DESENVOLVIMENTO BÁSICO EM JAVA

Ferramentas utilizadas para instalação do Java:

* GANDLE - sistema avançado de automatização de builds
* MAVEN - **O Apache Maven é uma ferramenta de automação e gerenciamento de projetos Java**, embora também possa ser utilizada com outras linguagens. Ela fornece às equipes de desenvolvimento uma forma padronizada de automação, construção e publicação de suas aplicações, agregando agilidade e qualidade ao produto final
* INTELLIJ IDE

JAVA – Criação de Projetos

<https://start.spring.io/>

O QUE PRECISAMOS SABER SOBRE JAVA?

O que é JAVA?

É uma linguagem de programação e plataforma computacional criada em 95 pela San Microsystems, em um time comandado por James Gosling. Comprada anos depois pela Oracle.

Diferente de outras linguagens que são compiladas para código nativo, Java é compilado para um *bytecode* interpretado por uma máquina virtual.

O que é o COMPILADOR?

Programa que, a partir de um código fonte, cria um programa semanticamente equivalente, porém escrito em outra linguagem, código objeto. Ele traduz uma linguagem textual para uma linguagem de máquina específica para um processador e SO.

O nome compilador é usado principalmente para os programas que traduzem o código fonte de uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível (ex.: ASSEMBLY, CÓDIGO DE MÁQUINA, etc).

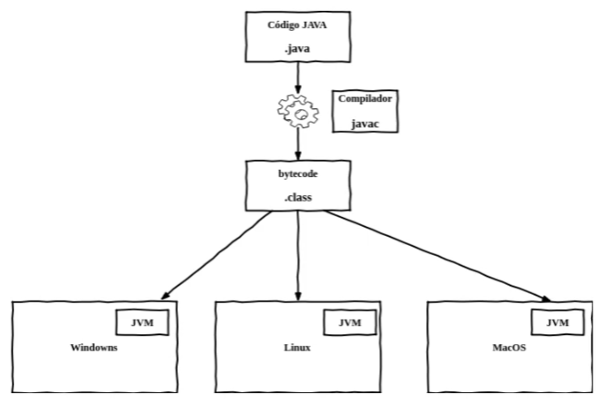
O que é o BYTECODE?

É o código originado da compilação de programas JAVA. É um programa interpretado e executado pela Máquina Virtual Java, JVM.

O que é uma VM?

Uma Máquina Virtual, ou Virtual Machine, é um software de simulação de uma máquina física que consegue executar vários programas, gerenciar processos, memória e arquivos. Isso tudo faz parte de uma plataforma com memória, processador e outros recursos totalmente virtuais, sem dependência do hardware.

O que é a JVM?

 É a máquina virtual que executa programas JAVA, executando os bytecodes em linguagem de máquina para sistema operacional.

Em linguagens compiladas diretamente para um SO específico, esse programa não irá executar em outro SO, sendo necessário então a compilação de uma versão do software para cada SO.

Com o JAVA, compilamos para a JVM, o bytecode será executado pela máquina virtual, e não diretamente no SO, desta forma o código escrito em JAVA possui portabilidade para qualquer SO, porém cada JVM deve ser construída para um SO específico.

JAVAC é o compilador do JAVA.

O que é a JRE?

JRE significa Java Runtime Environment, ou Ambiente de Execução do Java. Ele é composto pela Java Virtual Machine (JVM), bibliotecas e APIs da linguagem e outros componentes para suporte da plataforma JAVA.

É a parte responsável pela execução do software JAVA.

O que é o JAVA SE?

Java Standard Edition (SE), é a distribuição mínima da plataforma de desenvolvimento de aplicações JAVA.

OpenJDK é a implementação de referência opensource da plataforma JAVA, JAVA SE, que ainda é mantida pela Oracle.

O que é JAVA EE?

Java Enterprise Edition, é uma extensão do JAVA SE que possui suporte a desenvolvimento de sistemas corporativos.

Além do mínimo da plataforma JAVA, ele possui diversas especificações de partes da infraestrutura de aplicações, como acesso a banco de dados, mensageria, serviços web, parser de arquivos e outras.

Servidores de Aplicações Java EE, sabem seguir essas especificações e implementar os recursos para outros usuários.

Exemplos de servidores de aplicação Java EE: JBoss (RedHat), Weblogic (Oracle), WebSphere (IBM) e Glassfish (implementação de referência opensourse).

O que é JAKARTA EE?

Com a falta de investimento da Oracle no Java, ela cedeu todo o código, implementações e especificações do Java EE para a Eclipse Foundation, mas como o nome Java EE é uma marca registradas, foi escolhido no nome Jakarta EE.

Agora a evolução de especificações e padrões do Java será feita sob o nome Jakarta EE, com compatibilidade com o Java EE.

COMPILANDO CÓDIGO JAVA NO TERMINAL

LINUX - TERMINAL

Mkdir hello-java – cria o diretório

Cd hello-java – entra no diretório criado

Hello-java vim Hello.java – cria o arquivo .java

Após criar o arquivo .java inserimos um código dentro do arquivo para ser executado.

Javac Hello.java – compila o código do arquivo .java.

Java Hello – executa o código após ser compilado

Cat Hello.java – utilizado para ver o código fonte do arquivo .java.

INICIANDO UM PROJETO EM JAVA

1. Inicie o Intellij;
2. New Project
3. Escolha Maven ou Gradle
4. Name = empresa.da.nome – exemplo: one.innovation.digital
5. Finish
6. Verifique se as estruturas do projeto estão todas para o Java 11
   1. File -> Project Structure
   2. Project SDK = 11
   3. Project Language Level = 11 – Local variable syntax for lambda parametes

CLASSES

package one.digitalinnovation.classes;

public class Programa {

public static void main(String[] args){

System.out.println(“Hello world”);

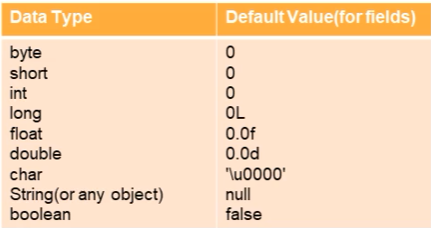
}

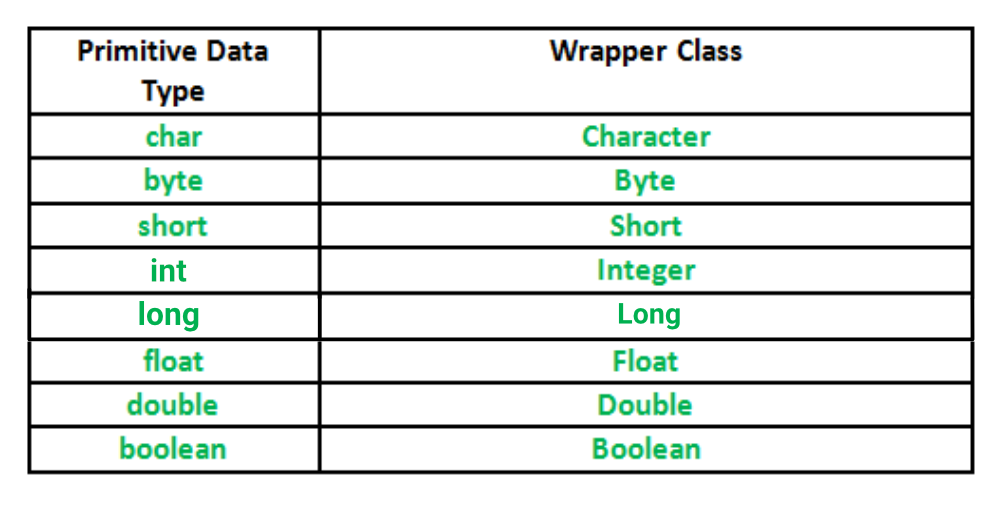
}

* Por convenção classes em Java sempre iniciam por letra maiúscula, nomes compostos também. Ex.:ImprimeTela, etc;
* O método principal para executar um programa em Java é o public static void main;
* (String[] args) – são os argumentos do método, sua assinatura, não necessariamente seu retorno
* Compõem uma classe: classe, atributos, construtores e métodos

TIPOS

* Primitivos
  + Não aceitam null
  + Possuem valores default



* Byte possuí 8 bits, portanto seu valor máximo vai de -128 até 127
* Short possuí 16 bits indo de -32768 até 32767
* Int possuí 32 bits indo de -2147483648 até 2147483647
* Long possuí 64 bits indo de -9223372036854745808L até 9223372036854775807L. É necessário inserir um L ao seu final
* Float possuí 32 bits e pode ser usado para números não inteiros. É necessário inserir um F ao final do número
* Double possuí 64 bits
* Char possuí 16 bits
* Boolean – TRUE ou FALSE
* Wrappers
  + São objetos que representam os primitivos através de uma classe
  + Auto-boxing – o compilador é responsável por autoconstruir o objeto, seguindo as mesmas regras dos tipos primitivos, quanto a seus tamanhos máximos e quantidade de caracteres. Aqui conseguimos atribuir null devido a variável se tornar um objeto.
  + Unboxing – conseguimos atribuir um wrapper a um tipo primitivo
    - Ex.: int inteiro = Integer.valueof(1024);
    - boolean b2 = Boolean.TRUE;

Então o unboxing é passar uma classe (instanciar um objeto) a um tipo inicialmente primitivo.

* Não primitivos
  + String – sequência de caracteres
  + Number -
  + Object – objeto principal do Java
  + Quaisquer outros objetos
* Tipagem forte (uma vez atribuído um tipo a variável não é possível alterá-lo) e estática (os tipos das variáveis são verificados em tempo de compilação, ou seja, não permite a variável receber outro tipo de atribuição além da qual foi criada. Ex.: Integer numero = “123456”; retorna erro porque não recebe String apenas números)
* Tipo Inferido – conseguimos criar variáveis a atribuir tipos a elas sem ter de falar de forma explícita qual o seu tipo, utilizando apenas VAR

var numero = Integer.valueOf(“123456”);

var texto = “O número é: “;

MODIFICADORES DE ACESSO

* Public
  + Pode ser acessado de qualquer lugar por qualquer entidade que possa visualizar a classe a que ela pertence
* Private
  + Métodos e atributos da classe não podem ser acessados ou usados por nenhuma outra classe. Esses atributos e métodos também não podem ser visualizados pelas classes herdadas.
* Protected
  + Torna o membro acessível às classes do mesmo pacote ou através de herança, seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.
* Default (padrão)
  + A classe e/ou seus membros são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador
* Abstract
  + Não pode ser aplicado em variáveis apenas em classes e métodos. Não pode ser instanciada. Caso houver algum método declarado como abstract a classe também deve ser marcada como abstract.
  + Ser uma classe abstrata significa que ela é uma ideia de uma classe, uma forma geométrica, por exemplo, onde podemos ter classes que são herdeiras dela, como um quadrado.
  + As classes herdeiras dela devem sobrescrever os métodos herdados.
* Static
  + É utilizado para criação de uma variável que estará disponível a todas as instâncias de objetos desta classe como uma variável comum. Ela será a mesma em todas as instâncias, ou seja, se modificada em uma parte essa alteração ocorre em todas as outras. Nas declarações de métodos ajudam no acesso direto à classe, portanto não é necessário instanciar um objeto para acessar o método.
* Final
  + Não permite estender, nos métodos impede que o mesmo seja sobrescrito (overriding) na subclasse, e nos valores de variáveis não pode ser alterado depois que já tenha sido atribuído um valor.

INTERFACES

A interface é um recurso muito utilizado em Java, bem como na maioria das linguagens orientadas a objeto, para “obrigar” a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente. Pode-se dizer, a grosso modo, que uma interface é um contrato que quando assumido por uma classe deve ser implementado.

* Métodos Abstratos
  + Devem ser implementados por todos
  + Novos métodos quebram as implementações
* Métodos Default
  + São herdados a todos que implementam
  + Novos métodos não quebram as implementações
* Herança Múltipla – herda de mais de uma classe, em Java só é possível no conceito de interfaces
  + Ex.:

public class Fiesta implements Carro, Veiculo {

@Override

public String marca() {

return “Ford”;

}

@Override

public void ligar() {

Carro.super.ligar();

Veiculo.super.ligar();

}

}

INTERFACES vs ABSTRAÇÕES

Interfaces – é uma espécie de classe, que tem nome e informações, aqui não conseguimos colocar modificadores de acesso, portanto tudo inserido nela é public e static. Ex.: interface cliente possui nome, cpf, rg, etc.

Classe Abstrata – é uma classe que herda de outras classes. Ex.: classe Pessoa dela são herdadas outras classes como Cliente e Cidadão, por exemplo, que são herdeiras da classe Pessoa. Temos aqui o conceito de herança onde Cliente e Cidão herdam da classe Pessoa suas características, como por exemplo, atributos como Nome, Tipo Sanguíneo, Idade, Data de Nascimento, etc., métodos como andar, falar, pensar, etc..

Nela também é possível termos métodos privados, métodos públicos, métodos protegidos, etc. Não conseguimos instanciar a classe (new Cidadão). Nem toda linguagem suporta herança múltipla.

Quando usar interfaces?

Quando a classe necessitar de um contrato, ou seja, quando for preciso garantir que uma classe vai possuir exatamente aqueles métodos e atributos. Então basicamente ela é um contrato de garantia que o que for usado por ela terá exatamente aqueles métodos e atributos.

Quando usar classes abstratas?

Quando for aplicar padrões de projetos, para boas práticas de programação, principalmente pensando em reuso. Ela é uma abstração, serve efetivamente para conter informações abstratas, códigos abstratos, coisas que vão se concretizar futuramente, ou coisas que são efetivamente comuns para vários objetos.

É um contrato? Use interface

É uma abstração? Use uma classe abstrata

ENUMS

* São conjuntos de valores pré-definidos;
* É uma espécie de dicionário de dados imutável;
* Não é permitido criar novas instâncias;
* O construtor é sempre declarado como private;
* Por convenção, por serem objetos constantes e imutáveis (static final), os nomes são em MAIÚSCULOS.

Exemplo: Estamos fazendo o sistema de um curso, suas aulas podem acontecer de MANHÃ, TARDE ou NOITE.

public enum PeriodoEnum {

MANHA,

TARDE,

NOITE

}

Utilizamos enum para padronizar determinadas informações que serão padrão em todo o sistema e para garantir que serão sempre as mesmas informações.

STRINGS

É uma sequência de caracteres. Está inserida dentro do pacote java.lang.

var nome = “André”;

var sobreNome = “Gomes”;

final var nomeCompleto = nome + sobreNome; --concatenando strings

MÉTODOS EM STRINGS

var string = “ Minha String ”;

System.out.println(“Char na posição : “ + string.charAt(5)); --retorna o caractere na posição 5 (a), Strings são arrays onde cada índice armazena um caractere do texto.

System.out.println(“Quantidade =: “ + string.length()); --retorna o tamanho da String

System.out.println(“Trim [“ + string.trim() + “]”); --remove os espaços nas extremidades. Retorna “[Minha String]”

System.out.println(“Lower “ + string.toLowerCase()); --retorna a String minúscula

System.out.println(“Upper “ + string.toUpperCase()); --retorna a String maiúscula

System.out.println(“Contém M? “ + string.contains(“M”)); --retorna TRUE devido a String conter M, porém se fosse X, por exemplo, retornaria FALSE

System.out.println(“Replace “ + string.replace(charSequence:”n”, charSequence1:”$”)); --troca o n por $ -> “Mi$ha Stri$g”

Existem diversos outros

IF

if (condição) {

} else {

}

TERNARIO (IF)

final var ternario = condição ? “é verdadeira” : “é falsa”;

OPERADORES

|  |  |  |
| --- | --- | --- |
| == | variavel.equals(valor) | Igual |
| != | !variavel.equals(valor) | Diferente |
| + | x = x + 2 ou x += 2 | Adição |
| - | x = x – 2 ou x -= 2 | Subtração |
| \* | x = x \* 2 ou x \*=2 | Multiplicação |
| / | x = x / 2 ou x /= 2 | Divisão |
| % | x = x % 2 ou x %= 2 | Módulo / Resto da Divisão |
| > |  | Maior que |
| < |  | Menor que |
| >= |  | Maior ou igual a |
| <= |  | Menor ou igual a |
| && |  | e |
| || |  | ou |
| ++numero | Incrementa primeiro depois avalia a expressão | Incremento |
| numero++ | Avalia primeiro incrementa depois | Incremento |
| --numero |  | Decremento |
| numero-- |  | Decremento |

Exemplos de incremento/decremento:

var variavel = 10;

System.out.println(variavel--)

System.out.println(variavel)

//10 imprimiu primeiro decrementou depois

//9 imprimiu

Short Circuit

Se a avaliação de uma expressão lógica sair no meio antes da avaliação completa, é conhecido como short circuit. Caso a condição de execução de um if seja && ou || temos a seguinte lógica:

* &&
  + Caso a primeira condição seja false ele para a execução e vai para o else, uma vez que ambas as condições precisam ser verdadeiras para serem executadas
* ||
  + Caso a primeira condição já seja verdadeira ele já executa.

Tendo esses conceitos em mente entendemos o que o short circuit é.

Non Short Circuit

Utilizando apenas & e | ele verifica as duas condições para poder executar.

FOR

for (int i = 0; i <= 10; i++) {

System.out.println(“I=” + i);

}

WHILE

var x = 0;

While (x < 1) {

System.out.println(“Dentro do while...”);

x++;

}

var y = 0;

//testa a condição depois

Do {

System.out.println(“Dentro do do/while...”);

} while (y++ < 1)

IntStream

São uma espécie de for.

IntStream.*of*(1,2,3,4,5).forEach(n -> {  
 System.*out*.println(n);  
});

1

2

3

4

5

IntStream.*range*(0,3).forEach(n -> {  
 System.*out*.println("Número: " +n);  
});

Número: 0

Número: 1

Número: 2

CONVENÇÕES DE NOMES

* Nomes de Classes
  + Classe – primeira letra maiúscula demais minúsculas
  + CamelCase – em classes como nome composto
* Nomes de Métodos
  + método – sempre minúsculo
  + nomeComposto – primeira letra minúscula o restante CamelCase
* Nomes de Variáveis
  + variavel

PLUGINS

No arquivo build.gradle insere-se os plugins Checkstyle (ajuda a seguir um padrão de codificação (Sun Code ou Google Java Style)) e PMD (analisador de código-fonte. Ele encontra falhas de programação comuns, como variáveis ​​não utilizadas, blocos catch vazios, criação desnecessária de objetos e assim por diante).

plugins **{** id 'java'  
 id 'checkstyle'  
 id 'pmd'  
**}**

DEBUG

Quando se é necessário olhar o código linha a linha, utilizamos o debug para isso. Isso é feito para encontrarmos bugs dentro do código.

ORIENTAÇÃO A OBJETOS

Segundo Maurício Aniche em Código Procedural o que mais importa é a escolha do melhor algoritmo. Já em OO pensar no projeto de classes, em como se encaixam, e como serão estendidas é o que mais importa.

CLASSE

É um modelo a ser seguido. Ela será o molde para construção de um objeto. Por exemplo ao construirmos uma casa, primeiro construímos sua planta baixa que será o modelo a ser seguido para sua construção, para construir algo concreto.

OBJETO

Após termos uma planta baixa iniciamos a construção da casa, o resultado que construirmos chamaremos de objeto. Por exemplo a classe CASA será o molde para diversas outras casas iguais a ela.

Quando utilizamos a classe para criar um objeto, dizemos que estamos INSTANCIANDO um objeto da classe. Podemos criar vários objetos de uma mesma classe, ou seja, várias instâncias de objetos.

ATRIBUTO

São as “características”, propriedades da classe. Por exemplo uma classe Pessoa tem como atributo nome.

MÉTODOS

São as “ações” de uma classe. Uma classe Pessoa teria como método falar, andar, respirar, etc.

CONSTRUTORES

Servem para construirmos um objeto. Através dele criamos um objeto baseado em uma Classe e assim o alocamos em memória.

Public Class Pessoa {

public Pessoa{ construtor padrão

}

}

É possível utilizarmos construtores parametrizados também. Dessa forma conseguimos definir um contrato onde sempre será obrigatórios passar alguma informação na hora de instanciar a classe.

Public Class Pessoa {

public Pessoa{ construtor padrão

}

public Pessoa (String nome) {

this.nome = nome;

}

}

Termos dois construtores dentro da classe, como no exemplo acima, permite que possamos instanciar o objeto das duas maneiras, passando ou não o parâmetro nome. Caso a classe possua apenas o construtor com passagem de parâmetro, torna-se obrigatório, ao instanciar o objeto, passarmos o parâmetro solicitado.

ENCAPSULAMENTO

É encapsular uma variável dentro da classe, ou seja, não permitir que todos tenham acesso direto as variável, deixa-los protegidos.

HERANÇA

É a capacidade de uma classe herdar o comportamento de outra.

public class Veiculo {

private String modelo;

private String marca;

}

public class Carro extends Veiculo{

private int quantidadeDePortas;

}

public class Motocicleta extends Veiculo {

private String cilindradas;

}

Como herança de Veículo, as classes Carro e Motocicleta herdam os atributos modelo e marca. Definimos nessas classes herdeiras os getters e setters de acesso as variáveis. O conceito de herança também se estende aos métodos da classe pai. Por exemplo, a classe Veículo pode possuir os métodos acelerar e parar que seria comum em qualquer veículo, portanto suas classes filhas podem herdar esses métodos.

HERANÇA vs COMPOSIÇÃO

Há algumas bibliografias que dizem que herança nunca deve ser utilizada. O grande problema tem relação a questão do encapsulamento. A subclasse em muitos momentos acaba tendo que conhecer a implementação da superclasse, o que cria um acoplamento que quebra a primícia básica do isolamento.

POLIMORFISMO

Ao falarmos sobre herança, o verbo ser é mandatório na forma de falarmos sobre classe. Entendemos, portanto, que um carro é um veículo e uma motocicleta também é um veículo.

Além disso é necessário haver implementações diferentes de acordo com a classe que estamos trabalhando. Por exemplo, taxas de IPVA, onde veículos possuem pelo menos 0.01% de taxa, carros pelo menos 0.07% e motocicletas pelo menos 0.03%. É ai que entra o polimorfismo, ele nos garante que um objeto poderá ser referenciado de múltiplas formas. O Java é capaz de identificar qual objeto foi instanciado e, assim, escolher qual método será utilizado.

Portanto o método será escrito na classe Veiculo e reescrito, de acordo com as regras de cada classe, nas classes Carro e Motocicleta caso as regras sejam diferentes.

THIS

Ao utilizarmos o this estamos fazendo uma auto referência. Ele faz sentido nos conceitos de construtores e métodos.

public Funcionario(String nome, Double salario) {  
 this.nome = nome;  
 this.salario = salario;  
}

Acima estamos atribuindo ao escopo da classe Funcionario o this.nome, que é o nome do objeto a ser criado, recebe o parâmetro que o usuário forneceu. Por isso utilizamos o this.

SUPER

Análogo ao this, só que utilizado em construtores de classes herdeiras onde os atributos passados a elas são os mesmos da classe pai.

public Gerente(String nome, Double salario) {  
 super(nome, salario);  
}

A classe Gerente é herdeira da classe Funcionario, portanto seu construtor recebe os mesmos parâmetros de instanciamento. Então o super esta fazendo referência a superclasse.

EQUALS

É um método herdado da classe Object, como todas as classes em Java herdam dela ele está disponível para todas as classes. O Equals é utilizado para fazermos comparações entre objetos.

Quando comparamos dois objetos, estamos comparando referências deles. Então se instanciarmos dois carros, por mais que eles tenham exatamente as mesmas informações, o Java não é capaz de identificar.

Para resolvermos isso podemos sobrescrever o método equals() para que nossa lógica funcione do jeito que gostaríamos. Fixe que é uma boa prática sobrescrever esse método.

@Override  
public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
  
 Veiculo comparavel;  
 if (obj instanceof Veiculo) {  
 comparavel = (Veiculo)obj;  
 } else {  
 return false;  
 }  
  
 if (comparavel.marca == this.marca && comparavel.modelo == this.modelo && comparavel.valorVenal == this.valorVenal) {  
 return true;  
 }  
  
 return false;  
}

HashCode

Quando falamos em hashCode, precisamos pensar em um código gerado que garanta um caráter único ao nosso objeto. Essa pode ser uma forma muito interessante para que possamos comparar se realmente um objeto é igual ao outro.

Temos que garantir que a implementação da lógica de hashCode sempre respeite as mesmas regras, pois quando comparamos os nossos objetos, o nosso fator de comparações será ele.

Por exemplo anteriormente utilizamos o método equals() para fazer a comparação entre dois objetos. Entretanto, nós fizemos essa comparação utilizando explicitamente 3 atributos: modelo, marca e valor venal. Sendo que poderíamos transferir essa responsabilidade para nosso hashCode.

Veiculo comparavel;  
if (obj instanceof Veiculo) {  
 comparavel = (Veiculo)obj;  
} else {  
 return false;  
}  
  
if (this.hashCode() == obj.hashCode()) {  
 return true;  
}  
  
return false;

Ou sobrescreve-lo

public int hashCode() {

return Objects.hash(modelo, marcam, valorVenal);

}

PRINCIPIOS DE SOLID

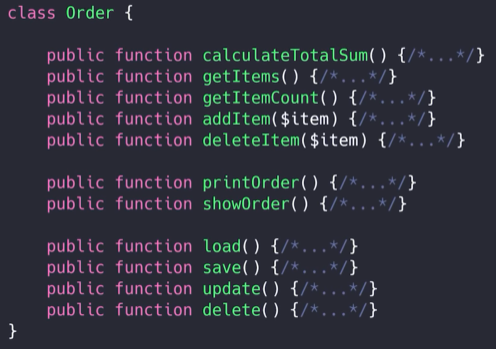
SOLID é um acrônimo dos princípios de POO descritas por Robert C. Martin (“Uncle Bob”). Auxiliam o programador a escrever códigos mais limpos, facilitando a refatoração e estimulando o reaproveitamento do código.

SINGLE RESPONSABILITY PRINCIPLE - SRP

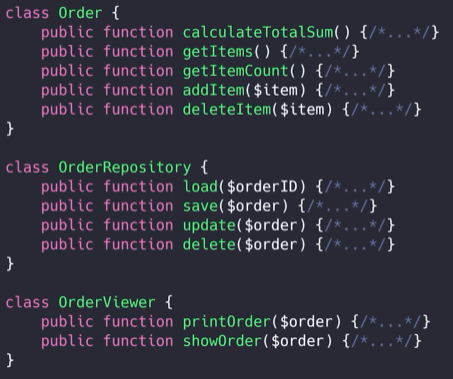
PRINCIPIO DA RESPONSABILIDADE ÚNICA

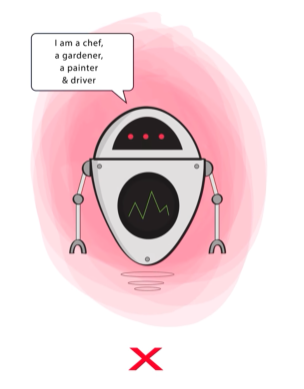
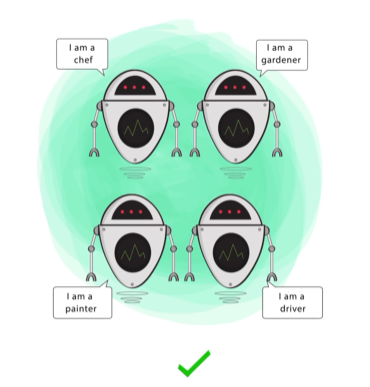
“Uma classe deve ter um, e somente um, motivo para mudar.”

A classe deve possuir uma única responsabilidade dentro do software. Ela deve ter uma única tarefa ou ação para executar.



Classe sem o SRP.

Aplicando o SRP.



OPEN-CLOSED PRINCIPLE - OCP

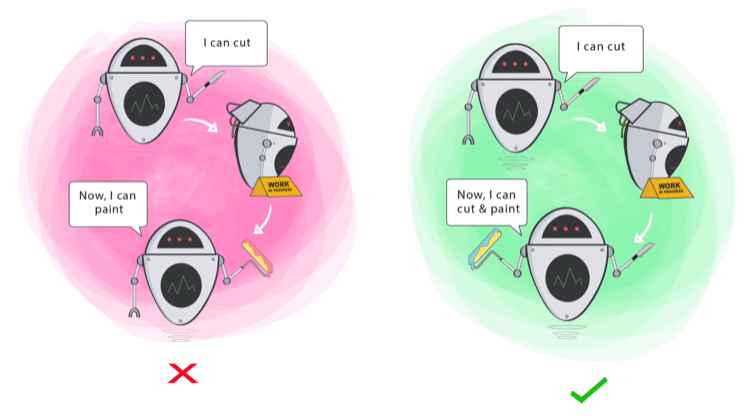
PRINCIPIO ABERTO FECHADO

“Você deve poder estender um comportamento de classe, sem modifica-lo.”

Objetos ou entidades devem estar abertos para extensão e fechados para modificação.

Quando novos comportamentos precisam ser adicionados no software, devemos estender e não alterar o código fonte original. Ao alterar uma classe já existente para inserir novos comportamentos, corremos o risco de inserir bugs no código em algo que já funcionava.

Então, segundo “Uncle Bob”, o que deve ser feito é separar o comportamento extensível através de uma interface, inverta as dependências. O que devemos fazer é abstrair o contexto em uma interface. Se as abstrações estão bem definidas ela está aberta para ser estendida. Então teremos uma nova classe que se estende de outra e cria novos métodos. Entende-se então que essa nova classe herda os métodos da antiga e implementa novos podendo realizar qualquer um deles.



Pense como exemplo o Google Chrome, ao instalar plugins você não alterou o código fonte dele, mas mesmo assim adicionou funcionalidades novas. Então o navegador está OPEN para estender novas funcionalidades, mas a base dele está CLOSED.

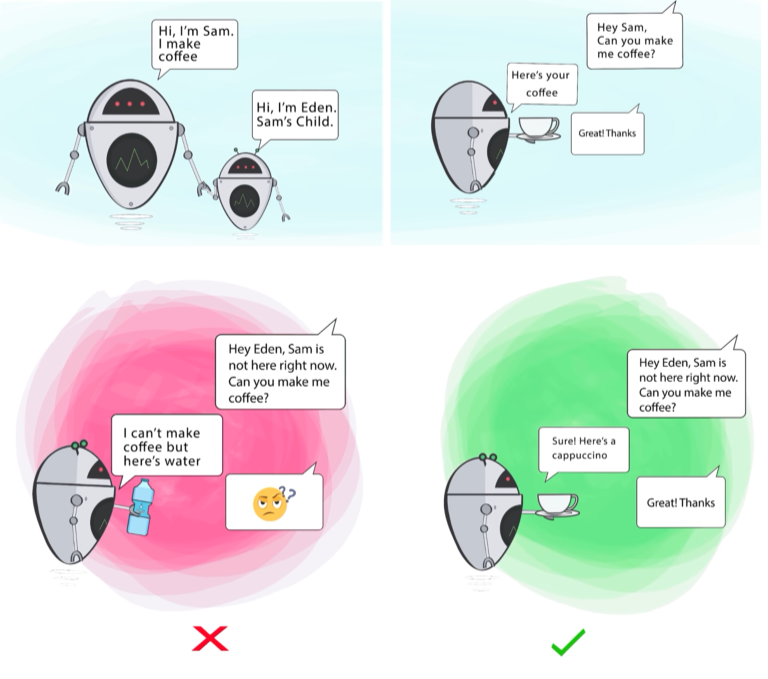
LISKOV SUBSTITUTION PRINCIPLE - LSP

PRINCIPIO DA SUBSTITUIÇÃO DE LISKOV

“Classes derivadas dever sem substituíveis por suas classes base.”

Esse principio foi introduzido por Barbara Liskov em 1987.

“Se para cada objeto O1 do tipo S há um objeto O2 do tipo T de forma que, para todos os programas P, o comportamento de P é inalterado quando O1 é substituído por O2, então S é um subtipo de T”



Respeitar o principio de Liskov força o seu programa a ter as abstrações no nível certo e ser mais consistente.

Por exemplo, criamos uma classe Ave() e nela os métodos bicar() e voar(). Tendo uma classe por herança PicaPau() todos os métodos se encaixam por fazerem sentido. Já criando outra classe por herança Pinguim() o método voar já não se encaixa.

Então se a cada herança que fizer tiver essa luta com o que se herda da superclasse e precisar desfazer o que herdou é bem possível que a abstração já começou errada. Então esse principio vai forçar a pensar o que a classe pai deve ter de comum para todas as outras subclasses porque isso será fundamental para respeitar o princípio de Liskov, levando em consideração que a classe filha pode e deve fazer tudo o que a classe pai faz, além de ser capaz de fazer além com novos métodos implementados.

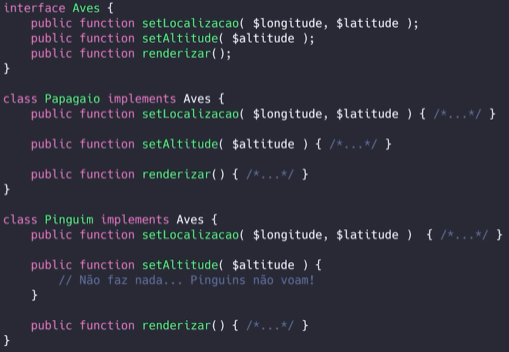
INTERFACE SEGREGATION PRINCIPLE

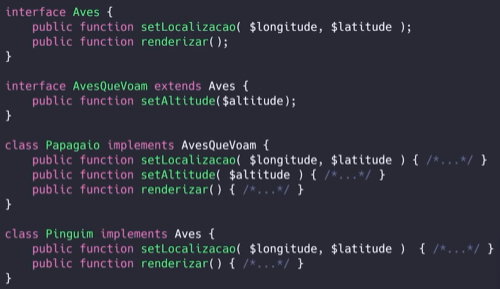
PRINCIPIO DA SEGREGAÇÃO DA INTERFACE

“Faça interfaces refinadas que são específicas do cliente.”

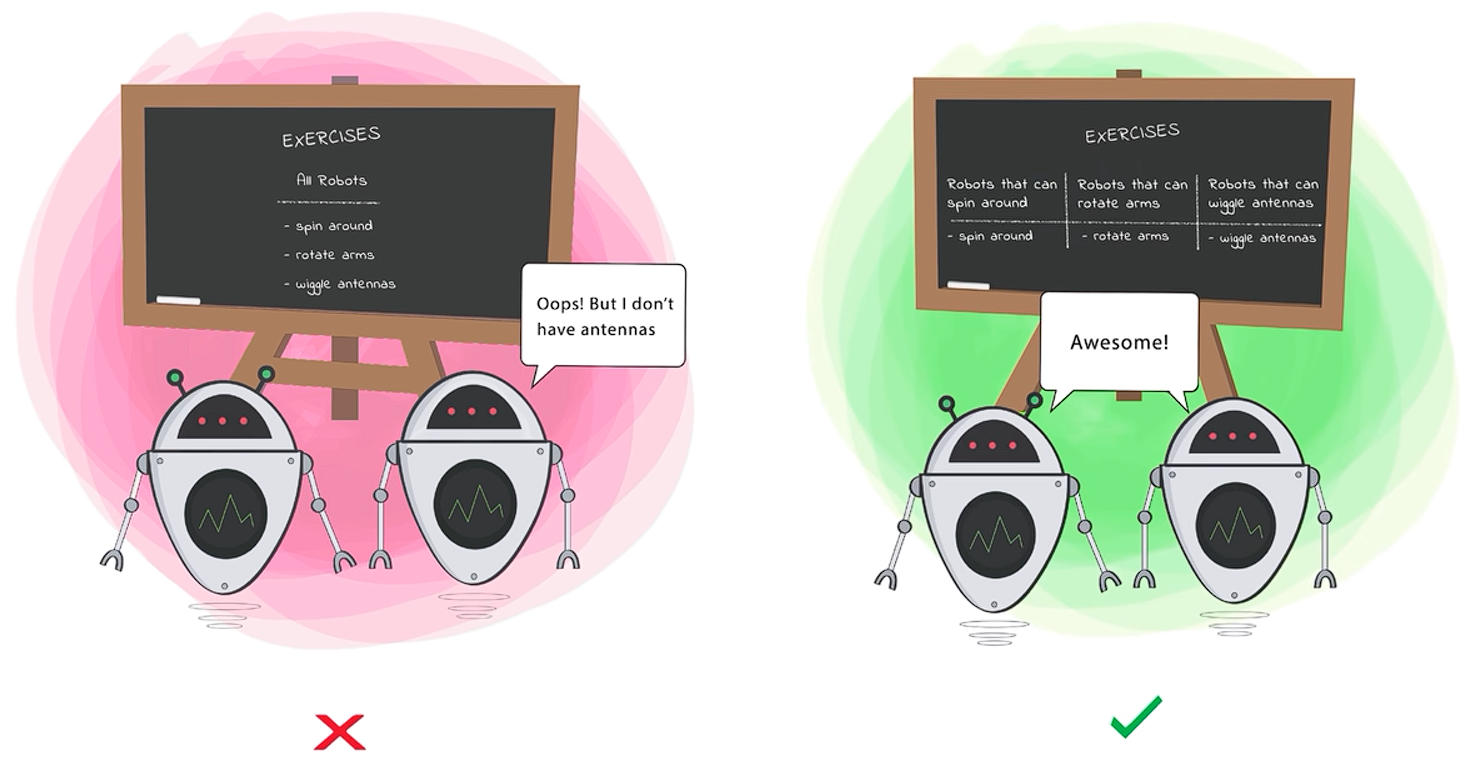
Uma interface não deve ser forçada a implementar interfaces e métodos que não serão utilizados por ela.

É melhor criar interfaces mais especificas ao invés de termos uma única interface genérica.

Aqui Pinguim acaba implementando setAltitude, mesmo que ele não voe.

O que deve ser feito é uma outra interface que se extenda de Aves e implemente o método setAltitude. Assim classes que são implementadas por ela recebem todas as características de Aves além do método setAltitude.

Clientes não devem ser forçados a depender de métodos que eles não usem.



DEPENDENCY INVERSION PRINCIPLE - DIP

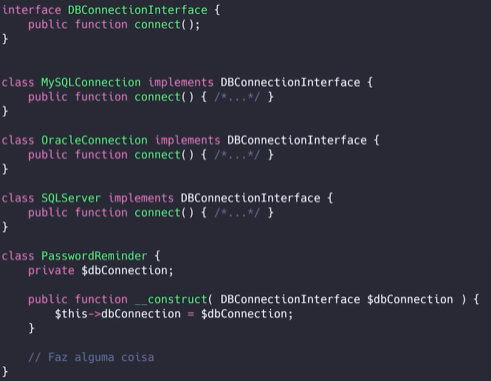
PRINCIPIO DA INVERSÃO DA DEPENDÊNCIA

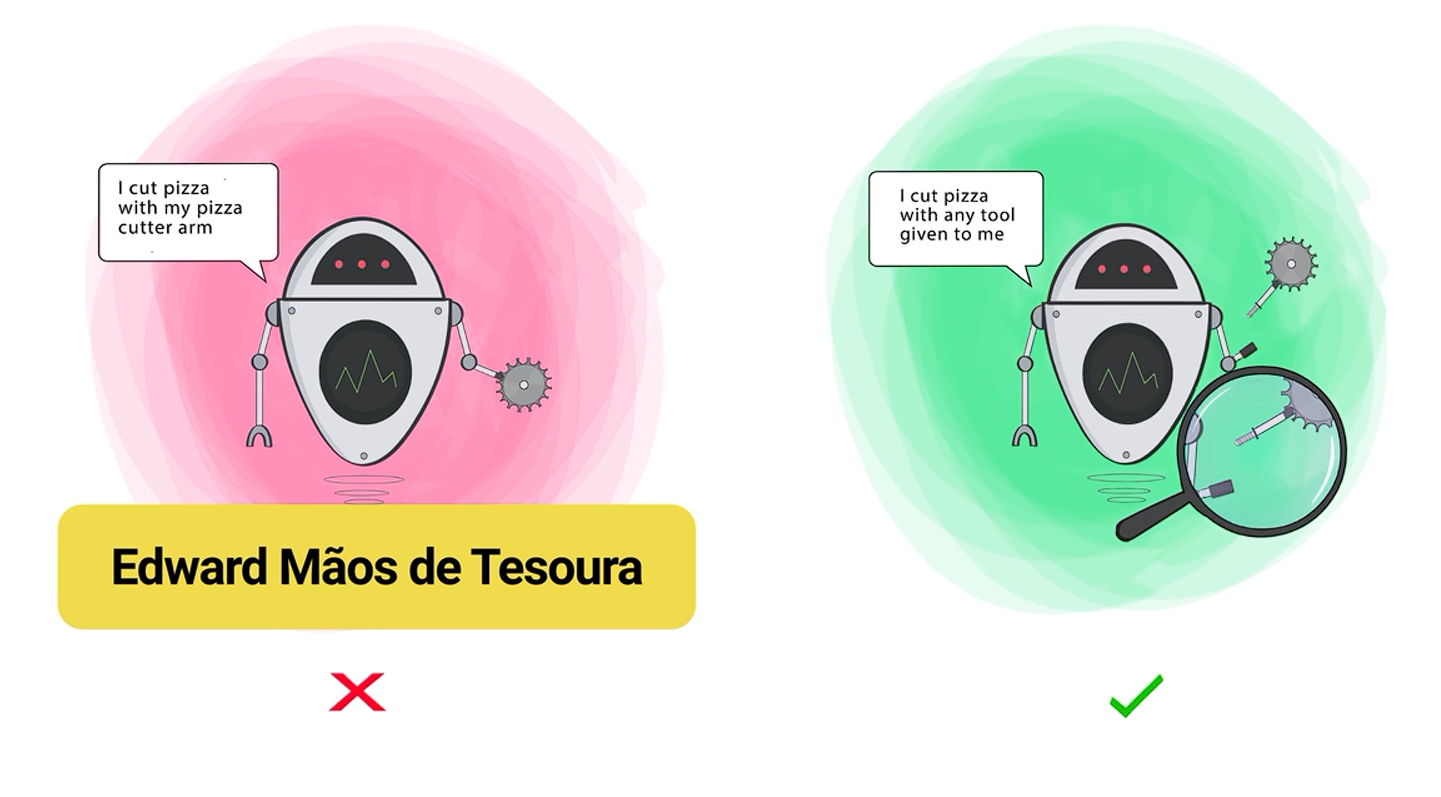
“Dependa de abstrações e não de implementações.”

Um módulo de alto nível não deve depender de módulos de baixo nível, ambos devem depender da abstração. Abstrações não devem depender de detalhes, detalhes devem depender de abstrações.

 PS: Inversão de Dependência não é igual a Injeção de Dependência

Aqui temos uma classe que depende da implementação da classe MySQLConnection.

Aplicando os conceitos do DIP, criamos uma interface que será a responsável por criar o método connect(). Sendo as outras classes que a implementarem, responsáveis por rescrever seu próprio método connect();



TRABALHANDO COM DATAS

O java.util.Date foi implementado na JDK desde sua versão 1.0, ou seja, é de se esperar que algumas coisas não se mostrem tão interessantes nos dias atuais, dado a sua idade.

Usaremos como referência o Java 8.

<http://docs.oracle.com/javase/8/docs/api/java/util/Date.html>

Date()

Esse construtor irá alocar um objeto da classe Date e o inicializará com o milissegundo mais próximo do período da sua execução.

Date novaData = new Date();

//retorna: Thu Jul 08 09:55:08 BRT 2019

Dia da semana, mês, dia, hora e ano.

Date(long date)

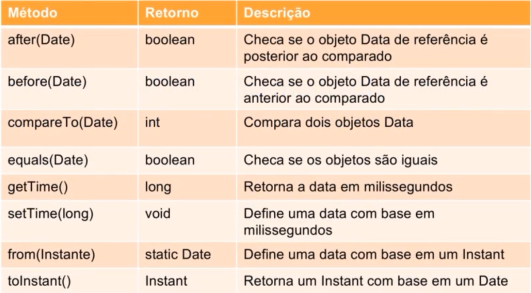
Diferente do construtor anterior, esse construtor espera que você passe os milissegundos com base padrão de tempo (epoch) que usa como referência 1 de Janeiro de 1970 00:00:00.

O que é o Epoch?

“O epoch timestamp é um padrão largamente aceito para representar uma data como um inteiro 32-bits a partir do inicio do Unix Epoch (1 de Janeiro de 1970 00:00:00) ...”

Vamos testar com base no System.currentTimeMillis(). Esse método estático vai nos retornar o milissegundo mais próximo de sua execução com base no SO.

MÉTODOS ÚTEIS

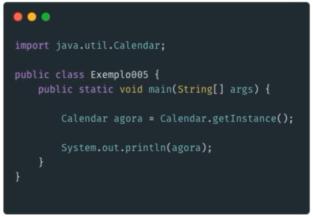


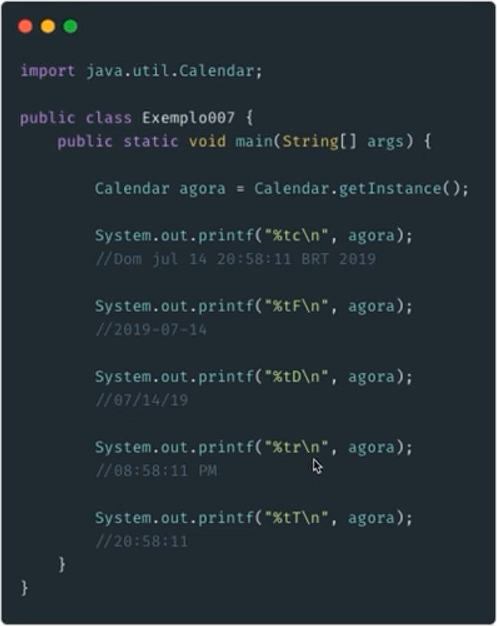
Class Instant

* Surgiu na JDK 1.8;
* Imutável e Thread safe;
* Modela um ponto instantâneo de uma linha do tempo;
* Indicado para gravar marcações temporais em eventos da sua aplicação.

JAVA.UTIL.CALENDAR

Calendar é uma classe abstrata que provê métodos para converter data entre um instante específico. O Calendar possui alguns campos específicos para manipulação como MONTH, YEAR, HOUR, etc.



IMPRIMINDO DATAS E HORAS

JAVA.TEXT.DATEFORMAT

Aqui existem duas classes para formatação de datas: o DateFormat e o SimpleDateFormat. Ambos oferecem maneiras de formatar e parsear a saída das datas.

DateFormat



SimpleDateFormat

Já o SimpleDateFormat traz uma grande facilidade que é definir um padrão de formatação para a saída de data que você deseja.



DATAS NO JAVA 8+

A grande melhoria do trabalho de datas no Java está no pacote java.time que foi herdado do projeto Joda Time.

<https://www.joda.org/joda-time/>

Neste pacote se destacam três classes:

-LocalDate;

-LocalTime;

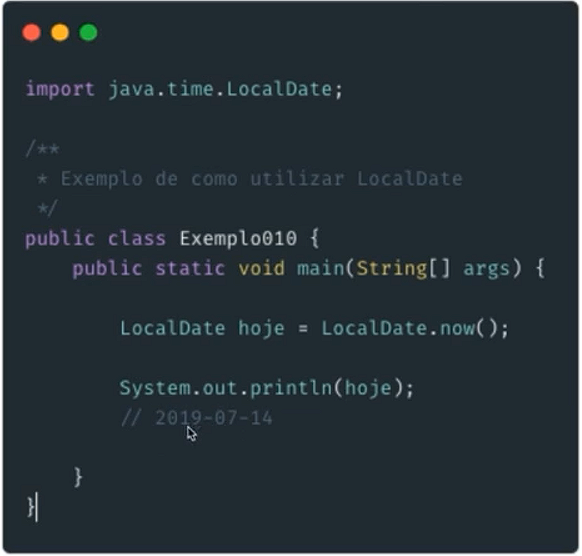
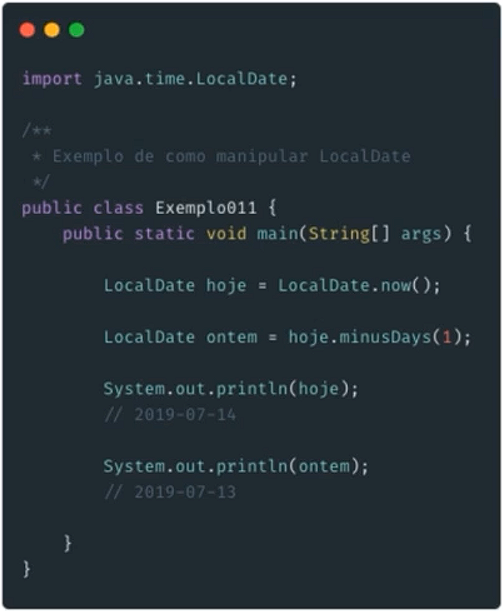
-LocalDateTime;

O que tínhamos eram as classes: Date e Calendar.

Com o uso constante, elas se mostram confusas e trabalhosas. Além de serem mutáveis.

LocalDate

É uma classe imutável para representar uma data, ou seja, trabalhamos exclusivamente com data. Seu formato padrão é **yyyy-MM-dd.**



LocalTime

É uma classe imutável que representa um padrão de hora-minuto-segundo. Pode ser representado até o nível de nanossegundos. Exemplo: **12:22:10:123212345**



Sua utilização é similar ao LocalDate.

LocalDateTime

É uma espécie de junção entre o LocalTime e o LocalDate. Também é uma classe imutável e você consegue trabalhar com dia e hora de uma vez só. Você pode manipular data e hora com precisão de nanossegundos. Exemplo: **2nd October 2007 at 13:45:30.123456789**





ARRAYS

Array ou matriz é uma estrutura de dados que nos permite organizar valores na memória. Fazem parte da biblioteca java.util que é nativa do Java. Armazenam elementos do mesmo tipo. Podem ser unidimensionais ou multidimensionais.

A principal diferença entre Arrays e Listas é que após ser declarado de um tipo os elementos dentro do Array devem ser de mesmo tipo e na Lista isso não é obrigatório.

Cada item de um array é chamado de elemento. Cada elemento é acessado pela posição numérica (índice). O índice inicia a partir de 0. Ao se declarar um array, todos os índices são inicializados em 0.

Arrays não podem mudar de tamanho, a partir do momento que um array foi criado, ele não pode mudar de tamanho. Caso necessite de mais espaço, será necessário criar um novo array e, antes de se referir a ele, copiar os elementos do array velho.

DECLARAÇÃO DE ARRAYS

dataType[] arrayName = new dataType[arraySize];

int meuArray = new int[7];

Outra forma

dataType[] arrayName = {value0, value1, ..., valueN};

int[] meuArray = {12,32,54,6,8,89,64};

Para alterar o valor em uma posição específica no array:

meuArray[6] = 50;

COMPRIMENTO DO ARRAY

meuArray.length

PERCORRENDO ARRAYS

Para processar um array, devemos usar um laço de repetição (for, for each, etc)

public class MyClass {

public static void main(String[] args) {

int[] meuArray = {12,32,54,6,8,89,64};

for (int i=0; i < meuArray.length; i++){

System.out.println(meuArray[i]);

}

}

}

ARRAYS MULTIDIMENSIONAIS

Array Multidimensional é um array contendo um ou mais arrays internos.

Exemplo de array bidimensional

Int[][] meuArrayMulti = {{1,2,3,4}, {5,6,7}};

public class MyClass {

public static void main(String[] args) {

int[][] meuArrayMulti = {{1,2,3,4}, {5,6,7}};

for (int i=0; i < meuArrayMulti.length; ++i){

for(int j=0; i<meuArrayMulti[i].length; ++j){

System.out.println(meuArray[i][j]);

}

}

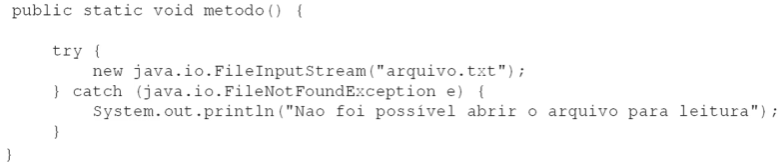
}

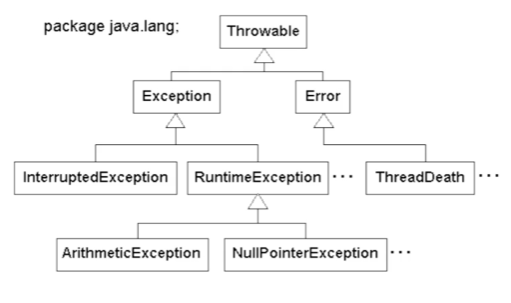
}

TRATAMENTO DE EXCEÇÕES

São todos os erros que ocorrem durante o processamento de um método que podem ser esperados ou não esperados. Como o termo já diz, “Exceptions” são exceções. Falhas que não devem ocorrer rotineiramente no fluxo de um sistema.

Exemplo de tratamento de exceção:





É fundamental que todo método criado possua um tratamento de exceção para que não apareça um erro muito grande para o usuário.

As exceções ocorrem quando algo imprevisto acontece, elas podem ser provenientes de erros de lógica ou acesso a recursos que talvez não estejam disponíveis.

Alguns possíveis motivos externos para ocorrer uma exceção são:

* Tentar abrir um arquivo que não existe.
* Tentar fazer consulta a um banco de dados que não está disponível.
* Tentar escrever algo em um arquivo sobre o qual não se tem permissão de escrita.
* Tentar conectar em servidor inexistente.

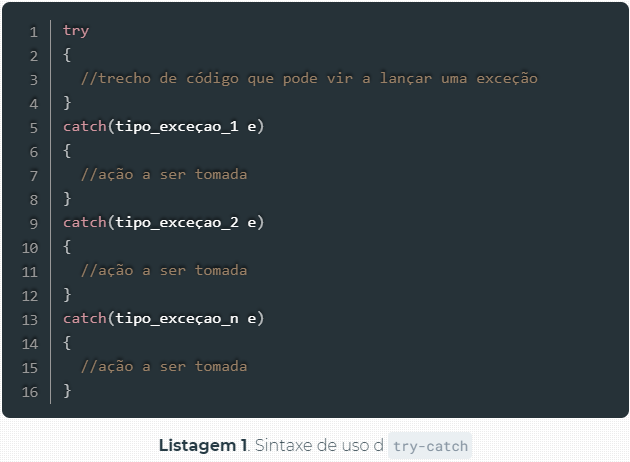
Alguns possíveis erros de lógica para ocorrer uma exceção são:

* Tentar manipular um objeto que está com o valor nulo.
* Dividir um número por zero.
* Tentar manipular um tipo de dado como se fosse outro.
* Tentar utilizar um método ou classe não existentes.

Uma maneira de tentar contornar esses imprevistos é realizar o tratamento dos locais no código que podem vir a lançar possíveis exceções, como por exemplo, campo de consulta a [**banco de dados**](https://www.devmedia.com.br/cursos/banco-de-dados), locais em que há divisões, consulta a arquivos de propriedades ou arquivos dentro do próprio computador.

Para **tratar as exceções em Java** são utilizados os comandos try e catch.

Sintaxe:

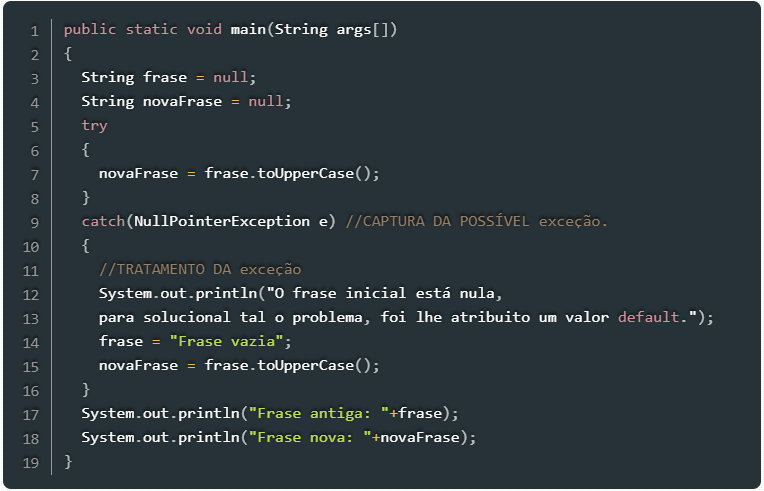


Onde:

* try{ … } - Neste bloco são introduzidas todas as linhas de código que podem vir a lançar uma exceção.
* catch(tipo\_excessao e) { … } - Neste bloco é descrita a ação que ocorrerá quando a exceção for capturada.

Imagine uma classe que tem um método principal main que tem como seu único objetivo alterar todas as letras de um frase para maiúsculas utilizando o método toUpperCase() da classe String, caso a frase esteja nula e se tente usar o método toUpperCase() na mesma será lançada uma exceção de NullPointerException.

Quando este código for executado, o mesmo lançará uma NullPointerException na saída do console quando executamos tal programa. Ou seja, o mesmo tentou acessar um atributo de um objeto que estava nulo.



Quando este código for executado, o mesmo lançará uma NullPointerException, porém esta exceção será tratada desta vez, sendo a mesma capturada pelo catch{} e dentro deste bloco as devidas providências são tomadas. Neste caso é atribuído um valor default à variável frase. A saída deste programa seria a seguinte:



CHECKED EXCEPTIONS

São exceções esperadas, cujo fluxo ou método de um sistema foi preparado para receber. Um bom exemplo é uma exceção de negócio, onde se deseja informar um erro caso a exceção esperada ocorra.

UNCHECKED EXCEPTIONS

São exceções inesperadas para o fluxo ou método de um sistema, um bom exemplo é a famosa NullPointException que ocorre quando se tenta acessar uma referência de memória vazia, ou recuperar uma instância que não existe, dentre outros motivos.

BLOCO TRY CATCH

É um bloco utilizado quando no processo que será executado dentro de um método é esperado um erro, então cria-se um bloco “protegido” onde qualquer erro que ocorra dentro do trecho “try” é direcionado para o trecho “catch” e sofrerá o devido tratamento de erro.

FINALY

É um bloco de código que pode ou não ser utilizado junto ao try catch, este trecho de código sempre será executado independente se ocorrer erro ou não dentro do fluxo onde existe o try catch. Normalmente o finally é usado para liberar recursos ou para dar continuidade em um fluxo que deve ocorrer independente de erro.

THROW E THROWS

Throws: é a assinatura do método que será retornado caso ocorra erro para o método que fez a chamada, dentro de um fluxo encadeado.

Throw: é usado para lançar a exceção desejada, juntamente com a mensagem de erro, para o método que fez a chamada.