "alibibar@com", "06/07/1971"));
    }

    public static void main(String[] args) {
        // simulando a inclusão de contatos...
        addContacts();
        // mostrando todos os contatos...
        System.out.println(contactBook.toString());
        // e testando emails
        contactBook.email("Happy New Year!", "Dear Friends, blah blah blah...");
    }

}
```

Não importa a classe, desde que o objeto seja `Contactable`.

Apenas para mostrar como o `toString` fica simples quando implementamos o `toString` dos elementos.

11. Para testar nosso programa no NetBeans, clique com o botão direito sobre qualquer área em branco do arquivo aberto e selecione [Executar Arquivo]. Analise os resultados.

    Veja que não faria sentido termos uma classe abstrata `Contactable`, a qual, em algum momento teria que ser herdada por `Person` e `Company`. A ideia é que pessoas ou empresas sejam capazes de ser contatadas. Além disso, estaríamos forçando a herança e dificultando manutenções futuras, pois herança propaga as mudanças. Por exemplo, regras de validação para um nome de empresa provavelmente são diferentes daquelas usadas para pessoas físicas.

    Interfaces têm sido amplamente utilizadas no desenvolvimento orientado a objetos. Leia mais sobre elas (nunca use uma única fonte), acompanhe discussões sobre a utilização de herança e interfaces. Porém, avalie com cuidado propostas radicais como: “não use herança, use interfaces”. Seja equilibrado e ponderado. Um bom desenvolvedor precisa analisar o contexto do problema, identificar possíveis soluções e escolha aquela que possui o melhor custo/benefício para o momento e que facilite o caminho para mudanças no futuro.

Roteiro 06 – Atividade 04/05

1. Uma lista `list` do tipo `List` (onde `List` é uma interface da API Java) pode ser ordenada com a operação estática `sort` da classe `Collections`:

   ```
   Collections.sort(list);
   ```

   Se `list` possui elementos `String`, ela será ordenada alfabeticamente. Se ela é formada de elementos `LocalDate`, ela será ordenada cronologicamente. Isso acontece porque, como vimos anteriormente, ambas implementam a interface `Comparable`. Ou seja, esta operação só funciona se os elementos da lista forem comparáveis. No nosso caso, se quiséssemos que nossos contatos fossem ordenados por nome, poderíamos implementar a interface `Comparable` nas classes `Person` e `Company`. A imagem abaixo apresenta como seria na classe `Person`. Para a classe `Company`, o procedimento é o mesmo.

```
public abstract class Person implements Contactable, Comparable {
...
    /**
     * Compara objetos Contactable pelo nome. Utilizado para ordenação.
     *
     * @param obj o objeto Person a ser comparado
     * @return valor negativo quando este objeto for menor do que o passado, zero quando
     *          forem iguais e valor positivo quando este objeto for maior
     *
     * @throws NullPointerException se o objeto passado for null
     * @throws ClassCastException se o objeto passado não for Person
     */
    @Override
    public int compareTo(Object obj) {
        if (obj == this) {
            return 0;
        }
        if (obj instanceof Contactable) {
            return name.compareTo(((Contactable)obj).getName());
        }
        throw new ClassCastException("Contactable object expected. "
                + obj.getClass().getName() + " instead.");
    }
```

   Repare que a verificação da compatibilidade é com a interface `Contactable` e não com `Person`. Lembre-se que a ideia aqui é comparar objetos de diferentes classes, mas com uma interface em comum. Isto nos traz mais um problema: a lista `CONTACTS` não é uma lista de `Person` ou de `Company`, mas de `Contactable`. E uma interface não pode implementar outra Interface, mas ela pode especializar uma existente. Se alterarmos nossa interface `Contactable` para:

```
public interface Contactable extends Comparable {

    public String getName();
    public String getEmail();

}
```

   a nossa lista `CONTACTS` passará a ser uma lista também de elementos `Comparable`. A contrapartida aqui é que todas as classes que forem `Contactable` precisarão também oferecer um método `compareTo`. Mas isso faz sentido se queremos permitir a ordenação dos nossos contatos pelo nome. Não esqueça de retirar o `implements Comparable` nas duas classes, pois fica redundante (elas já implementam `Contactable`).

Roteiro 06 – Atividade 04/05

Agora, o método `sort` em nossa classe `ContactBook` fica muito simples. Podemos utilizar a operação estática `sort` da classe `Collections`:

```
public void sort() {
    Collections.sort(CONTACTS);
}
```

2. Uma limitação da solução anterior é que precisamos implementar um método praticamente igual em todas as classes `Contactable`. Como alternativa, podemos implementar uma classe que implementa a comparação desejada, retirando esta responsabilidade de todas as classes `Contactable`. Para isso, você deve antes:

   a) Excluir a herança de `Contactable` para `Comparable`. Nossas classes não precisam saber comparar mais.
   b) Excluir o método redefinido `compareTo` nas classes `Person` e `Company`.

   Agora, vamos criar a nossa classe que fará a comparação de qualquer `Contactable`. Esta classe precisa implementar a interface genérica `Comparator<T>`, a qual declara uma operação `compare(T o1, T o2)`.

```
/**
 * Classe que compara nome dos contatos.
 *
 * @author Marcello Thiry
 */
public class ContactableNameComparator implements Comparator<Contactable> {

    @Override
    public int compare(Contactable c1, Contactable c2) {
        //Nos testes, tire o comentário abaixo e acompanhe as comparações
        //System.out.println(c1.getName() + " comparing to " + c2.getName());
        return c1.getName().compareTo(c2.getName());
    }

}
```

   Note que implementamos a interface já passando o parâmetro de tipo `<Contactable>`. Desta forma, podemos redefinir a operação `compare`, utilizando o tipo adequado nos parâmetros. A comparação em si é trivial, pois utilizamos o próprio método `compareTo` da classe `String` (lembre-se que ela implementa `Comparable`).

   Mas não vimos ainda como utilizar a nossa classe criada. Isso será muito simples, pois a operação `sort` de `Collections` possui uma sobrecarga que aceita como parâmetros, um objeto `List` e um objeto `Comparator`. O método irá utilizar o objeto `Comparator` para ordenar os elementos do objeto `List`. A nossa operação `sort` ficará assim:

```
public void sort() {
    Collections.sort(CONTACTS, new ContactableNameComparator());
}
```

   Entretanto, também podemos utilizar a operação `sort`, disponibilizada pela interface `List`:

```
public void sort() {
    CONTACTS.sort(new ContactableNameComparator());
}
```

   Note que a segunda opção é mais interessante, pois não precisamos da classe *helper* `Collections`. Uma **classe *helper*** é usada para fornecer alguma funcionalidade, a qual não seria uma responsabilidade direta da classe na qual ela é aplicada. Entretanto, ordenar elementos é uma responsabilidade tipicamente esperada por uma classe lista. Logo, solicitar a ordenação para a própria lista deixa a implementação mais orientada a objetos.

Roteiro 06 – Atividade 04/05

A partir do Java 8, uma interface pode declarar métodos *default*, ou seja, um comportamento *default* para uma operação declarada. Abaixo, podemos ver o método default para a operação `sort` declarada na interface `List`.

```
public interface List<E> extends Collection<E> {
...
  default void sort(Comparator<? super E> c) {
      Object[] a = this.toArray();
      Arrays.sort(a, (Comparator) c);
      ListIterator<E> i = this.listIterator();
      for (Object e : a) {
          i.next();
          i.set((E) e);
      }
  }
...
```

Monta um vertor temporário de `Object` com os elementos da lista atual (`this`).

Utiliza a classe `Arrays` para ordenar o vetor temporário. `Arrays` é outra classe *helper* em Java.

Troca os elementos da lista atual um a um com os elementos do vetor temporário agora ordenado.

Com esta solução, podemos implementar diferentes classes de comparação, conforme as necessidades do negócio. O mais interessante é que não precisaremos alterar as classes dos objetos que são armazenados na lista.

3. Uma terceira forma de abordar a mesma questão é utilizando uma classe anônima em Java. **Classes anônimas** são expressões que permitem a declaração e instanciação de uma classe ao mesmo tempo. Elas funcionam como classes internas (*inner classes*), mas sem um nome. Você deve utilizá-las somente quando uma classe interna for utilizada uma única vez. Observe o trecho de código abaixo incluído na classe `ContactBook`.

```
private static final Comparator<Contactable> NAME_ORDER = new Comparator<Contactable>() {
    @Override
    public int compare(Contactable c1, Contactable c2) {
        return c1.getName().compareTo(c2.getName());
    }
};

private static final Comparator<Contactable> EMAIL_ORDER = new Comparator<Contactable>() {
    @Override
    public int compare(Contactable c1, Contactable c2) {
        return c1.getEmail().compareTo(c2.getEmail());
    }
};
```

Veja que foram definidos atributos do tipo `Comparator<Contactable>`. Entretanto, a diferença aqui é nós estamos instanciando estes objetos a partir de uma classe sem nome que implementa `Comparator<Contactable>`. Observe que não estamos instanciando a interface, mas uma classe que a implementa (poderia ser também o nome de uma classe a ser estendida). Ao mesmo tempo que estamos instanciando o objeto, estamos inserindo a implementação da classe (no nosso caso, a implementação da operação `compare` que faz parte do contrato de `Comparator`). Por ser considerada uma declaração, note que ela termina com ";". Os atributos foram definidos como `static final` por uma decisão de projeto. Você pode encontrar mais detalhes em https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html.

Para utilizar a nossa classe anônima, basta utilizarmos os atributos definidos:

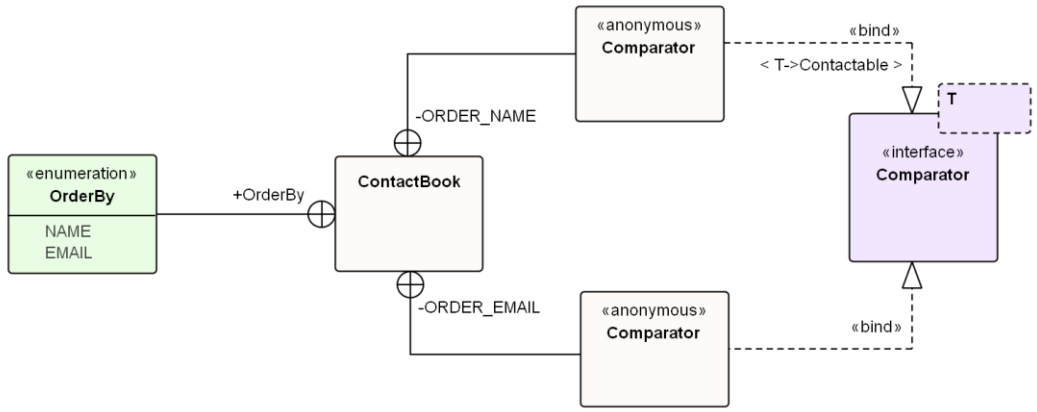
```
public void sort() {
    CONTACTS.sort(NAME_ORDER);
}
```

Roteiro 06 – Atividade 04/05

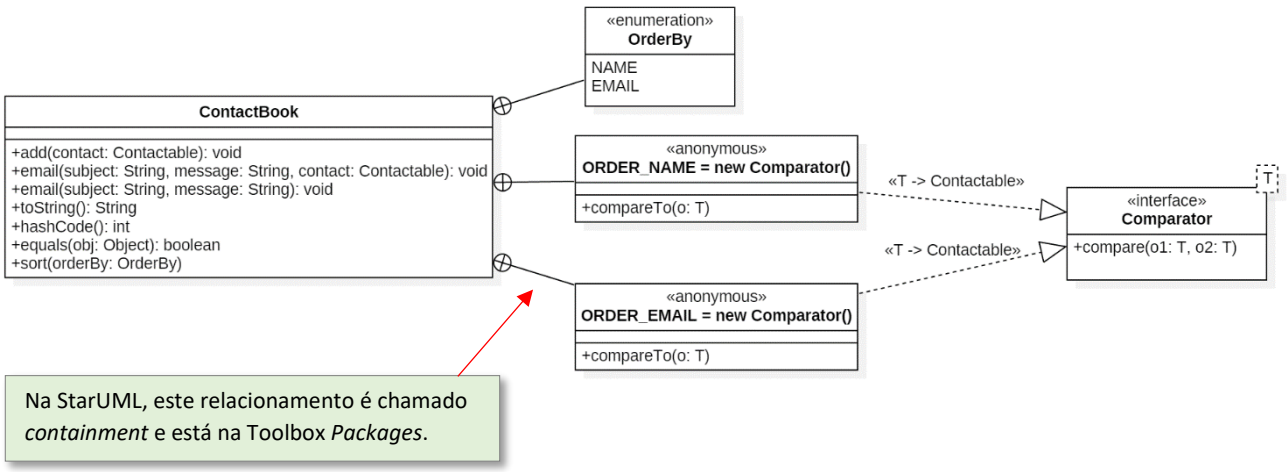
Mas você deve ter reparado que criamos dois objetos `Comparator`, sendo um para a ordenação pelo nome e outro para ordenarmos pelo e-mail. Para deixar nossa classe `ContactBook` mais flexível, vamos criar um atributo **enum** que conterá os tipos de ordenação possíveis:

```
public static enum OrderBy { NAME, EMAIL };
```

4. O diagrama de classe com a notação UML para as classes anônimas e o **enum** local ficaria como a imagem abaixo.



Entretanto, por limitações da StarUML, modelaremos da seguinte forma nestes casos:



5. E para finalizar, precisamos ajustar o método `sort`:

```
public void sort(OrderBy orderBy) {
    if (orderBy == OrderBy.NAME) {
        CONTACTS.sort(NAME_ORDER);
    } else {
        CONTACTS.sort(EMAIL_ORDER);
    }
}
```

ou, como só temos duas opções, utilizar um operador ternário (o código fica mais conciso, sem perder legibilidade):

```
public void sort(OrderBy orderBy) {
    CONTACTS.sort(orderBy == OrderBy.NAME ? NAME_ORDER : EMAIL_ORDER);
}
```

Roteiro 06 – Atividade 05/05

Com base no que vimos até agora...

1. Modele e implemente operações “finders” (nome e email) na classe `ContactBook`. No roteiro 2, foi mostrado um exemplo sobre como isso pode ser feito.

2. Faça uma versão da classe `ContactBook`, utilizando a classe `HashMap<K, V>`, `K` poderia ser uma String representando o email e `V`, a referência `Contactable`. Da mesma forma que a classe ArrayList implementa a interface `List`, a classe `HashMap` implementa a interface `Map<K, V>`. Veja mais sobre esta classe na documentação javadoc da API Java: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. Veja também nos tutoriais do Java: https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html.

3. Pesquise sobre métodos default em interfaces. Este recurso está disponível na API Java somente a partir da versão 8. Implemente testes para verificar o comportamento. Explique suas vantagens e limitações. Um bom ponto de partida é http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html.