Introdução a POO

Paradigmas de programação

- Linguagens de programação fornecem funcionalidades diferentes e são categorizadas com base nesse aspecto. Existem duas grandes categorias.

- Linguagens imperativas: descrevem como o programa deve fazer algo, possuem os seguintes paradigmas.
    - Procedural: ênfase em procedimentos, modela a solução de problemas com base nos algoritmos, estado é manipulado diretamente por meio de variáveis, laços, etc. Exemplos: Pascal, C, Fortran.
    - Orientado a Objetos: programa composto por objetos que se comunicam através de mensagens, ênfase na abstração de conceitos, objetos caracterizados por propriedades e comportamento. Exemplos: C++, Java.

- Linguagens declarativas: descrevem o que o programa deve fazer, possuem os seguintes paradigmas.
    - Lógico: programas baseados em relações matemáticas e axiomas, regras escritas como cláusulas lógicas. Exemplo: PROLOG.
    - Funcional: Uso de expressões e funções no lugar de variáveis, ênfase na avaliação e composição de expressões e funções. Exemplos: Lisp, Haskell.

Princípios básicos de POO

Abstração, Modularização, Encapsulamento, Reusabilidade, Manutenibilidade programas são compostos por objetos que são uma abstração do mundo real.

Objetos são classificados pelas suas propriedades(informações de estado), comportamentos(o que o objeto faz ou recebe), identidade única.

- TAD (tipo abstrato de dado): tipo de dado em que o que importa é apenas a operação que se pode realizar, sem se preocupar com a implementação interna. Exemplos: listas, pilhas, etc.

Estrutura de um objeto

Objetos possuem atributos(estado), operações privadas e operações públicas, podem ser manipulados apenas com operações públicas e se comunicam através de mensagens. Possuem o conceito de encapsulamento, que une dados + operações para formar um contexto protegido e organizado.

- Atributos: variáveis de estado, exs: nome, velocidade…

- métodos: definem comportamento e são o único meio de se manipular atributos do objeto.

Classes

Objetos são instâncias de uma classe abstrata, por exemplo, a classe Bola pode possuir objetos como Bola de Basquete, Bola de futebol, etc.
Explicando os princípios básicos:

- Abstração: modelo de um conjunto de objetos com características semelhantes. Destaca os aspectos importantes de um objeto real segundo o observador(depende do observador).
- Modularidade: Construção do programa por módulos diferentes e independentes, facilita gerenciamento, se comunicam pela interface(métodos públicos).
- Encapsulamento: esconde detalhes de estruturas complexas, tirar o conhecimento do cliente acerca da implementação de operações, fornecer apenas a cápsula, dados não podem ser alterados de forma direta, nem mesmo acidentalmente.
- Reusabilidade: a independência gerada pela abstração e modularidade, permite reutilizar implementações, entre objetos da classe, entre classes diferentes de projetos anteriores.
- Manutenibilidade: implementação de uma classe pode ser alterada sem afetar os outros objetos que fazem uso dela. Objetos problemáticos podem ser substituídos.

Pacotes

Pacotes são utilizados como mecanismo para organizar classes relacionadas, um pacote é composto por classes e podem estar contidos um dentro do outro, formam uma hierarquia. Em Java, pacotes correspondem a pastas físicas. Classes podem ter o mesmo nome se estiverem em pacotes diferentes. Por padrão, classes devem estar contidas em pacotes, os pacotes são nomeadas utilizando o nome de domínio web reverso, por exemplo, package br.unicamp.ic.mc322. Para execução deve ser especificado o nome completamente qualificado, ou seja, o nome completo, exemplo br.unicamp.ic.mc322.Calculadora.

Declaração de atributos

- Modificadores: definem características do atributo. Visibilidade, pode possuir os valores public, protected e private, definem quais outras classes podem acessar os atributos. Se pode ser alterado ou não (final).
- Tipo: tipo primitivo ou por referência. Ex: int, float, outra classe, etc.
- Identificação: nome do atributo.

Métodos

Implementam aspectos do comportamento do objeto, podem receber mensagens de outros objetos ou deles mesmos. Métodos são chamados utilizando o padrão Object.method(parameters). Métodos possuem os mesmos modificadores de visibilidade que os atributos, possuem tipo de retorno.

Sobrecarga de métodos (overloading)

Métodos podem possuir o mesmo nome desde que tenham assinaturas diferentes. A assinatura de um método é dada pelo nome do método, número de argumentos, tipo de argumentos e ordem dos argumentos, o tipo de retorno não faz parte, o compilador entende qual método está sendo chamado diferenciando a quantidade e tipo de argumentos. Ex:

```
public void deposit (float amount){
        balance += amount;
}

public void deposit (float amount, String currency){
        /.../
}
```

Visibilidade

Os valores public, private e protected, definem quais outras classes podem acessar atributos e métodos. Pelo princípio do encapsulamento, todos os atributos de uma classe sempre devem ser privados, apenas a própria classe pode acessá-los. A opção protected permite apenas as classes contidas no mesmo pacote e classes de fora do pacote que herdam da classe original, verem o atributo/método. Public permite todas as classes acessarem o atributo/método. Se não for especificado visibilidade, automaticamente é atribuído a visibilidade default, que permite apenas as classes do mesmo pacote verem o atributo/método, porém, nesse caso as classes que herdam da classe original não tem esse acesso, diferente da opção protected.

Construtores

Construtores são métodos especiais que instanciam (criam) um objeto de uma classe. Possuem obrigatoriamente o mesmo nome da classe. Se não for especificado um construtor, o compilador fornece um construtor padrão que inicializa os atributos com valores padrões. É boa prática sempre definir o construtor explicitamente, podendo passar parâmetros e valores para inicializar atributos. Construtores geralmente são públicos mas podem ser protegidos para instanciação apenas por classes do mesmo pacote. Construtores podem ser sobrecarregados. Construtores são chamados pela palavra-chave new. Ex:

BankAccount conta1 = new BankAccount(“1234”);

Destrutores

“Destroem” objetos, liberando a memória e recursos alocados. Em C++ destrutores tem o mesmo nome da classe precedido de um ~ e não recebem parâmetros. Java não possui destrutores por conter a coleta de lixo automática (garbage collector), no qual objetos são destruídos automaticamente quando não existem referências que apontam para eles.

A palavra-chave this pode ser utilizada para referenciar um objeto dentro de seu próprio contexto. Ex:

Palavra-chave this

dentro da classe Carro
Carro.dirigir(); equivale a this.dirigir();

Métodos de acesso (getters e setters)

São métodos que permitem obter e alterar valores de atributos. Pelos princípios de POO, um atributo deve na maioria dos casos ser privado, atributos públicos violam gravemente os conceitos de POO. Por isso, para acessar e alterar atributos de uma classe, são criados métodos get e set públicos, garantindo segurança ao impedir o acesso direto a variável do atributo. Deve se atentar ao fato de que o uso de getters e setters não deve ser levado como regra em todos os casos, cada situação deve ser analisada individualmente para não haver quebra dos paradigmas de POO. Como por exemplo uma classe BankAccount que possui os atributos number e balance. Utilizar um método para alterar o número da conta não faria sentido nesse caso.

Data Transfer Objects(DTO)

Objetos de transferência de dados são utilizados como estruturas de dados apenas para armazenar informações que serão passadas para outras áreas na aplicação, esses tipos de objetos geralmente sempre possuem getters e setters.

Coesão de módulos

Coesão é uma característica importante para a aplicação. Módulos devem ser consistentes com as suas partes, com responsabilidade única e bem definida. Existem indicações de que há coesão, os métodos da classe chamam com frequência outros métodos da mesma classe, os atributos da classe são usados com frequência pela maioria dos métodos da classe.

Acoplamento

Acoplamento não deve ocorrer, representa o quanto um módulo é dependente de outros módulo. Existe responsabilidade espalhada entre módulos, alterações impactam várias áreas do software. Indicadores de acoplamento, a classe possui muitas variáveis com outras classes como tipo, a classe chama muitos métodos de outras classes.

Um método m de uma classe C deve chamar apenas métodos que pertencem a própria classe C, qualquer objeto instanciado dentro de m, qualquer objeto recebido como parâmetro pelo próprio m, qualquer objeto em uma variável de instância(atributo) de C.

Vetores e Strings

Vetores em java são objetos gerenciados pelo garbage collector, são instanciados com a palavra-chave new igual aos outros objetos:

int[] numeros = new int[12];

Um array possui como atributo público seu próprio tamanho(length).

Vetores podem ser inicializados com valores:

Point[] pontos = {new Point(10,20) , new Point(123, -7)};

vetores podem formar matrizes, seguindo o padrão de outras linguagens de programação.

Notação for simplificada

Para percorrer valores de um vetor podemos utilizar:

```
double[] notas = {7.0, 9.8, 6.2};
duble media = 0;
for(double n : notas){
        media = media+n;
}
media = media/notas.length;
```

Classe String

Em java String é uma classe(java.lang.String). A classe String possui vários construtores, podemos utilizar:

String disciplina = new String(mc322);
A classe String possui vários métodos.

Comparação entre objetos

Objetos não podem ser comparados com operador de igualdade. Todas as classes possuem o método equals, utilizado para comparar objetos. Esse método compara o significado dos objetos e pode ser sobrescrito pelo programador. Retorna true ou false.

```
Ex: public class Point
        private double x;
        private double y;

        private Point(int i, int j){
                x = i;
                y = j;
        }

        public boolean equals(Point p){
                return x == p.x && y == p.y; // retorna true se os atributos são iguais
        }
```

Método toString

O método toString é utilizado para “traduzir” um objeto em uma representação String. Por padrão retorna tipo do objeto + “@” + hashcode. Pode ser sobrescrito e personalizado pelo programador.

Herança

Herança define um conjunto de classes que são especializações de uma classe mais genérica, exemplo, Veículo pode ter as classes herdeiras Carro e Caminhão. Classes herdadas podem herdar atributos e métodos da sua superclasse(nomenclatura para a classe principal). Dizemos que uma subclasse é uma especialização de uma superclasse, assim como, uma superclasse é uma generalização de uma subclasse.

Em java a herança é declarada pela palavra-chave extends. Ex:

```
public class Caminhao extends Veiculo{
```

Lembrando, classes são definidas no código fonte, objetos são instânciados em memória em tempo de execução.

Quando aplicar herança

Princípio da substituição de Liskov: se B é uma subclasse de A, então objetos do tipo A podem ser substituídos por objetos do tipo B sem alterar nenhuma das propriedades desejáveis do programa. Ao utilizar herança, devemos sempre analisar a frase “é um tipo de”, se B é um tipo de A, a herança é correta, senão, pode quebrar conceitos de POO.

Modelar Classes

Duas etapas são fundamentais em POO:
análise do problema que deve ser resolvido (o que?)
projeto do programa que resolve tal problema (como?)

quais classes? quais métodos?

Devemos documentar essas escolhas. Isso pode ser feito por um tipo muito utilizado de representação gráfica, a notação UML.

Unified Modeling Language (UML)

Linguagem de modelagem para descrever software, possui vários tipos de diagramas que podem ser construídos.

Diagrama de classes: permite descrever as classes de um programa e as relações entre elas.

Em UML a relação de herança é representada por uma flecha saindo da subclasse para a superclasse, com uma ponta triangular oca.

Efeitos da herança

Sub-classes herdam atributos e métodos da superclasse. Atributos e operações comuns devem ser implementados no nivel mais alto da hierarquia.
Subclasses não podem acessar atributos privados, para subclasses acessarem atributos da superclasse, devem ser declarados como protegidos.

Subclasses definem subtipos, assim como classes definem tipos. Objetos de subclasses podem ser usados quando se espera um objeto da classe base, mas nunca o oposto. Ex:

CORRETO
```
Veiculo meuVeiculo;
meuVeiculo = new Veiculo();
```

meuVeiculo = new Aviao(); // Aviao também é um veiculo

ERRADO
Aviao meuAviao;
meuAviao = new Aviao();
meuAviao = new Veiculo(); // Veiculo não é um tipo de avião

Benefícios da herança

Herança é um conceito fundamental de POO e proporciona vários benefícios. Alguns deles são, reuso de código, extensibilidade, abstração.

Propriedades da herança

Relação transitiva e anti-simétrica entre classes:

Transitiva: Se A é subclasse de B e B é subclasse de C, então A é subclasse de C

Anti-simétrica: Se A é subclasse de B e B é subclasse de C, C não pode ser subclasse de A. Não podem existir ciclos.

Pode existir hierarquia dentro dessa árvore de heranças.

Herança múltipla

Caso em que uma classe herda diretamente de mais de uma classe, linguagens como C++ possuem essa possibilidade, Java não possui herança múltipla.

Construtores de subclasses

Objetos de subclasses também precisam ser instânciados. Construtores não são herdados e devem ser definidos na subclasse. Podemos utilizar a palavra-chave super() para chamar o construtor da superclasse, dentro do construtor da subclasse.

Classe Object

Em java toda classe é subclasse diretamente ou indiretamente da classe java.lang.Object, a herança da classe Object é implícita em todas novas classes. A classe Object é a classe primordial em Java, possui métodos como toString() e equals().

Sobrescrita de métodos(overriding)

Uma classe pode customizar métodos que ela herda da superclasse. Esse processo é chamado de sobrescrita de método. Para ocorrer sobrescrita, o método da subclasse deve possuir a mesma assinatura da superclasse(diferente da sobrecarga).

Sobrescrita x Sobrecarga

Sobrecarga: ocorre na mesma classe(geralmente), a implementação preexistente é mantida, métodos com assinaturas necessariamente diferentes.

Sobrescrita: ocorre apenas entre subclassse e superclasse, a implementação preexistente é escondida, métodos necessariamente com a mesma assinatura.

A assinatura em sobrescrita de método deve ser exatamente igual a da superclasse, caso haja algo diferente, ocorrerá uma sobrecarga e a subclasse terá 2 implementações diferentes do mesmo método.

Anotação @Override

Ao se utilizar sobrescrita, é muito recomendado a utilização da notação @Override acima da implementação do método. Isso indica explicitamente a intenção de sobrescrita e o compilador java retorna um erro caso houver algum erro na sobrescrita.

Tipo de retorno

Na sobrescrita de métodos é permitido alterar o tipo de retorno na subclasse, com uma restrição, o tipo de retorno na subclasse deve ser um subtipo do tipo de retorno do método da superclasse. Ex:

```
public class Figura{

	public Figura criaCopia(){
		…
	}
}

public class Triangulo extends Figura{

	@Override
	public Triangulo criaCopia(){
		…
	}
}
```

Nesse caso Triangulo retornado pelo método sobrescrito é um subtipo de Figura, portanto o uso está correto.

Polimorfismo

Em POO, polimorfismo se refere à capacidade dos objetos de exibirem tipos diferentes. Por exemplo, um objeto da classe Aviao pode ser manipulado como um avião, porém ele também pode ser atribuído a um objeto da classe Veiculo, sem quebrar os conceitos de POO, isso é o polimorfismo.

Operador instanceof

Em java existe o operador instanceof, que permite verificar se um objeto é instância de uma classe. Ex:
Veiculo v = new Veiculo();
System.out.println(v instanceof Veiculo); // retorna true

A utilização de instanceof é quase sempre indicador de uma péssima organização de código, um código bem projetado não deve precisar retornar o tipo de um objeto em tempo de execução. O uso desse tipo de recurso é solucionado com a utilização de métodos polimórficos. De forma que um método possa ser executado para diferentes formas de um mesmo objeto seguindo os princípios de POO.

Open-Closed Principle

“Uma classe reusável deve ser aberta para extensão, mas fechada para modificação”.

Aberta para extensão: deve ser fácil estender os módulos com novas funcionalidades

Fechada para modificação: introduzir novas funcionalidades não deve precisar de alterações no código original

Com isso surge o conceito de classes abstratas.

Classes abstratas

Algumas vezes desejamos declarar classes que são utilizadas apenas como superclasse em uma hierarquia de herança, classes que não possuem um significado concreto e sem nunca instanciar um objeto dessa classe, apenas objetos de subclasses. Podemos realizar isso utilizando as classes abstratas.

Classes abstratas não podem ser instanciadas, podem possuir métodos sem implementação, podem possuir atributos, podem herdar de outras classes. Elas definem um conjunto de métodos comuns para objetos de um certo tipo, sem especificar a implementação (os métodos serão sobrescritos pelas subclasses).

Em java classes abstratas são declaradas com a palavra-chave abstract. Ex:

```
public abstract class Veiculo{
        /***/
}
```

Classes abstratas podem ter construtores, porém, eles não podem ser chamados diretamente, já que a classe não pode ser instanciada. Por isso, construtores de classes abstratas devem ser chamados apenas dentro dos construtores das subclasses com a palavra-chave super().

Métodos abstratos

Como dito, classes abstratas podem ter métodos abstratos. Métodos abstratos possuem apenas uma assinatura e um tipo de retorno. Métodos abstratos apenas podem existir em classes abstratas, porém uma classe abstrata pode conter apenas métodos concretos, sem nenhum método abstrato.

Subclasses que herdam de classes abstratas, devem implementar todos os métodos abstratos da superclasse, caso contrário, ocorrerá um erro de compilação. Métodos abstratos podem ser chamados dentro da classe abstrata por métodos concretos, mesmo não tendo implementação, já que, quando as subclasses sobrescreverem o método abstrato e utilizarem a chamada ao método concreto, esse identificará a implementação de cada subclasse.

Classes abstratas em notação UML

Em UML, classes abstratas podem ser representadas com o nome da classe escrito em itálico, ou com a anotação <<abstract>> acima da classe, métodos abstratos são representados em itálico.

Constantes em Java

Em java, constantes são declaradas com a palavra-chave final. Por padrão, constantes devem ter o nome todo maiúsculo com palavras separadas por um underline. Qualquer tipo de variável pode ser declarada como constante e constantes podem receber qualquer tipo de visibilidade. Ex:

```
private int final NUMERO_PI = 3.1415;
```

Constantes x Objetos imutáveis

O modificador final afeta apenas o valor da variável. Para tipos referenciados, é apenas a referência que não pode ser alterada. Ex:

```
final int ID = 10;
ID = 42; // erro de compilação

final int[] USER_IDS = new int[] {23, 32, 5};
USER_IDS[0] = 42; // possível
USER_IDS[1] = 43; // possível
USER_IDS = new int[40] // erro de compilação
```

Para garantir que um objeto seja realmente imutável, é necessário que seus atributos não possam ser modificados. Isso pode ser feito com a ausência de métodos que alterem os atributos.

Classes e Métodos constantes

A palavra-chave final também pode ser utilizada em classes e métodos. Classes definidas com final não podem ser estendidas, ou seja, não podem ser herdadas por outras classes. Métodos com a palavra-chave final não podem ser sobrescritos.

Métodos Estáticos

Métodos estáticos são métodos que não se aplicam a um objeto específico da classe. Funcionam como funções normais como vemos na programação procedural. Em java utilizamos a palavra-chave static para definir métodos estáticos. Ex:

```
public class Calculadora {

	public static int[] numerosPrimos(int n){

		int[] numeros = new int[n];
		// calcula os primeiros n numeros primos
		return numeros;
	}
}
```

chamada:

```
int[] primos = Calculadora.numerosPrimos(10);
```

Métodos estáticos são chamados também de métodos da classe. Em contrapartida, os métodos comuns vistos até agora para manipular objetos são chamados de métodos da instância.

Um exemplo de método estático é o método main definido para um programa em Java.

Quando usar métodos estáticos?

Geralmente são uma boa escolha para operações procedurais que não envolvem estado. Por exemplo:

- bibliotecas matemáticas ou de auxílio
- gerenciamento de entradas e saídas
- operações relacionadas a uma certa classe, mas que não se aplicam a nenhum objeto específico
- logging, debugging, testing

quando estiver na dúvida entre usar ou não, não use.

Particularidades de métodos estáticos

Métodos estáticos não são herdados, portanto não podem ser sobrescritos e nem ser abstratos. Métodos estáticos podem ser sobrecarregados. Um método estático não pode acessar variáveis ou métodos da instância, para isso precisa de uma referência para um objeto.

Atributos estáticos

Assim como os métodos, os atributos também podem ser declarados como estáticos. Possui o mesmo conceito de atributos da classe x atributos da instância. Atributos estáticos podem ser acessados por métodos estáticos e por métodos da instância.

Quando usar?

Atributos estáticos se aproximam muito de variáveis globais. São adequados para utilização em variáveis de configuração(ex: número máximo de itens em uma estrutura), objetos que devem existir em apenas uma única instância(ex: um gerenciador de conexões com o banco), logging, debugging, testing.

É comum o uso da junção public static final para definir constantes relacionadas a uma classe. Como por exemplo na definição do número PI na classe Math da

biblioteca java.lang (java.lang.math). Nesse caso, utilizar atributos com visibilidade pública é aceitável

Enumerações

Enumerações são conjuntos de constantes representadas por identificadores únicos, esses identificadores são descritos em maiúsculas. Em Java, as constantes de uma enumeração não estão associadas a números, diferente de outras linguagens.

Enumerações são declaradas com a palavra-chave enum. Ex:

```
public enum WeekDay{
	MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
MONDAY
}
```

Exemplo de uso:

```
public class LembreteDiaDaSemana extends LembreteComum{

	private WeekDay dia;

	public LembreteDiaDaSemana(String message, WeekDay dia){
		super(message);
		this.dia = dia
	}

	public WeekDay getDiaDaSemana(){
		return dia;
	}
}
```

cliente:

```
LembreteDiaDaSemana l = new LembreteDiaDaSemana(“Lab MC322”,
WeekDay.TUESDAY);
```

Enumerações e Classes

Em java, enumerações são tipos especiais de classes. Podem ter métodos e atributos, porém, normalmente não tem. Constantes de enumerações são definidas como variáveis estáticas e constantes (static final). Enumerações podem ser comparadas com operador de igualdade, diferente das classes comuns.

Método values()

Toda enumeração possui um método estático values(). Esse método retorna um array com as constantes enumeradas, na mesma ordem em que elas foram declaradas.

Relações entre classes

A forma com que dois objetos podem se comunicar através da invocação de métodos, define relações entre as classes. Existem três principais tipos de relações entre classes, composição, agregação e associação. Identificar relacionamentos entre classes antes de implementar o código é fundamental para uma boa estrutura. Diferente da herança, as relações entre classes são conceituais, ou seja, não são definidas explicitamente no código.

Composição

Objetos muitas vezes são formados por partes, por exemplo, um objeto que instância uma classe Carro pode possuir atributos como peças e métodos como ações. Composição indica uma relação de todo-parte indivisível. Nesse caso a parte existe apenas como componente do objeto contenedor, ou seja, se o contenedor deixar de existir, a parte também deixará de existir. Um objeto não pode ser parte de vários contenedores ao mesmo tempo. Exemplos: um motor não pode ser parte de vários carros ao mesmo tempo. Se uma universidade deixar de existir, o instituto também deixa de existir. Se o livro deixar de existir, as páginas também deixam de existir.

Identificamos composições pela relação “tem um”. Carro tem um motor, universidade tem um instituto, livro tem uma página. Se diferencia da relação “é um” aplicada na herança. Em notação UML a composição é representada por uma ligação com ponta em formato de diamante preenchido, que fica ligado ao contenedor e vai até a parte contida.

Na prática em código, dados uma classe contenedora A e uma parte contida B, a composição ocorre quando A contém um atributo do tipo B, instâncias de B são criadas e destruídas por A. Ex:

```
public class Sala{

	private Poltrona[][] assentos;

	public Sala(int linhas, int colunas){
		assentos = new Poltrona[linhas][colunas];
		for(int i=0; i<linhas; i++){
```

```
                        for(int j=0; j < colunas; j++){
                                assentos[i][j] = new Poltrona();
                        }

                }
        }
}
```

Poltrona é a parte contenedora e Sala é o contenedor.

Cuidados que devemos ter ao utilizar composição:

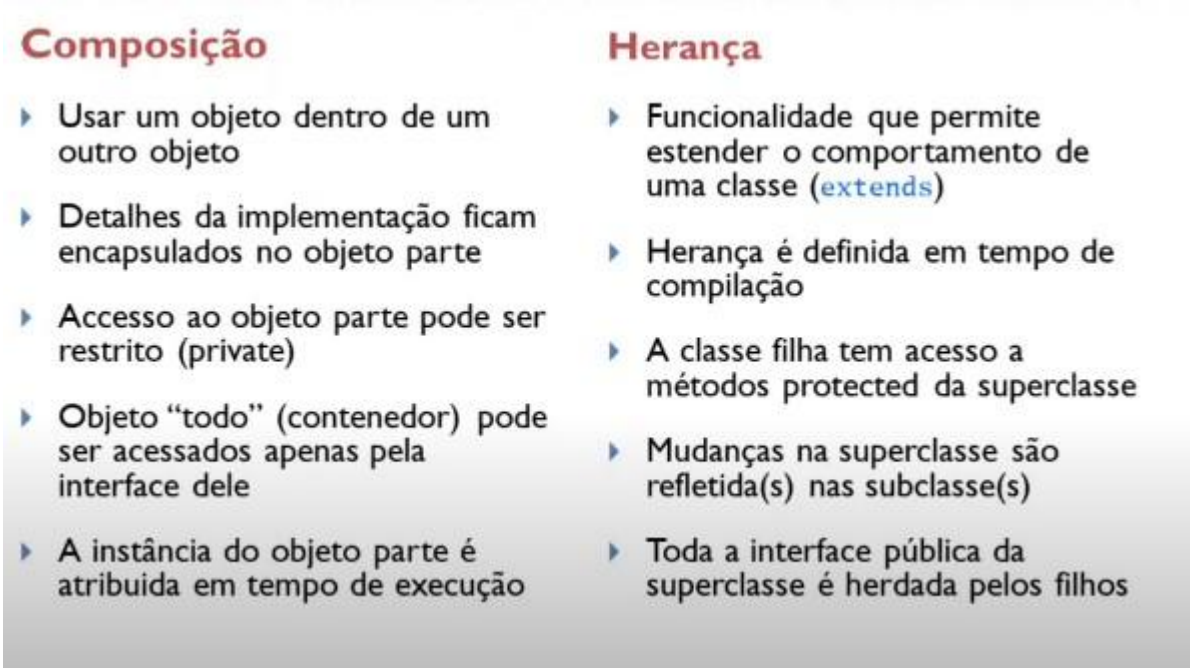
- não referenciar partes da classe contenedora fora da classe do contenedor, inclusive dentro da própria classe contida, isso pode quebrar o encapsulamento.

Reuso por composição

Composição também é uma forma de reutilizar código, o código da parte é reutilizado pelo contenedor.

Delegação de tarefas é uma forma de usar composição para reutilizar código. Realizamos isso utilizando os métodos da classe parte, podendo realizar alterações trocando apenas o objeto interno.

Herança x Composição



Multiplicidade

Todas as relações entre classes podem ter uma multiplicidade. Define quantos objetos daquele tipo podem ser contidos. Quando não houver especificação, por padrão entende-se que a multiplicidade é de um objeto.

Em UML a notação para multiplicidade é implementada colocando o número de objetos na parte da ligação na classe parte. Se omitirmos um número, assumimos que existe um objeto. Podemos utilizar também um asterisco para indicar que um número qualquer de objetos fazem parte da relação. Outras notações de multiplicidade:

Notação	Alternativa	Significado
0..1		Zero ou uma instância
1..1	1	Exatamente uma instância
0..*	*	Zero ou mais instâncias (qualquer número)
1..*		Pelo menos uma instância
k..k	k	Exatamente k instâncias
k..*		Pelo menos k instâncias
m..n		Pelo menos m e no máximo n instâncias

Efeito da herança nas relações

Subclasses herdam todas as relações das próprias superclasses. Uma subclasse pode possuir relações com todas as classes que tem relação com sua superclasse e com todas as classes herdeiras da classe que possui relação com sua superclasse.

Relações múltiplas

Podem existir múltiplas relações do mesmo tipo entre classes, isso acontece quando cada uma possui um significado específico. Por exemplo, um livro tem uma capa e várias páginas de conteúdo:

```
public class Livro{

        private Pagina capa;
        private Pagina[] conteudo;
}
```

Nesse caso, em notação UML, o papel de cada relação deve ser especificado no diagrama, com o nome capa e conteudo abaixo da multiplicidade do lado da classe parte.

Agregação

Agregação é um tipo de relação entre classes, mais fraca que composição. Ainda possui a semântica do todo-parte, porém o ciclo de vida da parte é independente. Ex: rodas de um carro podem ser retiradas do carro, guardadas de forma independente e até montadas em outro carro. Algumas vezes, agregação e composição são descritas como dois tipos de agregação, agregação compósita(composite aggregation = composição) e agregação compartilhada(shared aggregation = agregação).

Em notação UML uma relação de agregação é representada por uma linha possuindo uma forma de diamante oco na ponta do lado da classe contenedora que se liga na classe parte.

A agregação “relaxa” os vínculos da composição, a parte não depende mais do contenedor para sobreviver e pode pertencer a vários contenedores. Assim como na composição, a implementação consiste apenas em ter um atributo da classe parte na classe contenedora e o significado da relação é conceitual, sem especificação explícita no código.

Associação

A associação é a relação mais genérica entre as três citadas. Reflete o conceito geral “objetos da classe x mantém uma referência para objetos da classe y”. Essa relação ocorre entre objetos completamente independentes, possuindo apenas algum tipo de conexão lógica entre objetos. Agregação e composição são tipos específicos de associação, agregação = associação + semântica todo-parte, composição = associação + semântica do todo-parte + ciclo de vida gerenciado pelo contenedor.

Em notação UML, a associação é representada por uma linha contínua simples, sem formas em nenhuma das pontas.

Exemplo de associação: relação entre uma classe Carro que possui associação com uma classe Pessoa.

Relações Reflexivas

Uma relação reflexiva ocorre quando uma classe realiza relação com ela mesma, esse tipo de relação é permitido e comum de ocorrer. Todos os três tipos de relações podem realizar relação reflexiva. Ex: uma classe Pessoa que possui referência a uma classe Pessoa que guarda os pais de uma pessoa e um vetor de Pessoa que guarda os filhos, essa classe possui duas relações reflexivas.

Navegabilidade em associações

Associações podem ser bidirecionais ou unidirecionais, isso é chamado de navegabilidade. Assumindo duas classes A e B em associação, a extremidade do lado de B é navegável se um objeto de A possui referência a instâncias de B. Em UML representamos isso com uma flecha aberta que aponta de A para B. Nesse caso temos uma associação unidirecional. Se B também possuir referências de A, então representamos uma seta dupla em UML e a associação é dita bidirecional.

Se a navegabilidade for omitida em uma associação, por padrão se considera uma associação bidirecional. Em relações de composição e agregação, a parte não sabe quem é o seu contenedor, por isso, a navegabilidade por padrão é unidirecional. Isso é implícito e não deve ser alterado.

Identificação das relações

A identificação de relações é feita na análise e projeto de sistemas a objetos, traduzir os requisitos em um conjunto de classes e relações. Alguns padrões são:

- substantivos identificam classes e atributos
- verbos e expressões que indicam posse identificam composição/agregação. Ex: tem, possui, contém, é formado por
- verbos ou expressões que indicam especialização ou generalização, identificam herança. Ex: é um, existem vários tipos de.
- outros verbos, principalmente transitivos, identificam associação. Ex: professor ministra disciplina.

Padrões de projeto - Design Patterns

Existem padrões que são aplicados para se resolver problemas, que são estabelecidos para determinados contextos porque sabemos que eles funcionam bem.

Design Patterns são soluções para problemas comuns encontrados durante o projeto e desenvolvimento de software, validadas através do uso em diversas circunstâncias distintas. Especificadas de uma maneira estruturada que descreve a solução e o problema que ela se propõe a resolver. Design Patterns são

classificados com nomes, propósito, motivação, estrutura(conjunto de classes e relações entre elas) e consequências.

O primeiro livro que descreveu um Design Pattern em projeto de software foi escrito por quatro autores. Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm escreveram Design Patterns: Elements of Reusable Object-Oriented Software, ficaram conhecidos como gang of fours(GoF). Eles organizaram os padrões de projeto em três categorias:

- Criacionais: definem formas alternativas de criar(instanciar) objetos
- Estruturais: definem formas de compor objetos para alcançar algum objetivo específico
- Comportamentais: Definem como um grupo de objetos devem interagir para realizar algum comportamento complexo.
-

GoF Patterns

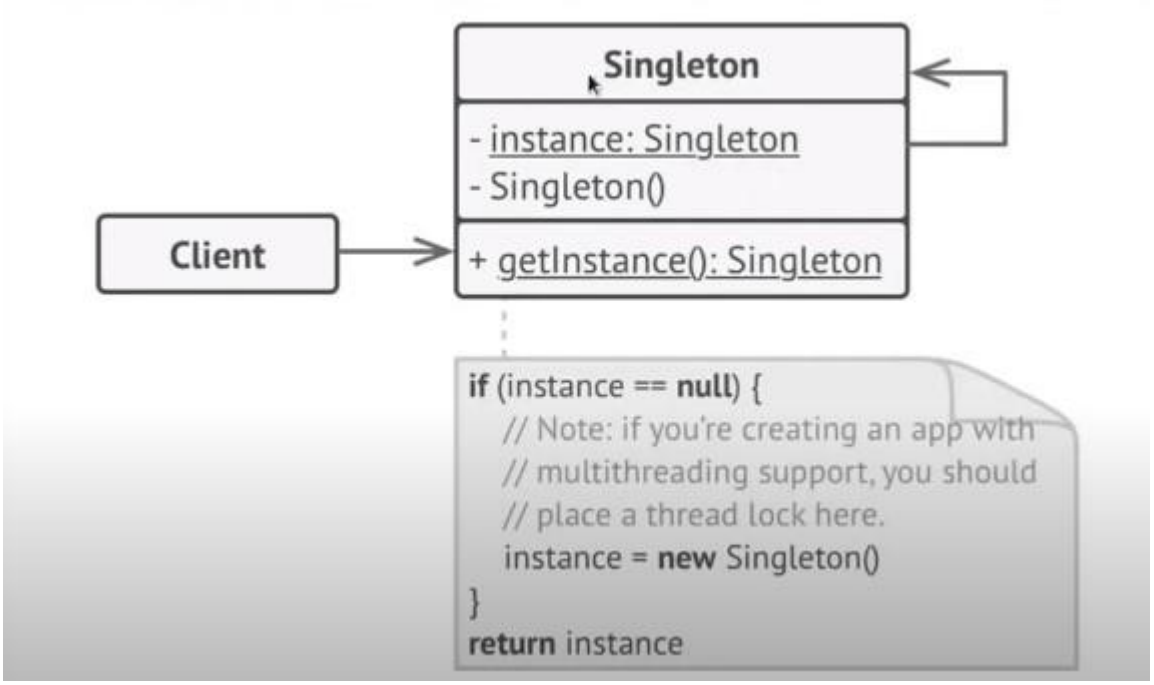
Purpose		
Creational	**Structural**	**Behavioral**
Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

É importante conhecer os principais padrões para um bom programador. Para isso pode se usar o GoF Book como enciclopédia.

Padrão Singleton

Em alguns casos, é importante que uma classe tenha exatamente uma instância e que seja facilmente acessível, por exemplo em drivers de conexão com um banco de dados, gerenciador de janelas, gerenciador de dispositivos. Uma variável estática, permite um fácil acesso a um determinado objeto, mas ainda não impede a criação de múltiplas instâncias. O propósito do padrão singleton é impor que uma classe tenha apenas uma única instância e fornecer um ponto de acesso único a tal instância.

Para implementar essa ideia, o construtor da classe é declarado como private, assim apenas a própria classe poderá criar instâncias, essa instância é guardada em uma variável estática e um método estático retorna sempre a mesma referência.




```java
public class Database {

    private static Database db;

    private boolean connected;

    private Database() {
        connected = false;
    }

    public static Database getInstance() {
        if(db == null) {
            db = new Database();
        }
        return db;
    }

    public void connect() {
        /* ... */
        connected = true;
    }

    public void disconnect() {
        /*...*/
        connected = false;
    }
}
```

```java
Database appDB = Database.getInstance();
appDB.connect();

/* ... */

Database myDB = Database.getInstance();
myDB.disconnect();
```

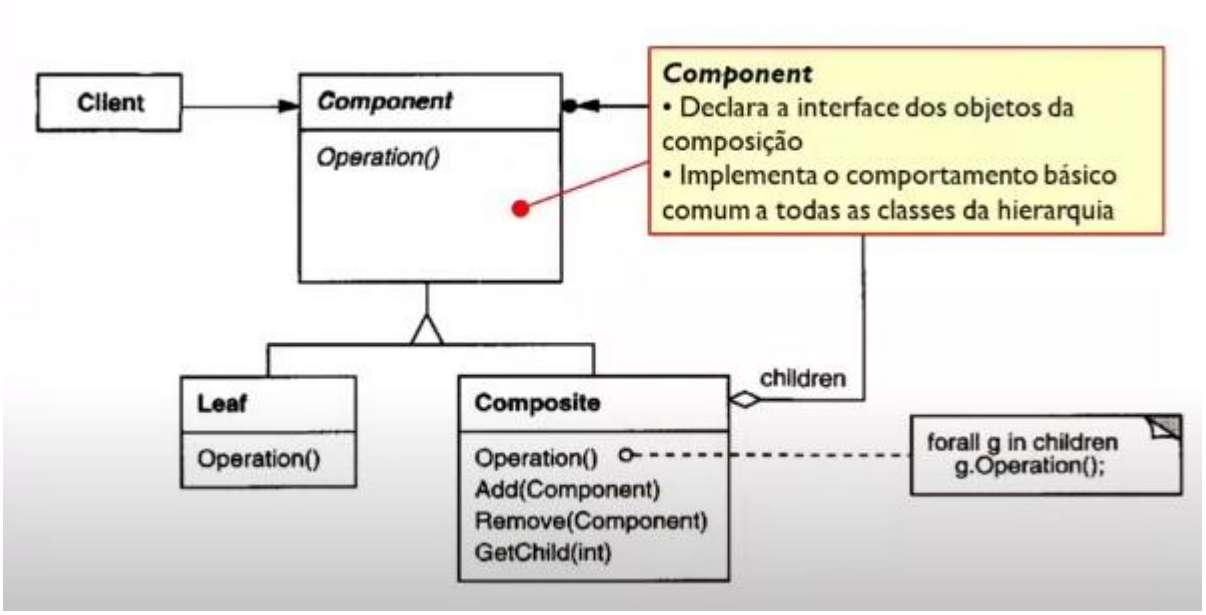
Nesse caso utilizamos o padrão Singleton para instanciar um gerenciador de banco de dados. Como todas as instâncias sempre estarão na verdade em apenas uma instância do banco, evitamos múltiplas conexões com o banco, o que poderia causar problemas de desempenho e integridade ao sistema.

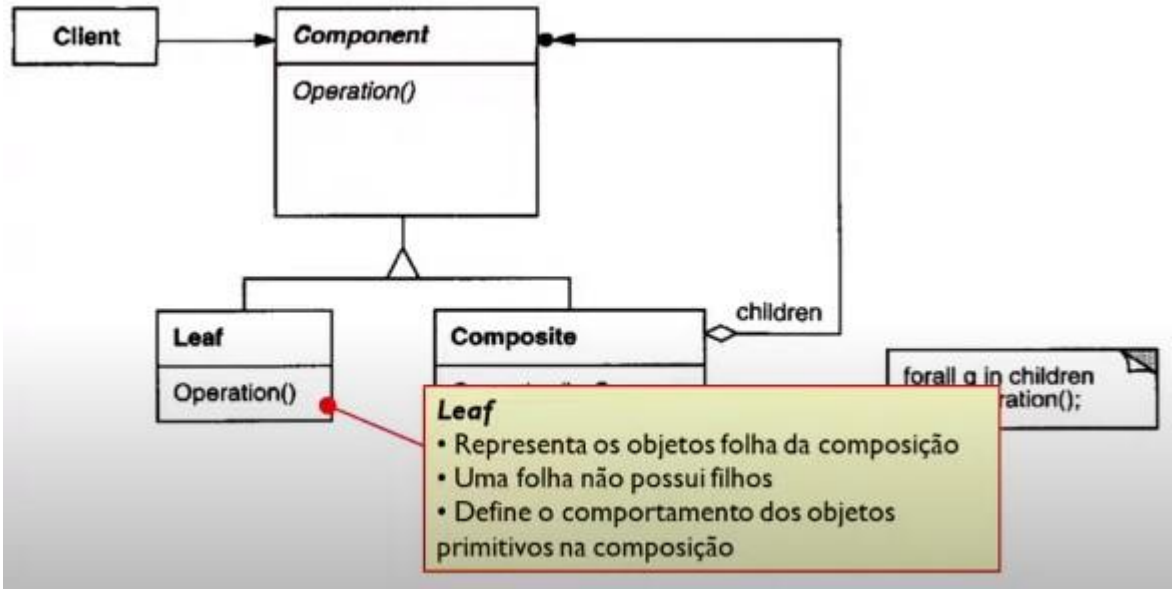
Multiton

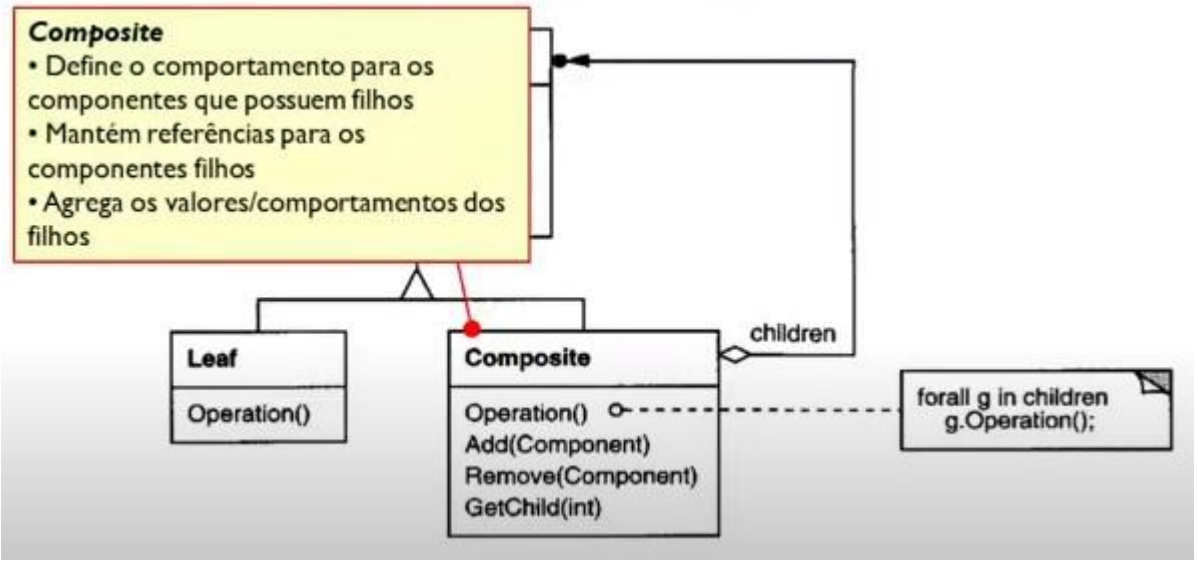
Seguindo essa mesma ideia, é possível limitar o número de instâncias para um número qualquer, chamamos de Multiton, pode ser usado para limitar conexões, threads, etc.

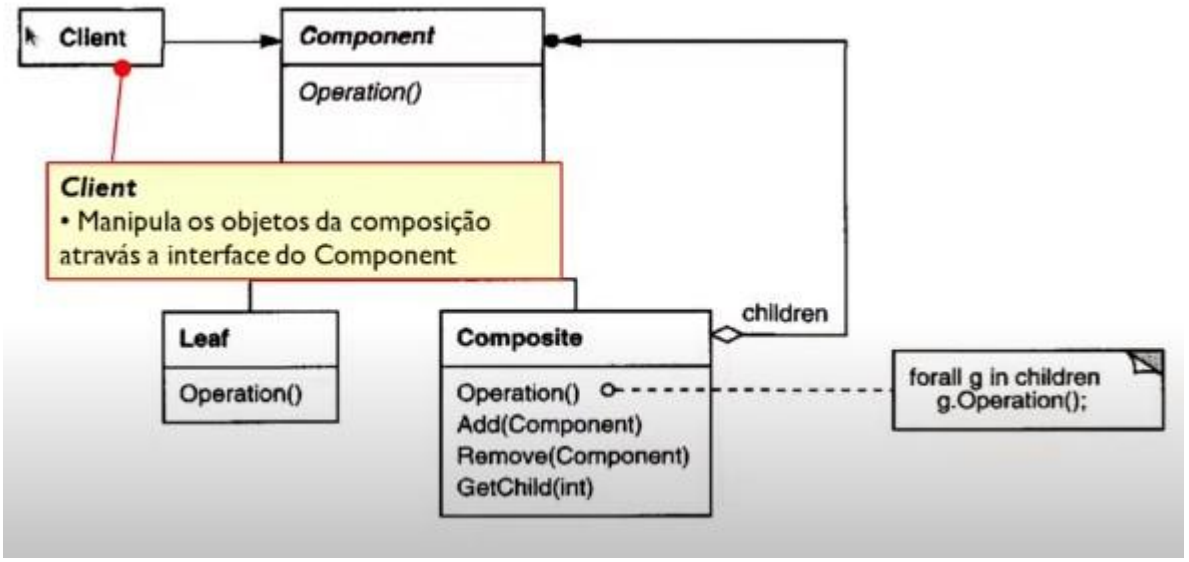
Composite

Em algumas situações, desejamos trabalhar com objetos complexos, organizados em uma estrutura de árvore, por exemplo: Categorias de produtos, elementos de uma interface gráfica, estrutura de uma ordem, expressões matemáticas. O propósito do padrão composite é permitir que o código que utiliza os objetos que pertencem ao objeto complexo, possa operar neles, desconhecendo a estrutura interna.





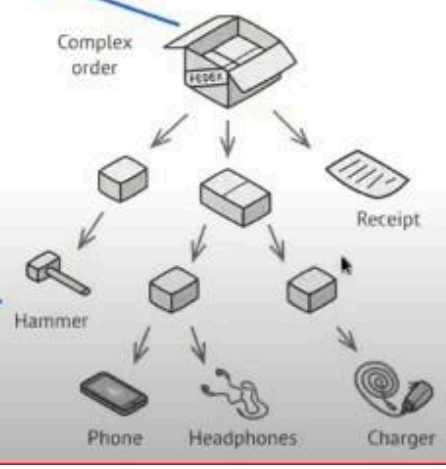





```java
public class Box extends Item {

    private Item[] content;

    public double calculatePrice() {
        double sum = 0;
        for(Item element : content) {
            sum = sum + element.calculatePrice();
        }
        return sum;
    }
}
```

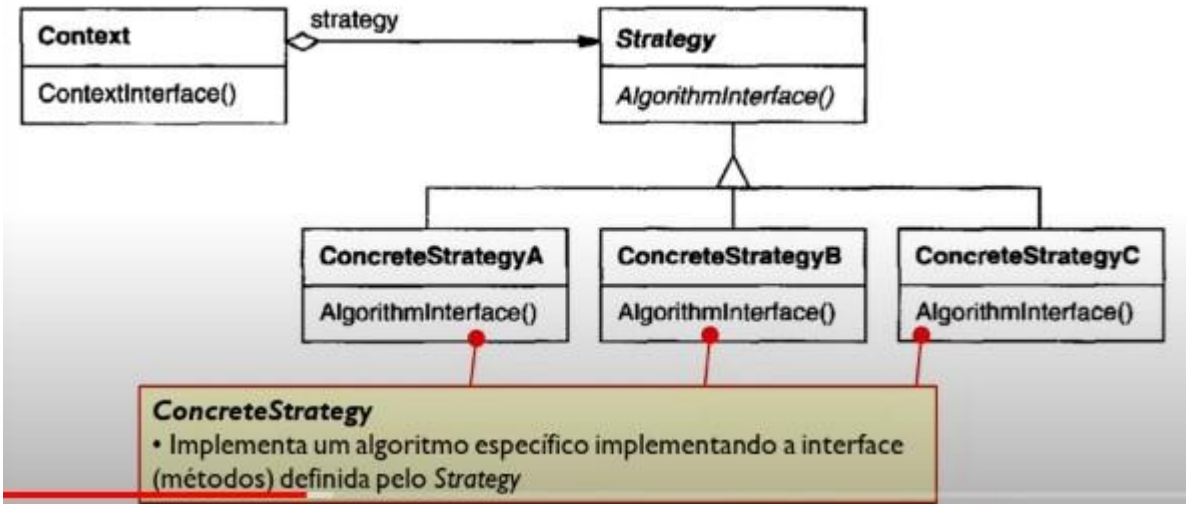
```java
public class Product extends Item {

    private double price;

    public Product(double price) {
        this.price = price;
    }

    public double calculatePrice() {
        return price;
    }
}
```

Ordem
Calcular de forma transparente o preço total da ordem

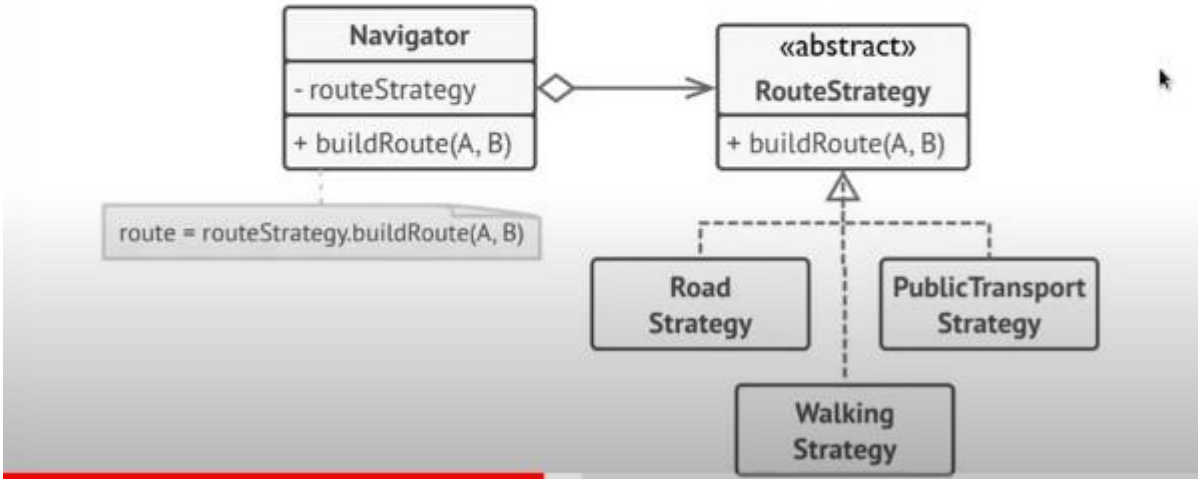


Strategy

Em geral, podem existir vários algoritmos para fazer a mesma tarefa(ex: ordenação), pode ser necessário alterar esses algoritmos, por razões como versões diferentes do software, configurações diferentes do usuário, em tempo execução. O propósito do padrão strategy é definir uma família de algoritmos e permitir que eles sejam intercambiáveis, faz com que o algoritmo possa ser alterado sem alterar as chamadas dos clientes, mesmo em tempo de execução.



- Exemplo
  - Navegador
  - Estratégias: Algoritmos de busca do melhor caminho



- Para trocar o algoritmo e, portanto, alterar o comportamento do sistema, é suficiente trocar uma linha de código

```java
public class Navigator {

    private RouteStrategy routeStrategy;

    public Navigator() {
        routeStrategy = new RoadStrategy();
    }

    public Route buildRoute(Point a, Point b) {
        return routeStrategy.buildRoute(a,b);
    }
}
```

```java
Navigator nav = new Navigator();

Point start = new Point(-22.8147168,-47.0649212);
Point end = new Point(-22.8174758,-47.0691752);

Route route = nav.buildRoute(start, end);
```

- Para trocar o algoritmo e, portanto, alterar o comportamento do sistema, é suficiente trocar uma linha de código

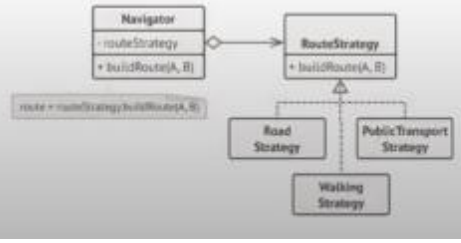
```
public class Navigator {

    private RouteStrategy routeStrategy;

    public Navigator() {
        routeStrategy = new WalkingStrategy();
    }

    public Route buildRoute(Point a, Point b) {
        return routeStrategy.buildRoute(a,b);
    }
}
```

```
Navigator nav = new Navigator();

Point start = new Point(-22.8147168,-47.0649212);
Point end = new Point(-22.8174758,-47.0691752);

Route route = nav.buildRoute(start, end);
```

- As estratégias podem ser configuradas em tempo de execução

```
public class Navigator {

    private RouteStrategy routeStrategy;

    public Navigator(RouteStrategy rs) {
        routeStrategy = rs;
    }

    public Route buildRoute(Point a, Point b) {
        return routeStrategy.buildRoute(a,b);
    }

    // configura estrategia "a pé"
    public void setWalkingStrategy() {
        routeStrategy = new WalkingStrategy();
    }

    // configura estrategia "de carro"
    public void setRoadStrategy() {
        routeStrategy = new RoadStrategy();
    }

    // configura estrategia "com transporte publico"
    public void setPublicTransportStrategy() {
        routeStrategy = new PublicTransportStrategy();
    }
}
```

```
Navigator nav = new Navigator(new RoadStrategy());

Point start = new Point(-22.8147168,-47.0649212);
Point end = new Point(-22.8174758,-47.0691752);

Route routeByCar = nav.buildRoute(start, end);

nav.setWalkingStrategy();
Route routeByFoot = nav.buildRoute(start, end);
```



Cuidados ao se utilizar padrões

Existem alguns cuidados que devemos ter antes de utilizar um padrão de projeto.

- Nem sempre é necessário aplicar padrões, uma solução mais simples pode ser possível
- é necessário entender se o padrão que se deseja utilizar realmente é a melhor solução para o problema. ex:
    - singleton: realmente precisamos de uma única instância da classe? Isso é um fato ou pode mudar no futuro?

- composite: Estou realmente lidando com uma estrutura do tipo árvore? Ou existe um número de níveis predeterminado?

Exceção (Exception)

Exceções são lançadas quando um método não consegue realizar sua função, fornece um padrão de resposta a quem chamou o método. Podemos ter vários tipos de soluções para erros de execução, como impressão de mensagem via terminal ou código de erro. Porém, essas soluções não são adequadas para POO, já que, a responsabilidade de tratar o erro pode não ser da classe que o gerou, o que quebraria o encapsulamento, nesse caso, o erro deveria ser descrito por um tipo de dado adequado. Uma exceção fornece um fluxo alternativo a ser executado pelo programa ao ocorrer um erro, evitando o encerramento do programa. Chamamos a implementação dessa prática de tratamento de exceção (exception handling).

Tratamento de exceção (exception handling)

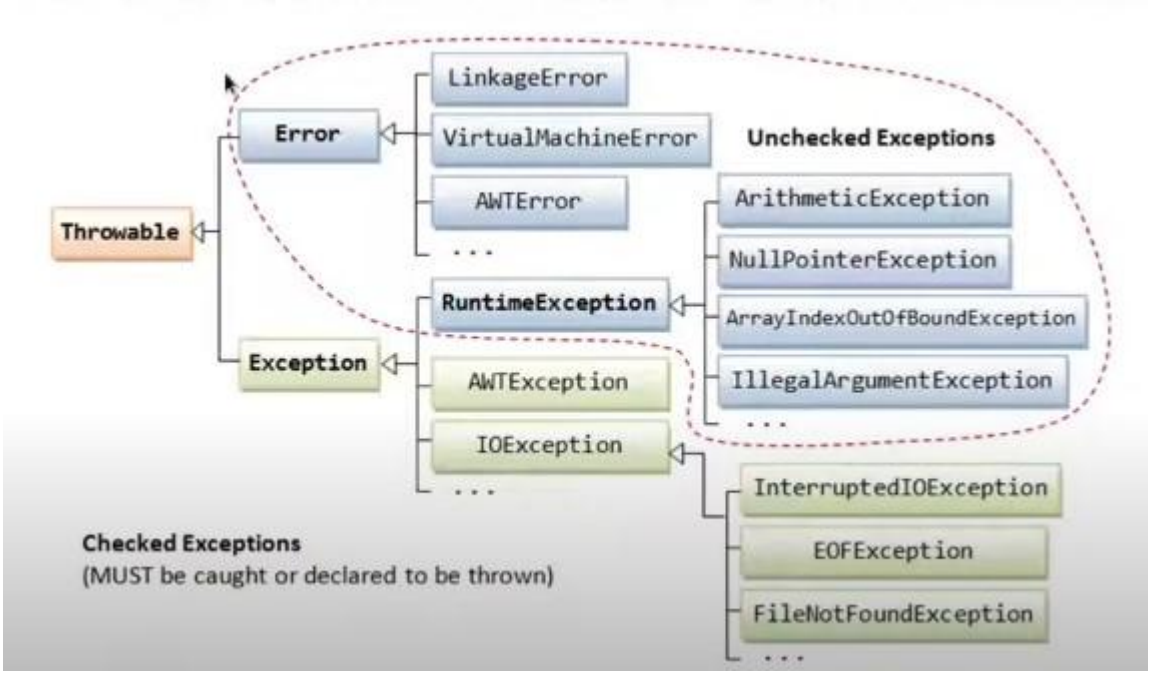
ciclo de vida da excessão:

- geração: sinalização de que uma condição excepcional aconteceu
- propagação: exceções podem se propagar em outras partes do programa através da pilha de chamada(call stack)
- tratamento: fluxo alternativo é definido por meio de construtores específicos da linguagem, blocos try-catch, dizemos que a exceção é pega(caught) ou tratada(handled)

Tipos de exceções em java

Em java, exceções são objetos. Toda exceção em java é instância da classe Throwable ou de uma subclasse dela. Existe uma hierarquia de exceções e classes diferentes para sinalizar problemas diferentes, ex: NullPointerException, ArrayIndexOutOfBoundsException. O programador pode definir suas próprias exceções, estendendo algumas das classes existentes.

Hierarquia de exceções



Checked exceptions

São situações de alguma forma esperadas, que a aplicação deveria antecipar. Por exemplo, ao tentar ler um arquivo devemos considerar a possibilidade dele não existir, lançamos FileNotFoundException. Exceções desse tipo devem sempre ser tratadas.

Unchecked exceptions

São situações inesperadas e normalmente fora do controle da aplicação. Indicam defeitos da aplicação, bugs, entradas inválidas ou falhas relacionadas a plataforma de execução.

Errors

Como vimos na tabela, existem dois objetos que herdam de Throwable, além de Exception, temos Error. Errors representam geralmente problemas da plataforma que a aplicação não pode tratar e nem deve. Erros são do tipo unchecked.

Geração de exceções

Exceções podem ser lançadas, automaticamente pelo sistema de execução java, ou, pelo programador. Lançar uma exceção envolve criar uma instância da exceção que representa o problema e lançá-lá através da palavra-chave throw. Ex:

- Classe ContaBancaria
  - Não pode completar a operação de saque se o saldo for menor do valor informado

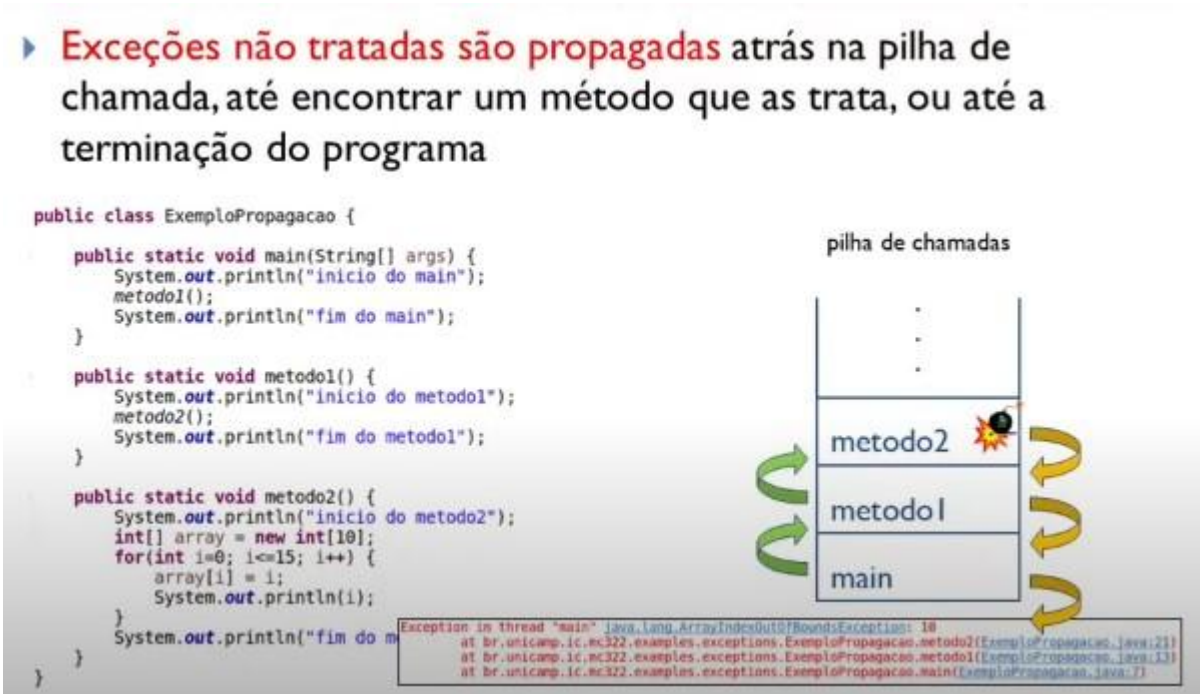
```
public class ContaBancaria {

    private float saldo;

    public ContaBancaria(float saldoInicial) {
        saldo = saldoInicial;
    }

    public void sacar(float valor) {
        if(valor > saldo) {
            // Saldo não é suficiente, o fluxo nominal
            // da operação não pode ser executado
            throw new IllegalArgumentException();
        }
        saldo = saldo - valor;
    }
}
```

Construtores da classe Exception

A classe exception contém quatro construtores públicos.
Exception() : cria uma exceção sem uma mensagem de erro específica
Exception(String message) : cria uma exceção com mensagem de erro específica
Exception(Throwable cause) : cria uma exceção com uma causa, passada por objeto do tipo Throwable
Exception(String message, Throwable cause) : cria exceção com mensagem de erro e causa.

Propagação

A pilha de chamada(call stack) armazena o contexto de execução dos métodos ativos, principalmente ponto de retorno e parâmetros. Exceções não tratadas são propagadas voltando na pilha de chamadas, até encontrar um método que as trata, ou até o programa terminar. Ex:



Tratamento

O tratamento de exceções é feito por um bloco try-catch-finally. As exceções são geradas no try e interceptadas em catch, um bloco finally pode ser opcionalmente adicionado ao final e abrange qualquer caso. Ex:

```
try {
    // codigo com possível erro
} catch (ExceptionType1 obj1) {
    // fluxo alternativo 1
} catch (ExceptionType2 obj2) {
    // fluxo alternativo 2
} finally {
    // funcionará para ambos os catchs
}
```

bloco try: identifica exceções geradas, sempre seguido por pelo menos um catch. Como outros blocos fechados por chaves {}, define um escopo, ou seja, as variáveis geradas dentro do bloco são válidas apenas ali dentro.

bloco catch: exception handler. Possui um parâmetro cujo tipo determina o tipo de erro tratado.

Ao ocorrer uma exceção, o bloco try é interrompido, o objeto exceção é passado para o primeiro bloco catch capaz de tratá-lo, se nenhum bloco catch é compatível, a exceção é propaganda para o método que chamou, se houver bloco finally, ele é executado ao fim.

Boas práticas

Blocos catch devem ser ordenados seguindo a exceção mais específica possível, até a mais genérica. Como o primeiro bloco compatível é executado, se a exceção genérica for passada primeiro, ela será sempre executada. Ex:

```
try {
    int n = a/b;
}
catch (ArithmeticException e) {
    System.out.println(e.getMessage());
    // Fluxo alternativo em caso de divisão por 0
    // ...
}
catch (Exception e) {
    System.out.println(e.getMessage());
    // Fluxo alternativo em caso de outros erros
    // ...
}
```

Não é boa prática criar tratador para exceções gerais do tipo Exception, sempre é melhor criar tratamentos bem específicos. Se um método catch foi adicionado, a exceção deve sempre ser tratada, é melhor nenhum bloco catch do que um bloco catch vazio, nesse caso estaríamos escondendo um erro e não tratando.

Métodos da classe Throwable