**Stack (Pilha):**

A Collections Stack (Pilha ) usa o método Last in/ First Out(o último que entra, é o primeiro que sai), e para declara uma pilha é:

Deque<String> livros = new ArrayDeque<>();

Aqui nos declaramos uma pilha do tipo String, e os métodos “.add”, “.poll”, “.remove”, “.element”, “.peek” das outras Collections também se aplicam na Collection Pilha, além também do “.size” para ver o tamanho da pilha, “.clear” para limpar a pilha, “.contains” para verificar se tem algo dentro daquela Collection, “.isEmpty” para verificar se aquela Collection está vazio, além de outros métodos. A diferença é que a pilha também tem o método “.push” para adicionar no começo(ou no caso da pilha, no topo dela) da pilha, e o “.add” vai adicionar no final(no caso da pilha, em baixo).

livros .add(“O Pequeno Príncipe”);

livros.push(“O Hobbit”);

O livro “O Hobbit” vai ser adicionado no começo da pilha(na teoria, no topo dela), e se você adicionar outro valor com o “.push”, ela via ficar antes do “O Hobbit”, e se você usar o “.add” vai ficar depois do “O Pequeno Príncipe”. E também tem o “.pop” que vai funcionar que vai funcionar da mesma forma que o “.remove”, vai remover o primeiro item da pilha, e se a pilha estiver vazia vai retornar um erro.

**Map:**

O Map é um objeto que mapeia valores para chaves, ou seja, através da chave consegue ser acessado o valor configurado, sendo que a chave não pode ser repetida ao contrário do valor, mas se caso tiver uma chave repetida é sobrescrito pela última chamada. Para declarar um Map é:

Map <Integer, String> usuarios = new HashMap<>();

A Collection Map não tem o “.add”, para você adicionar algo nele se usa o “.put”:

usuarios.put(1, “Roberto”);

Sendo o primeiro a chave, e o segundo o valor.

usuarios.put(20, “Ricardo ”);

usuarios.put(3, “Rafaela ”);

usuarios.put(4, “Rebeca ”);

Para ver todas as chaves do Map, se usa o “.keySet”:

System.out.println(usuarios .keySetl());

Ele vai imprimir como um Array com todas as chaves do Map. Para pegar apenas os valores se usa o “.values”:

System.out.println(usuarios .values());

Ele vai imprimir todos os valores do Map como “Roberto”, “Rafaela” e etc. E caso você queira pegar os dois valores se usa o “.entrySet”:

System.out.println(usuarios .entrySet());

Ele vai imprimir tanto a chave quanto o valor do Map.

Para remover um valor dentro do Map, você pode passar apenas a chave, que ele vai remover aquele item que tem a chave que foi passada, ou a chave e o valor, que ele só vai remover se o valor for referente aquela chave:

System.out.println(usuarios .remove(1));

Aqui ele vai remover o item cuja chave é igual a 1.

System.out.println(usuarios .remove(4, “Gui ”));

E aqui ele só vai remover se o valor da chave 4 for igual a “Gui”.

No caso do Map, não temos o “.contains” e sim o “.containsKey” e o “.containsValue” para verificar se contem um valor ou chave:

System.out.println(usuarios.containsKey(4));

Aqui estamos verificando se o Map usuarios tem a chave 4.

System.out.println(usuarios.containsValue(“Rebeca ”));

Aqui estamos verificando se o Map usuarios tem o valor “Rebeca ”.

E também podemos pegar um valor pela chave usando o “.get”:

System.out.println(usuarios.get (4));

Aqui retornara o valor “Rebeca”, que está na chave 4.

E temos 3 formas de percorrer um Map usando o forEach, pela chave, pelo valor ou pela chave e pelo valor:

for (int chave : usuarios .keySetl()) {

System.out.println(chave);

}

Aqui será retornado cada uma das chaves do Map usuarios.

for (String valor : usuarios.values()) {

System.out.println(chave);

}

Aqui será retornado cada uma dos valores do Map usuarios.

for (Map.Entry<Integer, String> registro : usuarios .entrySet()) {

System.out.println(chave.getKey);

System.out.println(chave.getValues);

}

Aqui será retornado cada uma das chaves e valores do Map usuarios.

**Orientação a Objetos**

**Composição**

Na composição codificamos pequenos comportamentos padrão e usamos composição para definir comportamentos mais complexos. Com isso podemos alterar a associação entre classes em tempo de execução, assim um objeto pode assumir mais de um comportamento.

Quando, em uma linguagem de programação, os objetos são tipados, os tipos geralmente podem ser divididos em tipos compostos e não-compostos e a composição pode ser considerada uma relação entre os tipos: um objeto de um tipo composto (e.g. *carro*) "tem um" objeto de um tipo mais simples (e.g. roda).

**Formas de identificar de quando usar composição:**

...TEM UM(A)...

* Carro TEM UM( motor.
* Carro TÊM portas.
* Casa TEM UMA uma cozinha.

**Pilares:**

**Encapsulamento:**

Conceitua-se **encapsulamento** como sendo o processo utilizado para proteger os campos e operações de uma classe (atributos e métodos), permitindo que apenas os membros públicos - em **Java** métodos Get / Set - sejam acessados pelos usuários de determinada classe.

Encapsulamento é o ato de privar(deixar oculto) algo que só pode ser acessado pelo próprio objeto, mas não pode ser acessado por outro objeto que faz referencia ao objeto que tem uma classe privada.

**Modificadores de Acesso:**

**Public:**

Ao criar um atributo publico, significa que qualquer classe do sistema pode acessar o atributo ou método publico de outra classe.

**Protected :**

Um atributo protegido é visível no próprio pacote, na própria classe e é transmitido por herança para um outro pacote, ou seja, ele só é visível em outro pacote desde que haja herança .

**Package(Default):**

Um atributo de nível pacote(padrão), só pode ser acessado por outra classe do mesmo pacote, e não por outras classes de pacotes diferentes.

**Private:**

Ao criar um atributo privado dentro da classe, ele só pode ser acessado dentro da classe, ou seja, não pode ser acessado por outras classes.

**Herança:**

Herança é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos, através de "heranças". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.

Na Herança uma classe pode herdar os atributos e métodos de sua superclasse. A herança é transitiva, então uma classe pode herdar de outra classe que herda de outra classe, e assim por diante, até uma classe base. Subclasses podem substituir alguns métodos e/ou atributos para alterar o comportamento padrão.

**Formas de identificar de quando usar Herança:**

...É UM(A)...

* Civic É UM carro.
* Gato É UM mamífero.
* Maça É UMA fruta.

Super Classe (+Genérica)

Sub Classe (+Específica)

**Polimorfismo:**

**Polimorfismo** é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.

Polimorfismo Estático:

Polimorfismo Estático = Sobrecarga.

Um polimorfismo estático seria quando você escreve estaticamente o mesmo método de diferentes formas diferentes(cada um com sua assinatura diferente) por exemplo.

Poliformismo Dinâmico:

Para ter um polimorfismo dinâmico, precisa ter Herança.

Exemplo:

Tipo especifico:

Civic c = new Civic();

Tipo mais genérico:

Carro c = new Civic();

c = new Ferrari();

Ao criar uma variável mais genérico, você pode atribuir outros tipos diferente para o objeto, uma hora ele pode ser um Civic, outra hora pode ser uma Ferrari, por o tipo dele ser Carro, e Ferrari e Civic serem herança em Carro.

Mas não pode criar uma variável do tipo Civic e atribuir para ela uma Ferrari.

**Abstração:**

A **abstração** é o ponto de partida para a criação de programas utilizando **POO**. Trata-se da capacidade de extrair dos personagens ou dos itens presentes no contexto, suas principais características, criando, dessa forma, objetos.

Se estivéssemos lidando com um software que precisará armazenar dados de um carro, poderíamos pensar nos seguintes métodos:

* Frear.
* Acelerar.
* Buzinar.
* Acender Farol.

Além de métodos, este nosso carro também teria propriedades, que poderiam ser

* Cor
* Quantidade de Portas
* Combustível utilizado
* Modelo
* Marca

A abstração é, portanto, a captura das principais características do personagem ou item envolvido no contexto, de forma que elas possam ser facilmente descritas em uma classe, que gerará um objeto.

**ENUM**

São tipos de campos que consistem em um conjunto fixo de constantes (static final), sendo como uma lista de valores pré-definidos. Na linguagem de programação **Java**, pode ser definido um tipo de enumeração usando a palavra chave **enum**. Todos os tipos **enums** implicitamente estendem a classe **java**.

**Interface**

Em java uma “Interface” sempre vai ter métodos públicos e abstratos, ou seja, mesmo que você não diga explicitamente que o método da interface é publica, ela é publica implicitamente, e é abstrata pois não tem corpo.

Exemplo:

public interface Esportivo {

void ligarTurbo();

void desligarTurbo();

}

public interface Ferrari implements Esportivo {

}

Sendo possível uma classe herdar de mais de uma interface.

public interface Ferrari implements Esportivo, Luxo {

}

Sendo obrigatório as classes que herdam de uma interface implementarem seus métodos que não foram implementados, da mesma forma que uma classe que herda de outra classe abstrata ela é obrigada a implementar o método que não foi finalizada do classe pai.

Para definir um método “padrão” dentro de um interface, se usa a palavra reservada “default”:

public interface Luxo {

void ligarAr();

void desligarAr();

default int velocidadeDoAr() {

return 1;

}

}

**Classes abstratas**

Pode-se dizer que as [**classes abstratas**](http://www.devmedia.com.br/classe-abstrata-curso-basico-de-java-e-orientacao-a-objetos-parte-29/25322) servem como “modelo” para outras classes que dela herdem, não podendo ser instanciada por si só. Para ter um objeto de uma classe abstrata é necessário criar uma classe mais especializada herdando dela e então instanciar essa nova classe. Os métodos da classe abstrata devem então serem sobrescritos nas classes filhas.

Por exemplo, é definido que a classe “Animal” seja herdada pelas subclasses “Gato”, “Cachorro”, “Cavalo”, mas ela mesma nunca pode ser instanciada.

**Lambdas**

Uma expressão **Lambda** permite definir uma interface funcional (novamente, um método abstrato) que o compilador identifica pela estrutura. O compilador pode determinar a interface funcional representada a partir de sua posição. O tipo de uma expressão **lambda** é o da interface funcional associada.

**Interface funcional:**

I**nterfaces funcionais** são **interfaces** que têm um método a ser implementado, em outras palavras, um método abstrato. Isso significa que toda **interface** criada que respeite esta premissa, tornando-se automaticamente uma **interface funcional**. E para ser uma interface funcional, ela tem que ter apenas 1 único método abstrato dentro dela.

**ForEach:**

Para usar uma função lambda com um forEach é bem prático, em vez de fazer um forEach tradicional como:

List<String> aprovados = Arrays.asList(“Ana”, “Bia”, “Lia”, “Gui”):

for ( String nome : aprovados) {

System.out.println(nome);

}

Basta colocar da seguinte forma:

aprovados.forEach(nome -> System.out.println(nome));

Dessa forma você tem um forEach no formato de função lambda, sendo muito mais prático e rápido de se usar, fazendo a mesma coisa de primeiro forEach.

Também tem o “Method Reference” ou “Método por Referencia”, é quando vc passa dentro do forEach uma referencia para o valor, em vez de passar explicitamente o parâmetro:

aprovados.forEach(System.out::println);

Dessa forma ele vai fazer a mesma coisa que os outros 2 forEach, mas de forma por referencia.

**Stream API**

Entre as diversas funcionalidades adicionadas à linguagem **Java** em sua versão 8 está a **Streams API**, recurso que oferece ao desenvolvedor a possibilidade de trabalhar com conjuntos de elementos de forma mais simples e com um número menor de linhas de código.Isso se tornou possível graças à incorporação do [**paradigma funcional**](http://www.devmedia.com.br/programacao-funcional-com-java/32176), combinado com as [**expressões lambda**](http://www.devmedia.com.br/trabalhando-com-expressoes-lambdas-no-java-8/34322), o que facilita a manutenção do código e aumenta a eficiência no processamento devido ao uso de paralelismo.

A [**proposta em torno da Streams API**](http://www.devmedia.com.br/streams-api-trabalhando-com-colecoes-de-forma-flexivel-em-java/31980) é reduzir a preocupação do desenvolvedor com a forma de implementar controle de fluxo ao lidar com coleções, deixando isso a cargo da API. A ideia é iterar sobre essas coleções de objetos e, a cada elemento, realizar alguma ação, seja ela de filtragem, mapeamento, transformação, etc.

Exemplo:

List<String> aprovados = Arrays.asList(“Lu”, “Gui”, “Luca”, “Ana”);

Stream<String> stream = aprovados.stream();

stream.forEach(System.out::println);

Aqui está sendo passado para a variável stream do tipo “Stream”, o retorno do método “stream()” da lista “aprovados”, e depois sendo usado o “forEach” para imprimir cada elemento da stream.

**Map:**

Retorna uma stream consistindo no resultado da aplicação de uma função de mapeamento nos elementos da stream.

A stream “Map” vai transformar cada elemento da stream, ou seja, ele vai gerar uma nova stream transformada para oque foi passado na lambda expression.

Exemplo:

Consumer<String> print = System.out::print;

List<String> marcas = Arrays.asList(“BMW “, “Audi “, “Honda “);

marcas.stream().map(m -> m.toUpperCase()).forEach(print);

No exemplo, a lista “marcas” será feito um mapeamento e criado uma nova Stream contendo todas as Strings da lista “marcas”, só que tudo em maiúsculo.