Engenharia de Software

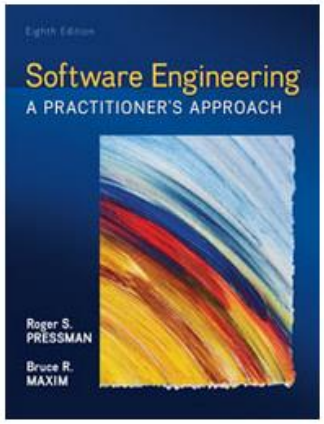
Unidade 5 - Conceitos de Orientação a Objetos

Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP

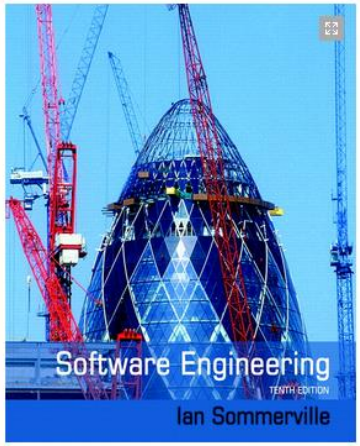
ENGENHARIA DE SOFTWARE

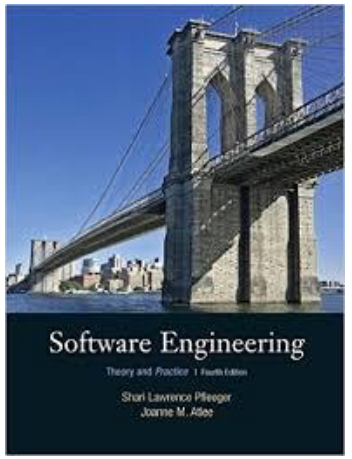
Bibliografia

- **Software Engineering – A Practitioner's Approach – Roger S. Pressman – Eight Edition – 2014**
- **Software Engineering – Ian Sommerville – 10th edition – 2015**
- **Software Engineering – Pfleeger & Atlee – Theory and Practice – 4th edition – Prentice Hall 2009**
- Engenharia de Software – Uma abordagem profissional – Roger Pressman - McGraw Hill, Sétima Edição - 2011
- Engenharia de Software – Ian Sommerville – Nona Edição – Addison Wesley, 2007
- Engenharia de Software – Teoria e Prática - Shari Lawrence Pfleeger – Editora Pearson – 3ª edição



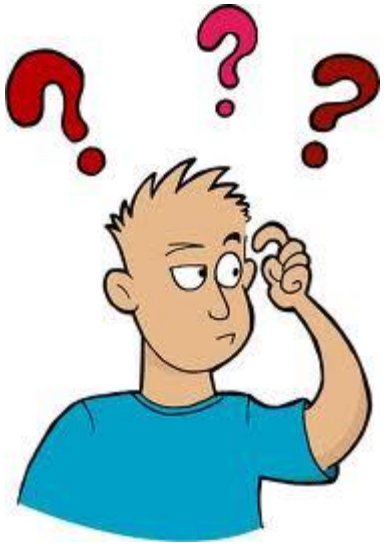
Software Engineering: A Practitioner's Approach, 8/e





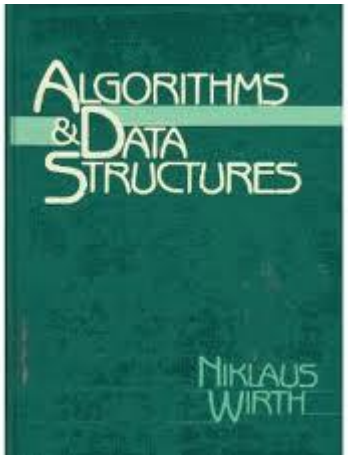
ENGENHARIA DE SOFTWARE

Como era a programação nos anos 70 ?



Programação Estruturada

- Focada no processo de transformação de dados.
- Após os procedimentos serem definidos, os próximos passos consistiam em se determinar as formas de armazenamento dos dados.
- **Niklaus Wirth**, projetista do Pascal, escreveu o famos livro “Algorithms + Data Structures = Programs”. (Observe que Algorithms vem antes de Data Structures…).

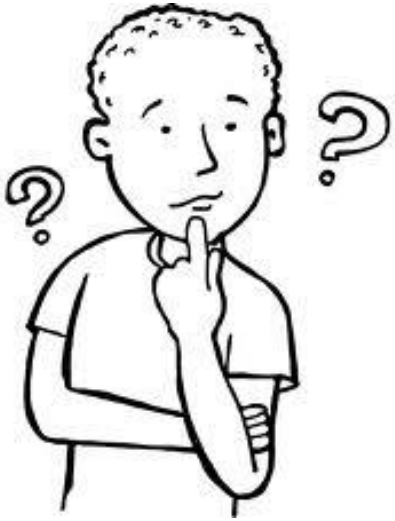


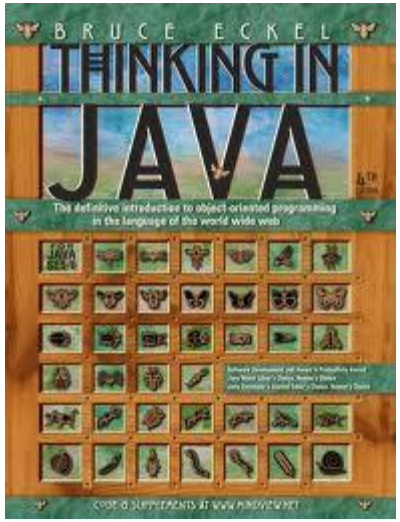
O que mudou no paradigma OOP ?

Paradigma OOP

- OOP inverte a ordem. Primeiro foco nos dados e em seguida nos algoritmos que operam sobre os dados.
- A implementação da funcionalidade é, em geral, escondida dos usuários.
- Este conceito é chamado **Encapsulamento**.

OOP exige uma nova forma de pensar...





Scala

C++

As linguagens modernas são, na maioria, orientadas a objetos...

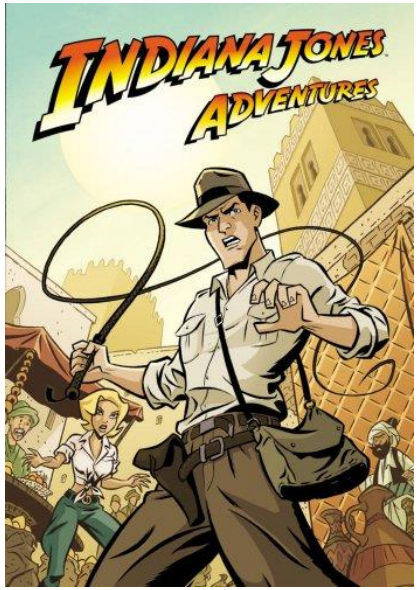
C#

python™

Ruby

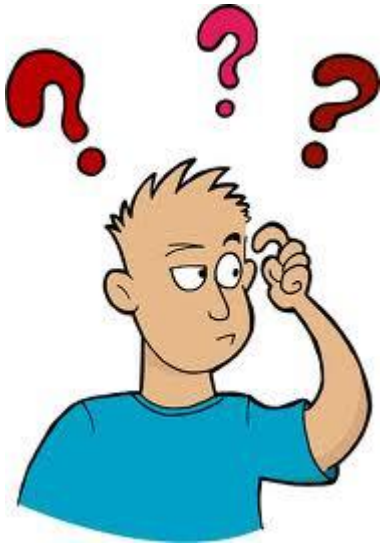
RAILS

Para ser produtivo nessas linguagens, é necessário conhecer o paradigma OOP...



Afinal, o que é um programa OOP ?

Antes, uma outra questão ...



Para que serve um programa OOP ?

para resolver um problema...

Imagine uma empresa com um determinado problema...

A empresa pode contratar um funcionário (objeto) ...

O funcionário irá resolver o problema desempenhando uma função...

Para a empresa o mais importante é resolver o problema...

Os detalhes de como o funcionário desempenha a função não são tão importantes...

Os funcionários da empresa operam como uma comunidade, trocando mensagens para a solução do problema...

Como um programa OOP resolve um problema ?

C++

Da mesma forma que em uma empresa!
Com o emprego de objetos que executam ações (operações)...

Afinal, o que é um programa OOP ?

Um programa OOP

- É um conjunto de objetos que trocam mensagens para, ao final do processamento, resolver o problema do usuário.
- Cada objeto tem uma **funcionalidade** que é exposta aos usuários (interface) e a implementação é, em geral, escondida dos mesmos (encapsulamento).

C++

JAVA™

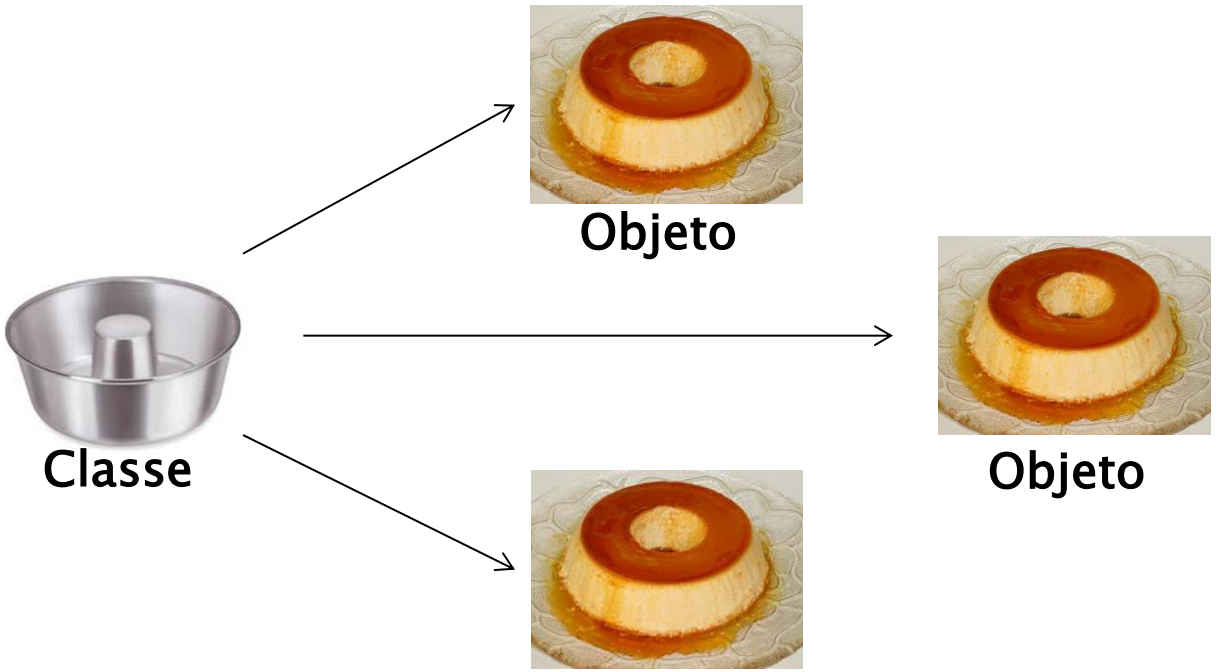
Objetos

- Podem ser obtidos de uma prateleira (**biblioteca**) com objetos prontos.
- Podem também ser construídos internamente no programa.

Como se cria um objeto num programa OOP ?

Por meio de classes...

Uma classe é um modelo, uma especificação, um molde, a partir do qual criam-se objetos...



Instâncias

- Um objeto é uma instância de uma classe

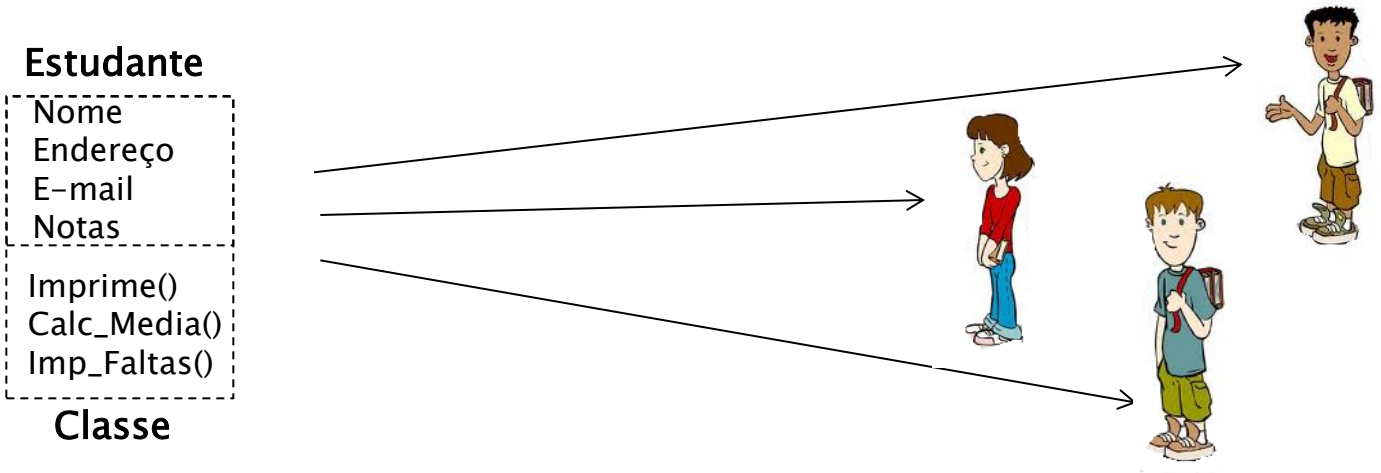
Classe

Instância →

Objeto

Objetos

- Os dados (propriedades, atributos) de um objeto são os **campos instância**.
- Os procedimentos que operam os dados do objeto são os **métodos**.
- Um objeto específico (instância de uma classe) tem seus campos instância (valores) particulares e isto os tornam distintos de outros objetos (**individualidade**).
- O conjunto de valores de um objeto específico constitue o seu **estado**.

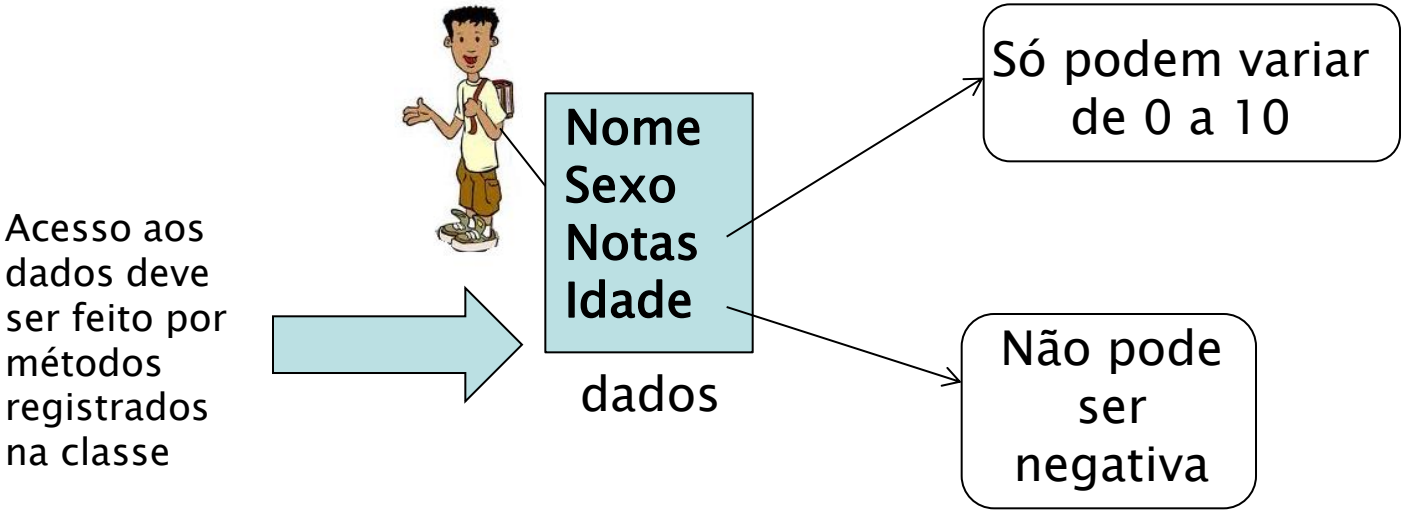


Como implementar encapsulamento ?

Encapsulamento

- Programas devem interagir somente com os dados do objeto por meio de **métodos definidos na classe**.

- Isto define uma "Caixa-preta" para acesso aos dados o qual é a chave para reusabilidade e confiabilidade.



Como um objeto é construído ?

Construção de Objetos

- Objetos são construídos por meio de ***construtores*** .
- Um ***construtor*** é um método especial cujo propósito é criar e inicializar objetos.
- Construtores têm sempre o mesmo nome da classe.
- Para criarmos objetos, combinamos o construtor com o operador **new**.

Construtores

- Uma classe pode ter mais de um construtor.
- Um construtor pode ter zero, um ou mais parâmetros.
- Um construtor nunca retorna valor.
- Sempre é chamado pelo operador **new**.

Construtores

- Há uma importante diferença entre construtores e outros métodos.
- Um construtor pode somente ser chamado por meio do operador **new**.
- Você não pode aplicar um construtor em um objeto existente para resetar os campos instância do objeto.
- Exemplo:

x.Estudante(“André“, 10, “Rua Santos, 34”, 8.5); **// Erro !!!**

Construtor – Exemplo em Java

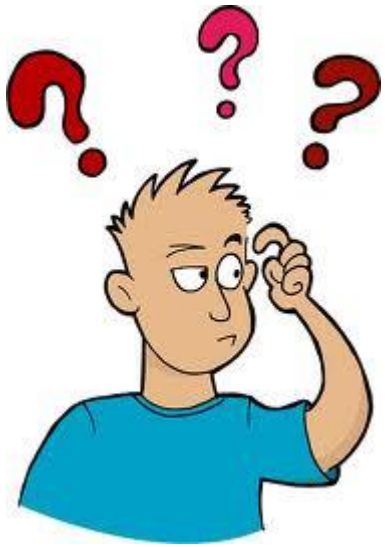
```
package maua;

import java.util.Date;

        public class Const_Object {

                public static void main(String[] args) {

                        Date x = new Date();

                        System.out.println(new Date());

                        System.out.println(new Date().toString());

        }

}
```

Overloading

- Uma classe pode ter diversos construtores.
- Overloading ocorre quando diversos métodos têm o mesmo nome, mas diferentes parâmetros.
- Exemplo:

```
public Estudante ( )  {
        ...
}

public Estudante ( String Nome, double NotaExame) {
        ...
}
```

Objetos e Variáveis Objeto são a mesma coisa ?

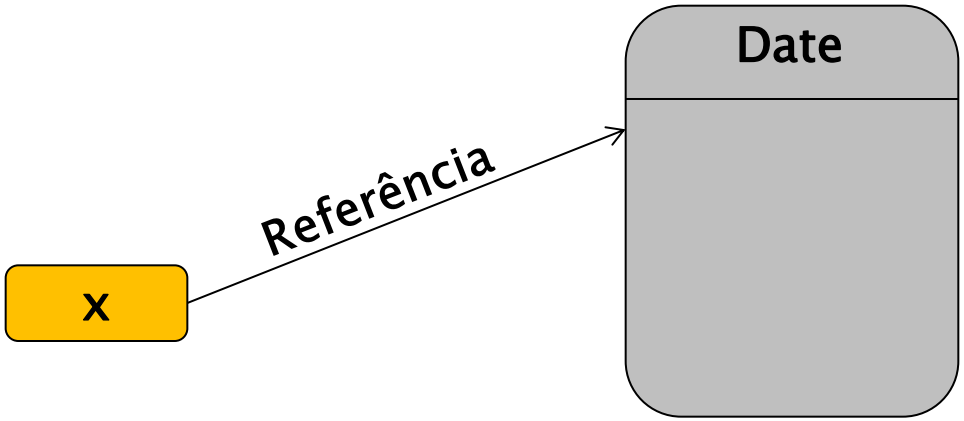


Cuidado...

```
Date x;          // x ainda não aponta para nenhum objeto...

String S = x.toString() ;  // erro de compilação !!!
```

x é uma variável que pode fazer referência à um objeto do tipo Date !

x não é um objeto!!!



Variáveis Instância

- Uma classe modela objetos. Estes objetos têm atributos de dados.
- Na classe, os atributos são definidos por variáveis.
- Variáveis instância são variáveis associadas aos atributos de um objeto.
- Cada instância da classe (objeto) contém sua própria cópia destas variáveis instância.
- Estas variáveis permitem a diferenciação entre os objetos criados num programa, dando-lhes sua **individualidade**.

Variáveis Instância

```
public class Estudante {
```

```
String      Nome;
String      Endereco;
int         Matricula;
double      Nota_Aproveitamento;
double      Nota_Semestre;
double      Nota_Exame;
int         Total_faltas;
```

Para cada objeto instanciado, estas variáveis serão preenchidas com os valores particulares do objeto criado...

Variáveis de Classe

- São variáveis associadas com a classe.
- Há somente uma cópia de cada uma destas variáveis independentemente do número de objetos criados pelo programa.
- Os valores associados a estas variáveis existem na memória, mesmo se nenhum objeto ainda tiver sido criado pelo programa.
- Ou seja, estas variáveis **são independentes da existência de objetos**.

Definindo classes
Variáveis de Classe

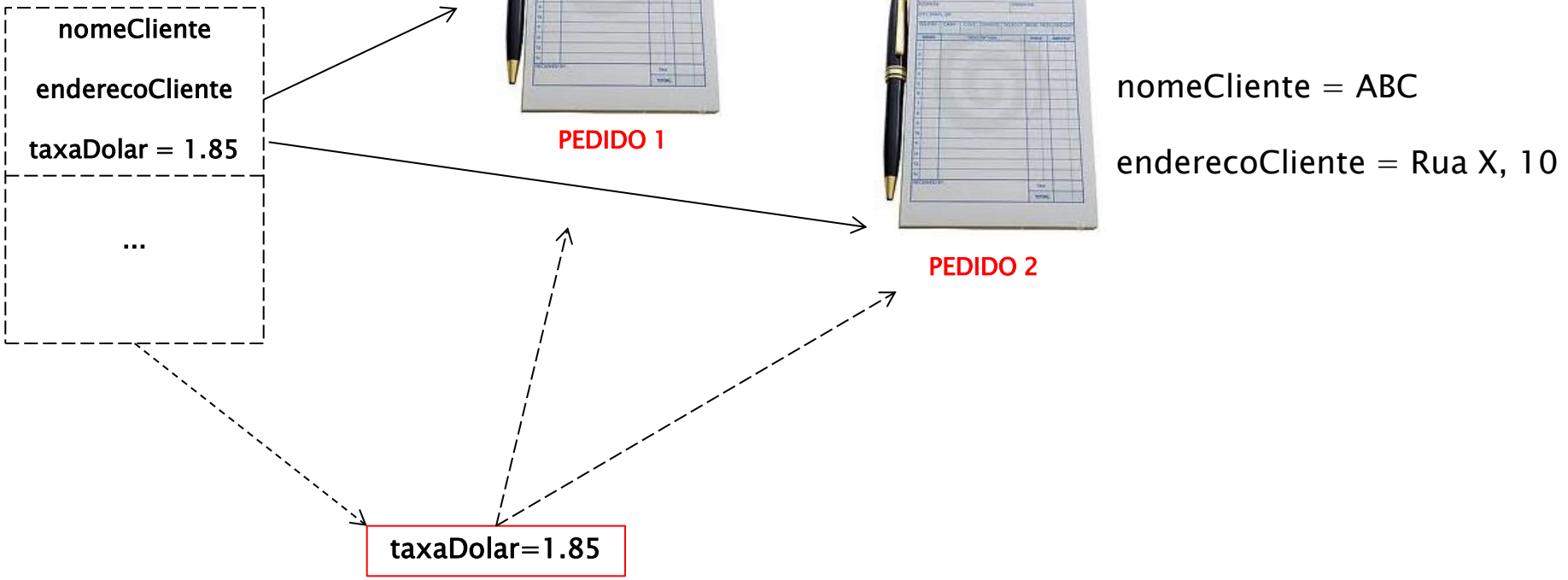
- Se o valor de uma variável de classe é modificado, então o novo valor estará disponibilizado para todos os objetos da classe.
- Em Java, uma variável de classe deve ser definida com a keyword **static** precedendo o nome.

Variáveis de Classe

```
public class PedidoCompra{

        String              nomeCliente;

        String              enderecoCliente;

        static double       taxaDolar = 1.85;

        ...
}
```

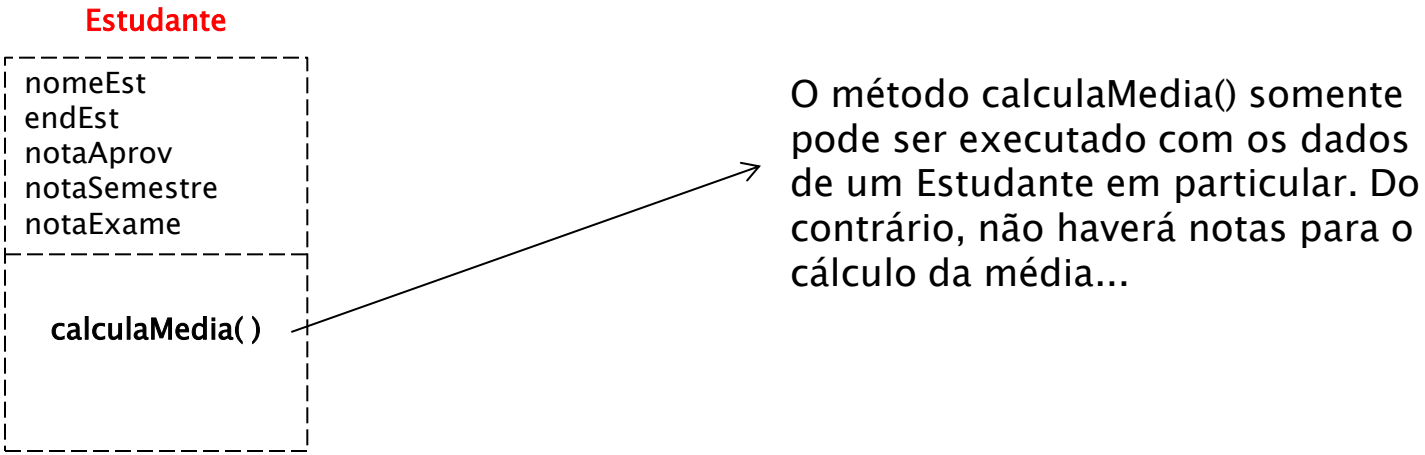
A taxa do dólar é compartilhada entre todos os objetos da classe. Esta informação existe mesmo se nenhum pedido de compra for instanciado.

Variáveis de Classe



Métodos Instância

- São os métodos que somente podem ser executados quando houver alguma instância (objeto) criado.
- Portanto, estão associados à um objeto em particular.
- Se não houver objetos, estes métodos não podem ser executados.



Métodos de Classe

- São métodos que pertencem a classe.
- Podem ser executados mesmo quando não existirem objetos.
- Em Java são declarados através da keyword **static**.
- Exemplo: Método main.

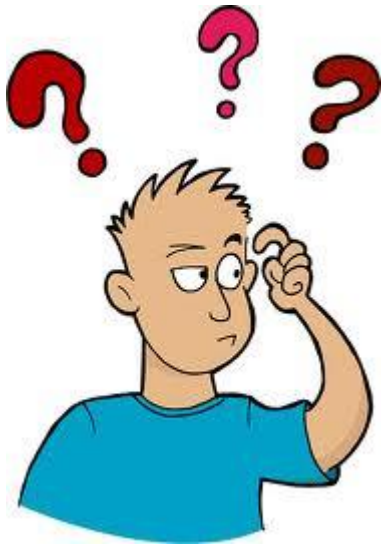
Definindo classes

```
package maua;

public class Estudante {
        String Nome;
        String Endereco;
        int Matricula;
        double Nota_Aproveitamento;
        double Nota_Semestre;
        double Nota_Exame;
        int Total_faltas;
```

```
public Estudante(String nome,
        String endereco,
        int matricula,
        double nota_Aproveitamento,
        double nota_Semestre,
        double nota_Exame,
        int total_faltas)  {

                Nome = nome;
                Endereco = endereco;
                Matricula = matricula;
                Nota_Aproveitamento = nota_Aproveitamento;
                Nota_Semestre = nota_Semestre;
                Nota_Exame = nota_Exame;
                Total_faltas = total_faltas;
}
```

```
    public double Calcula_Media() {
          return (this.Nota_Aproveitamento + Nota_Semestre)/2.0;
    }

    public void Imprime_Estudante() {
          System.out.println(
          "Nome:  " + Nome +
          "Endereco: " + Endereco +
          "Matricula: " + Matricula +
          "Nota de Aproveitamento: " + Nota_Aproveitamento +
          "Nota do Semestre:  "  + Nota_Semestre +
          "Nota do Exame: " + Nota_Exame +
          "Total de Faltas: " + Total_faltas);
    }

}
```

Eu consigo executar a classe Estudante ?



Executando a classe Estudante...

```
package maua;

public class Test_Estudante {

        public static void main(String[] args) {

                Estudante A = new Estudante("Paulo",
                        "Rua Brasil, 10 ",
                        55670,
                        5.7 ,
                        5.9 ,
                        8.0 ,
                        6 );

                A.Imprime_Estudante();
                System.out.println(A.Calcula_Media());

        }

}
```

Como altero a nota do estudante ?

```
package maua;

public class Test_Estudante {

public static void main(String[] args) {

        Estudante A = new Estudante("Paulo",
                "Rua Brasil, 10 ",
                55670,
                5.7 ,
                5.9 ,
                8.0 ,
                6 );

            A.Nota_Aproveitamento = 9.9;

                A.Imprime_Estudante();
                System.out.println(A.Calcula_Media());

        }

}
```

Alteração feita diretamente nos dados do objeto...

Que princípio foi violado ? ? ?

Princípio do Encapsulamento....

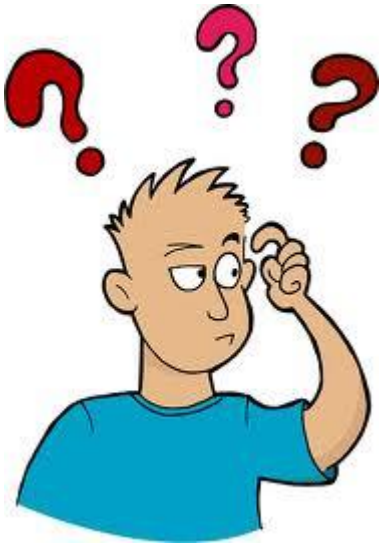
Sem encapsulamento...

- Os dados do objeto poderiam ter estado inconsistente. Por exemplo, a nota do estudante não pode ser negativa. Mas o programa abaixo é legal...

```
package maua;

public class Test_Estudante {

public static void main(String[] args) {

        Estudante A = new Estudante("Paulo",
                "Rua Brasil, 10 ", 55670, 5.7 , 5.9 , 8.0 , 6 );

            A.Nota_Aproveitamento = -1.5 ;

            A.Imprime_Estudante();
            System.out.println(A.Calcula_Media());

        }

}
```

Nota Inválida...

Então, como implementar o Encapsulamento ? ? ?



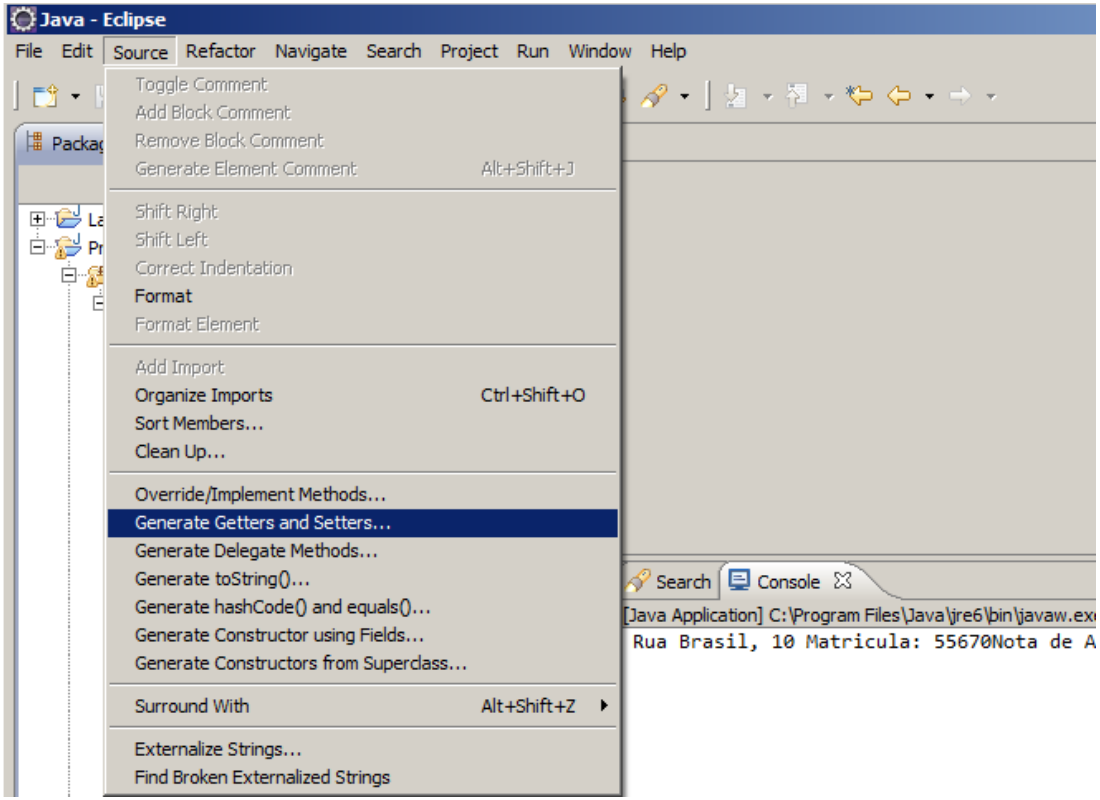
Keyword private

```
package maua;

public class Estudante {
        private String Nome;
        private String Endereco;
        private int Matricula;
        private double Nota_Aproveitamento;
        private double Nota_Semestre;
        private double Nota_Exame;
        private int Total_faltas;
```

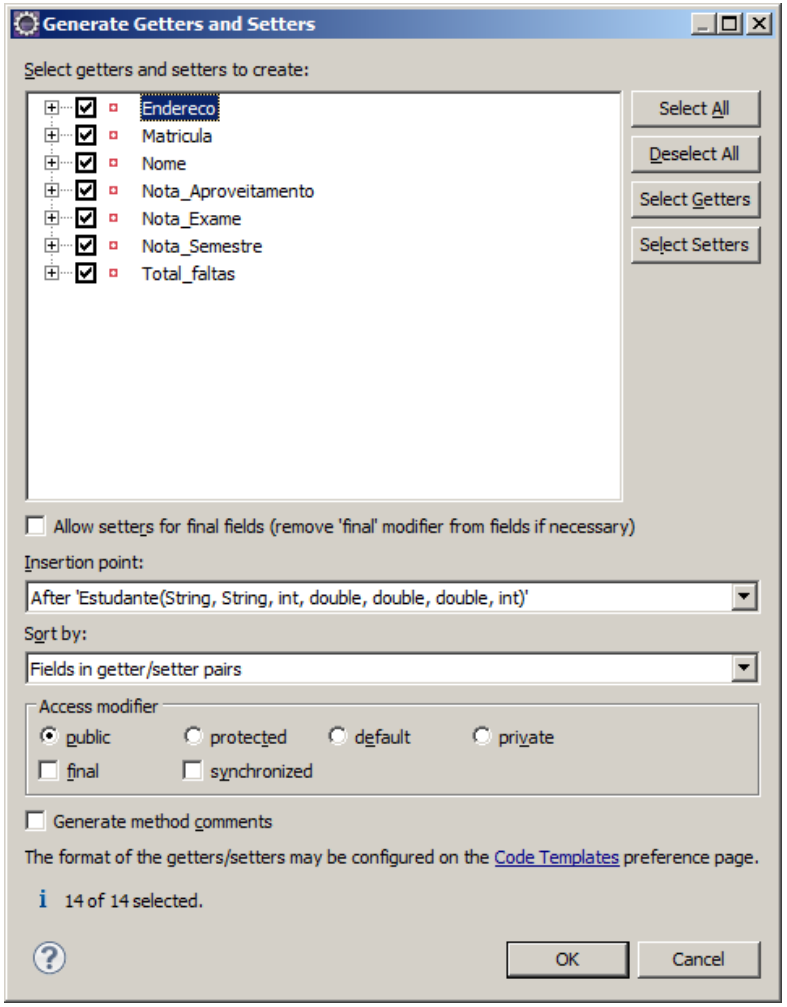
- A keyword **private** assegura que os únicos métodos que podem acessar os campos instância do objeto são os métodos da própria classe.

Getters e Setters

- São métodos auxiliares que provêem a interface aos dados do objeto.
- A IDE Eclipse automaticamente gera os métodos **Getters** e **Setters**.



Getters e Setters



Getters e Setters

```
public String getNome() {
        return Nome;
}

public void setNome(String nome) {
        Nome = nome;
}

public String getEndereco() {
        return Endereco;
}

public void setEndereco(String endereco) {
        Endereco = endereco;
}
```

Alteração da nota do estudante

```
package maua;

public class Test_Estudante {

  public static void main(String[] args) {

        Estudante A = new Estudante("Paulo",
              "Rua Brasil, 10 ", 55670, 5.7 , 5.9 , 8.0 , 6 );

        A.setNota_Aproveitamento(9.9);

        A.Imprime_Estudante();
        System.out.println(A.Calcula_Media());

  }

}
```

A função setNota_Aproveitamento() deverá prover o código de consistência do campo...

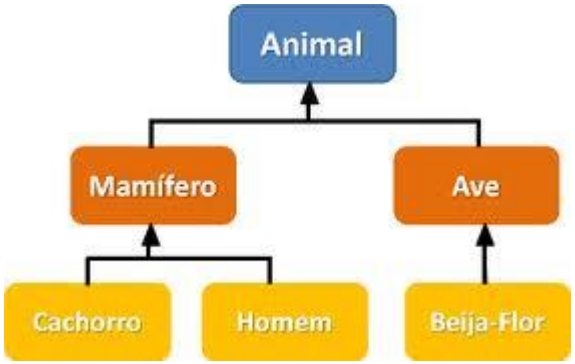
Controlando acesso a membros de classes em Java

Atributo	Acesso Permitido
Nenhum atributo de acesso definido	Para qualquer classe dentro do mesmo package.
public	Para qualquer classe de qualquer lugar.
private	nenhum acesso fora da classe.
protected	Para qualquer classe dentro do package e de qualquer subclasse.

Destruição de Objetos

- Java faz regeneração (aproveitamento) de memória de forma automática.
- Isto é feito automaticamente pelo procedimento Garbage Collector.
- No entanto, você pode – a qualquer momento – adicionar o método **finalize()** antes do Garbage Collector proceder a limpeza de objetos em memória.
- Na prática, a ação do método **finalize()** é duvidosa, pois não se sabe ao certo o momento em que a máquina virtual efetua a chamada do Garbage Collector.

Herança



Introdução

- Por meio de herança, podemos criar novas classes a partir de classes existentes.
- Isto permite o **reuso** de métodos e campos das classes existentes.
- Na classe nova, pode-se também criar novos métodos e campos para adaptar à novas situações.
- Esta técnica é de extrema importância em Orientação a Objetos.

Usando classes existentes

- Este procedimento é conhecido por **derivação**.
- A classe nova é chamada classe derivada ou subclasse.
- A classe existente é chamada base ou superclasse .

Um gerente é um funcionário comum em uma Empresa ?

Gerentes e Empregados certamente têm muitas coisas em comum ...

Ambos têm um salário ...

Ambos têm código funcional ...

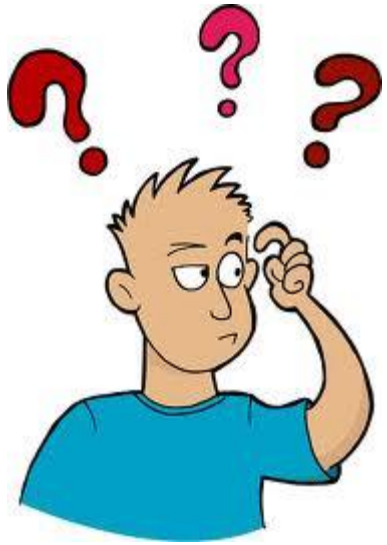
Ambos têm dados pessoais...

Mas, gerentes têm algo a mais...

Todo gerente é um empregado...

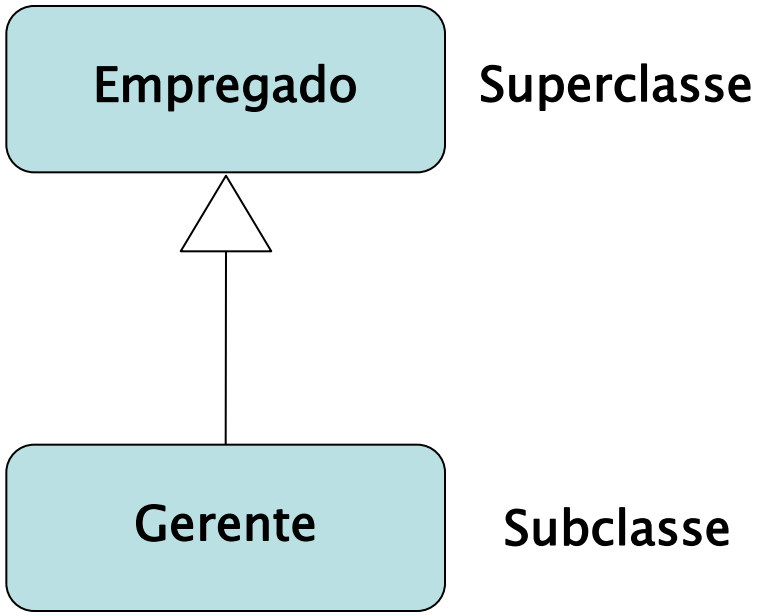
- Esta é uma situação típica do uso de herança.
- Precisamos definir uma nova classe – **Gerente** – e adicionar a ela funcionalidades.
- Mas, podemos aproveitar o que já está definindo na classe **Empregado**.
- Os campos e funções da Classe **Empregado** são aproveitados para a classe **Gerente**.
- Há um relacionamento "**is-a**" entre Gerente e Empregado.
- Ou seja, todo Gerente também **é um** Empregado.

Como definir Herança em Java ?



Por meio da keyword extends

- A keyword **extends** indica que se está criando uma nova classe a partir de outra classe já existente.



Sem herança

Empregado	
nome:	String
salario:	double
codfunc:	int
Imprime_Func()	
calculaSalario( )	

Gerente	
nome:	String
salario:	double
codfunc:	int
bonus:	double
calculaSalario( )	

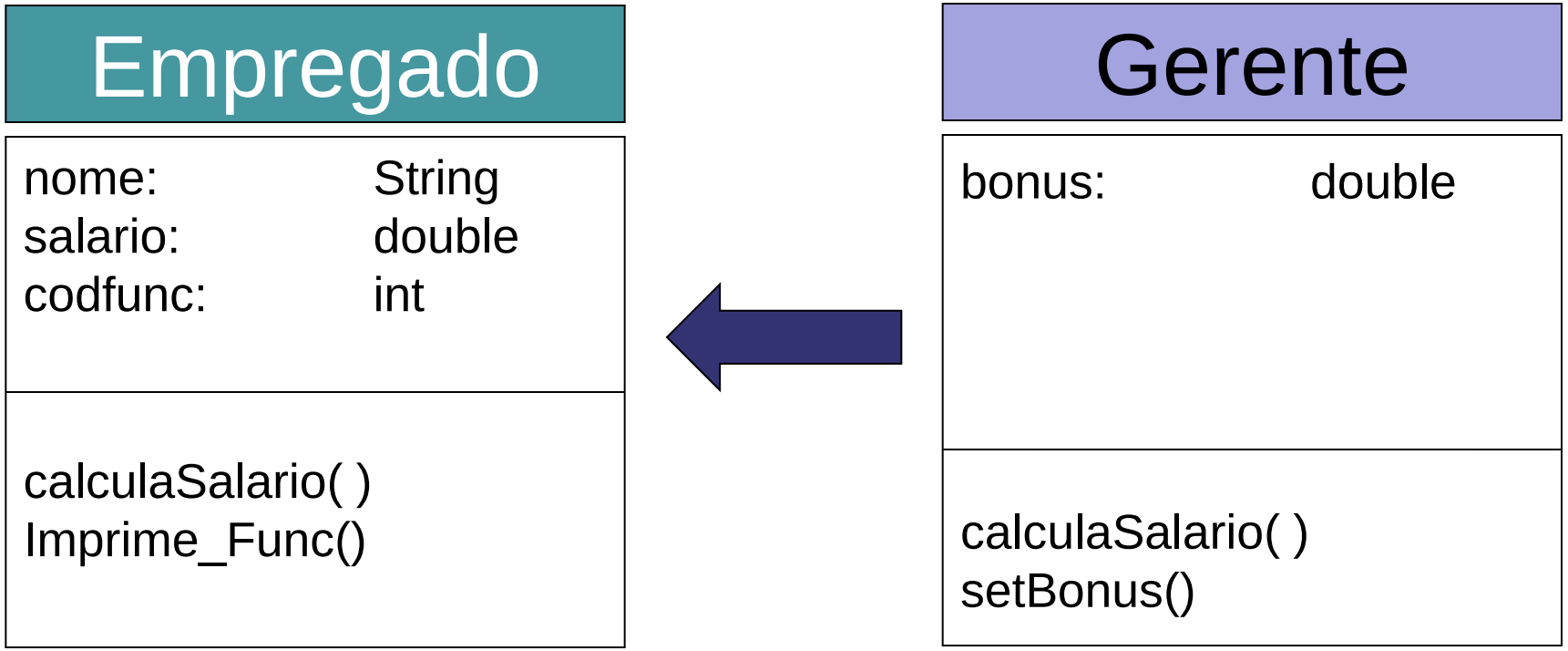
Sem herança

```
class Empregado {
   private String nome;
   private double salario;
   private int codfunc;

   public double calculaSalario( ) {
       ...
   }
   public void Imprime_Func( ) {
       ...
   }
}
```

Sem herança

```
class Gerente {
   private String nome;
   private double salario;
   private int codfunc;
   private double bonus;

   public double CalculaSalario( ) {
       …
   }
   public void SetBonus(double b) {
       bonus = b;
   }
}
```

Com herança


Empregado
nome: String
salario: double
codfunc: int
calculaSalario( )
Imprime_Func()
Gerente
bonus: double
calculaSalario( )
setBonus()

Com herança

```
class Gerente extends Empregado{

   private double bonus;

   public double calculaSalario( ) {
       …
   }
   public void SetBonus(double b) {
       bonus = b;
   }
}
```

Overriding

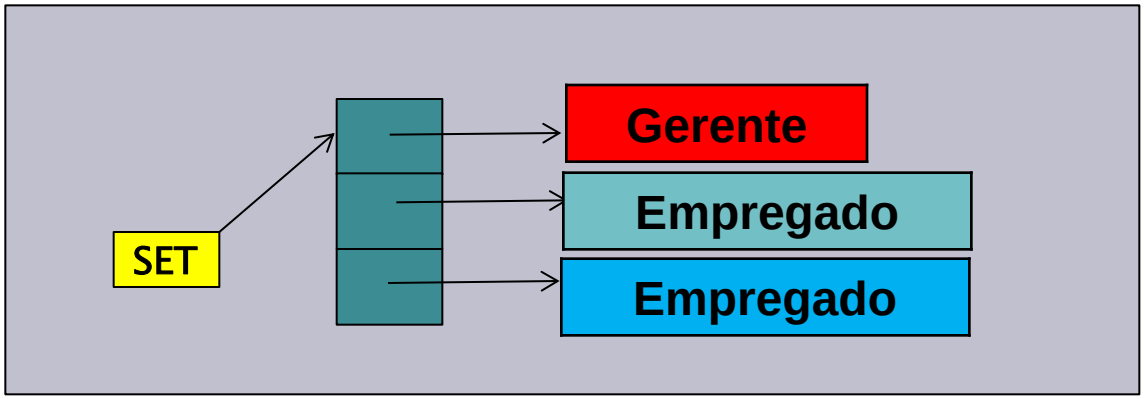
- Pode haver funções definidas em Empregado que não são apropriadas para Gerente. Em particular, a função **calculaSalario**(), pois o cálculo para Gerente é diferente (Gerente tem bonus).
- Neste caso, definimos um outro método **calculaSalario**() na classe Gerente que sobrepõe a função na superclasse. Este conceito é chamado **Overriding**.

Polimorfismo

Um exemplo...

```
Empregado[] set = new Empregado[3];
```

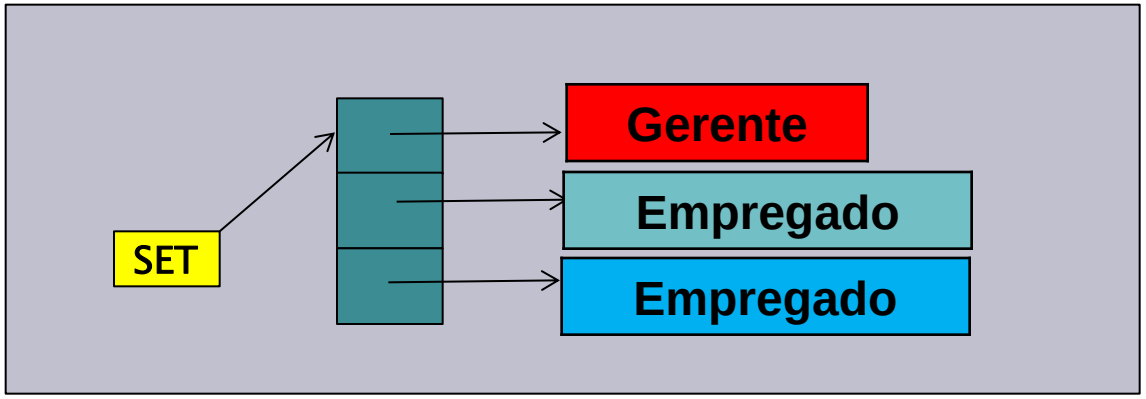
- O tipo declarado de **set** é Empregado, mas o tipo real do objeto para o qual **set** aponta pode ser Gerente ou Empregado.



Polimorfismo

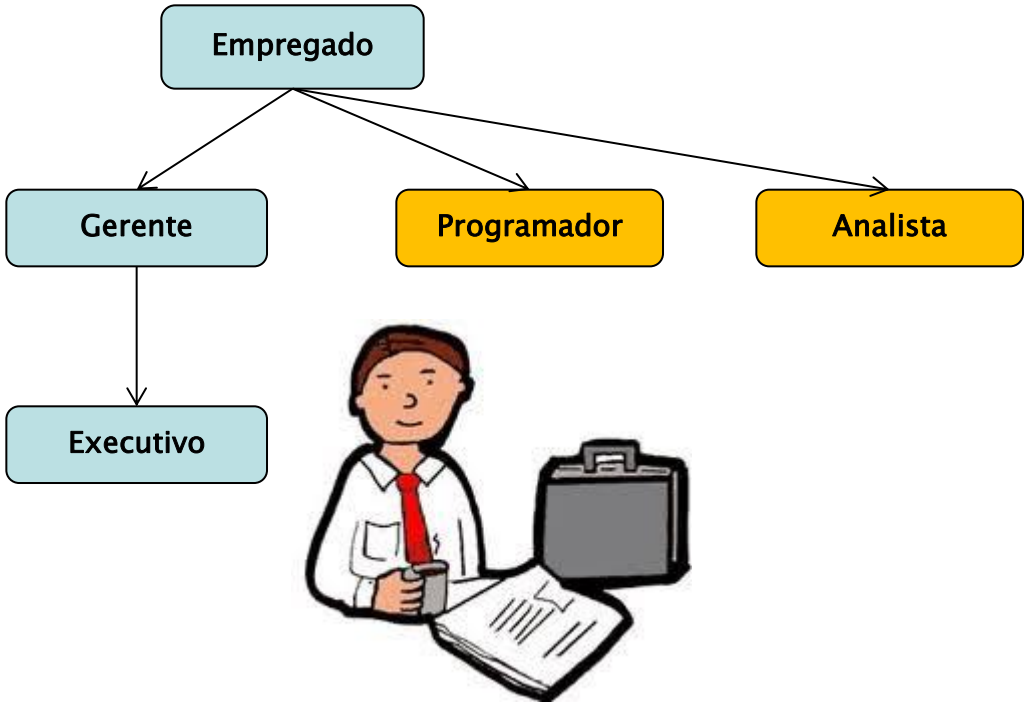
```
for (int i=0; i< set.length ; i++)
    System.out.println (set[i].getNome() + " " + set[i].calculaSalario());
```

- Set[**0**].**calculaSalario**() efetua chamada do salário de Gerente.
- Set[**1**].**calculaSalario**() efetua chamada do salário de Empregado.
- A JVM sabe qual o tipo em tempo de execução e chama o método adequado.
- Este conceito é chamado **POLIMORFISMO**.



Estrutura de Herança

- A herança pode se estender em vários níveis.
- Por exemplo, poderíamos criar uma classe **Executivo** que é filha de Gerente.
- Java não suporta múltipla herança. Esta funcionalidade é tratada com o conceito de interfaces.



Como saber se herança é a forma correta do projeto dos dados ?

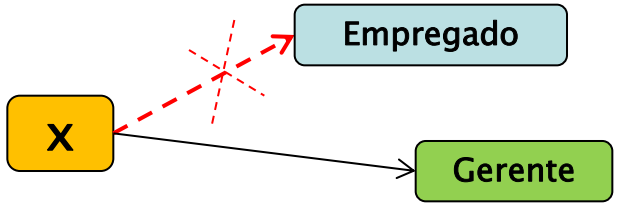
Primeira Regra

- A regra "**is-a**" estabelece que todo objeto da sub-classe é também um objeto da superclasse.
- Por exemplo, todo **gerente** também é um **empregado**.
- Naturalmente, o oposto não é verdade – nem todo empregado é gerente.

Segunda Regra

- O **princípio da Substituição** estabelece que você pode usar um objeto de uma subclasse sempre que o programa espera um objeto da superclasse.

- Exemplo:



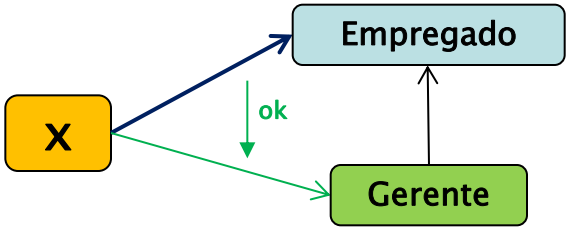

```
package maua;

public class Polimorf_01 {

        public static void main(String[] args) {

                Empregado x = new Empregado("Marcos", 2300.5,55150) ;

                x = new Gerente("Mike", 9300.5, 51234, 500);

                System.out.println(x.getDetalhes() ) ;
        }
}
```

Variáveis Polimórficas

- O **princípio da Substituição** estabelece que variáveis objeto em Java são polimórficas, ou seja uma variável do tipo Empregado (superclasse) pode fazer referência à uma variável do tipo Gerente (subclasse).

x é do tipo Empregado !

Todo Gerente também é Empregado !

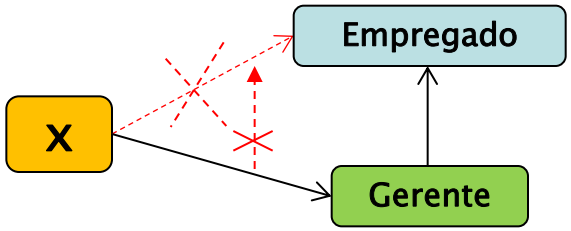



```
Empregado x = new Empregado("Marcos", 2300.5,55150) ;

x = new Gerente("Mike", 9300.5, 51234, 500);
```

E o contrário ?

É possível uma variável do tipo subclasse fazer referência a um objeto da superclasse ?

Não !



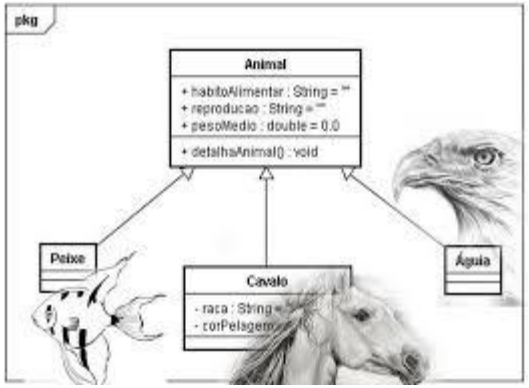

```
package maua;

public class Polimorf_02 {


        public static void main(String[] args) {

                Gerente x = new Gerente("Marcos",2300.5,55150,500);

                x = new Empregado("Mike", 9300.5, 51234);

                System.out.println(x.getDetalhes() ) ;

        }
}
```

x é do tipo Gerente !

Porém, nem todo Empregado é Gerente!

- Não é possível assinalar uma referência de subclasse (Gerente) à uma variável superclasse (Empregado).
- Nem todos os empregados são gerentes !!!

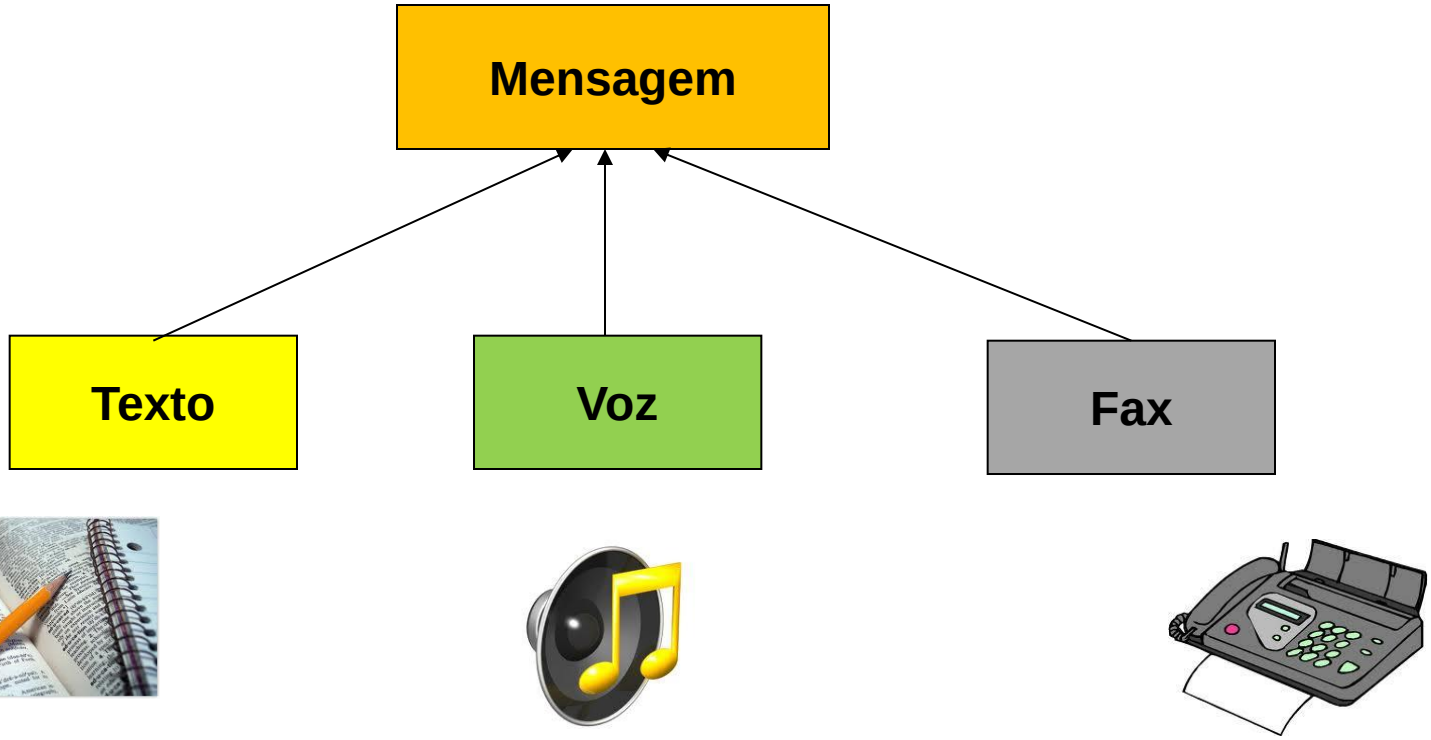
Classes Abstratas e Interfaces

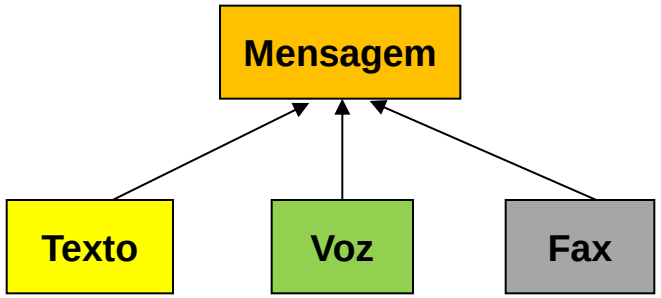


Classes Abstratas

- À medida em que se sobe na hierarquia de herança, as classes se tornam mais gerais e mais **abstratas**.
- Em algum ponto, as classes ancestrais tendem a se tornar tão gerais que podemos imaginá-las mais como um **framework** para outras classes do que uma classe com instâncias específicas que desejamos utilizar.

Classes Abstratas





Classes Abstratas

- Consideremos que todas as classes têm um método comum **exibe().**
- Seguindo os princípios da OOP, deveríamos implementar **exibe()** na classe pai.
- No entanto:

p/ texto => exibir a mensagem é enviá-la a uma janela de texto;

p/fax => exibir a mensagem é enviá-la para uma janela gráfica;

p/voz => exibir a mensagem é enviá-la para um sistema de som;

Como então implementar a função exibe( ) na classe pai ?

A resposta é:
Não podemos implementar na superclasse.

Keyword abstract

- Em Java usamos a keyword **abstract** para indicar que um método não pode ser implementado em uma classe.

```
public abstract class Mensagem  {
      public abstract void exibe();
}
```

Classes abstratas

- Podem também ter dados e funções concretas.

```
public abstract class Mensagem  {

      public abstract void exibe();

      public String getEmissor() {
            return emissor;
      {

      private String emissor;
}
```

Classes abstratas

- Em adição aos métodos ordinários já vistos, **uma classe abstrata tem ao menos um método abstrato**.
- Certamente métodos de subclasses derivadas da classe abstrata providenciarão a implementação do método.

A classe Object

- A classe **Object** é a última ancestral.
- Todas as classes em Java são extensões da classe **Object**.
- Entretanto, você não precisa escrever:

```
class Empregado extends Object
```

Interfaces

Interfaces

- Imagine que você irá desenvolver uma aplicação no qual duas equipes irão desenvolver o software de forma simultânea.
- Cada equipe irá desenvolver seus códigos de forma independente.

Interfaces

- No entanto, deverá haver um "**contrato**" entre as equipes de tal modo que haja interação entre os códigos. Este contrato é conhecido por **interface**.
- Interface é uma forma de descrever o quê as classes devem fazer, sem especificar como elas devem fazê-lo.
- Uma classe pode implementar uma ou mais interfaces.
- Uma interface não é uma classe mas um conjunto de requisitos para as classes que quiserem implementá-la.
- Em Java, uma interface é uma definição de tipo, semelhante à classe, que pode conter apenas constantes e assinatura de métodos (protótipos).
- Numa interface não há corpo de definição de método. Não podem ser instanciadas. São implementadas por classes ou ainda estendidas em outras interfaces.

Interface

- Todos os métodos de uma interface são automaticamente **public**.
- Por esta razão não há necessidade de incluir a keyword **public** quando estivermos declarando um método em uma interface.
- Tendo em vista que interfaces não são classes, nunca se pode usar o operador **new** para instanciar uma interface.

Exemplo

```
public interface A12 {
        Integer  func (Integer  n);
    }
```

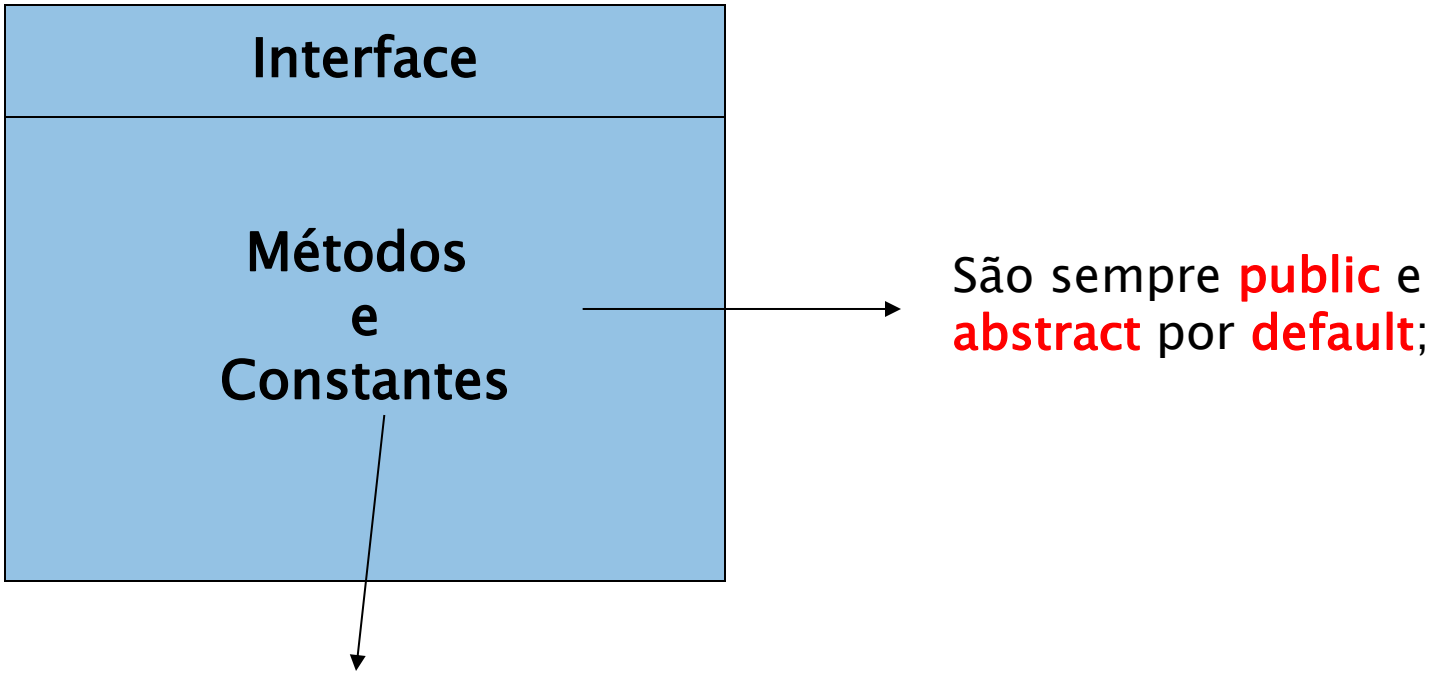
- Isto significa que para qualquer classe que implementa a interface **A12** é requerido que se tenha a definição do método **func** e o método deve receber um argumento **Integer** e retornar um objeto do tipo **Integer**.

Interfaces

- Uma interface é essencialmente uma coleção de constantes e métodos abstratos.
- Para fazer uso de uma interface, você implementa a interface em alguma classe.
- Ou seja, você declara que a classe implementa a interface e escreve o código para cada método declarado na interface.
- Quando uma classe implementa uma interface, quaisquer constantes que foram definidas na interface são diretamente disponíveis na classe, como se fossem herdados de uma classe base.

O que pode conter uma interface ?

Interface com métodos e constantes

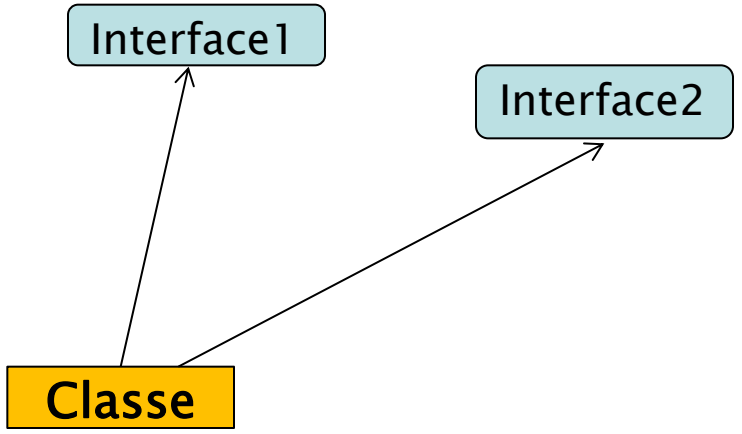


Para que serve Interface?

- Em Java, uma classe A pode somente estender uma simples classe B.
- A classe A não pode estender uma segunda classe C. (A não pode ter dois pais...)
- Outras linguagens, tais como C++, permitem que uma classe tenha mais de uma superclasse.
- Esta feature, não suportada por Java, denomina-se **herança múltipla**.
- Java no entanto, por razões de eficiência e simplicidade oferece o mecanismo de **interface** para suportar herança múltipla.

Para que serve Interface?

- Java não permite herança múltipla, mas interfaces provêem uma alternativa.
- Em Java, uma classe pode ser herdada de somente uma classe mas ela pode implementar mais de uma interface.

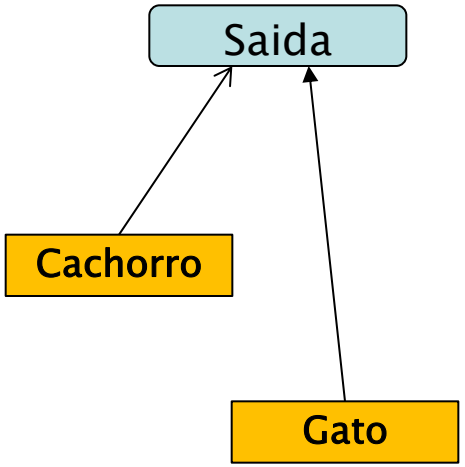


Para que serve Interface?

- É um meio de empacotar constantes.
- Você pode usar uma interface contendo constantes em qualquer número de diferentes classes que tenham acesso à interface.
- As constantes são **static** e assim são compartilhadas entre todos os objetos da classe.

Exemplo - Interface

```
public interface Saida {
        void som();
}
public class Cachorro implements Saida {
        public void som() {
                . . .
        }
}
public class Gato implements Saida {
        public void som() {
                . . .
        }
}
```



Exemplo - Interface

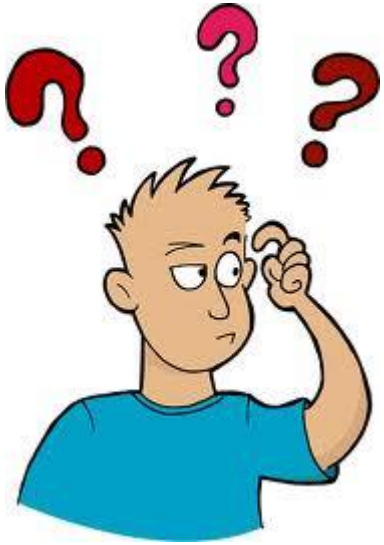
```
public class Cachorro implements  Saida {

        Cachorro (String nome) {
                this.nome = nome;
                this.raca = " Desconhecida ";
        }
        Cachorro (String nome, String raca) {
                this.nome = nome;
                this.raca = raca;
        }
        public void som() {
                System.out.println(" Uau.. Uau… ");
        }
} // fim da classe Cachorro
```

Exemplo - Interface

```
public class Gato implements  Saida   {
        Gato (String nome) {
                this.nome = nome;
                this.raca = " Desconhecida ";
        }
        Gato (String nome, String raca) {
                this.nome = nome;
                this.raca = raca;
        }
        public void som() {
                System.out.println(" Miau.. Miau… ");
        }
} // fim da classe Gato
```

Classes Genéricas

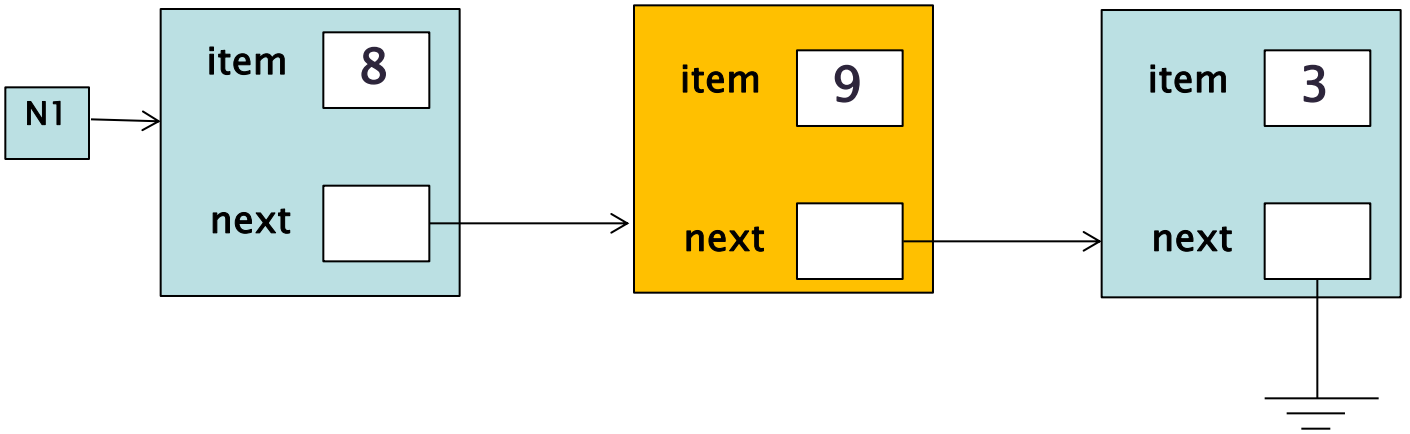
O que são tipos genéricos ?



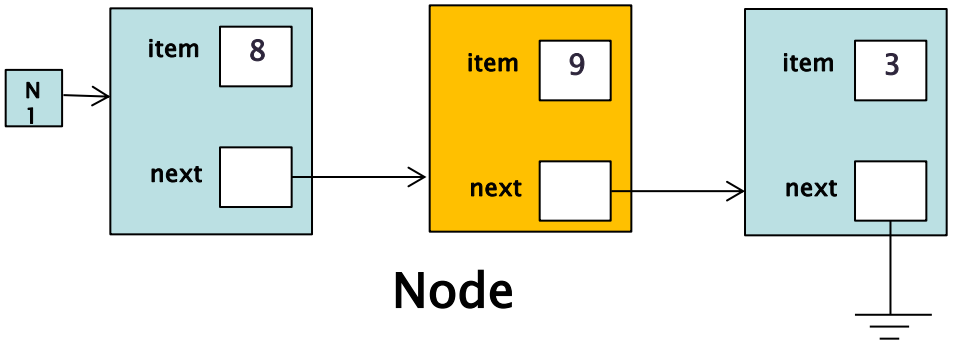
Tipo Genérico

- Também chamado de **tipo parametrizado**, é uma definição de classe que tem um ou mais tipos de parâmetros.
- Por exemplo, considere uma lista ligada de inteiros com um conjunto de operações definidas.
- Poderíamos necessitar da mesma lista ligada para implementar Strings, e assim por diante.

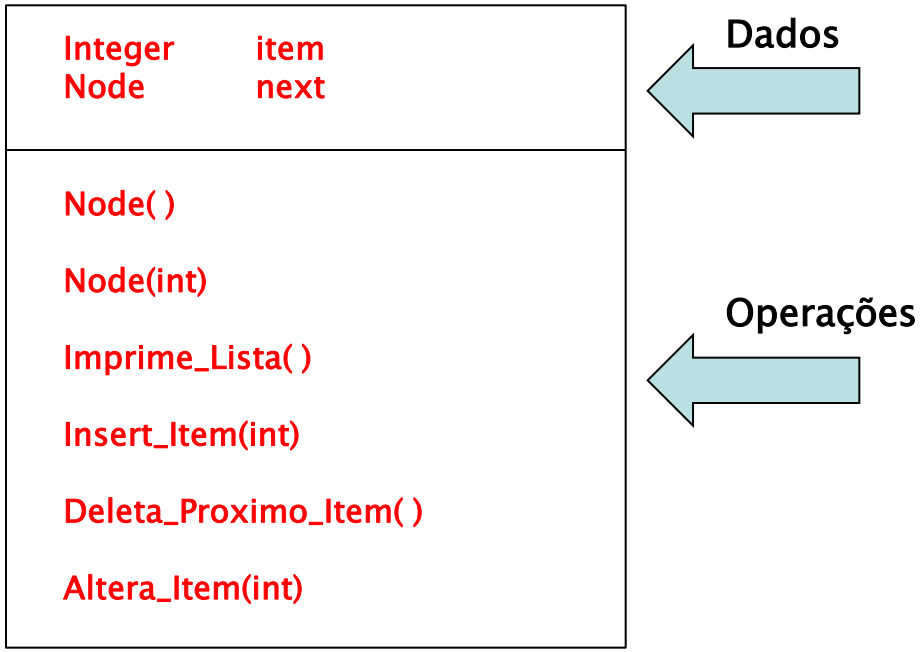
Lista ligada de Inteiros



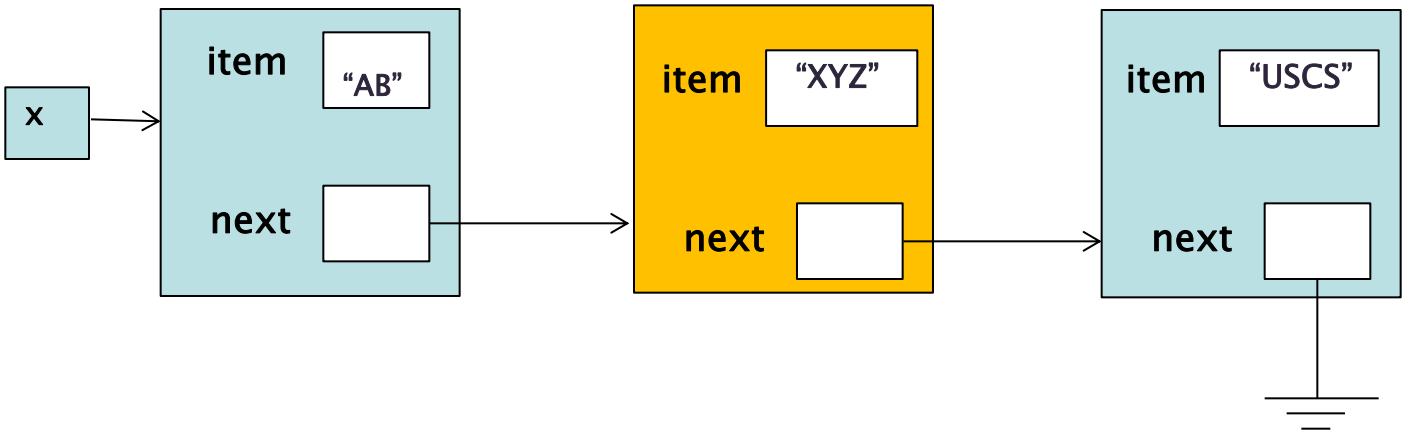
Lista ligada de Inteiros



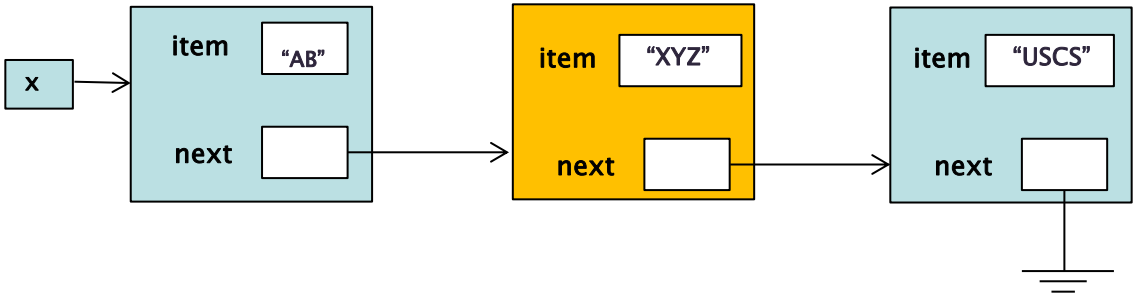
TAD



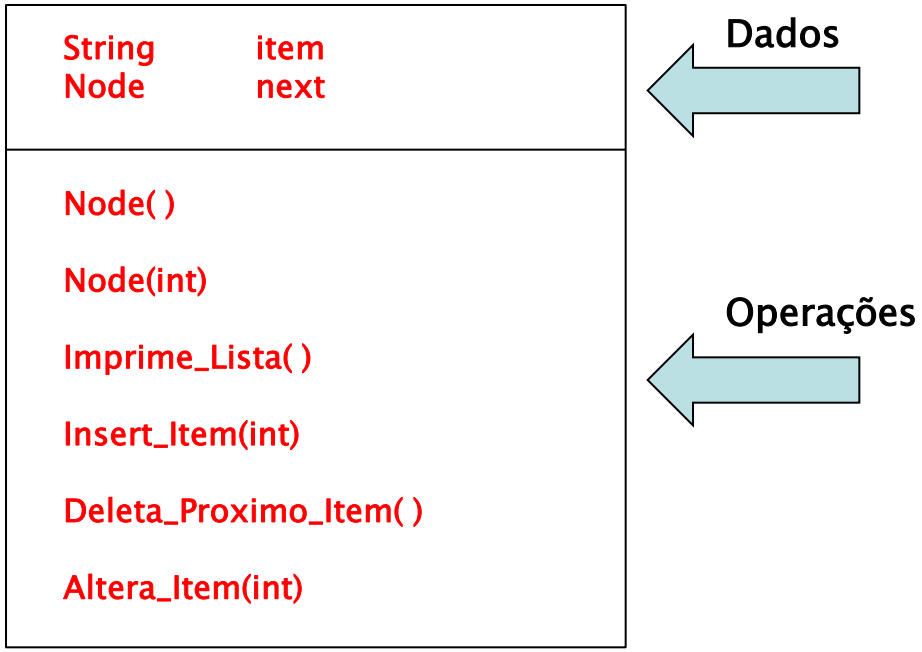
Lista ligada de Strings



Lista ligada de Strings



TAD



Coleção de classes

- A lista ligada é um exemplo de classe que pode ser definida como uma **classe de tipo genérico**.
- Uma classe de tipo genérico define coleção de classes (**collection**).
- Uma lista ligada com tipos genéricos apresenta a vantagem de ser aplicável à qualquer tipo de dados que quizermos implementar a lista.
- São semelhantes aos templates em C++ (**STL** = Standard Template Library).

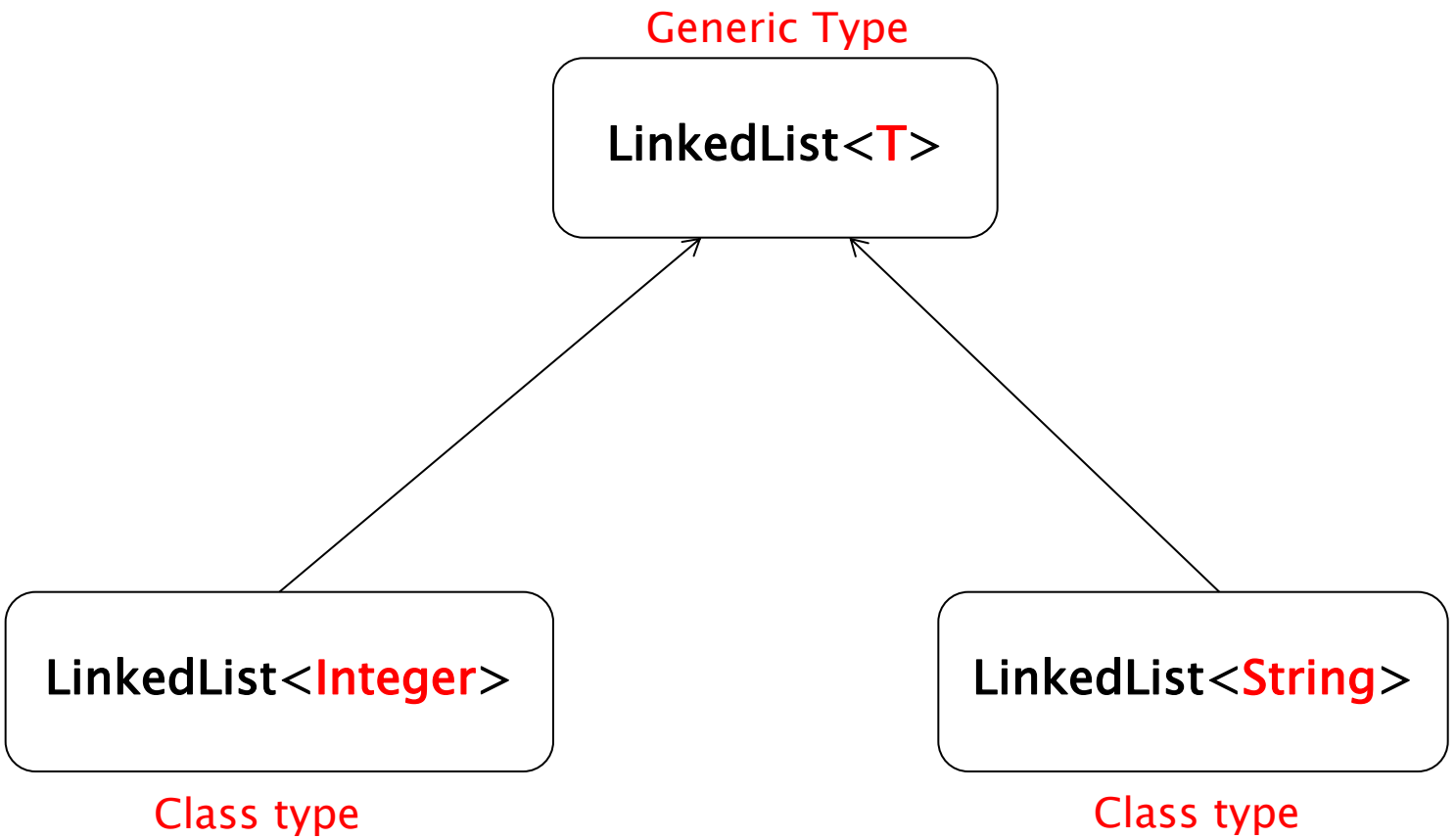
Como definir uma classe de tipo genérico ?

Classe Genérica - Definição

```
public    class    UserClass<T>  {

          //definicao de tipo genérico
}
```

- **T** entre **< >** é chamado tipo genérico (type parameter).
- Para se criar uma classe a partir da classe genérica, devemos simplesmente fornecer um apropriado argumento para o parâmetro entre **<** e **>**.
- Por exemplo: LinkedList<int> ou LinkedList<String> .

Classes Genéricas



Classes Genéricas – Observações

- O argumento para o tipo genérico deve ser uma **classe** ou **interface**.
- Ou seja, não é permitido o uso de tipos primitivos, por exemplo, int ou double. Deve-se usar wrapper classes, tais como Integer, Double, etc.
- Ao se criar um tipo particular a partir de um genérico, o argumento é substituído em toda a ocorrência de **T** na especificação genérica de tipo.

Exemplo - Lista ligada Genérica

```
package maua;

public class Node<T>     {

        T item;
        Node<T> next;

        public Node(T item) {

                this.item = item;
                this.next = null;
        }

        public Node() {

                this.item = null;
                this.next = null;
        }
```

Função Genérica de Inserção

```
public void insert_Item(T item) {

        Node<T> no_trab = new Node<T>(item);

        Node<T> p = this;

        while (p.next != null)

                p = p.next;

        p.next = no_trab;

}
```

Função Genérica de Impressão

```
public void Imprime_Lista() {
        Node<T> no_trab = this;
        System.out.print("Lista: ");
        while (no_trab != null ) {
                System.out.print("   " + no_trab.item);
                no_trab = no_trab.next ;
        }
        System.out.println("");
 }


}
```

Classe para execução

```
package maua;

public class Teste_Node {

   public static void main(String[] args ){

        Node<Integer> x = new Node<>(10);

        x.insert_Item(20);
        x.insert_Item(30);

        x.Imprime_Lista();

        Node<String> y = new Node<>("Corinthians");
        y.insert_Item("Sao Paulo");
        y.insert_Item("Santos");

        y.Imprime_Lista();

    }
}
```

```
Lista:    10   20   30
Lista:    Corinthians   Sao Paulo   Santos
```

API Collections Framework

- Exemplos de tipos genéricos são encontrados na **Collections Framework Java SE 5.0**.
- A necessidade de tipos genéricos surgiu na implementação e uso de **collections**.
- Uma coleção sempre contém elementos de um determinado tipo, tais como uma lista de inteiros, ou uma lista de Strings, etc.
- **ArrayList** é uma classe genérica com um type parameter que pertence ao Framework.
- **ArrayList** implementa uma estrutura de dados que modifica dinamicamente o seu tamanho em tempo de execução.

Collections Framework

- É uma arquitetura unificada para representar e manipular **collections**.
- São constituídas por: interfaces, implementações e algoritmos.
- Interfaces correspondem aos tipos abstratos de dados e permitem que **collections** sejam manipuladas de forma independente dos detalhes de implementação.
- Implementações são as classes concretas do framework e em essência correspondem às estruturas de dados reusáveis.
- Algoritmos correspondem aos métodos que executam as computações úteis sobre as **collections**.

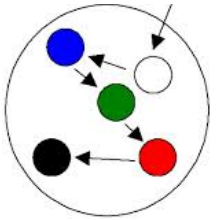
Quais os benefícios de se usar Collections Framework?

Vantagens

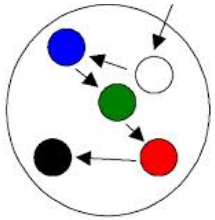
- Redução do esforço de programação, pois o **framework** oferece um conjunto de algoritmos eficientes, prontos para serem utilizados e reutilizados.
- Aumento da qualidade e produtividade no desenvolvimento.
- Interoperabilidade de uso entre as classes do framework.
- Estruturas de dados dinâmicas, ou seja, a adição ou remoção de objetos ocorre de forma automática, sem que o programador tenha que se preocupar com a alocação de espaço em memória.

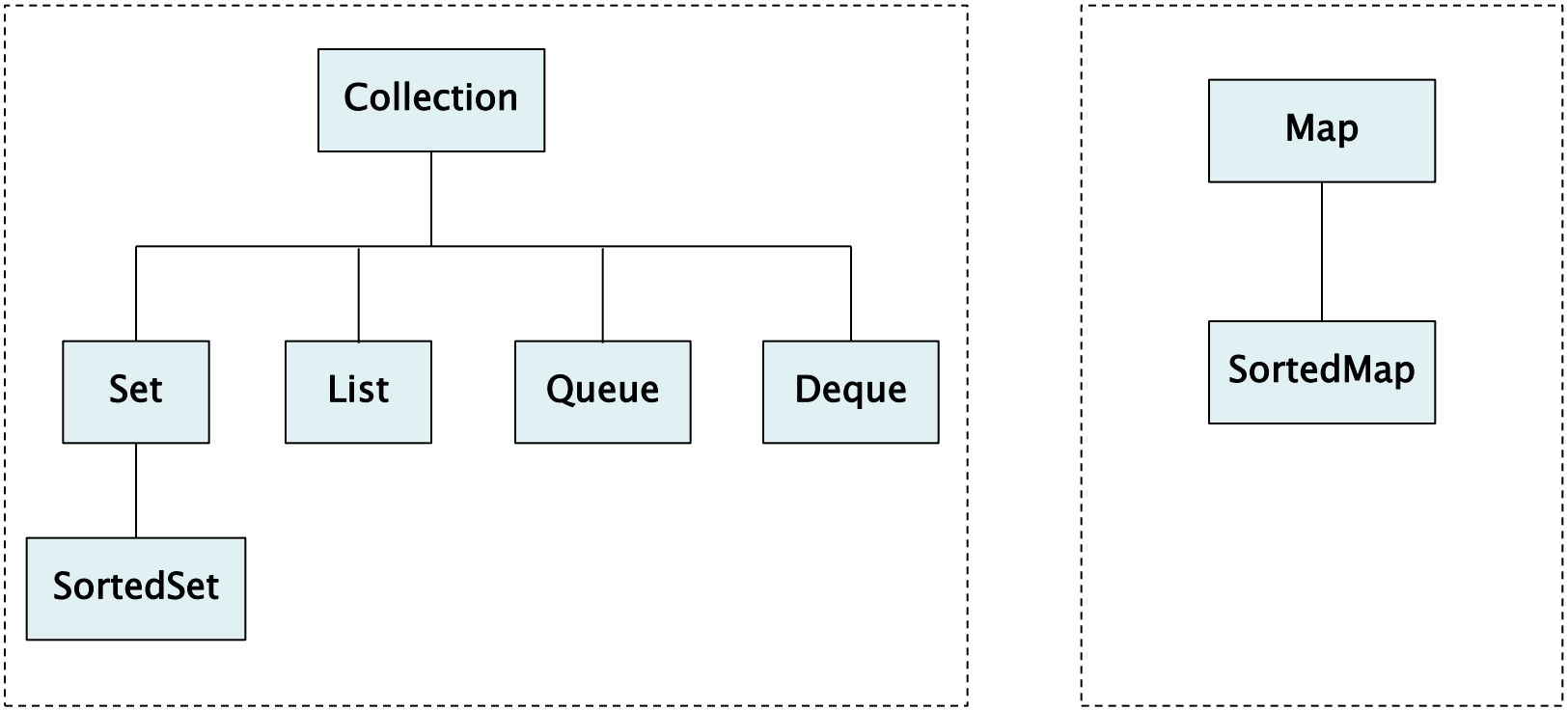
Interfaces

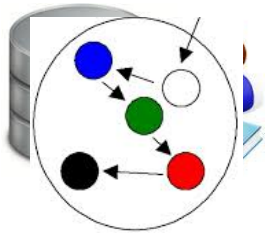
- Encapsulam diferentes tipos de **collections**.
- Permitem que as **collections** sejam manipuladas independentemente dos detalhes de como foram implementadas.
- Representam a base da Java Collections Framework.



Interfaces

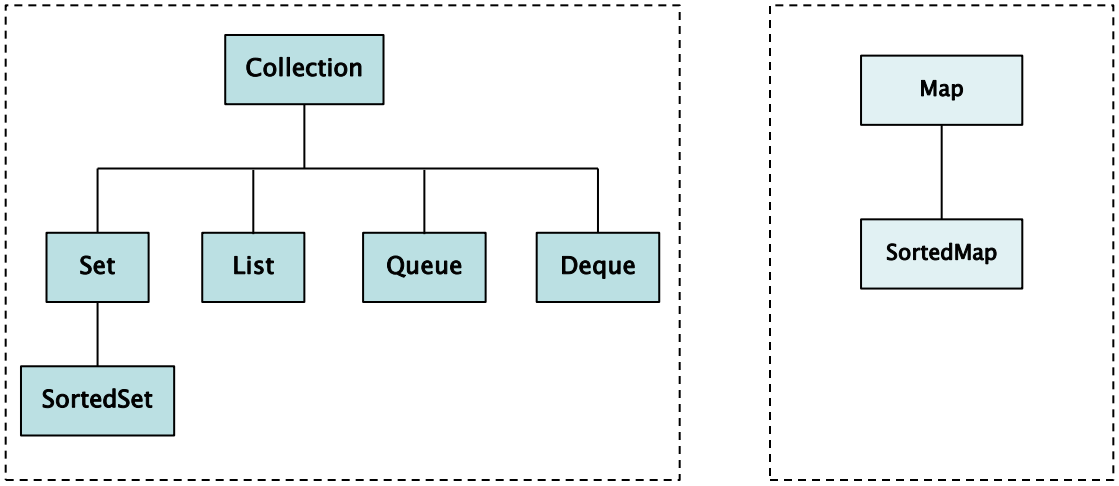






Interfaces

- As interfaces estão estruturadas na forma de hierarquia.
- Assim, um **SortedSet** é um tipo especial de **Set**.
- Um **Set** é um tipo especial de **Collection**
- A hierarquia consiste de duas árvores distintas. Map não é filha de **Collection**.

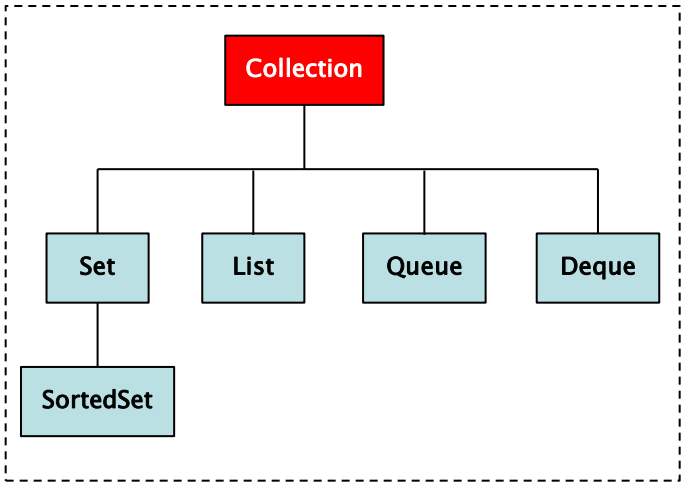


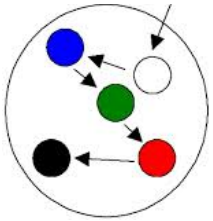
Interface Collection

- A interface **Collection** está no topo da hierarquia das interfaces e representa um grupo de interfaces.

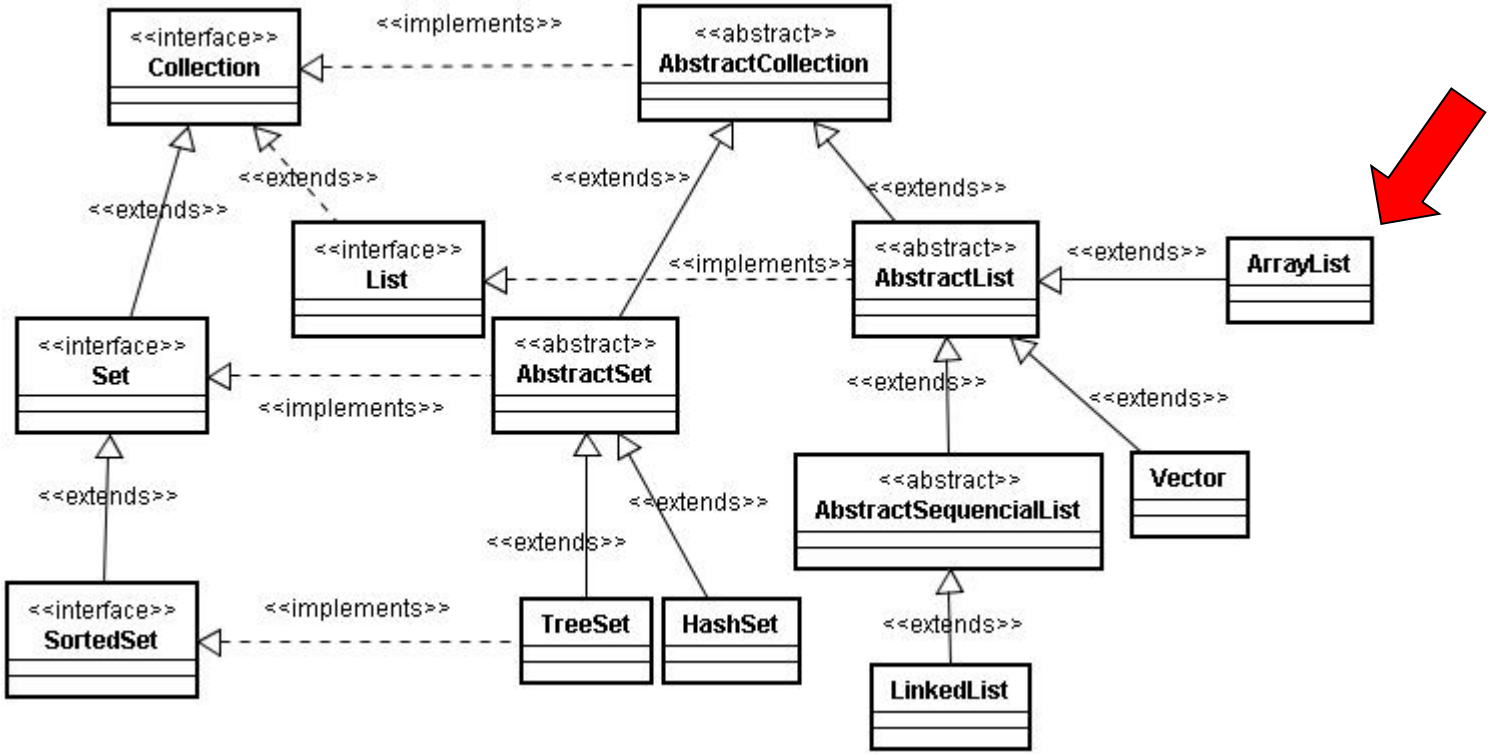
- Interface base para todos os tipos de coleção. Possui um conjunto mínimo de métodos que todas as collections implementam. Trazem assim funcionalidades genéricas para todas as collections.

- Ela define as operações mais básicas para coleções de objetos, como adição (**add**) e remoção (**remove**), esvaziamento (**clear**), tamanho (**size**), conversão para array (**toArray**), objeto de iteração (**iterator**), e verificações de existência (**contains** e **isEmpty**).





Classes e interfaces que estendem ou implementam a interface Collection



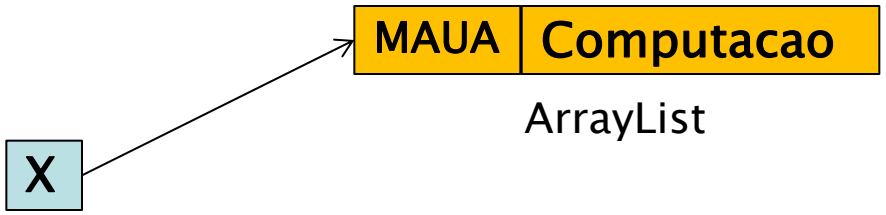
Exemplo - Classe ArrayList

- ArrayList é uma classe concreta da **API Framework Collections**.
- Diferentemente da estrutura Array que tem tamanho fixo, um arraylist é um objeto que pode modificar seu tamanho e, portanto, é adequado para situações onde se necessita de comportamento dinâmico.
- Assim, um ArrayList tem o mesmo propósito de um Array, mas seu tamanho pode se modificar em tempo de execução.

ArrayList – método add

boolean	add(E e)Appends the specified element to the end of this list.

```
package maua;

import java.util.ArrayList;

public class ArrayList_01 {

        public static void main(String[] args) {

                ArrayList<String> x = new ArrayList<String>();
                x.add("MAUA");
                x.add("Computacao");
                System.out.println(x.toString()) ;
        }
}
```



[MAUA, Computacao]

Tratamento de Exceções

Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP

aparecidovfreitas@gmail.com

Bibliografia

- Beginning Java 2 – Ivor Horton – 1999 WROX
- Java2 – The Complete Reference – 7th Edition – Herbert Schildt – Oracle Press
- Core Java Fundamentals – Horstmann / Cornell – PTR- Volumes 1 e 2 – 8th Edition
- Inside the Java 2 – Virtual Machine Venners – McGrawHill
- Understanding Object-Oriented Programming with JAVA – Timothy Budd – Addison Wesley
- Head First Java, 2nd Edition by Kathy Sierra and Bert Bates
- Effective Java, 2nd Edition by Joshua Bloch
- Thinking in Java (4th Edition) by Bruce Eckel
- Java How to Program - 9th Edition by Paul Deitel and Harvey Deitel

Introdução

- Numa aplicação em um mundo ideal, o usuário jamais entraria com dados em formato inválido, arquivos sempre existiriam, e o código nunca teria bugs.

Erros ...

- Infelizmente, durante a execução de uma aplicação podem ocorrer erros.

Tente executar este código !

```
public class Erro1 {
  public static void main(String[] args){

     int x = 4, y=0;
     System.out.println(x/y);

  }
}
```

Tente executar este código !

```
public class Erro2 {

  public static void main(String[] args){

     int[] vet = new int[3];
     for (int i=0; i<4; i++)
         vet[i] = i+1;
  }
}
```

Quais as causas de erros ?

Causas de erros...

- O erro pode ser causado por um arquivo com informação inválida, um problema de conexão na rede, ou índice de array inválido, ou ainda a tentativa de referenciar um objeto que aponta para null, ou erros de input do usuário, erros de devices, limitações físicas, erros de codificação, etc.

Vé ? Mas os métodos não retornam sempre um Return Code ?

RC = -1

Return Code

- Infelizmente, nem sempre é possível para um método retornar um Código de Retorno.
- Java, no entanto, permite que um caminho alternativo seja utilizado quando uma tarefa não puder ser completada.
- Nesta situação, um método não retorna valor. Ao invés disto, um objeto é lançado (**throws**) e este objeto encapsula todas as informações do erro.

Como prevenir erros ?

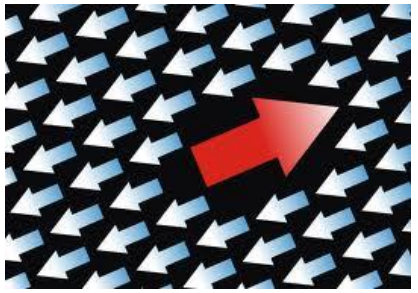
Como prevenir erros ?

- Java usa uma forma de tratamento de erros, chamada **exception-handling**. (manuseio de exceções...)
- Este mecanismo é semelhante a C++.

O que é uma exception ?

Exception

- Quando um programa viola uma regra semântica da linguagem, a Máquina Virtual sinaliza o erro na forma de uma **exception**.
- Uma **exception** é um objeto, que ocorre durante a execução de um programa, e encapsula todas as informações do erro ocorrido.
- Uma **exception** é sempre uma instância da classe **Throwable**.



Qual o impacto de uma exception durante a execução de um programa ?

Impacto da Exception

- Uma **exception** causa uma **transferência de controle** do programa a partir do ponto onde a **exception** ocorreu até o ponto em que tenha sido especificada pelo programador.

Terminologia

- Uma exceção é dita ser lançada (**thrown**) a partir do ponto onde ocorreu e é dita apanhada (**caught**) no ponto para o qual o controle é transferido.

thrown

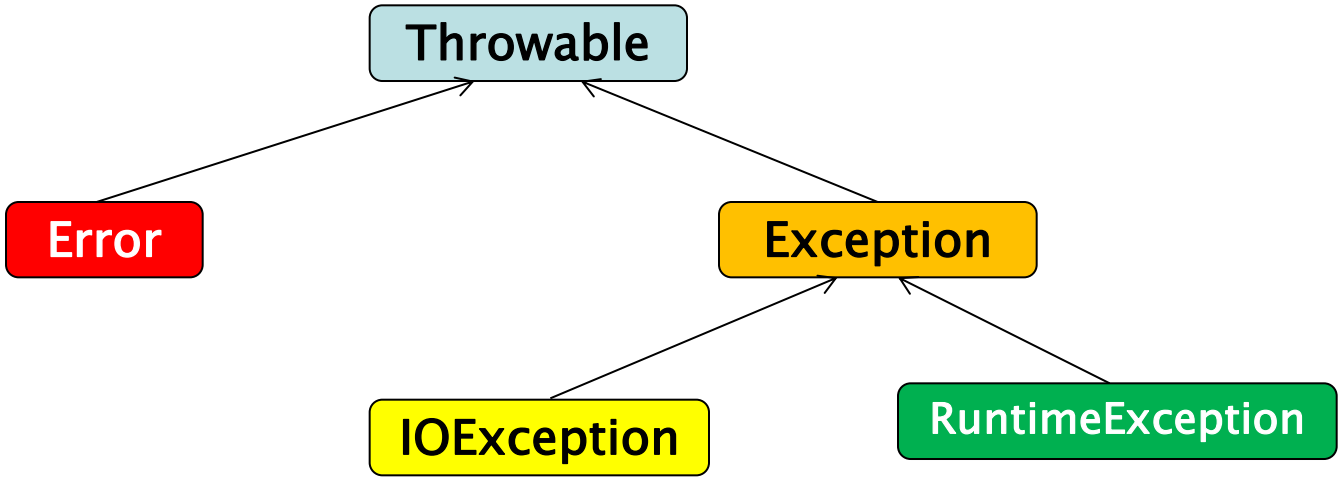
caught

Comando throw

- Programas podem lançar exceções explicitamente, por meio do comando **throw**.
- O uso explícito do comando **throw** torna os programas mais robustos e menos propensos a comportamento indesejado.

A classe Throwable

- Todas as exceções descendem da classe **Throwable**.
- Logo abaixo dela, descendem duas hierarquias: **Error** e **Exception**.

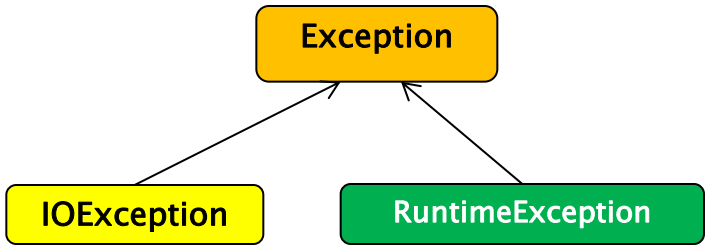


A classe Error

- Descreve erros internos e esgotamento de recursos dentro do ambiente de execução (run-time) da **Máquina Virtual**.
- Você não deve lançar (**throw**) objetos deste tipo.
- Há muito pouco o que se deve fazer quando errros deste tipo ocorrerem, além de notificar o usuário e tentar terminar o programa de forma agradável.
- Estas situações raramente ocorrem.

A classe Exception

- É a classe onde a programação Java é focada.
- Também tem duas ramificações: aquelas que são derivadas de **RuntimeException** e aquelas que não são.
- A regra geral é: Uma **RuntimeException** ocorreu porque houve um erro de programação.
- Qualquer outra má coisa que ocorreu não foi por culpa do programa. Por exemplo, um **I/O error**.



Quais exceptions são herdadas da classe RuntimeException ?

Exceptions herdadas de RuntimeException

- Uma operação de casting incorreta.
- Erro de endereçamento de acesso à arrays.
- Acesso a objetos com pointer null.

Subclasses da RunTimeException

- ArithmeticException
- IndexOutOfBoundException
- Negative ArraySizeException
- NullPointerException
- ArrayStoreException
- ClassCastException
- IllegalArgumentException
- SecurityException
- IllegalMonitorStateException
- IllegalStateException
- UnSupportedOperationException

Qual a missão do Error-Handling ?

Missão do Error-Handling

- Transferir o controle do ponto onde ocorreu o erro para um tratador de erros (error-handler) que irá tratar da situação sob erro.

Ciclo de vida de uma Exception

- Quando o sistema de exceptions detecta um erro dentro de um método, o método cria um objeto e o encaminha para o sistema de runtime.

- Este objeto é denominado objeto **exception**.

Ciclo de vida de uma Exception

- O objeto **exception** contém informações sobre o erro, incluindo o tipo e o estado do programa quando ocorreu o erro.
- A criação de um objeto **exception** e o seu emprego no sistema de runtime é conhecido por throwing.

Ciclo de vida de uma Exception

- Após um método encaminhar uma **exception** (throws), o sistema de runtime tenta encontrar algum procedimento para manuseá-lo.
- O conjunto de possíveis procedimentos para tratar a exceção é uma lista ordenada de métodos conhecida por **call stack**.

Ciclo de vida de uma Exception

- O sistema de runtime pesquisa a **call stack** visando encontrar algum método que contenha um bloco de código que possa manusear (tratar) a exceção.
- Este bloco de código é conhecido por **exception handler**.

Ciclo de vida de uma Exception

- Quando um apropriado **handler** é encontrado, o sistema de runtime passa o objeto **exception** para o **handler**.
- Um **handler** é considerado apropriado se o tipo do objeto exception coincidir com o tipo do objeto exception manuseado pelo **handler**.
- O exception handler escolhido é dito aceitar a exception (catch the exception).

Ciclo de vida de uma Exception

- Se o sistema de runtime exaustivamente pesquisar todos os métodos na **call stack** sem encontrar um apropriado exception handler, o sistema de runtime irá abortar o programa.

Principal vantagem da Manipulação de erros através de Exceções

- Separação do código para manipulação de erros do código "normal" do programa.

Principal vantagem da Manipulação de erros através de Exceções

```
Processo_Entrada ( ) {
   abrir arquivo;
   determinar seu tamanho;
   alocar memória suficiente;
   ler o arquivo para a memória;
   fechar o arquivo;
}
```

Tratamento tradicional

```
int MeuMetodo ( ) {
        int  erro=0;
        abrir arquivo;
        se (conseguir abrir arquivo) então{
                obter tamanho do arquivo;
                se (conseguir obter tamanho) então{
                        alocar memória;
                        se (conseguir alocar memória então {
                                ler arquivo;
                                se (houve erro de leitura) então {
                                        erro = -10;
                                } senão erro = -20;
                        } senão erro = -30;
                fechar arquivo;
                se (houve erro de fechamento) então {
                        erro = -40;
                } senão erro = -50;
        Retorno do erro;
}
```

Tratamento com exceptions

```
int MeuMetodo ( ) {
        try {
                abrir arquivo;
                determinar seu tamanho;
                alocar memória suficiente;
                ler o arquivo para a memória;
                fechar o arquivo;
        }
        catch (Exceção falhaNaAberturaArquivo) {
                processaFalhaAbertura;
        }
        catch (Exceção falhaObtencaoTamanho) {
                processaFalhaObtencaoTamanho;
        }
        catch (Exceção falhaAlocacaoMemoria) {
                processaFalhaAlocacaoMemoria;
        }
        catch (Exceção falhaLeituraArquivo) {
                processaFalhaLeitura;
        }
        catch (Exceção falhaFechamentoArquivo) {
                processaFalhaFechamentoArquivo;
        }
}
```

É bom lembrar que...

- Exceções não fazem **milagres**, ou seja, diminuem o esforço necessário para se detectar, reportar e manipular erros.
- O que elas permitem é a separação do código fonte tradicional do código responsável para processar alguma ação quando algo grave ocorre no programa.

Manuseando exceções

- Para tratarmos exceções em um programa quando elas ocorrerem, há três tipos de blocos de código que podemos incluir em um método para manuseá-las:
  - ✓ bloco try
  - ✓ bloco catch
  - ✓ bloco finally

Bloco try

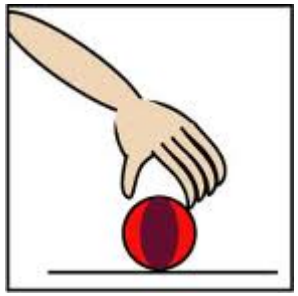
- Define um bloco de código que pode causar uma ou mais exceções.
- Ou seja, quando você quizer interceptar exceções (catch) o código que pode causá-las deve estar delimitado pelo bloco **try**.
- Código que cause exceção e que não esteja delimitado pelo bloco **try** não será capaz de capturá-la (por meio de catch correspondente).

Bloco try

```
try  {
        // código que pode disparar
        // uma ou mais exceções. . .
}
```

Bloco catch

- Define o código que irá manusear a exceção de um determinado tipo.
- Deve seguir imediatamente o bloco **try**.



Bloco catch

```
try {
        // codigo que pode disparar
        // uma ou mais exceções. . .
}
catch (ArithmeticException e) {
        // codigo para tratar a exceção
}
```

Bloco catch

- No exemplo anterior, o bloco **catch** somente manuseia (trata) exceções do tipo **ArithmeticException**.
- Isto implica que este é o único tipo de exceção que pode ser disparada (thrown) no bloco **try**.

Bloco catch

- Em geral, o parâmetro para um bloco **catch** deve ser do tipo **Throwable** ou uma de suas subclasses.
- Se a classe especificada como parâmetro tem subclasses,então o bloco catch irá tratar exceções desta classe mais todas as suas subclasses.

Exemplo

```
public class ExemploTryCatch  {
  public static void main(String[] args)   {
    int i = 1;
    int j = 0;
    try {
      System.out.println(" Entrada bloco try  " + "i = "+ i + " j = "+j);
      System.out.println(i/j);      // Divisao por 0 - exception thrown
      System.out.println(" Fim bloco try ");
    }
    // Catch the exception
    catch(ArithmeticException e) {
      System.out.println(" Arithmetic exception caught ");
    }
    System.out.println(" Apos bloco try ");
    return;
  }
}
```

Exemplo - saída

```
Entrada bloco try   i = 1 j = 0
Arithmetic exception caught
Apos bloco try
```

O par try/catch

- O par try catch formam um casal.
- Você não deve separá-los incluindo comandos entre os mesmos.

Exemplo

```
public class TestLoopTryCatch {
  public static void main(String[] args)  {
    int i = 12;

    for(int j=3 ;j>=-1 ; j--)
    try   {
      System.out.println("Try block entered " + "i = "+ i + " j = "+j);
      System.out.println(i/j);          // Divide by 0 - exception thrown
      System.out.println("Ending try block");
    }
    // Catch the exception
    catch(ArithmeticException e) {
      System.out.println("Arithmetic exception caught");
    }

    System.out.println("After try block");
    return;
  }
}
```

Exemplo - saída

```
Try block entered i = 12 j = 3
4
Ending try block
Try block entered i = 12 j = 2
6
Ending try block
Try block entered i = 12 j = 1
12
Ending try block
Try block entered i = 12 j = 0
Arithmetic exception caught
Try block entered i = 12 j = -1
-12
Ending try block
After try block
```

Exemplo

```
public class TestLoopTryCatch {
  public static void main(String[] args) {
    int i = 12;
    try {
      System.out.println("Try block entered.");
      for(int j=3 ;j>=-1 ; j--) {
        System.out.println("Loop entered " + "i = "+ i + " j = "+j);
        System.out.println(i/j);          // Divide by 0 - exception thrown
      }
      System.out.println("Ending try block");
    }
    // Catch the exception
    catch(ArithmeticException e) {
      System.out.println("Arithmetic exception caught");
    }
    System.out.println("After try block");
    return;
  }
}
```

Exemplo - saída

```
Try block entered.
Loop entered i = 12 j = 3
4
Loop entered i = 12 j = 2
6
Loop entered i = 12 j = 1
12
Loop entered i = 12 j= 0
Arithmetic exception caught
After try block
```

Múltiplos blocos Catch

- Se um bloco **try** pode disparar (throw) diversos tipos de exceções, então você poderá definir diversos blocos **try** para manusear estas diferentes exceções.

Múltiplos blocos Catch

```
try {
        // codigo que pode disparar
        // (throw) exceções
}
catch (ArithmeticException e) {
        // codigo para tratar as excecões do tipo
        // ArithmeticException
}
catch (IndexOutOfBoundsException e) {
        // codigo para tratar as excecões do tipo
        // IndexOutOfBoundsException
}
// execução continua aqui…
```

Múltiplos blocos Catch

- Quando você necessitar manusear (catch) exceções de diferentes tipos em um bloco **try**, a ordem dos blocos **catch** é importante.
- Quando uma exceção é disparada (thrown), ela será tratada pelo primeiro bloco **catch** com o parâmetro de mesmo tipo da exceção ou um tipo que é uma superclasse do tipo da exceção.

Múltiplos blocos Catch

- Isto tem uma importante implicação.
- Os blocos **catch** devem estar na seqüência do mais derivado tipo primeiro, e o tipo mais básico por último.
- Caso contrário, o código não compilará.
- A razão é simples: se o primeiro for mais geral e o segundo derivado do primeiro, então o segundo jamais será executado. . .

Múltiplos blocos Catch

```
try  {
        // bloco de codigo try
}
catch (Exception  e) {
        // manuseio generico de exceção
}
catch (AritmeticException  e) {
        // tratamento especializado
        // este catch jamais será executado. . .
        // compilador irá marcar como erro . . .
}
```

Múltiplos blocos Catch

- Em princípio, se você está interessado em exceções genéricas, todo o tratamento de erro pode ser localizado em um único bloco **catch** para exceções do tipo da superclasse.
- Entretanto, de forma mais útil você deve ter um bloco **catch** para cada tipo especializado de exceções que o bloco **try** pode disparar.

O bloco finally

- A exceção pode introduzir a possibilidade de deixar o program em estado insatisfatório.
- Por exemplo, você abriu um arquivo, e por causa de uma exceção, o código para fechar o arquivo não foi executado.
- O bloco **finally** provê o meio para sincronizar a execução do bloco **try**.

O bloco finally

- Um bloco **finally** é sempre executado, independentemente do que ocorre durante a execução do método.
- Se um arquivo precisa ser fechado, ou se algum recurso crítico precisa ser liberado, você pode garantir esta providência por meio do bloco **finally**.

O bloco finally

```
finally {
        // codigo de clean-up para ser
        // executado por ultimo
}
```

- Deve sempre ser seguido dos blocos **try** e **catch**.
- Se não houver bloco **catch**, então deve seguir imediatamente após o bloco **try**.

Estruturando um método

```
try {
        // codigo que pode disparar exceções...
}
catch  (ExceptionType1 e ){
       // codigo para tratar exceção do tipo
       // ExceptionType1 ou subclasses
}
catch  (ExceptionType2 e ){
       // codigo para tratar exceção do tipo
       // ExceptionType2 ou subclasses
}
// mais blocos catch se necessarios. . .
finally {
       // codigo para ser executado após
       // o bloco try
}
```

Estruturando um método

- Você não pode ter apenas um bloco **try**. Cada bloco **try** deve sempre ser seguido por ao menos um bloco catch ou **finally**.
- Você não pode incluir código entre um bloco **try** e seus blocos **catch**, ou entre o bloco **try** e o bloco **finally**.
- Você pode ter outros blocos **try** em um método.

Sequência de Execução

```
import java.io.IOException;
public class TryBlockTest {
  public static int divide(int[] array, int index) {
    try {
      System.out.println("\nFirst try block in divide() entered");
      array[index + 2] = array[index]/array[index + 1];
      System.out.println("Code at end of first try block in divide()");
      return array[index + 2];
    }
    catch(ArithmeticException e) {
      System.out.println("Arithmetic exception caught in divide()");
    }
```

```
    catch(ArrayIndexOutOfBoundsException e)   {
        System.out.println(
                "Index-out-of-bounds exception caught in divide()");
    }
    finally  {
        System.out.println("finally block in divide()");
    }
    System.out.println("Executing code after try block in divide()");
    return array[index + 2];
  }
}
```

```
public static void main(String[] args) {
   int[] x = {10, 5, 0};              // Array of three integers
   // This block only throws an exception if method divide() does
   try   {
     System.out.println("First try block in main() entered");
     System.out.println("result = " + divide(x,0));  // No error
     x[1] = 0;                        // Will cause a divide by zero
     System.out.println("result = " + divide(x,0));  // Arithmetic error
     x[1] = 1;                        // Reset to prevent divide by zero
     System.out.println("result = " + divide(x,1));  // Index error
   }
   catch(ArithmeticException e) {
     System.out.println("Arithmetic exception caught in main()");
   }
   catch(ArrayIndexOutOfBoundsException e) {
     System.out.println(
               "Index-out-of-bounds exception caught in main()");
   }
   System.out.println("Outside first try block in main()");
   System.out.println("\nPress Enter to exit");

   // This try block is just to pause the program before returning
```

```
try  {
      System.out.println("In second try block in main()");
      System.in.read();                  // Pauses waiting for input...
      return;
}
catch(IOException e)   {              // The read() method can throw exceptions
        System.out.println("I/O exception caught in main()");
}
finally     {                         // This will always be executed
        System.out.println("finally block for second try block in main()");
}

System.out.println("Code after second try block in main()");
}
```