Aula de Revisão

Introdução à Programação - IF669

Luís Gabriel Lima (lgnfl)
Vanessa Gomes (vgl2)

Centro de Informática
Universidade Federal de Pernambuco

27 de outubro de 2009

Tópicos

O que é?

- Técnica de programação orientada a objetos usada para organizar e criar classes visando o reuso de software;

O que é?

- Técnica de programação orientada a objetos usada para organizar e criar classes visando o reuso de software;
- Possibilita derivar uma classe de outra já existente;

O que é?

- Técnica de programação orientada a objetos usada para organizar e criar classes visando o reuso de software;
- Possibilita derivar uma classe de outra já existente;
- A classe existente é chamada de *super-classe*, *classe-base* ou *parent class*;

O que é?

- Técnica de programação orientada a objetos usada para organizar e criar classes visando o reuso de software;
- Possibilita derivar uma classe de outra já existente;
- A classe existente é chamada de *super-classe*, *classe-base* ou *parent class*;
- A classe derivada é chamada de *sub-classe* ou *child class*;

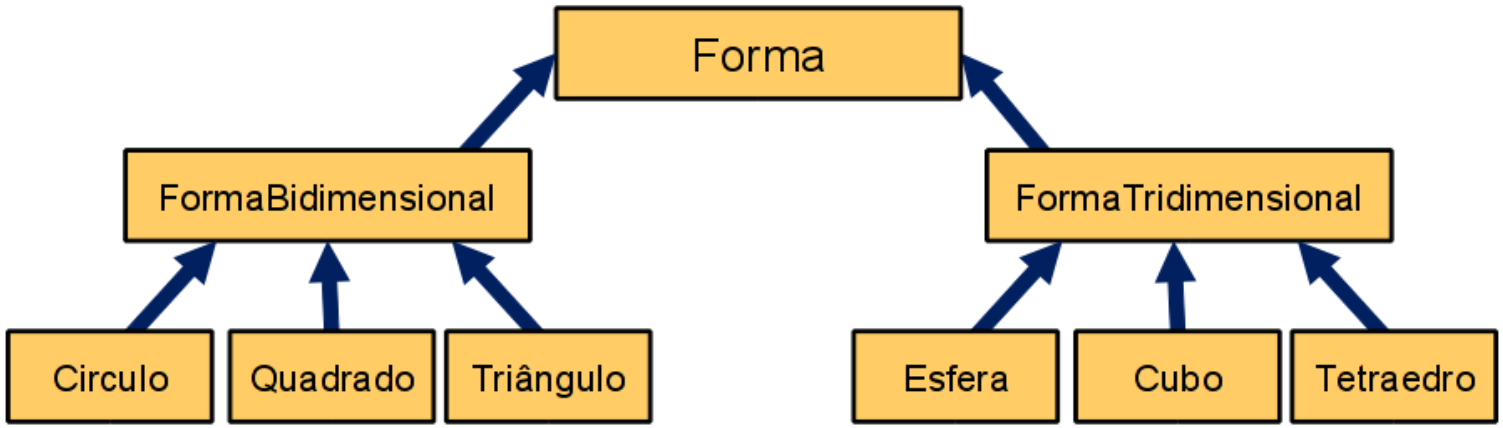
O que é?

- Técnica de programação orientada a objetos usada para organizar e criar classes visando o reuso de software;
- Possibilita derivar uma classe de outra já existente;
- A classe existente é chamada de *super-classe*, *classe-base* ou *parent class*;
- A classe derivada é chamada de *sub-classe* ou *child class*;
- A sub-classe herda as características da super-classe, ou seja, a sub-classe herda os métodos e os atributos definidos pela super-classe;

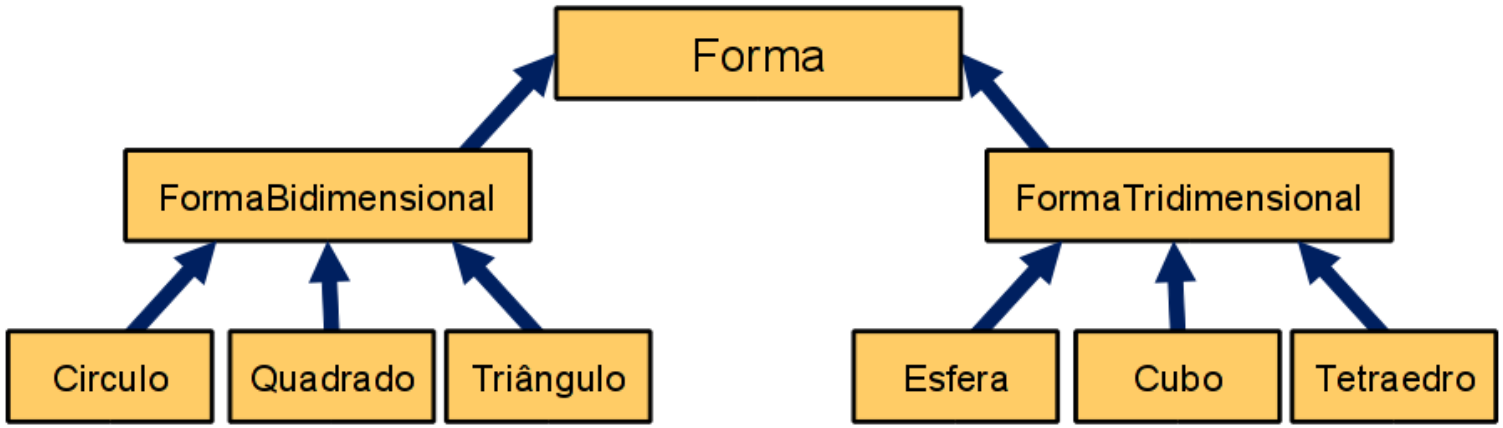
O que é?

- Técnica de programação orientada a objetos usada para organizar e criar classes visando o reuso de software;
- Possibilita derivar uma classe de outra já existente;
- A classe existente é chamada de *super-classe*, *classe-base* ou *parent class*;
- A classe derivada é chamada de *sub-classe* ou *child class*;
- A sub-classe herda as características da super-classe, ou seja, a sub-classe herda os métodos e os atributos definidos pela super-classe;
- A herança cria uma relação **é um**, isto é, a sub-classe **é uma** versão mais específica que a super-classe.

Modelagem


Forma
FormaBidimensional
FormaTridimensional
Circulo
Quadrado
Triângulo
Esfera
Cubo
Tetraedro

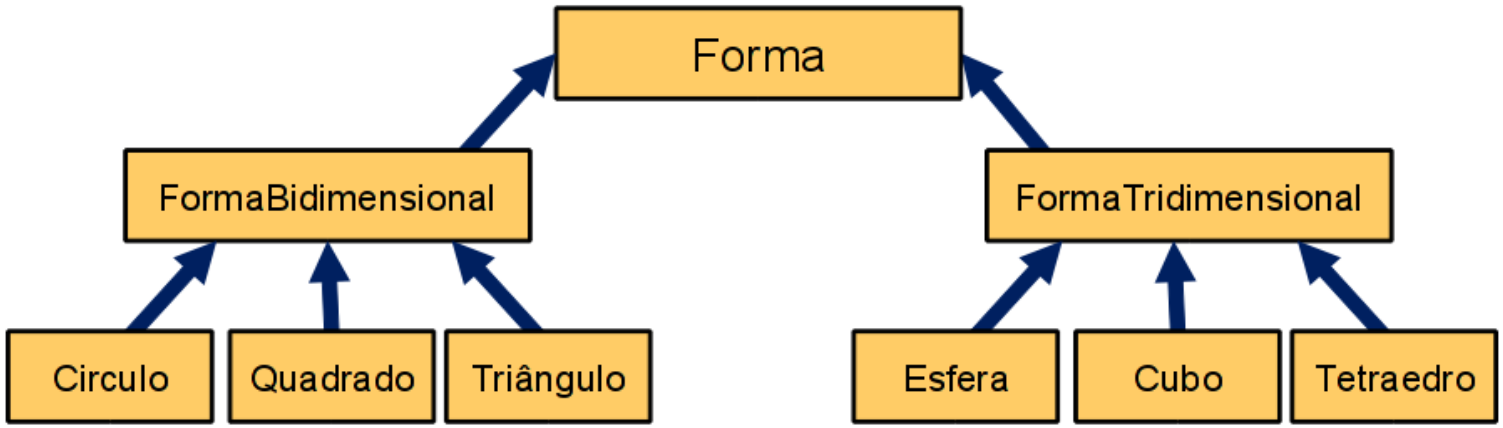
Modelagem



Exemplos de relacionamentos:

- Esfera **é uma** FormaTridimensional.

Modelagem



Exemplos de relacionamentos:

- Esfera **é uma** FormaTridimensional.
- FormaBidimensional **é uma** Forma.

Mais algumas palavras...

- Pode-se implementar uma classe derivada como for preciso, adicionando novos atributos ou métodos, ou até mesmo, modificando os herdados;

Mais algumas palavras...

- Pode-se implementar uma classe derivada como for preciso, adicionando novos atributos ou métodos, ou até mesmo, modificando os herdados;
- Atributos e métodos privados não podem ser referenciados diretamente em sub-classes;

Mais algumas palavras...

- Pode-se implementar uma classe derivada como for preciso, adicionando novos atributos ou métodos, ou até mesmo, modificando os herdados;
- Atributos e métodos privados não podem ser referenciados diretamente em sub-classes;
- Atributos e métodos públicos podem ser referenciados diretamente em sub-classes – mas isso fere o princípio do encapsulamento;

Mais algumas palavras...

- Pode-se implementar uma classe derivada como for preciso, adicionando novos atributos ou métodos, ou até mesmo, modificando os herdados;
- Atributos e métodos privados não podem ser referenciados diretamente em sub-classes;
- Atributos e métodos públicos podem ser referenciados diretamente em sub-classes – mas isso fere o princípio do encapsulamento;
- O modificador `protected` permite à sub-classe referenciar uma variável ou métodos da classe herdada diretamente, isso oferece um maior encapsulamento do que o uso do `public`, mas não tanto quanto o `private`;

Mais algumas palavras...

- Pode-se implementar uma classe derivada como for preciso, adicionando novos atributos ou métodos, ou até mesmo, modificando os herdados;
- Atributos e métodos privados não podem ser referenciados diretamente em sub-classes;
- Atributos e métodos públicos podem ser referenciados diretamente em sub-classes – mas isso fere o princípio do encapsulamento;
- O modificador `protected` permite à sub-classe referenciar uma variável ou métodos da classe herdada diretamente, isso oferece um maior encapsulamento do que o uso do `public`, mas não tanto quanto o `private`;
- Um atributo (ou método) `protected` é visível a qualquer classe dentro do mesmo pacote da super-classe.

Exemplo

```java
class Funcionario{
    protected String nome;
    protected String cpf;
    protected double salario;

    // alguns métodos
}
```

```java
class Gerente extends Funcionario{
    private int senha;

    public boolean autentica(int senha){
        boolean autenticacao = false;
        if(this.senha == senha)
            autenticacao = true;
        return autenticacao;
    }

}
```

Construtores em classes derivadas

- Construtores não são herdados, mesmo que sejam `public`;

Construtores em classes derivadas

- Construtores não são herdados, mesmo que sejam `public`;
- Ainda assim, geralmente precisamos usar o contrutor da super-classe para inicializar a parte herdada;

Construtores em classes derivadas

- Construtores não são herdados, mesmo que sejam `public`;
- Ainda assim, geralmente precisamos usar o contrutor da super-classe para inicializar a parte herdada;
- A referência `super` pode ser usada para referenciar a classe herdada ou para invocar o seu construtor;

Construtores em classes derivadas

- Construtores não são herdados, mesmo que sejam `public`;
- Ainda assim, geralmente precisamos usar o contrutor da super-classe para inicializar a parte herdada;
- A referência `super` pode ser usada para referenciar a classe herdada ou para invocar o seu construtor;
- A primeira linha do construtor da sub-classe deve conter a referência super para chamar o construtor da super-classe;

Construtores em classes derivadas

- Construtores não são herdados, mesmo que sejam `public`;
- Ainda assim, geralmente precisamos usar o contrutor da super-classe para inicializar a parte herdada;
- A referência `super` pode ser usada para referenciar a classe herdada ou para invocar o seu construtor;
- A primeira linha do construtor da sub-classe deve conter a referência super para chamar o construtor da super-classe;
- A referência super também pode ser usada para referenciar outros atributos ou métodos definidos na super-classe.

Exemplo

```java
class Funcionario{
    protected String nome;
    protected String cpf;
    protected double salario;

    public Funcionario(String n, String c, double s){
        this.nome = n;
        this.cpf = c;
        this.saladio = s;
    }

    public double getBonificacao(){
        return (this.salario * 0.1);
    }
}
```

```java
class Gerente extends Funcionario{
    private int senha;

    public Gerente(String n, String c, double s, int se){
        super(n, c, s);
        this.senha = se;
    }

    public double getBonificacao(){
        return (super.getBonificacao() + 500);
    }
}
```

Sobrescrevendo métodos

- Uma sub-classe pode sobrescrever (override) a definição de um método herdado;

Sobrescrevendo métodos

- Uma sub-classe pode sobrescrever (override) a definição de um método herdado;
- O novo método tem que possuir a mesma assinatura do método herdado, mas pode ter um corpo completamente diferente;

Sobrescrevendo métodos

- Uma sub-classe pode sobrescrever (override) a definição de um método herdado;
- O novo método tem que possuir a mesma assinatura do método herdado, mas pode ter um corpo completamente diferente;
- O tipo do objeto que executa o método é que determina qual das versões é realmente invocada;

Sobrescrevendo métodos

- Uma sub-classe pode sobrescrever (override) a definição de um método herdado;
- O novo método tem que possuir a mesma assinatura do método herdado, mas pode ter um corpo completamente diferente;
- O tipo do objeto que executa o método é que determina qual das versões é realmente invocada;
- Se um método for definido na super-classe como `final`, ele não pode ser sobrescrito;

O que são classes abstratas?

- Uma classe abstrata é uma classe que representa uma idéia (conceito) genérica;

O que são classes abstratas?

- Uma classe abstrata é uma classe que representa uma idéia (conceito) genérica;
- Uma classe abstrata não pode ser instanciada;

O que são classes abstratas?

- Uma classe abstrata é uma classe que representa uma idéia (conceito) genérica;
- Uma classe abstrata não pode ser instanciada;
- Usa-se o modificador `abstract` no cabeçalho da classe para declará-la como uma classe abstrata:

O que são classes abstratas?

- Uma classe abstrata é uma classe que representa uma idéia (conceito) genérica;
- Uma classe abstrata não pode ser instanciada;
- Usa-se o modificador `abstract` no cabeçalho da classe para declará-la como uma classe abstrata:

```
public abstract class Trabalhador{

    // corpo da classe

}
```

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);
- o modificador `abstract` tem que ser aplicado a todos os métodos abstratos;

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);
- o modificador `abstract` tem que ser aplicado a todos os métodos abstratos;
- As classes abstratas também podem possuir métodos não abstratos;

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);
- o modificador `abstract` tem que ser aplicado a todos os métodos abstratos;
- As classes abstratas também podem possuir métodos não abstratos;
- As classes abstratas não precisam necessariamente possuir métodos abstratos;

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);
- o modificador `abstract` tem que ser aplicado a todos os métodos abstratos;
- As classes abstratas também podem possuir métodos não abstratos;
- As classes abstratas não precisam necessariamente possuir métodos abstratos;
- Uma sub-classe de uma classe abstrata tem que sobrescrever seus métodos abstratos ou eles continuarão sendo considerados abstratos;

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);
- o modificador `abstract` tem que ser aplicado a todos os métodos abstratos;
- As classes abstratas também podem possuir métodos não abstratos;
- As classes abstratas não precisam necessariamente possuir métodos abstratos;
- Uma sub-classe de uma classe abstrata tem que sobrescrever seus métodos abstratos ou eles continuarão sendo considerados abstratos;
- Um método abstrato não pode ser definido como `final` ou `static`;

Características de classes abstratas

- Uma classe abstrata geralmente contém métodos abstratos (sem definição);
- o modificador `abstract` tem que ser aplicado a todos os métodos abstratos;
- As classes abstratas também podem possuir métodos não abstratos;
- As classes abstratas não precisam necessariamente possuir métodos abstratos;
- Uma sub-classe de uma classe abstrata tem que sobrescrever seus métodos abstratos ou eles continuarão sendo considerados abstratos;
- Um método abstrato não pode ser definido como `final` ou `static`;
- Não pode-se instanciar objetos do tipo de uma classe abstrata.

Considerações sobre herança

- Utilizar (corretamente) herança é uma boa prática de programação OO;
- Uma relação de herança bem feita pode contribuir bastante com a elegância, a manutenção e o reuso do software;
- Diferente de outras linguagens como C++, Java não possui suporte a herança múltipla.

Utilizando herança

- Pense sempre adiante, na potencialidade de uma classe ser herdada;
- Encontre características comuns entre as classes e empurre-as para cima na hierarquia;
- Sobrescreva métodos apropriadamente;
- Adicione novos atributos nas sub-classes, mas não redefina os herdados;
- Permita que cada classe gerencie seu próprio conteúdo
- Use o `super` para chamar métodos ou o construtor da classe herdada;
- Use os modificadores de acesso com cuidado para não violar o encapsulamento.

Binding

Considere a seguinte chamada de método:
`objeto.executar();`

- Em algum ponto, essa chamada é ligada (associada) à definição do método;

Binding

Considere a seguinte chamada de método:
`objeto.executar();`

- Em algum ponto, essa chamada é ligada (associada) à definição do método;
- Em Java, esta ligação é feita dinamicamente, ou seja, em tempo de execução;
- Por isto chamamos de *dynamic binding* (ligação dinâmica) essa ação.

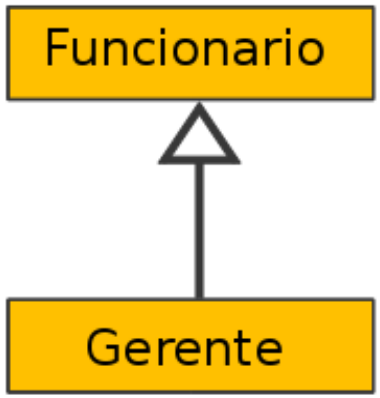
Referências

- Uma referência para um objeto pode apontar para um objeto de sua própria classe ou para um objeto relacionado a ele por herança;

Referências

- Uma referência para um objeto pode apontar para um objeto de sua própria classe ou para um objeto relacionado a ele por herança;
- Por exemplo, se a classe **Funcionario** é usada para derivar a classe **Gerente**, então uma referência de **Funcionario** pode apontar para um objeto **Gerente**:

```
Funcionario jorge;
jorge = new Gerente("Jorge", "00", 523.2, 1234);
```



O que é Polimorfismo?

- Polimorfismo é um conceito de orientação objeto que nos permite criar versáteis designs de softwares;

O que é Polimorfismo?

- Polimorfismo é um conceito de orientação objeto que nos permite criar versáteis designs de softwares;
- Uma referência polimórfica é uma variável que pode se referir a diferentes tipos de objetos em diferentes instantes;

O que é Polimorfismo?

- Polimorfismo é um conceito de orientação objeto que nos permite criar versáteis designs de softwares;
- Uma referência polimórfica é uma variável que pode se referir a diferentes tipos de objetos em diferentes instantes;
- O polimorfismo é usualmente caracterizado ao se atribuir a referência de um objeto de uma sub-classe à uma variável do tipo da super-classe desse objeto;

O que é Polimorfismo?

- Polimorfismo é um conceito de orientação objeto que nos permite criar versáteis designs de softwares;
- Uma referência polimórfica é uma variável que pode se referir a diferentes tipos de objetos em diferentes instantes;
- O polimorfismo é usualmente caracterizado ao se atribuir a referência de um objeto de uma sub-classe à uma variável do tipo da super-classe desse objeto;
- Um método invocado através de uma referência polimórfica pode mudar de chamada para chamada;

Polimorfismo via herança

- É o tipo do objeto que está sendo referenciado, não o tipo da referência, que determina qual método é chamado;

Polimorfismo via herança

- É o tipo do objeto que está sendo referenciado, não o tipo da referência, que determina qual método é chamado;
- A classe **Funcionario** tem um método chamado `getBonificacao`, e que a classe **Gerente** sobrescreve-o;

Polimorfismo via herança

- É o tipo do objeto que está sendo referenciado, não o tipo da referência, que determina qual método é chamado;
- A classe **Funcionario** tem um método chamado `getBonificacao`, e que a classe **Gerente** sobrescreve-o;
- Agora, considere a seguinte chamada:
  `trabalhador.getBonificacao();`

Polimorfismo via herança

- É o tipo do objeto que está sendo referenciado, não o tipo da referência, que determina qual método é chamado;
- A classe **Funcionario** tem um método chamado `getBonificacao`, e que a classe **Gerente** sobrescreve-o;
- Agora, considere a seguinte chamada: `trabalhador.getBonificacao();`
- Se *trabalhador* refere-se a um objeto **Funcionario**, então invoca-se a versão de `getBonificacao` de **Funcionario**; se *trabalhador* refere-se a um objeto **Gerente**, ele chama a versão de **Gerente**.

Casting

- *dynamic binding* garante que o método chamado será o método do objeto, porém o que acontece se a super-classe não tiver o método que queremos chamar?

Casting

- *dynamic binding* garante que o método chamado será o método do objeto, porém o que acontece se a super-classe não tiver o método que queremos chamar?
- Para invocar-se métodos de sub-classes que não existem na super-classe é necessário a atulização de casting.

```
Conta conta;
conta = new Poupanca("21.342-7");
((Poupanca) conta).renderJuros(0.01);
conta.imprimirSaldo();
```

instanceof

- `instanceof` é um operador (binário) de Java para verificação de tipos;
- Sintaxe: objeto instanceof classe;
- Retorna **true** quando `objeto` for do tipo `classe`.

```
Conta c = this.procurar("123.45-8");
if (c instanceof Poupanca)
    ((Poupanca) c).renderJuros(0.01);
else
    System.out.print("Poupança  inexistente!");
```

O que são exeções?

- Mecanismo utilizado por Java para tratamento de erros e situações excepcionais;

O que são exeções?

- Mecanismo utilizado por Java para tratamento de erros e situações excepcionais;
- Uma situação excepcional é um evento que ocorre durante a execução de um programa que interrompe o fluxo normal das instruções.

O que são exeções?

- Mecanismo utilizado por Java para tratamento de erros e situações excepcionais;
- Uma situação excepcional é um evento que ocorre durante a execução de um programa que interrompe o fluxo normal das instruções. Exemplos:
  - Erros de Programação:
    - Acesso a uma posição inválida de um array.
    - Chamada de método em uma referência nula.

O que são exeções?

- Mecanismo utilizado por Java para tratamento de erros e situações excepcionais;
- Uma situação excepcional é um evento que ocorre durante a execução de um programa que interrompe o fluxo normal das instruções. Exemplos:
  - Erros de Programação:
    - Acesso a uma posição inválida de um array.
    - Chamada de método em uma referência nula.
  - Situações Indesejadas:
    - Conexão com o servidor de banco de dados falhou.
    - Ausência de um arquivo.

Manipulando exeções

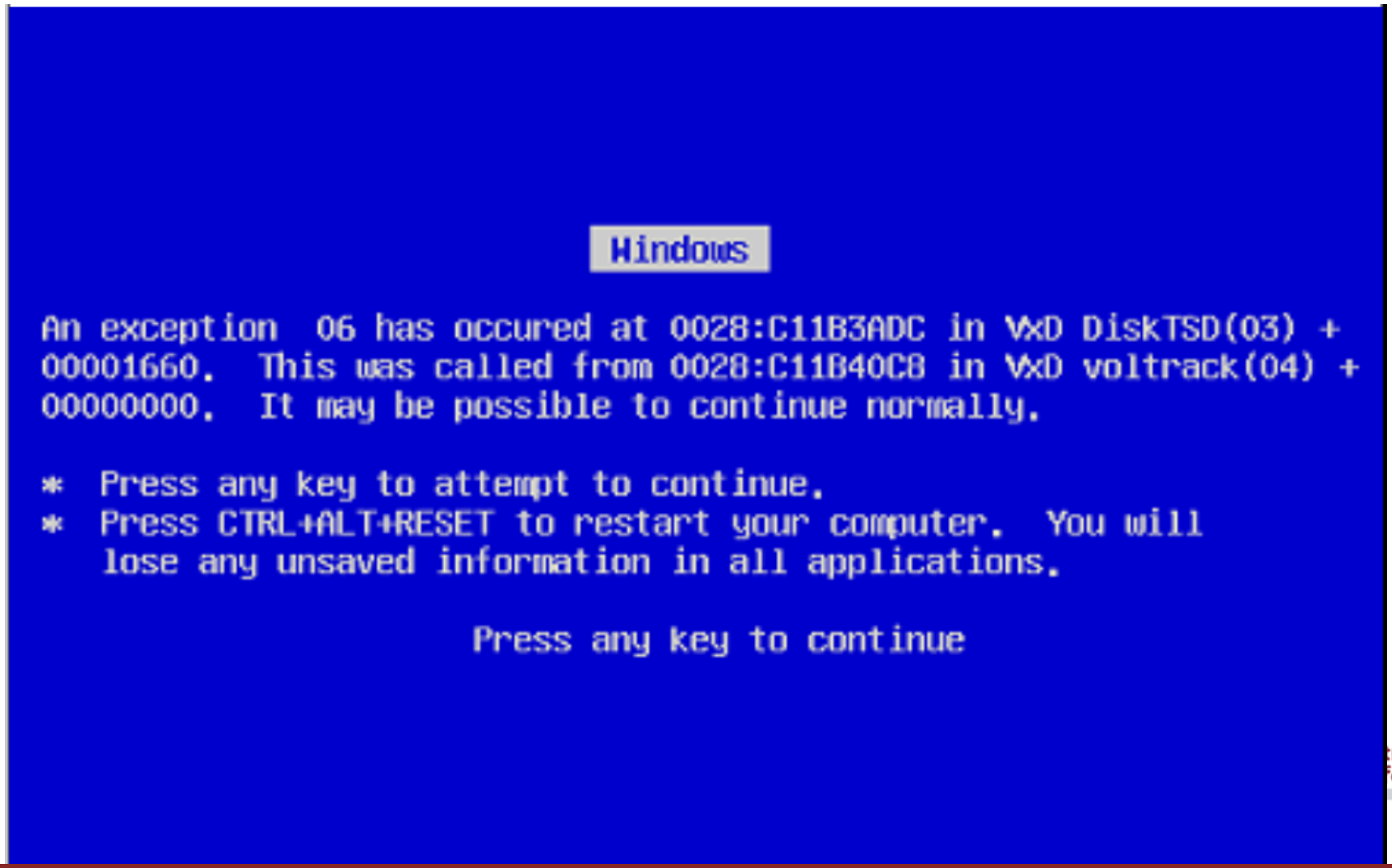
- Exeções também são classes em Java;

Manipulando exeções

- Exeções também são classes em Java;
- As exceções podem ser:
  - Declaradas
  - Lançadas
  - Tratadas

Como a Microsoft trata?

BSOD


Windows
An exception 06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000. It may be possible to continue normally.
* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue

Classes de Java

- Throwable
  - É a super classe de todas as exceções

Classes de Java

- Throwable
  - É a super classe de todas as exceções
- Error
  - Erros internos da máquina virtual

Classes de Java

- Throwable
  - É a super classe de todas as exceções
- Error
  - Erros internos da máquina virtual
- Exception
  - Devem ser declaradas e tratadas

Classes de Java

- Throwable
  - É a super classe de todas as exceções
- Error
  - Erros internos da máquina virtual
- Exception
  - Devem ser declaradas e tratadas
- RuntimeException
  - Erros de programação
  - Não precisam ser declaradas

Exceções Não Checadas

- Exceções do tipo **RuntimeException**;
- Qualquer método pode gerar essas exceções apesar de não explicitar isto em sua definição;

Exceções Não Checadas

- Exceções do tipo **RuntimeException**;
- Qualquer método pode gerar essas exceções apesar de não explicitar isto em sua definição;
- Tratar estas exceções é tentar corrigir um erro de programação durante a programação não faz muito sentido;

Exceções Não Checadas

- Exceções do tipo **RuntimeException**;
- Qualquer método pode gerar essas exceções apesar de não explicitar isto em sua definição;
- Tratar estas exceções é tentar corrigir um erro de programação durante a programação não faz muito sentido;
- O que se faz é capturar essa exceção e apresentar uma mensagem de erro agradável ao usuário indicando esta ocorrência.

Exceções Checadas

- Exceções do tipo **Exception**, exceto **RuntimeException** e suas subclasses;
- Devem ser declaradas e tratadas no código;

Exceções Checadas

- Exceções do tipo **Exception**, exceto **RuntimeException** e suas subclasses;
- Devem ser declaradas e tratadas no código;
- Exceções podem ser definidas pelo programador e devem ser subclasses de Exception;
  - Oferecem informações extra sobre o erro
  - Específicas para uma dada aplicação (exceções de negócio)

Exceções Checadas

- Exceções do tipo **Exception**, exceto **RuntimeException** e suas subclasses;
- Devem ser declaradas e tratadas no código;
- Exceções podem ser definidas pelo programador e devem ser subclasses de Exception;
  - Oferecem informações extra sobre o erro
  - Específicas para uma dada aplicação (exceções de negócio)
- Por convenção, é aconselhável que o nome de qualquer exceção definida pelo programador tenha o sufixo Exception:
  - SaldoInsuficienteException
  - ObjetoInvalidoException

Lançando uma exceção

- Um método que queira lançar uma exceção deverá ter duas coisas a mais:
  - Declarar no seu cabeçalho que pode lançar uma exceção (usando comando **throws**).
  - No corpor do método, ao detectar um erro, lançar a exceção (usando comando **throw**).

Lançando uma exceção

- Um método que queira lançar uma exceção deverá ter duas coisas a mais:
  - Declarar no seu cabeçalho que pode lançar uma exceção (usando comando **throws**).
  - No corpor do método, ao detectar um erro, lançar a exceção (usando comando **throw**).
- Um método que chama um outro método que pode lançar uma exceção PRECISA declarar no cabeçalho a possibilidade do lançamento, apesar de não ter o **throw** no seu corpo.

Tratando uma exceção

- Exceções são tratadas usando blocos **try-catch**.

Tratando uma exceção

- Exceções são tratadas usando blocos **try-catch**.
- A execução do **try** termina ao final do bloco ou assim que uma exceção é levantada;

Tratando uma exceção

- Exceções são tratadas usando blocos **try-catch**.
- A execução do **try** termina ao final do bloco ou assim que uma exceção é levantada;
- O primeiro catch de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último catch;

Tratando uma exceção

- Exceções são tratadas usando blocos **try-catch**.
- A execução do **try** termina ao final do bloco ou assim que uma exceção é levantada;
- O primeiro catch de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último catch;
- Exceções mais específicas devem ser capturadas primeiro. Caso contrário um erro de compilação é gerado.

Tratando uma exceção

- Exceções são tratadas usando blocos **try-catch**.
- A execução do **try** termina ao final do bloco ou assim que uma exceção é levantada;
- O primeiro catch de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último catch;
- Exceções mais específicas devem ser capturadas primeiro. Caso contrário um erro de compilação é gerado.
- **Se a exceção não for tratada em lugar nenhum, Java assume o controle e pára a aplicação**.

Declarando de uma exceção

```java
public class SaldoInsuficienteException extends Exception {
    private double saldo;
    private String numero;

    public SaldoInsuficienteException(double saldo, String numero) {
        super("Saldo Insuficiente!");
        this.saldo = saldo;
        this.numero = numero;
    }

    // ... Getters & Setters
}
```

Lançando de uma exceção

```
public class Conta {
    // ...

    public void debitar(double valor) throws SaldoInsuficienteException {
        if (valor <= saldo)
            saldo = saldo - valor;
        else
            throw new SaldoInsuficienteException(numero, saldo);
    }

    public void transferir(Conta c, double v) throws SaldoInsuficienteException {
        this.debitar(v);
        c.creditar(v);
    }
}
```

Tratando de uma exceção

```
public static void main(String args[]) {
    try {
        CadastroContas contas;
        // ...
        contas.debitar("123-0", 250.0);
        System.out.println("Débite efetuado");
    } catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
    } catch (ContaInexistenteException e) {
        System.out.println(e.getMessage());
    }
}
```

O que é interface?

- Uma interface em Java é uma coleção de métodos abstratos e constantes;

O que é interface?

- Uma interface em Java é uma coleção de métodos abstratos e constantes;
- Um método abstrato, como vimos antes, é apenas um cabeçalho de método, sem corpo (definição);

O que é interface?

- Uma interface em Java é uma coleção de métodos abstratos e constantes;
- Um método abstrato, como vimos antes, é apenas um cabeçalho de método, sem corpo (definição);
- Um método abstrato pode ser declarado usando o modificador `abstract`, mas como todos os métodos de uma interface são abstratos, normalmente não usa-se o modificador;

O que é interface?

- Uma interface em Java é uma coleção de métodos abstratos e constantes;
- Um método abstrato, como vimos antes, é apenas um cabeçalho de método, sem corpo (definição);
- Um método abstrato pode ser declarado usando o modificador `abstract`, mas como todos os métodos de uma interface são abstratos, normalmente não usa-se o modificador;
- Uma interface é usada para estabelecer um conjunto de métodos que uma classe irá implementar;

Exemplo

```java
public interface AreaCalculavel{
    public double calcularArea();
}
```

```java
public class Quadrado implements AreaCalculavel{
    private double lado;

    public Quadrado(double lado){
        this.lado = lado;
    }

    public double calcularArea(){
        return (this.lado * this.lado);
    }
}
```

```java
public class Circulo implements AreaCalculavel{
    private double raio;

    public Circulo(double raio){
        this.raio = raio;
    }

    public double calcularArea(){
        return (Math.PI * this.raio * this.raio);
    }
}
```

Mais algumas palavras sobre Interface

- Uma interface não pode ser instanciada;

Mais algumas palavras sobre Interface

- Uma interface não pode ser instanciada;
- Métodos em uma interface têm que ser `public` "por *default*";

Mais algumas palavras sobre Interface

- Uma interface não pode ser instanciada;
- Métodos em uma interface têm que ser `public` "por *default*";
- Uma classe implementa uma interface:
  - Declarando isto no cabeçalho da classe atraves da palavra reserveda `implements`;
  - Implementando cada um dos métodos abstratos da interface.

Mais algumas palavras sobre Interface

- Uma interface não pode ser instanciada;
- Métodos em uma interface têm que ser `public` "por *default*";
- Uma classe implementa uma interface:
  - Declarando isto no cabeçalho da classe atraves da palavra reserveda `implements`;
  - Implementando cada um dos métodos abstratos da interface.
- Uma classe que implementa uma interface deve definir **TODOS** os métodos da interface;

... mais algumas outras

- Uma classe que implementa uma interface pode definir outros métodos também;

... mais algumas outras

- Uma classe que implementa uma interface pode definir outros métodos também;
- Além de métodos abstratos, interfaces podem conter constantes;

... mais algumas outras

- Uma classe que implementa uma interface pode definir outros métodos também;
- Além de métodos abstratos, interfaces podem conter constantes;
- Quando uma classe implementa uma interface, ela ganha acesso a todas essas constantes;

Implementando mais de uma interface

- Uma classe pode implementar múltiplas interfaces;

Implementando mais de uma interface

- Uma classe pode implementar múltiplas interfaces;
- As interfaces são listadas após o `implements`;

Implementando mais de uma interface

- Uma classe pode implementar múltiplas interfaces;
- As interfaces são listadas após o `implements`;
- A classe deve implementar todos os métodos de todas as interfaces listadas.

Implementando mais de uma interface

- Uma classe pode implementar múltiplas interfaces;
- As interfaces são listadas após o `implements`;
- A classe deve implementar todos os métodos de todas as interfaces listadas.

```
public class Quadrado implements AreaCalculavel, PerimetroCalculavel{
    private double lado;

    public double calcularArea(){
        return (this.lado * this.lado);
    }

    public double calcularPerimetro(){
        return (4 * this.lado);
    }
}
```

Interfaces de Java

- A biblioteca padrão de Java contem uma série de interfaces úteis;

Interfaces de Java

- A biblioteca padrão de Java contem uma série de interfaces úteis;
- A interface **Comparable** contém um método abstrato chamado `compareTo`, que é usado para comparar dois objetos;

Interfaces de Java

- A biblioteca padrão de Java contem uma série de interfaces úteis;
- A interface **Comparable** contém um método abstrato chamado `compareTo`, que é usado para comparar dois objetos;
- A classe String implementa **Comparable**, dando a ela a habilidade de colocar strings em ordem lexicográfica;

Interfaces de Java

- A biblioteca padrão de Java contem uma série de interfaces úteis;
- A interface **Comparable** contém um método abstrato chamado `compareTo`, que é usado para comparar dois objetos;
- A classe String implementa **Comparable**, dando a ela a habilidade de colocar strings em ordem lexicográfica;
- Olhem a API de Java para mais detalhes... Dêem uma olhada na interface **Iterator** e na interface **Comparable**, pois sera útil no projeto. Qualquer dúvida procurem seus monitores.

Bibliografia

- Apostila Caelum.
- Apresentações de Igor Ebrahim.
- Apresentações de Guilherme Carvalho.