Roteiro 05 – Atividade 01/06

1. Vamos iniciar nosso roteiro de hoje com algumas melhorias no nosso estudo de caso do sistema acadêmico. Observe que as classes `Student` e `Employee` possuem um controle próprio de identificação: `rollNumber` e `id`, respectivamente. Além destes valores serem preenchidos manualmente, há também a questão de análise semântica sobre eles. Eles têm exatamente o mesmo objetivo e que não há uma regra específica para cada um. Logo, podemos pensar em **generalizar** este controle para a classe `Person`. Além disso, seria adequado implementarmos uma solução que gere automaticamente os valores dos identificadores, garantindo pelo programa que eles serão únicos.

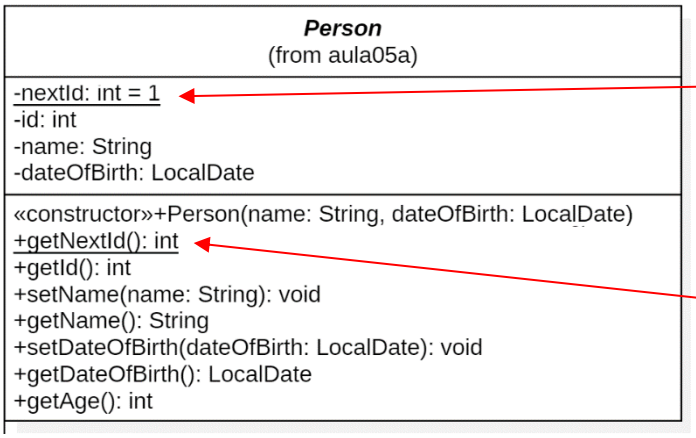
Antes de generalizar uma operação ou atributo (levar para uma superclasse), tenha certeza que o significado (semântica) e as regras são as mesmas para todas as classes especializadas (subclasses). Evite antecipar demais aquilo que pode ser generalizado sem uma análise mais profunda sobre o problema. Inicie sua modelagem com as classes que podem ser retiradas diretamente do negócio (ex: **`Student`**, **`Employee`**, **`Professor`**). Depois, verifique o que elas têm realmente e comum (mesmo significado). A partir desta análise é que você deveria partir para a generalização.

No nosso estudo de caso, iniciamos com a classe Person por questões didáticas. Em um sistema real, teríamos que entender o problema a ser resolvido e especificar os requisitos de uma solução de software para ele. Evite prejulgamentos, tentando antecipar os membros de uma classe terá sem que você tenha, pelo menos, um conhecimento inicial sobre o que você deve resolver. Sua formação aqui não é apenas de um programador, mais de um profissional capaz de analisar problemas, projetar alternativas de solução, escolher aquela com melhor custo/benefício, implementá-la e garantir que os usuários estão satisfeitos com o resultado.

Continuando a análise do nosso estudo de caso, podemos identificar alguma similaridade também entre as datas de matrícula/desligamento e admissão/demissão. Porém, elas têm semânticas diferentes e não iremos modificá-las por enquanto.

Colocar o atributo `id` na classe `Person` deve ser uma tarefa muito simples para você agora. Observe que só teremos a operação getter, pois o valor do `id` não deve ser modificado manualmente. Além disso, não esqueça de excluir os respectivos atributos e operações get/set nas classes `Student` e `Employee` (a classe `Professor` já herdava de `Employee`). Atualize o diagrama de classe e seu programa Java.

2. Precisamos modelar e implementar agora uma solução para gerar valores únicos para nossos identificadores. Neste momento, adotaremos uma solução simples com o uso de um atributo de classe (estático). O valor de um atributo de classe é compartilhado por todos os objetos de uma classe, uma vez que o atributo é da classe e não do objeto. Logo, ao incrementarmos este atributo no construtor de `Person` e atribuirmos o valor ao atributo `id`, teremos sempre um novo valor (mesmo que objetos criados anteriormente tenham sido coletados pelo *garbage collector*). Com o uso de `super` em nossos construtores de `Student`, `Employee` e `Professor`, também garantimos que este comportamento será herdado por estas classes (lembre-se que `Person` é abstrata e não podemos criar objetos a partir dela). Ajuste sua modelagem, implemente e teste a alteração.



Sempre terá o valor a ser utilizado no próximo identificador. O primeiro valor disponível será 1. Esta é a notação UML para um atributo de classe (estático).

Esta operação de classe (estática) permite apenas consultar qual o valor que será utilizado na próxima instanciação. **Operações de classe só podem manipular**

Roteiro 05 – Atividade 01/06

3. Se seu código ficou muito diferente da imagem abaixo, faça uma revisão para ter certeza que a solução foi implementada corretamente.

```
public abstract class Person {

    /**
     * Próximo ID a ser utilizado.
     */
    private static int nextId = 1;
    /**
     * ID único para cada objeto Pessoa (gerado automaticamente).
     */
    private int id;
    /**
     * Nome da pessoa.
     */
    private String name;
    /**
     * Data de nascimento da pessoa.
     */
    private LocalDate dateOfBirth;

    /**
     * Construtor para ser reutilizado pelas subclasses de
     * Person. Nome e data de nascimento são obrigatórios.
     *
     * @param name o nome da pessoa
     * @param dateOfBirth a data de nascimento da pessoa
     */
    public Person(String name, LocalDate dateOfBirth) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
        validateState();
        id = nextId++;
    }
```

Deixe a geração do ID para a última linha do seu construtor. Se você implementou a validação discutida no roteiro anterior, isso fará sentido.

4. Vamos agora modelar e implementar um construtor alternativo nas nossas classes. Se você executou os roteiros anteriores, deve ter percebido que os métodos de teste de criação dos objetos recebiam parâmetros do tipo String que eram então convertidos para os tipos corretos antes de serem passados aos construtores. Bom, por que não sobrecarregarmos um segundo construtor que faz esta conversão implicitamente para o método chamador? Veja a solução abaixo para `Person` e replique a solução em `Student`, `Employee` e `Professor`.

Person (from aula05a)
-nextId: int = 1 -id: int -name: String -dateOfBirth: LocalDate
«constructor»+Person(name: String, dateOfBirth: LocalDate) «constructor»+Person(name: String, dateOfBirth: String) +getNextId(): int +getId(): int +setName(name: String): void +getName(): String +setDateOfBirth(dateOfBirth: LocalDate): void +getDateOfBirth(): LocalDate +getAge(): int

```
/**
 * Construtor para ser reutilizado pelas subclasses de
 * Person. Utiliza formato string na data de nascimento para
 * facilitar chamada. Nome e data de nascimento são obrigatórios.
 *
 * @param name o nome da pessoa
 * @param dateOfBirth a data de nascimento da pessoa
 */
public Person(String name, String dateOfBirth) {
    this(name, LocalDate.parse(dateOfBirth, DateTimeFormatter.ofPattern("dd/MM/yyyy")));
}
```

5. Podemos ainda utilizar um atributo de classe para criar uma constante com o formato brasileiro de data. Desta forma, você pode utilizar esta constante também nos construtores de `Student`, `Employee` e `Professor`. Faça as alterações no diagrama e no seu código (não esqueça de utilizar a constante em seus construtores). Lembre-se que você também precisará converter valores **BigDecimal**.

Person (from aula05a)
+DATE_FORMAT_DDMMYYYY: DateTimeFormatter {readOnly} -nextId: int = 1 -id: int -name: String -dateOfBirth: LocalDate
«constructor»+Person(name: String, dateOfBirth: LocalDate) «constructor»+Person(name: String, dateOfBirth: String) +getNextId(): int +getId(): int +setName(name: String): void +getName(): String +setDateOfBirth(dateOfBirth: LocalDate): void +getDateOfBirth(): LocalDate +getAge(): int

```
/**
 * Constante com formato brasileiro de data.
 */
public static final DateTimeFormatter DATE_FORMAT_DDMMYYYY =
        DateTimeFormatter.ofPattern("dd/MM/yyyy");
```

Se você fez tudo certo, seus objetos estão recebendo um identificador único no momento em que eles são instanciados. Isso vale tanto para objetos `Student`, `Employee` e `Professor`.

Roteiro 05 – Atividade 01/06

6. No passo anterior, nós introduzimos a constante `DATE_FORMAT_DDMMYYYY` em `Person` para que ela fosse utilizada também nas subclasses. Entretanto, ao analisar a abstração, percebemos que perdemos coesão. Note que uma constante de formato de data não é uma informação de uma pessoa, mas de configuração do sistema. Logo, podemos melhorar nosso código ao separar esta informação de `Person` e colocá-la em uma classe que poderia ser utilizada por toda a aplicação. Para tal, vamos criar uma classe `AppConfig` que será responsável por conhecer informações gerais sobre a nossa aplicação. Uma solução seria utilizarmos uma classe para agrupar um conjunto de constantes (`static` + `final`) relacionadas com a configuração geral do sistema. Cuidado para não colocar todas as constantes da aplicação nesta classe, o que seria o oposto de coesão.

```java
/**
 * Configuração geral da aplicação.
 *
 * @author Marcello Thiry
 */
public class AppConfig  {

    /**
     * Nome da aplicação.
     */
    public static final String APP_NAME = "Sistema Acadêmico";
    /**
     * Versão da aplicação.
     */
    public static final String APP_VERSION = "1.0";
    /**
     * Constante com formato brasileiro de data.
     */
    public static final DateTimeFormatter DATE_FORMAT = DateTimeFormatter.ofPattern("dd/MM/yyyy");
    /**
     * Constante com formato de quebra de linha, conforme sistema operacional utilizado.
     */
    public static final String NEW_LINE = System.getProperty("line.separator");

}
```

Propriedade do sistema que retorna o código de quebra de linha utilizado no sistema operacional em que a aplicação está sendo executada. Por exemplo, o no Windows, o código de quebra de linha é diferente do Linux. Para mais detalhes sobre propriedades de System, acesse o link https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html

Embora funcional, a classe precisaria ser modificada a cada nova aplicação desenvolvida. Por ser uma série de constantes predefinidas, não seria possível, por exemplo, carregar a configuração a partir de um arquivo. Uma alternativa seria utilizarmos apenas o modificador `final` nos atributos, inicializando as constantes em um construtor. Entretanto, esta solução também é problemática, pois ela não garante uma única instância da classe. Programadores poderiam criar diferentes configurações ao longo da aplicação, quebrando totalmente a ideia inicial. A solução que adoremos segue a lógica da primeira, mas com a flexibilidade de carregarmos as configurações na hora que o sistema de execução JVM carregar as classes da aplicação.

```java
public final class AppConfig {

    /** Nome da aplicação ...3 lines */
    public static final String APP_NAME;
    /** Versão da aplicação ...3 lines */
    public static final String APP_VERSION;
    /** Constante com formato brasileiro de data ...3 lines */
    public static final DateTimeFormatter DATE_FORMAT;
    /** Constante com código de quebra de linha, conforme SO utilizado ...3 lines */
    public static final String NEW_LINE;
    /**
     * Armazenar as configurações lidas de alguma fonte.
     */
    private static final Object SETTINGS[] = new Object[10];

    /**
     * Um bloco static é um bloco normal de código cercado por chaves.
     * Uma classe pode ter vários blocos static, os quais podem aparecer em qualquer
     * local do corpo da classe. O sistema de execução JVM garante que a inicialização
     * feita nestes blocos seja feita na ordem em que eles aparecem no código.
     * Os atributos static final devem ser inicializados aqui.
     */
    static {
        loadSettings();
        APP_NAME = (String)SETTINGS[0];
        APP_VERSION = (String)SETTINGS[1];
        DATE_FORMAT = (DateTimeFormatter)SETTINGS[2];
        NEW_LINE = (String)SETTINGS[3];
    }

    /**
     * Carrega as configurações de alguma fonte como, por exemplo, um
     * arquivo XML ou de propriedades do sistema.
     */
    private static void loadSettings() {
        // Inserir aqui o código para carregar preencher settings a
        // partir da alguma fonte
        SETTINGS[0] = "Sistema Acadêmico";
        SETTINGS[1] = "1.0";
        SETTINGS[2] = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        SETTINGS[3] = System.getProperty("line.separator");
    }

}
```

Lista interna para armazenar as configurações que serão atribuídas às nossas constantes `static final`.

Utilizamos um bloco `static` (leia o javadoc) para inicializar as nossas constantes. Nesta solução, podemos chamar uma rotina que carregará, de alguma fonte, os valores que serão atribuídos.

Roteiro 05 – Atividade 01/06

Embora adotaremos esta solução neste momento, ela também tem limitações. Veja que não é possível alterar a configuração da aplicação em tempo de execução. Isso pode ser necessário, por exemplo, em aplicações que rodam 24x7 (24 horas por dia, todos os dias da semana). Na medida em que avançarmos na disciplina, você terá mais recursos para projetar outras soluções.

Agora, vamos ver como utilizar a nossa nova classe em nossa aplicação. Não foi mostrado aqui, mas você deve excluir a linha que define a constante na classe `Person`.

```java
/**
 * Construtor para ser reutilizado pelas subclasses de
 * Person. Nome e data de nascimento são obrigatórios.
 *
 * @param name o nome da pessoa
 * @param dateOfBirth a data de nascimento da pessoa
 */
public Person(String name, LocalDate dateOfBirth) {
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    validateState();
    id = nextId++;
}

/**
 * Construtor para ser reutilizado pelas subclasses de
 * Person. Utiliza formato string na data de nascimento para
 * facilitar chamada. Nome e data de nascimento são obrigatórios.
 *
 * @param name o nome da pessoa
 * @param dateOfBirth a data de nascimento da pessoa
 */
public Person(String name, String dateOfBirth) {
    this(name, LocalDate.parse(dateOfBirth, AppConfig.DATE_FORMAT));
}
```

Utilizando nossa nova classe de configuração.

Roteiro 05 – Atividade 02/06

1. Abra o arquivo **br.univali.kob.poo1.aula05a.StaticIDTest.java**, disponibilizado com este roteiro. Este arquivo mostra uma classe de teste para a atividade 01 deste roteiro. Entretanto, há uma diferença em relação ao que foi visto até o momento. Foi utilizada uma classe Java que implementa uma lista: **java.util.ArrayList**.

```
private final ArrayList<Person> PERSON_LIST = new ArrayList<>();
```

A classe **ArrayList** (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) é uma das várias classes disponíveis na **API**[1] Java para trabalhar com coleções de objetos (*collections*). Esta lista cresce dinamicamente conforme novos objetos são adicionados a ela. Como ela utiliza um vetor interno para armazenar os elementos, ela pode não ser adequada em situações onde desempenho é um requisito crítico do sistema (necessidade de processamento interno sempre que o vetor precisa ser expandido). Uma forma de minimizar este problema é instanciar a lista com uma capacidade inicial mais adequada à necessidade do programa (um dos construtores desta classe permite que você especifique a capacidade inicial). Existem outras questões que devem ser analisadas para a escolha da classe **Collection** a ser utilizada, mas elas não são pertinentes neste momento. Algumas opções que vale a pena conhecer:

- Vector (similar ao ArrayList) - http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html.
- LinkedList (duplamente encadeada) - http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html.
- Stack (pilha/LIFO) - http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html.
- HashSet (conjunto) - http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.

Você verá vários conceitos de listas na disciplina de Estrutura de Dados. Caso você queira depois estudar mais a fundo coleções em Java, veja o link abaixo:

- http://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html.

No nosso caso, utilizaremos **ArrayList** como nossa classe **Collection default**. Algumas observações para utilizarmos uma lista ArrayList:

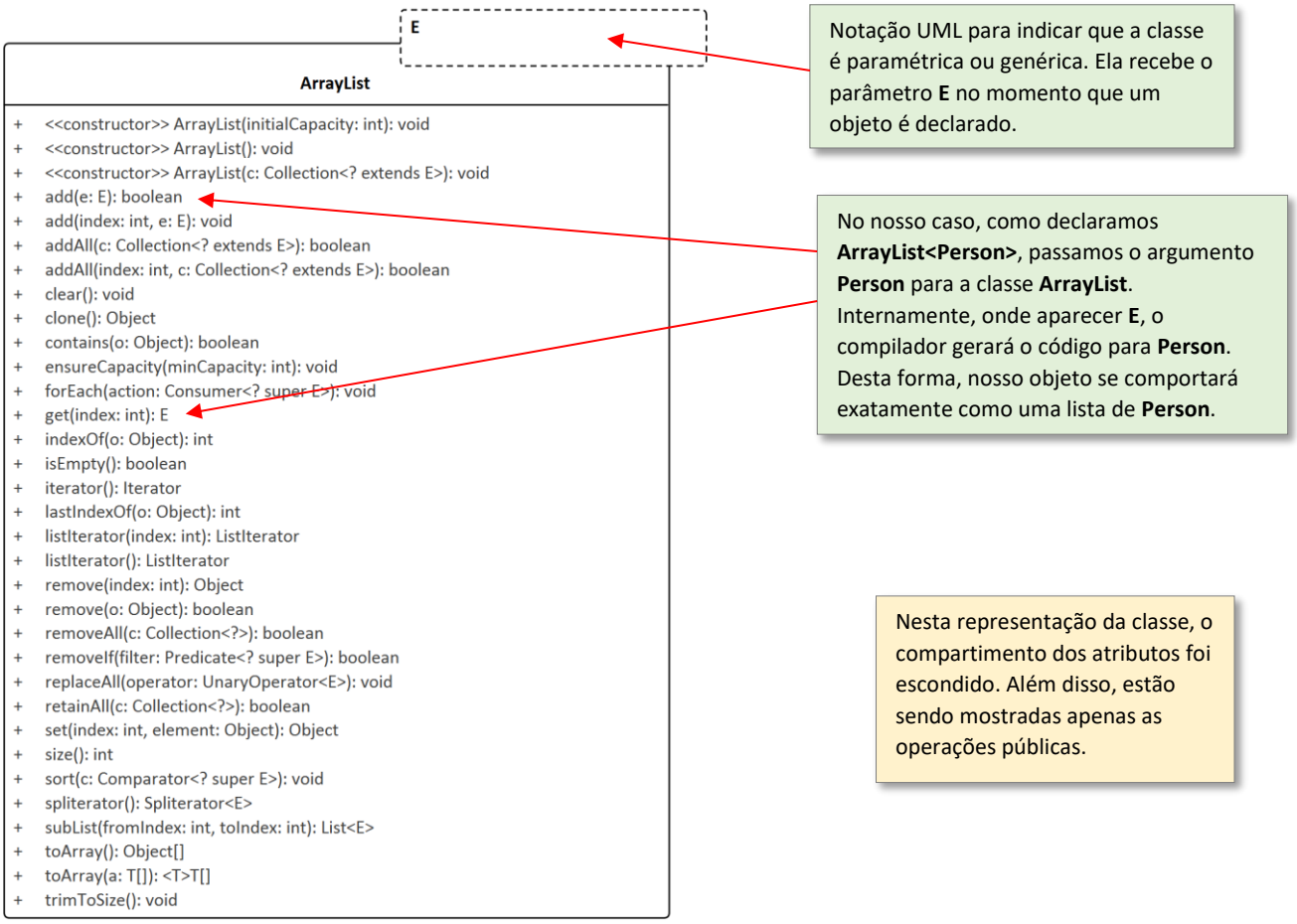
- Aceita apenas objetos. Se você quiser utilizar tipos primitivos, será necessário trabalhar com classes wrapper (Integer, Float, etc.).
- Aceita elementos null.
- Ela é uma classe genérica, pois funciona com qualquer tipo passado como parâmetro para ela (**polimorfismo paramétrico**). Veja que nosso exemplo, a lista foi declarada com um parâmetro para a classe **<Person>**. Este parâmetro faz com que a classe se comporte como uma lista de pessoas. Da mesma forma, ela poderia se comportar como uma lista de livros se fosse declarada como **ArrayList<Book>**. Todas as operações que manipulam elementos considerarão que o elemento é do mesmo tipo que aquele passado como parâmetro na declaração. Em um roteiro futuro, iremos aprender como criar a nossa própria classe paramétrica (genérica).

As operações tipicamente mais utilizadas são (considerando que utilizamos o parâmetro **Person**):

- `boolean add(Person element)` adiciona o elemento no final.
- `void add(int index, Person element)` insere o elemento na posição indicada.
- `void clear()` remove todos os elementos.
- `boolean contains(Object element)` verifica se a lista contém o elemento.
- `Person get(int index)` retorna o elemento que estiver na posição indicada.
- `int indexOf(Object element)` retorna a posição da primeira ocorrência do elemento.
- `boolean isEmpty()`: Retorna verdadeiro se a lista estiver vazia e falso caso contrário.
- `int lastIndexOf(Object element)` Retorna a posição da última ocorrência do elemento.
- `Object remove(int index)` remove o elemento que estiver na posição indicada.
- `Object set(int index, Object element)` substitui o elemento na posição indicada.
- `int size()` retorna o número de elementos.

[1] API (Interface de Programação de Aplicação, do Inglês *Application Programming Interface*) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização de suas funcionalidades por aplicativos sem que estes precisem conhecer detalhes de implementação.

Roteiro 05 – Atividade 02/06



2. Você também deve ter notado o seguinte trecho de código

```
if (PERSON_LIST.get(0) instanceof Employee) {
    ((Employee)PERSON_LIST.get(0)).setTerminationDate(LocalDate.parse("09/08/2001", DATE_FORMAT));
}
```

Uma vez que estamos utilizando polimorfismo, o compilador espera que todos os objetos da lista sejam Person. Logo, ela não conheceria a operação `setTerminationDate`, a qual é da classe `Employee`. Para ter acesso às operações implementadas (métodos) de `Employee`, foi necessário induzir o tipo (**type cast**). O retorno da expressão `(Employee)PERSON_LIST.get(0)` será uma referência `Employee`, desde que o objeto na posição 0 seja realmente um `Employee` ou de uma subclasse de `Employee` (como `Professor`). Caso contrário, teremos uma exceção não checada (*runtime*) **ClassCastException**.

Existem dois tipos de **type casting**:

- **Upcasting** é a indução de uma referência para uma superclasse (tipo acima).

```
Person person = new Employee();
```

- **Downcasting** é a indução de uma referência para uma subclasse (tipo abaixo).

```
Employee employee = (Employee)person;
```

Upcasting é sempre permitido, mas downcasting exige que o tipo seja verificado em tempo de execução. Se a verificação falhar, o sistema dispara a exceção não checada `ClassCastException`.

Se você precisar verificar de qual classe uma determinada instância pertence em tempo de execução, você pode utilizar a palavra reservada `instanceof`. No exemplo, verificamos se o elemento da posição 0 é realmente uma instância de `Employee` antes de forçar o tipo. Tome cuidado com o uso indiscriminado de `instanceof` e **downcasting** no seu programa. Isso pode indicar um problema no projeto da sua solução. Nestes casos, analise

Roteiro 05 – Atividade 02/06

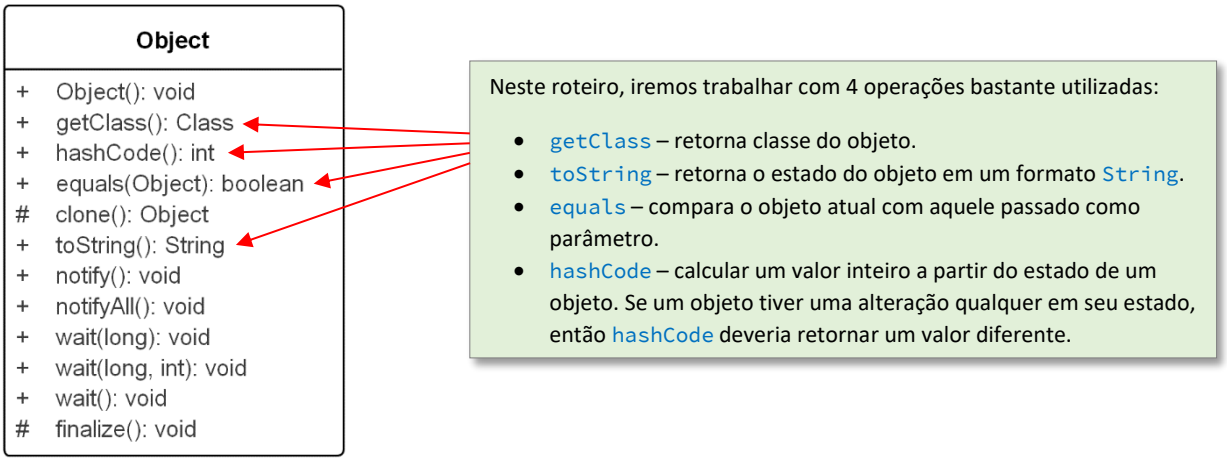
sua modelagem e respectivo código para verificar se existe espaço para **refatoração** (***refactoring***)[2]. **Refatoração** é ação de modificar um software com o objetivo de melhorar seu código sem alterar seu comportamento externo.

3. A última parte interessante deste exemplo está na forma em que a lista foi percorrida. Neste caso, não há necessidade de variável de controle e de verificar se chegamos ao fim da lista. A própria linguagem oferece um mecanismo para percorrer a lista até que todos os elementos tenham sido acessados. Vale notar que a variável `person` só existe no contexto do laço. Veja também os slides de referência que faziam parte do primeiro roteiro como você pode percorrer a lista utilizando `Iterator`. Há ainda outra alternativa onde podemos utilizar uma função lâmbda, mas deixaremos esta discussão para outro roteiro.

[2] Uma tradução mais adequada poderia ter sido *refabricação*, pois a proposta é melhorar a forma com a qual o código foi fabricado sem modificar, porém, seu comportamento externo. Entretanto, refatoração é o termo utilizado pela comunidade brasileira.

Roteiro 05 – Atividade 03/06

1. Nesta atividade, conheceremos em mais detalhes a classe `Object`. Se você tem feito todas as atividades e exercícios indicados, é impossível não ter passado alguma vez por ela. Se você ainda duvida, volte na notação UML da classe `ArrayList` e dê uma olhada novamente nas operações. Na imagem abaixo, podemos visualizar a classe `Object` e suas operações (novamente, o compartimento dos atributos foi escondido).



Experimente executar o código abaixo:

```
Employee el = new Employee("Orin Curry", "15/01/1976", "01/03/2000", 40, "22.80");
System.out.println("o estado de el é: " + el.toString());
```

Sua saída deve ter sido algo similar a:

```
o estado de el é: br.univali.kob.poo1.aula05b.Employee@4769b07b
```

O que aconteceu? Não era para o estado (valores dos atributos) ter sido impresso? Antes de entrarmos nesta questão, experimente adicionar também o seguinte código e execute novamente:

```
System.out.println(el.getClass().getName() + "@" + Integer.toHexString(el.hashCode()));
```

Você percebeu que os valores são iguais? Ou seja, a implementação default (herdada de `Object`) de `toString` apenas concatena o nome da classe do objeto `getClass().getName()` com o valor retornado de `hashCode` no formato hexadecimal `Integer.toHexString(int)`. Ele ainda separa os dois valores com o caractere "@". Ou seja, para o uso desta operação na prática, precisaremos redefini-la (`@Override`).

A ideia básica é retornar o estado do objeto como uma `String`, deixando o formato desta `String` como uma decisão de projeto. Esta operação é tipicamente utilizada para a depuração de classes durante o desenvolvimento, podendo ser passada em construtores de exceções ou para operações de log. Logo, tome cuidado se você pretende confiar no formato retornado. Verifique se ele está formalmente definido no **javadoc**. Caso você queira utilizar um modo formal, você deve fazer o mesmo.

Daremos prioridade ao uso da classe `StringBuilder`, pois ela simplifica a criação de *strings* mais complexas e deixa o código mais legível. Neste roteiro, adotaremos o seguinte formato:

```
br.univali.kob.poo1.aula05b.Employee {
  // Person
  id = 1
  name = Orin Curry
  dateOfBirth = 15/01/1976
  // Employee
  hireDate = 01/03/2000
  terminationDate = 09/08/2001
  hoursPerWorkWeek = 40
  hourlyRate = 22.80
}
```

```
Nome_da_Classe_Completo {
  Atributo = valor
  Atributo = valor
  ...
}
```

Roteiro 05 – Atividade 03/06

2. Para implementarmos o `toString` da nossa classe Person, utilizaremos o conceito de operação abstrata. Uma **operação abstrata** não possui um método (não tem implementação) na classe em que ela é definida. Operações abstratas só podem ser declaradas em classes abstratas, pois não seria possível instanciar um objeto sem que todas as suas operações estejam implementadas (tenham métodos). A declaração de uma operação abstrata obriga que todas as subclasses diretas a implementem (criar um método para ela). Passamos a ter a ideia de um **contrato**, pois quando decidimos que uma classe será uma subclasse da classe que possui a operação abstrata, estamos assumindo o dever de cumprir este contrato, ou seja, implementar todas as operações abstratas. Veja a imagem abaixo para a redefinição implementada de `toString` na classe `Person`.

```
/**
 * Monta uma String com todos os atributos e seus respectivos valores.
 * O formato retornado é:
 *
 * <pre>
 * {@code
 * Nome_da_Classe_Completo {
 *   Atributo = valor
 *   Atributo = valor
 *   ...
 * }}
 * </pre>
 *
 * @return o estado (atributos e respectivos valores) do objeto
 */
@Override
public String toString() {
    StringBuilder output = new StringBuilder();
    output.append(this.getClass().getName() + " {" + AppConfig.NEW_LINE);
    output.append("  // Person" + AppConfig.NEW_LINE);
    output.append("  id = " + id + AppConfig.NEW_LINE);
    output.append("  name = " + name + AppConfig.NEW_LINE);
    output.append("  dateOfBirth = " +
            dateOfBirth.format(AppConfig.DATE_FORMAT) + AppConfig.NEW_LINE);
    output.append(appendToString());
    output.append("}" + AppConfig.NEW_LINE);
    return output.toString();
}

/**
 * Cada subclasse de Person deve montar a string que será
 * adicionada à implementação herdada de toString(). Cada
 * atributo com seu respectivo valor deve estar em uma linha
 * separada. ver formato esperado da operação.
 *
 * @return um conjunto de linhas atributo = valor que será incluído
 *         na implementação herdada de toString().
 * @see #toString() para detalhes sobre o formato das linhas
 */
protected abstract String appendToString();
```

Um javadoc bem documentado na superclasse permite que programadores usuários de sua classe saibam o que esperar de `toString`. Além disso, você não precisará escrever o javadoc para as subclasses, a não ser que alguma coisa seja mudada a partir daquela classe

Utilizando nossa classe de configuração.

Declaração de uma operação abstrata em Java. Veja que não há método definido para ela. Mas como ela pode ser invocada em `toString` sem que ela tenha sido implementada? Alguma ideia? Se você está perdido, leia novamente sobre o conceito de **ligação tardia** quando discutimos o conceito de polimorfismo.

O que fizemos foi, basicamente, criar uma linha para cada atributo, concatenando seu nome com seu valor. A linha mais relevante da nossa implementação de `toString` é a invocação à operação `appendToString`. Como `Person` é abstrata, a operação `toString` nunca poderá ser invocada de um objeto `Person`. Como consequência, a operação `appendToString` também não será invocada (o que é esperado, pois ela não tem um método ainda).

Roteiro 05 – Atividade 03/06

Entretanto, qualquer subclasse direta de `Person` (ex: `Student` e `Employee`) será obrigada a implementar a operação `appendToString`. Ao redefinir esta operação, a subclasse deve apenas retornar suas linhas "atributo = valor". Todo o resto será feito pela implementação de `toString` em `Person`. Com esta solução, não precisaremos redefinir `toString` nas subclasses de Person, mas apenas `appendToString`. O mecanismo de ligação tardia será responsável por fazer a ligação em tempo de execução da chamada à operação `appendToString` com a implementação correta de acordo com a classe do objeto que está sendo invocado. Este tipo de solução é um **padrão de projeto** (*design pattern*) chamado **Método *Template***[3].

O padrão de projeto **Método *Template*** define o corpo principal de um algoritmo (*template* de um método), delegando a implementação de um ou mais passos deste algoritmo para as subclasses. O termo *template* aqui tem o significado de um modelo ou molde básico.

Para garantir que as subclasses implementem os passos, o método *template* invoca operações abstratas. Desta forma, cada subclasse pode implementar estes passos de modo diferente, permitindo variações de comportamento. Entretanto, a estrutura básica (template) do algoritmo é mantida.

A ideia é inverter a invocação. Ao invés de uma subclasse invocar um método herdado de sua superclasse, é o método herdado da superclasse que invoca um método implementado na subclasse.

3. Você lembrou de atualizar seu diagrama de classe? Se você esqueceu, este é um bom momento para colocar tudo em dia. Mantenha sua documentação atualizada para que ela seja realmente útil.

4. Agora vamos implementar a operação `appendToString` em `Employee`.

```
@Override
protected String appendToString() {
    StringBuilder output = new StringBuilder();
    output.append("  // Employee " + AppConfig.NEW_LINE);
    output.append("  hireDate = " + hireDate.format(AppConfig.DATE_FORMAT) + AppConfig.NEW_LINE);
    output.append("  terminationDate = " );
    output.append(((terminationDate == null) ? null : terminationDate.format(AppConfig.DATE_FORMAT)) + AppConfig.NEW_LINE);
    output.append("  hoursPerWorkWeek = " + hoursPerWorkWeek + AppConfig.NEW_LINE);
    output.append("  hourlyRate = " + hourlyRate.toPlainString() + AppConfig.NEW_LINE);
    return output.toString();
}
```

Nenhuma novidade aqui. Só adicionamos os atributos da própria classe. Todo o resto será garantido pela implementação (método *template*) de `toString` em `Person`.

5. A implementação para a classe `Professor` tem uma única diferença: a string a ser retornada é inicializada com a o retorno da implementação de `appendToString` na superclasse (neste caso, `Employee`). Desta forma, garantimos que a cada subclasse de `Employee`, teremos linhas adicionais (dependendo da quantidade de atributos), mas sem perder as linhas adicionadas pela implementação anterior.

```
@Override
protected String appendToString() {
    StringBuilder output = new StringBuilder(super.appendToString());
    output.append("  // Professor" + AppConfig.NEW_LINE);
    output.append("  academicDegree = " + academicDegree + AppConfig.NEW_LINE);
    return output.toString();
}
```

6. Agora implemente `appendToString` na classe `Student`. Depois, ajuste o teste da lista para que ele passe por todos os objetos e imprima apenas `person.toString()`. Analise os resultados e volte nos pontos que você ficou em dúvida.

[3] Não confundir com Template em C++, o qual tem o mesmo significado de uma classe genérica (generics) em Java (polimorfismo paramétrico).

Roteiro 05 – Atividade 03/06

7. A operação `equals` define uma relação de equivalência entre o objeto atual e aquele passado como parâmetro. Ela compara o estado (valor) do objeto atual com o objeto passado como parâmetro. Caso o objeto passado seja `null`, a comparação falhará. Se o objeto passado apontar para o mesmo endereço `(a == b)`, a comparação retornará sucesso. Se os objetos não pertencerem à mesma classe, a comparação falhará. Ou seja, ao ser redefinida (`@Override`) por uma classe, precisamos garantir que `equals` deva **SEMPRE**:

   - Ser **reflexiva**: `a.equals(a)` é sempre verdadeiro.
   - Ser **simétrica**: se `a.equals(b)`, então `b.equals(a)`.
   - Ser **transitiva**: se `a.equals(b)` e `b.equals(c)`, então `a.equals(c)`.
   - Ser **consistente**: se os objetos comparados não foram modificados, o valor retornado deve sempre ser o mesmo.
   - **Retornar *false* quando a comparação for feita com null**: `a.equals(null)` (pois nunca poderemos invocar um método a partir de um objeto `null`).
   - **Retornar false quando os objetos não pertencerem a mesma classe** (uma vez que não são comparáveis).

   Para atender a estas regras, adotaremos um algoritmo base que servirá como orientação para implementação de `equals` em qualquer classe. Obviamente, adaptações poderão ser necessárias, dependendo de como a classe implementa suas operações de acesso.

```
@Override
public boolean equals(Object obj) {

    if (obj == this) {
        return true;
    }

    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    MyClass c = (MyClasse)obj;

    return
        (atributo1 == c.atributo1 || (atributo1 != null && atributo1.equals(c.atributo1))) &&
        (atributo2 == c.atributo2 || (atributo2 != null && atributo2.equals(c.atributo2))) &&
        ...
        (atributoN == c.atributoN || (atributoN != null && atributoN.equals(getAtributoN)));
}
```

Teste mais eficiente. Se eles referenciam o mesmo objeto, já retorna `true`.

Se o objeto a ser comparado é `null` OU se eles não pertencem a mesma classe, já retorna `false`.

Typecast simplifica as comparações. Veja que o parâmetro na assinatura de `equals` é `Object`.

Todos os atributos devem ser comparados.

Primeira coisa é verificar se os atributos referenciam o mesmo objeto. Note que estamos acessando diretamente o atributo do objeto passado. Lembre-se que estamos na classe deste objeto, logo, temos acesso aos seus atributos.

Caso contrário, se o atributo não for `null`, compara com o atributo do objeto passado.

No caso de uma superclasse já ter redefinido `equals`, a subclasse pode ir direto para comparação dos atributos (claro que antes, ela precisa verificar se `super.equals` retornou `true`). Veremos isso com a implementação de `equals` em `Person` e `Employee`.

Roteiro 05 – Atividade 03/06

```java
/**
 * Define uma relação de equivalência entre o objeto atual e aquele passado
 * como parâmetro. Esta operação deve atender às seguintes regras:
 *
 * 1) Reflexiva: a.equals(a) é sempre verdadeiro.
 * 2) Simétrica: se a.equals(b), então b.equals(a).
 * 3) Transitiva: se a.equals(b) e b.equals(c)m então a.equals(c).
 * 4) Consistente: se os objetos não foram modificados, o valor retornado
 *    deve sempre ser o mesmo.
 * 5) a.equals(null) sempre retorna false.
 *
 * Compara o estado (valor) do objeto atual com o objeto passado como
 * parâmetro. Caso o objeto passado seja null, a comparação falhará. Se o
 * objeto passado apontar para o mesmo endereço (a == b), a comparação
 * retornará sucesso. Se os objetos não pertencerem à mesma classe, a
 * comparação falhará.
 *
 * @param obj o objeto a ser comparado
 * @return true quando os objetos têm o mesmo estado, false caso contrário
 */
@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    Person person = (Person)obj;
    // considerado que dateOfBirth nunca poderá ser null (validações)
    // note que estamos na classe Person, logo temos acesso também aos
    // atributos do outro objeto
    return
        id == person.id &&
        (name == person.name || name.equals(person.name)) &&
        (dateOfBirth == person.dateOfBirth || dateOfBirth.equals(person.dateOfBirth));
}
```

Note que `id` também está sendo comparado aqui. Formalmente, devemos comparar todos os atributos. Entretanto, como nosso `id` é gerado automaticamente, dois objetos `Person` nunca deverão retornar `true` em `equals`. Durante os testes, experimente deixar a linha que incrementa `nextId` no construtor comentada. Assim, você conseguirá simular a equivalência. Só não esqueça de retirar o comentário depois.

Veja que a comparação entre tipos primitivos é direta (apenas com `==`). Também não foi verificado se os atributos eram `null` antes da comparação. Se você implementou as validações, esta situação será impossível.

Agora, vamos ver como fica a implementação em uma subclasse (`Employee`):

```java
@Override
public boolean equals(Object obj) {
    if (!super.equals(obj)) {
        return false;
    }
    Employee employee = (Employee)obj;
    return
        // testa primeiro com == para ser mais eficiente
        (hireDate == employee.hireDate || hireDate.equals(employee.hireDate)) &&
        Objects.equals(terminationDate, employee.terminationDate) &&
        hoursPerWorkWeek == employee.hoursPerWorkWeek &&
        (hourlyRate == employee.hourlyRate || hourlyRate.equals(employee.hourlyRate));
}
```

Não precisa repetir os testes feitos na superclasse.

O único atributo que pode ser `null` em `Employee` é `terminationDate`. Observe que foi utilizado `Objects.equals(obj1, obj2)` para a comparação. Esta operação aceita `null` na comparação.

8. Ajuste o diagrama de classe e redefina `equals` também nas classes `Student` e `Professor`. Faça testes para confirmar que tudo está funcionando como esperado.

9. A última operação de `Object` que redefiniremos neste roteiro é `hashCode`. Para ser consistente, você sempre deve redefinir `equals` e `hashCode` (nunca redefina somente uma delas). Um **código hash** (***hash code***) é o resultado de uma função **hash**. Funções **hash** calculam valores (representadas sempre por um tamanho fixo) que mapeiam um conjunto de dados de tamanho variável. Este tipo de função é muito utilizado em computação: indexação, criptografia, autenticação de mensagens, etc. Se o valor retornado de uma função **hash** é o mesmo para dois conjuntos de dados diferentes, então é altamente provável que eles sejam iguais. Logo, não faria sentido termos `harshCodes` diferentes para **a** e **b**, se `a.equals(b)` retornou `true`.

Roteiro 05 – Atividade 03/06

Para calcular um valor inteiro a partir do valor dos atributos do objeto (estado), podemos fazer um XOR (OU exclusivo) entre seus valores (você já viu isso em Matemática Computacional e/ou Circuitos Digitais). Na linguagem Java, o símbolo "^" representa o operador binário XOR. Para evitar uma exceção quando o valor do atributo for `null`, colocamos um valor (positivo, sem ser zero e preferencialmente primo) arbitrário. Números primos (2, 3, 5, 7, 11, 13, 17, etc.) reduzem a possibilidade de calcularmos um mesmo valor para objetos com estados diferentes. O operador ternário pode ser utilizado para deixar o código mais legível (a alternativa seria um encadeamento de ifs). Se você fez as atividades anteriores, nossas classes já estão protegidas com validações nos **setters**. Deste modo, apenas alguns atributos poderão ser nulos. Assumiremos isso na nossa implementação. A imagem abaixo mostra a estrutura genérica para a implementação de `hashCode`.

```
@Override
public int hashCode() {
    return
        super.hashCode() ^
        (atributo1 == null ? 17 : atributo1.hashCode()) ^
        (atributo2 == null ? 19 : atributo2.hashCode()) ^
        ...
        (atributoN == null ? 89 : atributoN.hashCode());
}
```

No caso de `Person`, nossa implementação pode ser mais simples, pois não há como termos atributos nulos (considerando que você implementou as atividades anteriores).

```
/**
 * Calcula um valor inteiro a partir do valor dos atributos do objeto
 * (estado), por meio de um XOR entre seus valores. Esta função deve ser
 * consistente, ou seja, se o estado do objeto não for alterado, o valor
 * retornado deve sempre ser o mesmo. Também deve manter a regra que sempre
 * que a.equals(b), então a.hashCode() deve ser igual a b.hashCode().
 *
 * @return o valor calculado a partir do estado atual do objeto
 */
@Override
public int hashCode() {
    return id ^ (name.hashCode()) ^ (dateOfBirth.hashCode());
}
```

O atributo `id` sempre terá um valor, pois é um tipo primitivo. Já os atributos `name` e `dateOfBirth` devem ter sido validados no construtor e nunca poderão ser `null`.

E na classe `Employee`:

```
@Override
public int hashCode() {
    return
        super.hashCode() ^
        hireDate.hashCode() ^
        (terminationDate == null ? 19 : terminationDate.hashCode()) ^
        hoursPerWorkWeek ^
        getHourlyRate().hashCode();
}
```

Note a invocação de `super`. Também observe que o único atributo que pode ser `null` é `terminationDate`.

10. Ajuste o diagrama de classe e redefina `hashCode` também nas classes `Student` e `Professor`. No caso de `Professor`, utilize a operação `ordinal()` no item `enum` (ela retorna um inteiro referente a posição do item na lista). Faça testes para confirmar que tudo está funcionando como esperado. Volte nas implementações `equals` anteriores e veja o que pode ser simplificado por causa da garantia fornecida pelas validações.

Como boa prática, **SEMPRE** redefiniremos as operações `toString`, `equals` e `hashCode`. Obviamente, você pode utilizar alguma solução similar a apresentada neste roteiro para implementar `toString` em uma hierarquia de herança. Garanta que todas as regras discutidas neste roteiro estarão sempre atendidas.