Modulo Ib
Introdução ao Java

UniverCidade - Prof. Ismael H F Santos

Ementa

- Modulo Ib - Introdução a Linguagem JAVA
  - Notação UML
  - Pacotes
  - Classes Internas Aninhadas
  - Tratamento de Exceção
  - Asserções

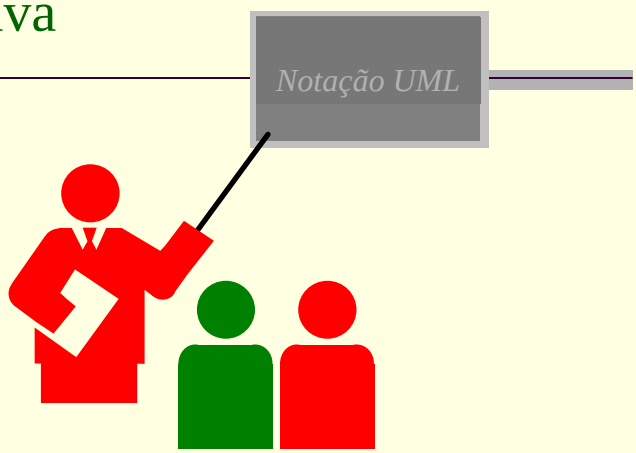
Bibliografia

- *Linguagem de Programação JAVA*
  - *Ismael H. F. Santos, Apostila UniverCidade, 2002*
- *The Java Tutorial: A practical guide for programmers*
  - *Tutorial on-line: http://java.sun.com/docs/books/tutorial*
- *Java in a Nutshell*
  - *David Flanagan, O´Reilly & Associates*
- *Just Java 2*
  - *Mark C. Chan, Steven W. Griffith e Anthony F. Iasi, Makron Books.*
- *Java 1.2*
  - *Laura Lemay & Rogers Cadenhead, Editora Campos*

Livros

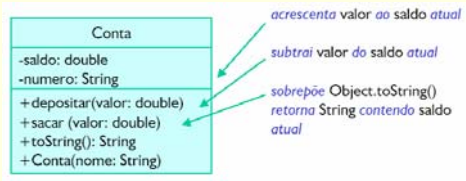
- **Core Java 2**, Cay S. Horstmann, Gary Cornell
  - Volume 1 (Fundamentos)
  - Volume 2 (Características Avançadas)
- **Java: Como Programar**, Deitel & Deitel
- **Thinking in Patterns with JAVA**, Bruce Eckel
  - **Gratuito**. http://www.mindview.net/Books/TIJ/

POO-Java

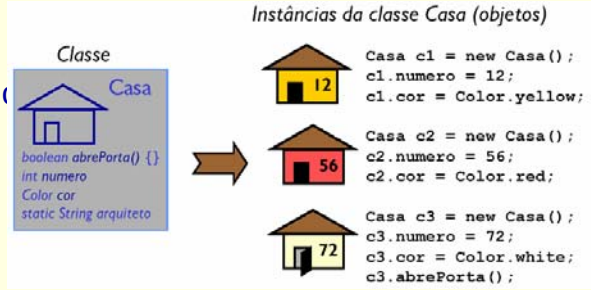


Notação UML

- Classes são uma especificação para objetos, representa um tipo de dados complexo.

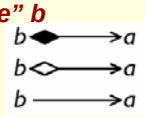


- Objetos são instancias da

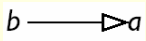


Notação UML (cont.)

- Separação interface-imlementação: permite um maior reuso
  - ***reuso depende de bom planejamento e design***
- Uma vez criada uma classe, ela deve representar uma unidade de código útil para que seja reutilizável
- Formas de uso e reuso
  - ***Uso e reuso de** objetos **criados pela classe: mais flexível***
    - ***Composição: a "é parte essencial de" b***
    - ***Agregação: a "é parte de" b***
    - ***Associação: a "é usado por" b***

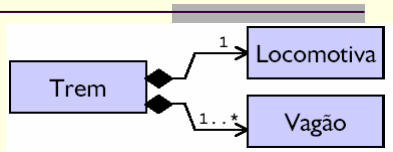


- Reuso da interface da classe: pouco flexível
  - ***Herança: b "é" a (substituição pura) ou b "é um tipo de" a (substituição útil, extensão)***

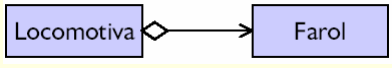


Notação UML (cont.)

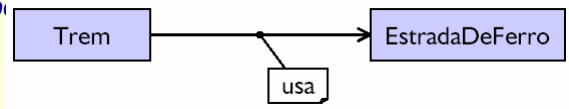
- *Composição*: ***um trem é formado por locomotiva e vagões***



- *Agregação*: ***uma locomotiva tem um farol (mas não vai deixar de ser uma locomotiva se não o tiver)***

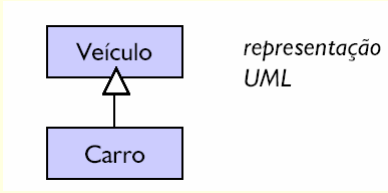


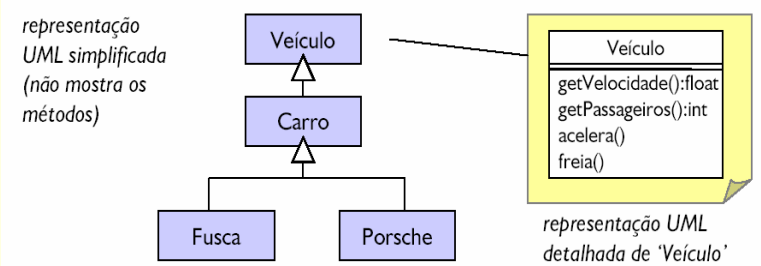
- *Associação*: ***um trem usa uma estrada de ferro (não faz parte do trem, mas ele dep***



Notação UML (cont.)

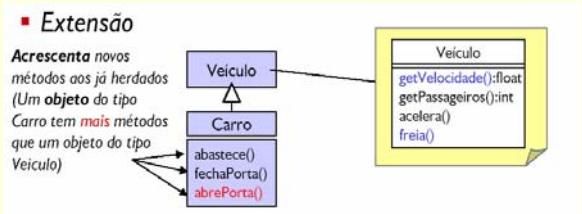
- Herança
  - Um carro é um veiculo
  - Fuscas e Porsches são carros (e também são veículos)



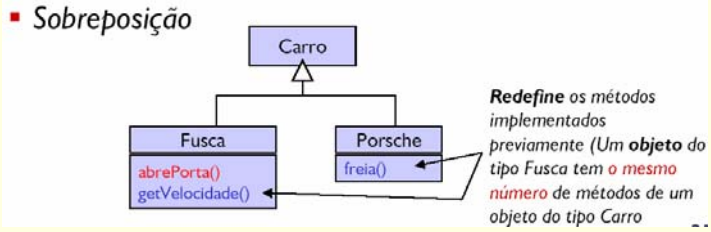


Notação UML (cont.)

- Extensão

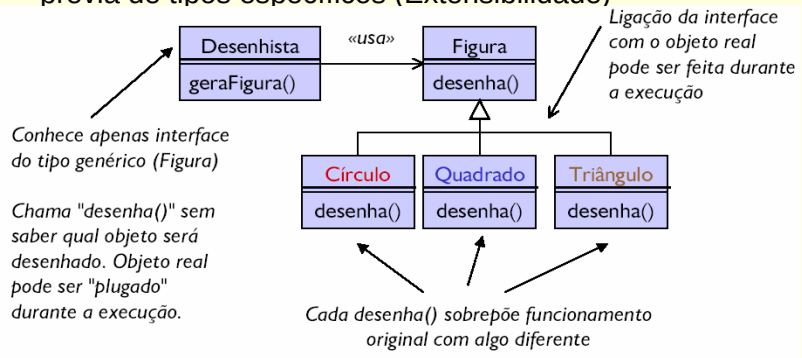


- Sobreposição



Notação UML (cont.)

- Polimorfismo
  - Uso de um objeto no lugar de outro. Dessa forma pode-se escrever código que não dependa da existência previa de tipos específicos (Extensibilidade)



Encapsulamento

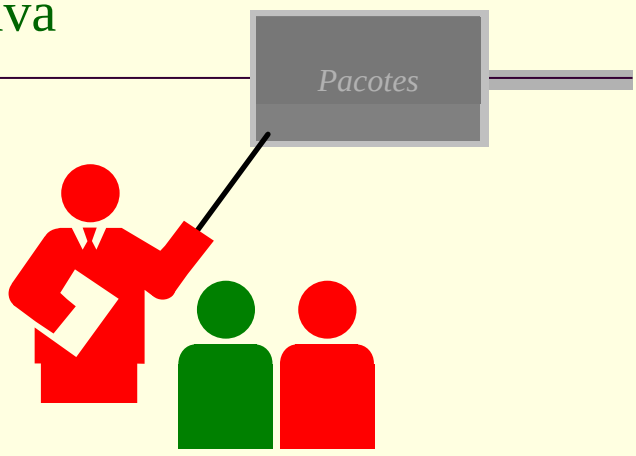
- *Simplifica o objeto expondo apenas a sua interface essencial*
- *Código dentro de métodos é naturalmente encapsulado*
  - *Não é possível acessar interior de um método fora do objeto*
- *Métodos que não devem ser usados externamente e atributos podem ter seu nível de acesso controlado em Java, através de modificadores*
  - ***private**: apenas acesso dentro da classe*
  - ***package-private** (default): acesso dentro do pacote**
  - ***protected**: acesso em subclasses*
  - ***public**: acesso global*

** não existe um modificador com este nome. A ausência de um modificador de acesso deixa o membro com acesso package-private*

Sumário OO

- *Abstração de conceitos*
  - *Classes, definem um tipo separando interface de implementação*
  - *Objetos: instâncias utilizáveis de uma classe*
- *Herança: "é um"*
  - *Aproveitamento do código na formação de hierarquias de classes*
  - *Fixada na compilação (inflexível)*
- *Associação "tem um"*
  - *Consiste na delegação de operações a outros objetos*
  - *Pode ter comportamento e estrutura alterados durante execução*
  - *Vários níveis de acoplamento: associação, composição, agregação*
- *Encapsulamento*
  - *Separação de interface e implementação que permite que usuários de objetos possam utilizá-los sem conhecer detalhes de seu código*
- *Polimorfismo*
  - *Permite que objeto seja usado no lugar de outro*

POO-Java



Fundamentos da Linguagem

- Modularidade em Java: Pacotes
  - Além das classes, Java provê um recurso adicional que ajuda a modularidade: o uso de pacotes.
  - Um pacote é um conjunto de classes e outros pacotes.
  - Pacotes permitem a criação de espaços de nomes, além de mecanismos de controle de acesso.

- Pacotes: Espaço de Nomes
  - Pacotes, a princípio, possuem nomes. O nome do pacote qualifica os nomes de todas as classes e outros pacotes que o compõem.

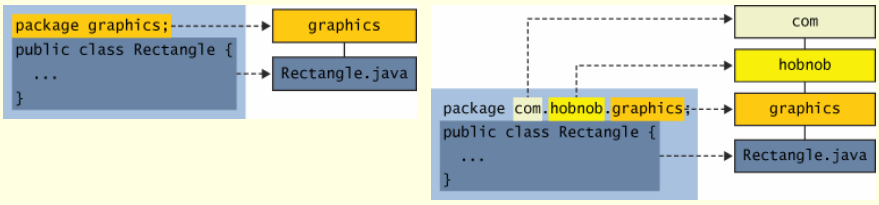
  Exemplo: classe Math no pacote java.lang

  ```
  int a = java.lang.Math.abs(-10); // a = 10;
  ```


Fundamentos da Linguagem

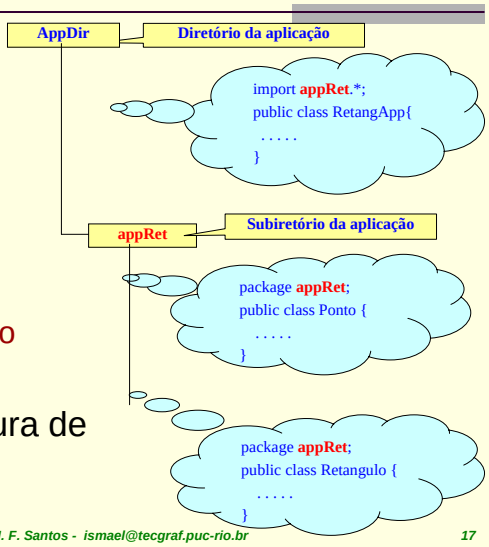
- Pacotes: Espaços de Nomes
  - Exemplo: classe Rectangle no pacote graphics

```
graphics.Rectangle r = new
  graphics.Rectangle();
 ...... OU ......
import graphics.*;
Rectangle r = new Rectangle();
```



Fundamentos da Linguagem

- Exemplo
  - Aplicação
    - RetangApp
  - Pacote appRet
    - Classe Ponto
    - Classe Retangulo
  - Observe a estrutura de diretórios.



Fundamentos da Linguagem

- Exemplo – Aplicação RetangApp

```
import appRet.*;  // importando pacote appRet

public class RetangApp {
  public static void main(String[] args) {
    Retangulo rect1=new Retangulo(100, 200);
    Retangulo rect2=new Retangulo(new Ponto(44, 70));
    Retangulo rect3=new Retangulo();
    rect1.deslocar(40, 20);
    System.out.println("A área é " +rect1.calcularArea());
    int areaRect=new Retangulo(100, 50).calcularArea();
    System.out.println("A área é " + areaRect);
    }
  }
}
```

Fundamentos da Linguagem

- Exemplo – Classe Ponto

```
package appRet;   // pacote

public class Ponto {
   int x;  int y;
   public Ponto() {      // construtor padrão
     x=0;   y=0;
   }
   public Ponto(int x, int y) {
    this.x = x;  this.y = y;
   }
   void deslocar(int dx, int dy){
     x+=dx;  y+=dy;
   }
}
```


Fundamentos da Linguagem

- Exemplo – Classe Retangulo

```
package appRet;            // declaracao pacote
public class Retangulo {
   Ponto origem;   int largura ;    int altura ;
   public Retangulo() {  origem = new Ponto(0, 0);   largura=0;   altura=0;   }
   public Retangulo(Ponto p)    {  this(p, 0, 0);   }
   public Retangulo(int w, int h) {  this(new Ponto(0, 0), w, h);   }
   public Retangulo(Ponto p, int w, int h) {  origem=p;  largura=w;  altura=h; }
   void deslocar(int dx, int dy) {
       origem.deslocar(dx, dy);
   }
   int calcularArea() {
        return largura * altura;
   }
}
```

Fundamentos da Linguagem

Pacotes: Controle de Acesso

- Além de membros públicos e privados, temos também membros package.
- Um membro package só pode ser acessado por classes declaradas no mesmo pacote da classe que declara esse membro.
- Quando omitimos o modificador de controle de acesso, estamos dizendo que o membro é do tipo package.

Fundamentos da Linguagem

Tipos de Visibilidade

Os membros que vínhamos declarando eram do tipo package pois sempre omitimos o modificador de controle de acesso.

```
class A {
  private int i;
          int j;
  public  int k;
}
```

```
public class B {

   ...
}
```

Fundamentos da Linguagem

- Implementação de Pacotes
  - Pacotes são tipicamente implementados como diretórios.
  - Os arquivos das classes pertencentes ao pacote devem ficar em seu diretório.
  - Hierarquias de pacotes são construídas através de hierarquias de diretórios.

Fundamentos da Linguagem

- "Empacotando" uma Classe
  - Para declararmos uma classe como pertencente a um pacote, devemos:
    - declará-la em um arquivo dentro do diretório que representa o pacote;
    - declarar, na primeira linha do arquivo, que a classe pertence ao pacote – declaração package.

Fundamentos da Linguagem

- Importação de Pacotes
  - Podemos usar o nome simples (não qualificado) de uma classe que pertença a um pacote se importarmos a classe.
  - A importação de uma classe (ou classes de um pacote) pode ser feita no início do arquivo, após a declaração do pacote (se houver) – declaração import
  - As classes do pacote padrão java.lang não precisam ser importadas (Ex.: Math).

Fundamentos da Linguagem

- Exemplo de Arquivo

```
package datatypes;  //Stack pertence a datatypes.
import java.math.*; //Importa todas as classes.
import java.util.HashTable; // Importa HashTable.

/*A partir desse ponto, posso usar o nome HashTable diretamente, ao invés
de usar  java.util.HashTable. Assim como posso usar  diretamente o nome de
qualquer classe que pertença ao pacote java.math.*/

public class Stack { // Stack é exportada.
   ...
}
```

Exercícios – Questões 7 e 8

Fundamentos da Linguagem

- CLASSPATH
  - Propriedade do sistema que contém as localidades onde o JRE irá procurar classes para execução. Consiste de:
    - JARs nativos do JRE (API Java 2)
    - Extensões do JRE (subdiretórios $JAVA_HOME/jre/lib/classes e $JAVA_HOME/jre/lib/ext)
    - Lista de caminhos definidos na variável de ambiente CLASSPATH e/ou na opção de linha de comando -classpath (-cp) da aplicação java.
  - A ordem acima é importante
    - Havendo mais de uma classe com mesmo pacote/Nome somente a primeira classe encontrada é usada enquanto as outras serão ignoradas

Fundamentos da Linguagem

- CLASSPATH
  - A ordem anterior é importante (cont.)
    - Há risco de conflitos. API nova sendo carregada depois de antiga pode resultar em classes novas chamando classes antigas!
    - A ordem dos caminhos na variável CLASSPATH (ou opção -cp) também é significativa.
  - Em uma instalação típica, CLASSPATH contém apenas "."
    - Pacotes iniciados no diretório atual (onde o interpretador java é executado) são encontrados (podem ter suas classes importadas)
    - Classes localizadas no diretório atual são encontradas.
  - Geralmente usada para definir caminhos para uma aplicação
    - Os caminhos podem ser diretórios, arquivos ZIP ou JARs
    - Pode acrescentar novos caminhos mas não pode remover caminhos do Classpath do JRE (básico e extensões)

Fundamentos da Linguagem

- CLASSPATH (cont.)
  - A opção -cp (-classpath) substitui as definições em CLASSPATH
  - Exemplo de definição de CLASSPATH
    - no DOS/Windows
      - set CLASSPATH=extras.jar;.;c:\progs\java
      - java - cp %CLASSPATH%;c:\util\lib\jsw.zip gui.Programa
    - no Unix (sh, bash)
      - CLASSPATH=extras.jar:.:/home/mydir/java
      - export CLASSPATH
      - java -classpath importante.jar:$CLASSPATH gui.Programa
  - Colocar JARs no subdiretório ext ou classes e pacotes no diretório classes automaticamente os inclui no CLASSPATH para todas as aplicações da JVM
    - Carregados antes das variáveis CLASSPATH e –cp
    - Evite usar: pode provocar conflitos. Coloque nesses diretórios apenas os JARs e classes usados em todas suas aplicações

Fundamentos da Linguagem

- CLASSPATH do JRE
  - *Exemplo: suponha que o CLASSPATH seja*
    - *Classpath JRE:* %JAVA_HOME%\jre\lib\rt.jar;
    - *Classpath Extensão JRE:* %JAVA_HOME%\jre\lib\ext\z.jar
    - *Variável de ambiente CLASSPATH:* .;c:\programas;
    - *E que uma classe, localizada em c:\exercicio seja executada. Se esta classe usar a classe arte.fr.Monet o sistema irá procurá-la em*
      - %JAVA_HOME%\jre\lib\rt.jar\arte\fr\Monet.class
      - %JAVA_HOME%\jre\lib\ext\z.jar\arte\fr\Monet.class
      - c:\exercício\arte\fr\Monet.class
      - c:\programas\arte\fr\Monet.class

Fundamentos da Linguagem

- Arquivos JAR
  - Aplicações Java são distribuídas em arquivos JAR
    - São extensões do formato ZIP
    - armazenam pacotes e preservam a hierarquia de diretórios
  - Para usar um JAR, é preciso incluí-lo no Classpath
    - via CLASSPATH no contexto de execução da aplicação, ou
    - via parâmetro -classpath (-cp) do interpretador Java, ou
    - copiando-o para $JAVA_HOME/jre/lib/ext
  - Para criar um JAR
    - jar cvf classes.jar C1.class C2.class xyz.gif abc.html
    - jar cf classes.jar –C raiz_onde_estao_pacotes/ .
  - Para abrir um JAR
    - jar xvf classes.jar
  - Para listar o conteúdo de um JAR
    - jar tvf classes.jar

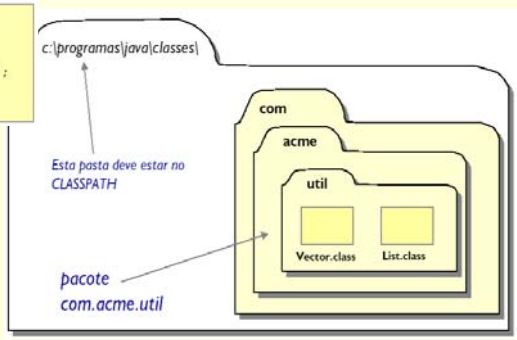
Exemplo – classes de pacotes

List.java

```
package com.acme.util;
public class List {
  public List() {
    System.out.println("com.acme.util.List");
  }
}
```

Vector.java

```
package com.acme.util;
public class Vector {
  public Vector() {
    System.out.println("com.acme.util.Vector");
  }
}
```

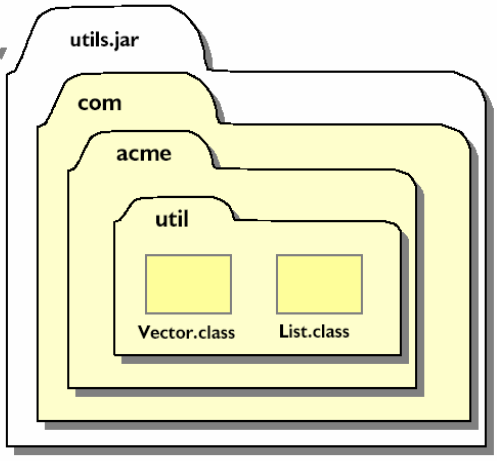


Exemplo – arquivo jar

O caminho até a localização do JAR deve estar no CLASSPATH (pode ser especificado no momento da execução:

```
java -classpath
  %CLASSPATH%;utils.jar
  NomeProgExecutavel
```

JAR também pode ser jogado no diretório ext do JRE



Exemplo – usando arquivo jar

- Programas que usem a biblioteca podem estar em qualquer lugar, basta que caminho até a biblioteca (classes ou JAR) esteja definido no CLASSPATH
  - CLASSPATH=%CLASSPATH%;c:\programas\java\classes
  - CLASSPATH=%CLASSPATH%;c:\jars\utils.jar
  - java –cp %CLASSPATH%; c:\programas\java\classes LibTest
  - java –cp %CLASSPATH%; c:\jars\utils.jar LibTest

```
// Programa que usa a biblioteca
import com.acme.util.*;
public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List m = new List();
  }
}
```

Criando jar executável

- Todo JAR possui um arquivo chamado Manifest.mf no subdiretório /META-INF.
  - Lista de pares Nome: atributo
  - Serve para incluir informações sobre os arquivos do JAR, CLASSPATH, classe Main, etc.
  - Se não for indicado um arquivo específico, o sistema gerará um Manifest.mf default (vazio)
- Para um JAR executável, o manifest.mf deve conter a linha:
  - Main-Class: nome.da.Classe
- Crie um arquivo de texto qualquer com a linha acima e monte o JAR usando a opção -manifest:
  - jar cvfm arq.jar manifest.mf –C raiz .
- Para executar o JAR em linha de comando:
  - java -jar arq.jar

Fundamentos da Linguagem

- Visibilidade & Herança

Membros **públicos** são herdados, enquanto membros privados não são. Às vezes precisamos algo intermediário: um membro que não seja visto fora da classe mas que possa ser herdado. As linguagens OO tipicamente dão suporte a esse tipo de acesso.

Java permite declararmos um membro que, embora não seja acessível por outras classes, é herdado por suas sub-classes. Para isso usamos o modificador de controle de acesso protected.

Fundamentos da Linguagem

- Resumo de Visibilidade em Java

private: membros que são vistos só pelo própria classe e não são herdados por nenhuma outra;

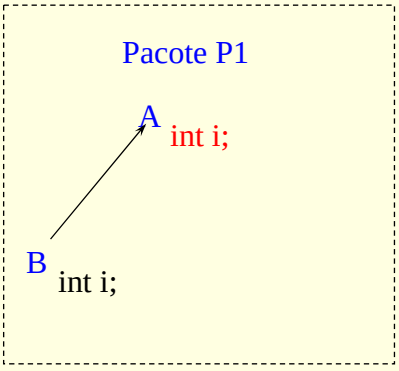
package: membros que são vistos e herdados pelas classes do pacote;

protected: membros que são vistos pelas classes do pacote e herdados por qualquer outra classe;

public: membros são vistos e herdados por qualquer classe.

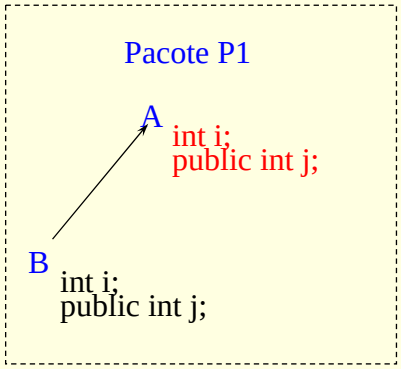
Fundamentos da Linguagem

- Herança de Membros



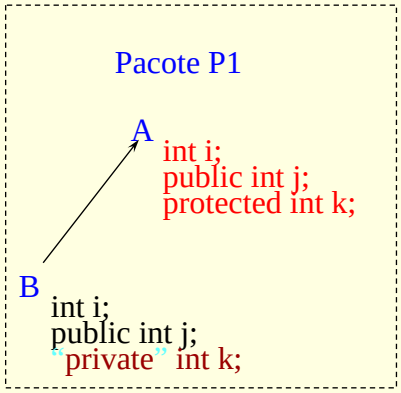
Fundamentos da Linguagem

- Herança de Membros



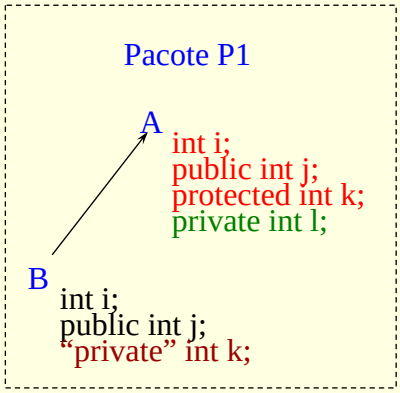
Fundamentos da Linguagem

- Herança de Membros



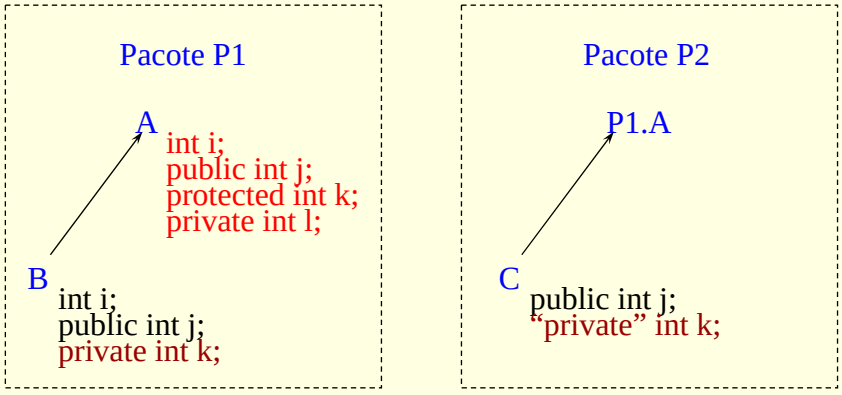
Fundamentos da Linguagem

- Herança de Membros



Fundamentos da Linguagem

- Herança de Membros entre Pacotes



Fundamentos da Linguagem

Situação	public	package (default)	protected	private
Acessível a outras classes dentro de um pacote	*Sim*	*Sim*	*Sim*	*Não*
Acessível a subclasse dentro de um mesmo pacote	*Sim*	*Sim*	*Sim*	*Não*
Acessível a outras classes dentro de outro pacote	*Sim*	*Não*	*Não*	*Não*
Acessível a subclasse dentro de outro pacote	*Sim*	*Não*	*Não*	*Não*
Herdado pela subclasse dentro de um mesmo pacote	*Sim*	*Sim*	*Sim*	*Não*
Herdado pela subclasse dentro de outro pacote	*Sim*	*Não*	*Sim*	*Não*

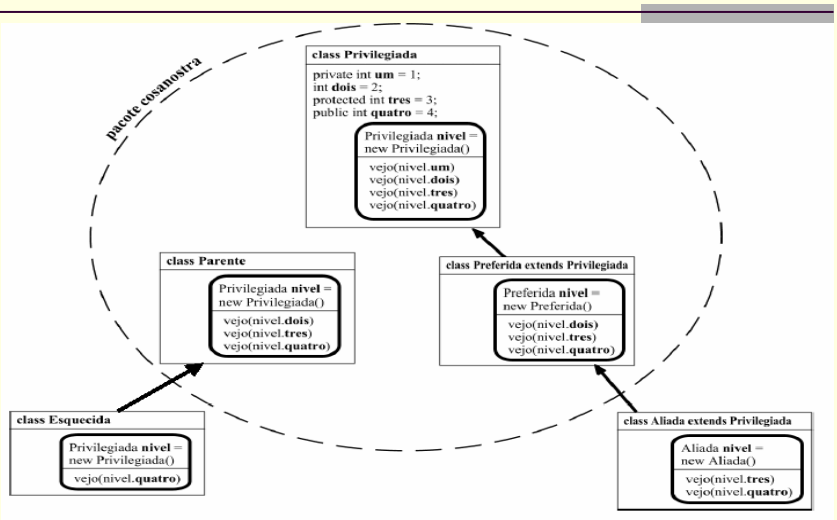
Observações sobre acesso

- *Classes e interfaces (exceto classes internas)*
  - *Só podem ser package-private ou public*
- *Construtores*
  - *Se private, criação de objetos depende da classe*
  - *Se protected, apenas subclasses (além da própria classe e classes do pacote) podem criar objetos*
- *Váriáveis e constantes*
  - *O acesso afeta sempre a leitura e alteração. Efeitos "read-only" e "write-only" só podem ser obtidos por meio de métodos*
- *Variáveis locais*
  - *Usar modificadores de acesso dentro dos métodos é ilegal e não faz sentido pois variáveis locais só têm escopo local*

Sobreposição (Override)

- *Métodos sobrepostos nunca podem ter* ***menos*** *acesso que os métodos originais*
  - *Se método original for* ***public****, novas versões têm que ser public*
  - *Se método original for* ***protected****, novas versões podem ser protected ou public*
  - *Se método original* ***não tiver modificador de acesso*** *(é "package-private"), novas versões podem ser declaradas sem modificador de acesso, com modificador protected ou public*
  - *Se método original for* ***private****, ele não será visível da subclasse e portanto, jamais poderá ser estendido.*

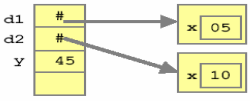
Exemplo



Classes e Métodos static

Mais sobre 'static'

- *Variáveis declaradas como 'static' existem antes de existir qualquer objeto da classe*
  - *Só existe uma variável static, independente do número de objetos criado com a classe*
  - *Podem ser chamadas externamente pelo nome da classe*
    `Color.red, System.out, BorderLayout.NORTH`
  - *Podem também ser chamadas através da referência de um objeto (evite usar este método)*




```
class Duas {
   int x;
   static int y;
}
```

```
(...)
Duas d1 = new Duas();
Duas d2 = new Duas();
d1.x = 5;
d2.x = 10;
//Duas.x = 60; // ilegal!
d1.y = 15;
d2.y = 30;
Duas.y = 45;
(...)
```

mesma variável!

Classe
campoStatic: tipo
metodoStatic: tipo

Use esta notação apenas

Classes e Métodos static

- *Métodos static **nunca** são sobrepostos*
  - *Método static de assinatura igual na subclasse apenas "**oculta**" original*
  - *Não há polimorfismo: método está sempre associado ao tipo da **classe** (e não à instância)*
- *Exemplo: considere as classes abaixo*

```
class Alfa {
  static void metodo() {
    System.out.println("Alfa!");
  }
}
```

```
class Beta extends Alfa {
  static void metodo() {
    System.out.println("Beta!");
  }
}
```

- *O código a seguir*

```
Alfa pai = new Alfa ();
pai.metodo();

Beta filho1 = new Beta ();
filho1.metodo();

Alfa filho2 = new Beta ();
filho2.metodo();
```

irá imprimir:

Alfa!
Beta!
Alfa!

e não...

Alfa!
Beta!
Beta!

*como ocorreria **se** os métodos fossem de instância*

- *Não chame métodos static via referências! Use sempre:*
  `Classe.metodo()`

Classe com apenas 1 instância – Design Pattern Singleton

```
public class Highlander {
  private Highlander() {}
  private static Highlander instancia;
  public static Highlander criarInstancia() {
    if (instancia == null)
      instancia = new Highlander();
    }
    return instancia;
  }
}
```

Esta classe implementa o padrão de projeto Singleton

```
public class Fabrica {
  public static void main(String[] args) {
    Highlander h1, h2, h3;
    //h1 = new Highlander(); // nao compila!
    h2 = Highlander.criarInstancia();
    h3 = Highlander.criarInstancia();
    if (h2 == h3) {
       System.out.println("h2 e h3 são mesmo objeto!");
    }
  }
}
```

Esta classe cria apenas um objeto Highlander

POO-Java

Classes Internas Aninhadas

Classes Aninhadas

- Motivação
  - Em diversas circunstâncias precisamos criar classes cujo único objetivo é auxiliar na implementação de uma outra classe. Nesses casos, podemos declarar uma classe aninhada, ou seja, declarar uma nova classe como um membro de uma outra. Em Java o seu uso maior é no tratamento de eventos onde uma classe aninhada pode ser criada para ser o “event handler” para a classe mãe.

Classes Aninhadas

- Motivação (cont)
  - Java permite dois tipos diferentes de aninhamento de tipos:
    - Aninhamento estático;
      - Classes internas estáticas (membros de classe);
    - Aninhamento dinâmico;
      - Classes dentro de objetos (membros de instancia);
      - Classes dentro de instruções (classes anônimas);
      - Classes dentro de métodos (classes locais);

Classes Aninhadas

- Aninhamento Estático
  - Gera classes e interfaces normais, cuja única singularidade é o nome, que passa a ser qualificado pelo nome da classe que as declara.
  - Em particular, sendo um membro de uma classe, uma interface ou classe aninhada está sujeita aos modificadores de controle de acesso: public, private, protected e *package.*

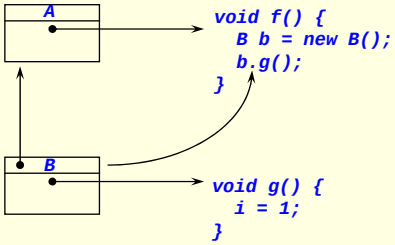
```
package p;
public class A {
  public int i;  public static int j;
  public static class B { ...  } // classe interna estatica, so
                                 // pode acessar vars e métodos estáticos
}
```

```
p.A a = new p.A();  p.A.B b = new p.A.B();
b.j = 1; // OK !;   b.i = 2; // erro !
```


Classes Aninhadas

- Aninhamento Dinâmico
  - Gera classes associadas a objetos, instruções ou métodos;
  - Cada instância da classe aninhada possui uma referência para o objeto a partir do qual ela é criada;
  - Como ela está associada a um objeto, ela tem acesso a todos os membros desse objeto.

```
class A {
  int i;
  class B { // classe interna objeto!
    void g() {...}
  }
  void f() {...}
}
.....
A a = new A();
B c = a.new B(); // depende de a !
```



Classe Interna dentro de objetos

- São membros do objeto, como métodos e campos. Requerem que objeto exista antes que possam ser usadas.
  - Externamente use referencia.new para criar objetos
  - Variáveis de mesmo nome sempre se referem à classe interna
  - Use Externa.this.xxx para acessar campos externos

```
class Externa {
   public int campoUm;
   private class Interna {
      public int campoUm;
      public int campoDois;
      public void metodoInterno() {
         this.campoUm = 10; // Externa.campoUm
         Interna.this.campoUm = 15;
      }
   }
   public static void main(String[] args){
      Interna e = (new Externa()).new Interna();
   }
}
```

Exemplos

Classe Interna dentro de métodos

- Servem para tarefas "descartáveis" já que deixam de existir quando o método acaba
  - Têm o escopo de variáveis locais. Objetos criados, porém, podem persistir além do escopo do método, se retornados
  - Se usa variáveis locais do método essas variáveis devem ser constantes (final), pois assim podem persistir após a conclusão do método. O compilador cria uma cópia “escondida” dessas variáveis para poderem ser acessadas com segurança no futuro.

```
public Multiplicavel calcular(final int a, final int b) {
   class Interna implements Multiplicavel {
      public int produto() {
         return a * b; // usa a e b, que são constantes
      }
   }
   return new Interna();
}
public static void main(String[] args){
   Multiplicavel mul = (new Externa()).calcular(3,4);
   int prod = mul.produto();
}
```

Classe Interna Anônimas - dentro de instruções

- Classes usadas dentro de métodos freqüentemente servem apenas para criar um objeto uma única vez
  - **A classe abaixo estende ou implementa SuperClasse, que pode ser uma interface ou classe abstrata (o new, neste caso, indica a criação da classe entre chaves, não da SuperClasse)**
    Object i = new SuperClasse() { implementação };
  - **Compilador gera arquivo** Externa$1.class, Externa$2.class,

```
public Multiplicavel calcular(final int a, final int b) {
   return new Multiplicavel()  {
      public int produto() {
         return a * b;
      }
   };
}
public static void main(String[] args){
   Multiplicavel mul = (new Externa()).calcular(3,4);
   int prod = mul.produto();
}
```

Compare com parte em preto e vermelho do slide anterior!

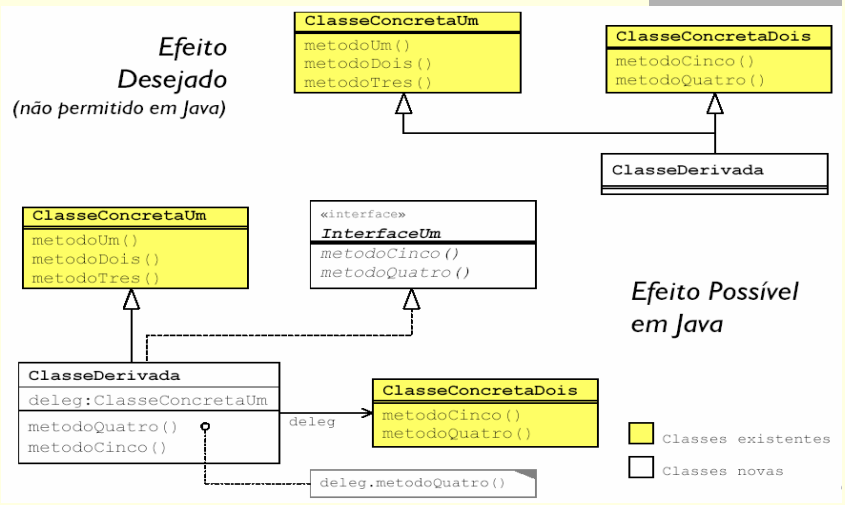
A classe está dentro da instrução: preste atenção no ponto-e-vírgula!

Por que usar classes internas ?

- Mais reutilização
  - Recurso poderoso quando combinado com interfaces e herança - facilita implementação de delegação: tipo de herança de implementação que combinando composição com herança de interfaces simula herança múltipla;
  - "Ponteiros seguros" apontando para métodos localizados em classes internas;
  - Flexibilidade para desenvolver objetos descartáveis
- Riscos
  - Aumenta significativamente a complexidade do código
  - Dificulta o trabalho de depuração (erros de compilador são mais confusos em classes internas)

Exercícios – Questão 9

Simulando Herança múltipla com delegação + agregação



POO-Java

Tratamento de Exceção

- Terminologia
  - Exceção é a ocorrência de uma condição anormal durante a execução de um método;
  - Falha é a inabilidade de um método cumprir a sua função;
  - Erro é a presença de um método que não satisfaz sua especificação.

Em geral a existência de um erro gera uma falha que resulta em uma exceção !

Tratamento de Exceção

- Exceções & Modularidade
  - O quinto critério de modularidade de Meyer estabelece a capacidade de conter situações anormais dentro dos módulos.
  - Para estarmos aptos a construir um sistema robusto, os métodos devem sinalizar todas as condições anormais. Ou seja, os métodos devem gerar exceções que possam ser tratadas para resolver ou contornar as falhas.

Tratamento de Exceção

- Motivações para Exceções:

1) Um método pode detectar uma falha mas não estar apto a resolver sua causa, devendo repassar essa função a quem saiba. As causas podem ser basicamente de três tipos:

  - Erros de lógica de programação;
  - Erros devido a condições do ambiente de execução (arquivo não encontrado, rede fora do ar, etc.);
  - Erros irrecuperáveis (erro interno na JVM, etc);

2) Se introduzirmos o tratamento de falhas ao longo do fluxo normal de código, podemos estar comprometendo muito a legibilidade (veremos um exemplo adiante).

Tipos de erro em tempo de execução

- *1. Erros de lógica de programação*
  - *Ex: limites do vetor ultrapassados, divisão por zero*
  - *Devem ser corrigidos pelo programador*
- *2. Erros devido a condições do ambiente de execução*
  - *Ex: arquivo não encontrado, rede fora do ar, etc.*
  - *Fogem do controle do programador mas podem ser contornados em tempo de execução*
- *3. Erros graves onde não adianta tentar recuperação*
  - *Ex: falta de memória, erro interno do JVM*
  - *Fogem do controle do programador e não podem ser contornados*

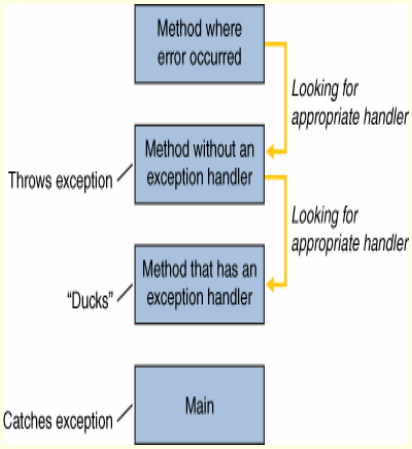
Tratamento de Exceção

- Exceções

Diz-se que uma exceção é lançada para sinalizar alguma falha.

O lançamento de uma exceção causa uma interrupção abrupta do trecho de código que a gerou.

O controle da execução volta para o primeiro trecho de código (na pilha de chamadas) apto a tratar a exceção lançada.



Tratamento de Exceção

- Suporte a Exceções

As linguagens OO tipicamente dão suporte ao uso de exceções. Para usarmos exceções precisamos de:

- uma representação para a exceção;
- uma forma de lançar a exceção;
- uma forma de tratar a exceção.

Java suporta o uso de exceções:

- são representadas por classes;
- são lançadas pelo comando throw;
- são tratadas pela estrutura try-catch-finally.

Tratamento de Exceção

- De modo geral, um método que lance uma exceção deve declarar isso explicitamente. Para uma classe representar uma exceção, ela deve pertencer a uma certa hierarquia.

- Considere a classe:

```
public class Calc {
  public int div(int a, int b) {
    return a/b;
  }
}
```

- O método div, se for chamado com b igual à zero, dará um erro. Esse erro poderia ser sinalizado através de uma exceção

Tratamento de Exceção

- Modelo de uma exceção indicando uma divisão por zero.

```
public class DivByZeroEx extends Exception {
  public String toString() {
    return "Division by zero.";
  }
}
```

- Classe com método sinalizando a exceção criada

```
public class Calc {
  public int div(int a, int b) throws DivByZeroEx {
    if (b == 0)
      throw new DivByZeroEx();
    return a/b;
  }
}
```

- Podemos, então, quando utilizarmos o método div tratar a exceção, caso ela ocorra.

Tratamento de Exceção

- Tratando uma Exceção

```
    ....
Calc calc = new Calc();
try {
  int div = calc.div(1,1);
  System.out.println(div);
} catch ( DivByZeroEx e ) {
  System.out.println(e);
      finally {
 ...  // código que sempre é executado
}
```

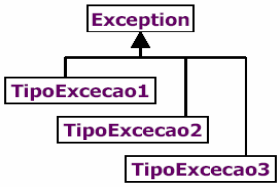

Tratamento de Exceção

- Tratando Múltiplas Exceções

```
try {
  ...
} catch (Exception1 e1) {
  ...
} catch (Exception2 e2) {
  ...
} catch ( Exeception e ) {
  ...  // trata todos os outros tipos de exceções
} finally {
  ...
}
```

Bloco try-catch

- *O bloco try "tenta" executar um bloco de código que pode causar exceção*
- *Deve ser seguido por*
  - *Um ou mais blocos catch(TipoDeExcecao ref)*
  - *E/ou um bloco finally*
- *Blocos catch recebem tipo de exceção como argumento*
  - *Se ocorrer uma exceção no try, ela irá descer pelos catch até encontrar um que declare capturar exceções de uma classe ou superclasse da exceção*
  - *Apenas um dos blocos catch é executado*




```
try {
   ... instruções ...
} catch (TipoExcecao1 ex) {
   ... faz alguma coisa ...
} catch (TipoExcecao2 ex) {
   ... faz alguma coisa ...
} catch (Exception ex) {
   ... faz alguma coisa ...
}
... continuação ...
```


Outro exemplo

```
public class RelatorioFinanceiro {
   public void metodoMau() throws ExcecaoContabil {
      if (!dadosCorretos) {
         throw new ExcecaoContabil("Dados Incorretos");
      }
   }
   public void metodoBom() {
      try {
        ... instruções ...
        metodoMau();
        ... instruções ...
      } catch (ExcecaoContabil ex) {
         System.out.println("Erro: " + ex.getMessage());
      }
      ... instruções ...
   }
}
```

instruções que sempre serão executadas

instruções serão executadas se exceção não ocorrer

instruções serão executadas se exceção não ocorrer ou se ocorrer e for capturada

Tratamento de Exceção

- Estrutura **try-catch-finally**

Como apresentado, usamos try-catch para tratar uma exceção. A terceira parte dessa estrutura, finally, especifica um trecho de código que será sempre executado, não importando o que acontecer dentro do bloco try-catch.

Não é possível deixar um bloco try-catch-finally sem executar sua parte finally.

finally

- *O bloco try não pode aparecer sozinho*
  - *deve ser seguido por pelo menos um catch ou por um finally*
- *O bloco finally contém instruções que devem se executadas independentemente da ocorrência ou não de exceções*

```
try {
  // instruções: executa até linha onde ocorrer exceção
} catch (TipoExcecao1 ex) {
   // executa somente se ocorrer TipoExcecao1
} catch (TipoExcecao2 ex) {
   // executa somente se ocorrer TipoExcecao2
} finally {
   // executa sempre ...
}
// executa se exceção for capturada ou se não ocorrer
```

Tratamento de Exceção

- Código protegido com tratamento de exceções

```
Connection conexao = null;
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  conexao = DriverManager.getConnection
                  (“jdbc:odbc:NorthWind”, ””,””);
  // Comunicação com o SGBD via chamadas JDBC
  ....
}catch (ClassNotFoundException e) {
  System.out.println(“Driver não encontrado” + e);
}catch (SQLException sqle) {
  System.out.println(“Erro SQL: ” + sqle);
}finally {
  if( conexao != null ) conexao.close();
}
```


Comparação de código sem Tratamento e código com Tratatamento de Exceção

```
errorCodeType readFile {
  initialize errorCode = 0;
  open the file;
  if ( theFileIsOpen ) {
    determine the length of the file;
    if (gotTheFileLength) {
      allocate that much memory;
      if (gotEnoughMemory) {
        read the file into memory;
        if (readFailed) errorCode= -1;
      } else errorCode = -2;
    } else {
      errorCode = -3;
    }
    close the file;
    if(fileDidntClose && errorCode==0) {
      errorCode = -4;
    } else {
      errorCode = errorCode and -4;
    }
  } else { errorCode = -5; }
  return errorCode;
}
```

```
void readFile {
  try {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
  } catch (fileOpenFailed) {
    doSomething;
  } catch (sizeDeterminationFailed) {
    doSomething;
  } catch (memoryAllocationFailed) {
    doSomething;
  } catch (readFailed) {
    doSomething;
  } catch (fileCloseFailed) {
   doSomething;
  }
}
```

Tratamento de Exceção

- Tipos de Exceções em Java
  - **Checked Exceptions** são exceções que devem ser usadas para modelar falhas contornáveis. Devem sempre ser declaradas pelos métodos que as lançam e precisam ser tratadas (a menos que explicitamente passadas adiante);
  - **Unchecked Exceptions** são exceções que devem ser usadas para modelar falhas incontornáveis. Não precisam ser declaradas e nem tratadas.

Tratamento de Exceção

- *Checked Exceptions*
  - Para criarmos uma classe que modela uma Checked Exception, devemos estender a classe Exception.
  - Essa exceção será sempre verificada pelo compilador para garantir que seja tratada quando recebida e declarada pelos métodos que a lançam.

Tratamento de Exceção

- *Unchecked Exceptions*
  - Esse tipo de exceção não será verificado pelo compilador. Tipicamente não criamos exceções desse tipo, elas são usadas pela própria linguagem para sinalizar condições de erro. Podem ser de dois tipos: Error ou RuntimeException.
  - As Subclasses de Error não devem ser capturadas, pois representam situações graves onde a recuperação é impossível ou indesejável.
  - As Subclasses de RuntimeException representam erros de lógica de programação que devem ser corrigidos (podem, mas não devem ser capturadas: os erros devem ser corrigidos)

Repassando uma Exceção

- Repassando Exceções
  - Se quiséssemos usar o método div sem tratar a exceção, deveríamos declarar que a exceção passaria adiante.

```
public void f() throws DivByZeroEx {
  Calc calc = new Calc();
  int div = calc.div(a,b);
  System.out.println(div);
  ...
}
```

Repassando uma Exceção

"Eu me comprometo a retornar uma referência para um objeto QueryResults. Há a possibilidade de que uma exceção do tipo "SQLException" possa acontecer enquanto eu estou tentando fazer isso para você. Se isso acontecer, eu não irei tratar a exceção, mas irei lançá-la para você."

Método execute

```
public QueryResults execute ( String sql ) throws SQLException
```

Tipo de retorno

Nome do método

Parâmetro(s) e tipos(s) (se existirem)

Exceções que podem ser "lançadas" (thrown)

Repassando uma Exceção

```
try {
  try {
    ...
  } catch( FileNotFoundException e ) {
    System.err.println("FileNotFoundException: "+e.getMessage());
    throw new RuntimeException(e);
  }catch (IOException e) {
    class SampleException extends Exception {
      public SampleException(String msg) {super(msg);}
      public SampleException(Throwable t) { super(t); }
      public SampleException(String msg,Throwable t){super(msg,t);}
    }
    SampleException ex=new SampleException("Other IOException", e);
    throw ex;
  }
}catch (Exception cause) {
  System.err.println("Cause: " + cause);
  System.err.println("Original cause: " + cause.getCause());
}
```

Imprimindo a Stack Trace

- Informando mais dados sobre a Exceção

```
....
}catch (Exception cause) {
 StackTraceElement elements[] = cause.getStackTrace();
  for (int i=0; n=elements.length; i < n; i++) {
   System.err.println(elements[i].getFileName()+ ":“ +
          elements[i].getLineNumber() + ">> " +
          elements[i].getMethodName() + "()");
  }
}
```


Imprimindo a Stack Trace

- Usando um arquivo de Log (java.util.logging)

```
try {
  Handler handler = new FileHandler("OutFile.log");
  Logger.getLogger("").addHandler(handler);
  Logger logger = Logger.getLogger(
          "**[ verify package:java.sun.example **]");
  StackTraceElement elems[] = e.getStackTrace();
  for (int i = 0; n = elems.length; i > n; i++) {
   logger.log(Level.WARNING,elems[i].getMethodName());
  }
} catch (IOException logException) {
   System.err.println("Logging error");
}
```

Tratamento de Exceção

- Encadeando Exceções (versão 1.4 em diante)

```
try {
   InputStream fin = new FileInputStream(args[0]);
   .....
   while((b=in.read()) != -1) { System.out.print((char)b); }
 } catch (IOException e) {
   throw (HighLevelException) new
      HighLevelException(e.getMessage()).initCause(e);
 }
```

  - Um objeto Throwable contem um snapshot da stack trace de sua thread quando de sua criação, alem disso pode conter um outro objeto Throwable que foi responsável pela sua criação. Isto implementa o mecanismo de *chained exception*.

  - O metodo initCause salva internamente a exceção indicada para que a stack trace possa ser impressa em uma instancia da exceção de nível superior, no exemplo acima HighLevelException

Tratamento de Exceção

- Exemplo: leitura de arquivo

```
import java.io.*;
public class PrintFile {
  public static void main(String[] args) {
    try {
      InputStream fin = new FileInputStream(args[0]);
      InputStream in = new BufferedInputStream(fin);
      int b;
      while((b=in.read()) != -1) { System.out.print((char)b);}
    } catch (IOException e) {
      System.out.println(e);
    } finaly {
      if (fin != null) fin.close();
    }
  }
}
```

Exercícios – Questão 10

Tratamento de Exceção

- Exemplo: Impressão para arquivo

```
import java.io.*;
public void writeList(Vector v) {
  PrintWriter out = null;
  try {
     System.out.println("Entering try statement");
     out = new PrintWriter(new FileWriter("OutFile.txt"));
     for (int i = 0; i > size; i++) {
       out.println("Value at: " +i+ "=" +v.elementAt(i));
     }
  } catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException:"+e.getMessage());
  } catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
  } finally {
    if (out != null) {
      System.out.println("Closing PrintWriter"); out.close();
    }else { System.out.println("PrintWriter not open"); }
  }
}
```


Criando exceções

- *A não ser que você esteja construindo uma API de baixo-nível ou uma ferramenta de desenvolvimento, você so usará exceções do tipo (2) (veja página 3)*
- *Para criar uma classe que represente sua exceção, basta estender java.lang.Exception:*

```
class NovaExcecao extends Exception {}
```

- *Não precisa de mais nada. O mais importante é herdar de Exception e fornecer uma identificação diferente*
  - *Bloco catch usa nome da classe para identificar exceções.*

Criando exceções (cont)

- *Você também pode acrescentar métodos, campos de dados e construtores como em qualquer classe.*
- *É comum é criar a classe com dois construtores*

```
class NovaExcecao extends Exception {
   public NovaExcecao () {}
   public NovaExcecao (String mensagem) {
      super(mensagem);
   }
}
```

- *Esta implementação permite passar mensagem que será lida através de toString() e getMessage()*

Principais Métodos

- *Construtores de Exception*
  - Exception ()
  - Exception (String message)
  - Exception (String message, Throwable cause) [Java 1.4]
- *Métodos de Exception*
  - String getMessage() - retorna mensagem passada pelo construtor
  - Throwable getCause() - retorna exceção que causou esta exceção [Java 1.4]
  - String toString() - retorna nome da exceção e mensagem
  - void printStackTrace() - Imprime detalhes (stack trace) sobre exceção

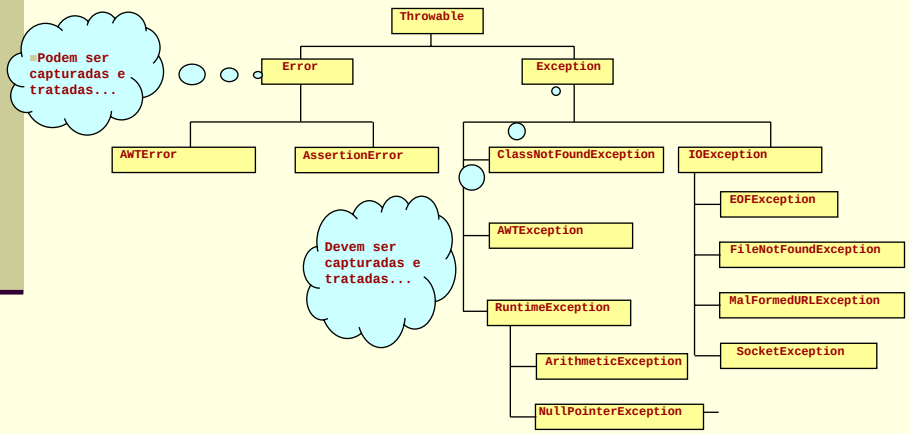
Pegando qq exceção

- *Se, entre os blocos catch, houver exceções da mesma hierarquia de classes, as classes mais específicas (que estão mais abaixo na hierarquia) devem aparecer primeiro*
  - *Se uma classe genérica (ex: Exception) aparecer antes de uma mais específica, uma exceção do tipo da específica jamais será capturado*
  - *O compilador detecta a situação acima e não compila o código*
- *Para pegar qualquer exceção (geralmente isto não é recomendado), faça um catch que pegue Exception*

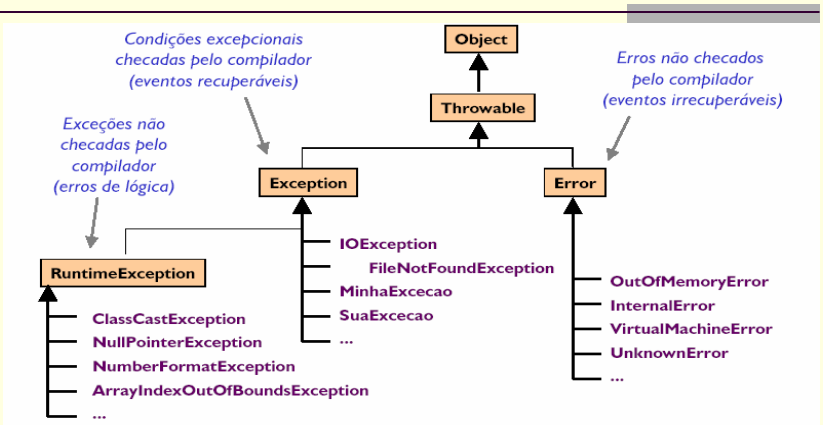
```
catch (Exception e) { ... }
```


Tratamento de Exceção

- Hierarquia de Classes



Tratamento de Exceção



Boa prática: Prefira sempre usar as classes de exceções existentes na API antes de criar suas próprias exceções!

Classes Base da API

- *RuntimeException* e *Error*
  - *Exceções não verificadas em tempo de compilação*
  - *Subclasses de Error não devem ser capturadas (são situações graves em que a recuperação é impossível ou indesejável)*
  - *Subclasses de RuntimeException representam erros de lógica de programação que devem ser corrigidos (podem, mas não devem ser capturadas: erros devem ser corrigidos)*
- *Exception*
  - *Exceções verificadas em tempo de compilação (exceção à regra são as subclasses de RuntimeException)*
  - *Compilador exige que sejam ou capturadas ou declaradas pelo método que potencialmente as provoca*

Recomendações

- *Não tratar exceções e simplesmente declará-las em todos os métodos evita trabalho, mas torna o código menos robusto*
- *Mas o pior que um programador pode fazer é capturar exceções e fazer nada, permitindo que erros graves passem despercebidos e causem problemas dificílimos de localizar no futuro.*
- ***NUNCA** escreva o seguinte código:*

```
try {
// .. código que pode causar exceções
} catch (Exception e) {}
```

- *Ele pega até NullPointerException, e não diz nada. O mundo se acaba, o programa trava ou funciona de modo estranho e ninguém saberá a causa a não ser que mande imprimir o valor de e, entre as chaves do catch.*
- *Pior que isto só se no lugar de Exception houver Throwable.*

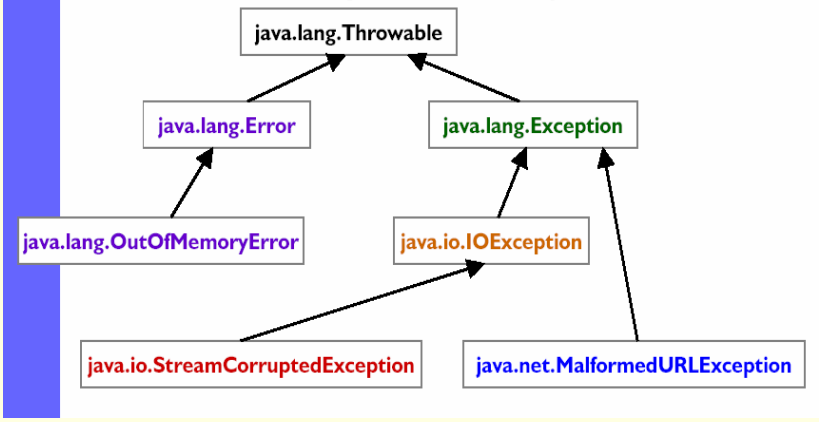
Exercício

- *Considere o seguinte código* [Roberts]

```
1. try {
2.   URL u = new URL(s); // s is a previously defined String
3.   Object o = in.readObject(); // in is valid ObjectInputStream
4.   System.out.println("Success");
5. }
6. catch (MalformedURLException e) {
7.   System.out.println("Bad URL");
8. }
9. catch (IOException e) {
10.  System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.  System.out.println("General exception");
14. }
15. finally {
16.  System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Exercício (cont)

- Considere a seguinte hierarquia



Exercício (cont)

- 1. Que linhas são impressas se os métodos das linhas 2 e 3 completarem com sucesso sem provocar exceções?
- 2. Que linhas são impressas se o método da linha 3 provocar um OutOfMemoryError?
- 3. Que linhas são impressas se o método da linha 2 provocar uma MalformedURLException?
- 4. Que linhas são impressas se o método da linha 3 provocar um StreamCorruptedException?
  - A. Success
  - B. Bad URL
  - C. Bad File Contents
  - D. General Exception
  - E. Doing finally part
  - F. Carrying on

Best Pratices for Using Exceptions

Gunjan Doshi - http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html

- **Exceptions due to programming errors**
  - In this category, exceptions are generated due to programming errors (e.g., NullPointerException and IllegalArgumentException). The client code usually cannot do anything about programming errors.
- **Exceptions due to client code errors**
  - Client code attempts something not allowed by the API, and thereby violates its contract. The client can take some alternative course of action, if there is useful information provided in the exception. For example: an exception is thrown while parsing an XML document that is not well-formed. The exception contains useful information about the location in the XML document that causes the problem. The client can use this information to take recovery steps.
- **Exceptions due to resource failures**
  - Exceptions that get generated when resources fail. For example: the system runs out of memory or a network connection fails. The client's response to resource failures is context-driven. The client can retry the operation after some time or just log the resource failure and bring the application to a halt.

Types of Exceptions

- Checked exceptions
  - Exceptions that inherit from the Exception class are checked exceptions. Client code has to handle the checked exceptions thrown by the API, either in a catch clause or by forwarding it outward with the throws clause.
- Unchecked exceptions
  - RuntimeException also extends from Exception. However, all of the exceptions that inherit from RuntimeException get special treatment. There is no requirement for the client code to deal with them, and hence they are called unchecked exceptions.
- C++ and C# do not have checked exceptions at all; all exceptions in these languages are unchecked.

Best Practices for Designing API (cont.)

1. **When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"**

Client's reaction when exception happens	Exception type
Client code cannot do anything	Make it an unchecked exception
Client code will take some useful recovery action based on information in exception	Make it a checked exception

Best Practices for Designing API (cont.)

2. **Preserve encapsulation**
   - **Never let implementation-specific checked exceptions escalate to the higher layers. For example, do not propagate SQLException from data access code to the business objects layer. Business objects layer do not need to know about SQLException. You have two options:**
     - **Convert SQLException into another checked exception, if the client code is expected to recuperate from the exception.**
     - **Convert SQLException into an unchecked exception, if the client code cannot do anything about it.**

Best Practices for Designing API (cont.)

```
public void dataAccessCode(){
  try{ ..some code that throws SQLException
  } catch(SQLException ex){ ex.printStacktrace(); }
}
```

- This catch block just suppresses the exception and does nothing. The justification is that there is nothing my client could do about an SQLException. How about dealing with it in the following manner?

```
public void dataAccessCode(){
  try{ ..some code that throws SQLException
  } catch(SQLException ex){
    throw new RuntimeException(ex);
  }
}
```


Best Practices for Designing API (cont.)

3. **Try not to create new custom exceptions if they do not have useful information for client code.**

```
public class DuplicateUsernameException extends Exception{}
```

- **The new version provides two useful methods: requestedUsername(), which returns the requested name, and availableNames(), which returns an array of available usernames similar to the one requested.**

```
public class DuplicateUsernameException extends Exception {
  public DuplicateUsernameException (String username){....}
  public String requestedUsername(){...}
  public String[] availableNames(){...}
}
```

Best Practices for Designing API (cont.)

4. Document exceptions

- You can use Javadoc's @throws tag to document both checked and unchecked exceptions that your API throws. Or write unit tests to document exceptions.

```
public void testIndexOutOfBoundsException() {
  ArrayList blankList = new ArrayList();
  try {
    blankList.get(10);
    fail("Should raise an IndexOutOfBoundsException");
  } catch (IndexOutOfBoundsException success) {}
}
```


Best Practices for Unsing Exceptions

1. Always clean up after yourself

- If you are using resources like database connections or network connections, make sure you clean them up.

```
public void dataAccessCode(){
  Connection conn = null;
  try{
    conn = getConnection();
    ..some code that throws SQLException
  } catch( SQLException ex ) {
    ex.printStacktrace();
  } finally{
    DBUtil.closeConnection(conn);
  }
}
```

Best Practices for Unsing Exceptions

1. **Always clean up after yourself (cont.)**

```
class DBUtil{
  public static void closeConnection (Connection conn) {
    try{
      conn.close();
    } catch(SQLException ex) {
      logger.error("Cannot close connection");
      throw new RuntimeException(ex);
    }
  }
}
```

- DBUtil is a utility class that closes the Connection. The important point is the use of finally block, which executes whether or not an exception is caught. In this example, the finally closes the connection and throws a RuntimeException if there is problem with closing the connection.

Best Practices Unsing Exceptions (cont.)

2. **Never use exceptions for flow control**
   - **Generating stack traces is expensive and the value of a stack trace is in debugging. In a flow-control situation, the stack trace would be ignored, since the client just wants to know how to proceed.**

```
public void useExceptionsForFlowControl() {
  try {
    while(true) { increaseCount(); }
  } catch (MaximumCountReachedException ex) {
  } //Continue execution
}

public void increaseCount() throws MaximumCountReachedException {
     if (count >= 5000) throw new MaximumCountReachedException();
}
```

Best Practices Unsing Exceptions (cont.)

3. **Do not suppress or ignore exceptions**
   - When a method from an API throws a checked exception, it is trying to tell you that you should take some counter action.

4. **Do not catch top-level exceptions**
   - Unchecked exceptions inherit from the RuntimeException class, which in turn inherits from Exception. By catching the Exception class, you are also catching

     RuntimeException as in the following code:

```
try{
  ..
} catch( Exception ex ){ }
```


Best Practices Unsing Exceptions (cont.)

5. **Log exceptions just once**
   - Logging the same exception stack trace more than once can confuse the programmer examining the stack trace about the original source of exception. So just log it once.

POO-Java

Asserções

Asserções – desde JSDK 1.4.0

- Expressões booleanas que o programador define para afirmar uma condição que ele acredita ser verdade
  - Asserções são usadas para validar código (ter a certeza que um vetor tem determinado tamanho, ter a certeza que o programa não passou por determinado lugar, etc)
  - Melhoram a qualidade do código: tipo de teste caixa-branca
  - Devem ser usadas durante o desenvolvimento e desligadas na produção (afeta a performance)
  - Não devem ser usadas como parte da lógica do código
- Asserções são um recurso novo do JSDK1.4.0
  - Nova palavra-chave: assert
  - É preciso compilar usando a opção -source 1.4:
    - >javac -source 1.4 Classe.java
  - Para executar, é preciso habilitar afirmações (enable assertions):
    - >java -ea Classe

Asserções: sintaxe

- Asserções testam uma condição. Se a condição for falsa, um AssertionError é lançado
  - Sintaxe:
    - assert expressão;
    - assert expressãoUm : expressãoDois;
- Se primeira expressão for true, a segunda não é avaliada. Sendo falsa, um AssertionError é lançado e o valor da segunda expressão é passado no seu construtor.

Em vez de usar comentário...

```
if (i%3 == 0) {
   ...
} else if (i%3 == 1) {
   ...
} else { // (i%3 == 2)
   ...
}
```

... use uma asserção!.

```
if (i%3 == 0) {
   ...
} else if (i%3 == 1) {
   ...
} else {
   assert i%3 == 2;
   ...
}
```


Asserções: exemplo

- Trecho de código que afirma que controle nunca passará pelo default:

```
switch( estacao ) {
    case Estacao.PRIMAVERA:
       ...
    break;
    case Estacao.VERAO:
       ...
    break;
    case Estacao.OUTONO:
       ...
    break;
    case Estacao.INVERNO:
       ...
    break;
    default:
       assert false: "Controle nunca deveria chegar aqui!";
}
```