Programação Orientada a Objetos

Prof. Paulo André Castro
pauloac@ita.br
www.comp.ita.br/~pauloac
ITA – Stefanini

Planejamento

- Aula 1
  - Introdução
  - Conceitos Básicos
    - Classe, Objeto, Método,Herança, interfaces, polimorfismo, Encapsulamento
  - Introdução a linguagem Java
- Aula 2
  - Modelagem de Programas Orientada a Objetos
  - Introdução a Padrões de Projeto (Design Patterns)
  - Introdução a linguagem Java
  -
- Aula 3
  - Introdução a Ambientes Integrados de Desenvolvimento
  - Desenvolvimento de Programas Básicos
  - Manipulação de E/S em Java
  - Tipos genéricos

Planejamento

- Aula 4
  - Programando Interfaces Gráficas com Java I
- Aula 5
  - Programando Interfaces Gráficas comJava - II
- Aula 6
  - Programação concorrente (Threads)
  - Conexão com outros programas em Rede
- Aula 7
  - Conectividade com Banco de Dados (JDBC)
  - Padrão de projeto para acesso a Dados: DAO Design Pattern
- Aula 8
  - XML
  - Introdução a Web Services

Referências

- “Conceitos Essenciais de Computação com Java”. C. Horstmann. Ed. Bookman.
- “Core Java – Volume I”, Cay Horstmann, Gary Cornell. Ed. SunSoft Press.
- “Core Java – Volume II – Advanced Features”, Cay Horstmann, Gary Cornell. Ed. SunSoft Press.
- “Core web Programming”, Marty Hall, Larry Brown, 2003.
- “ Java – Como Programar”, Deitel & Deitel, 6a. Edição, Ed. Prentice Hall. 2005.
- E Internet
  - java.sun.com, www.theserverside.com, etc.

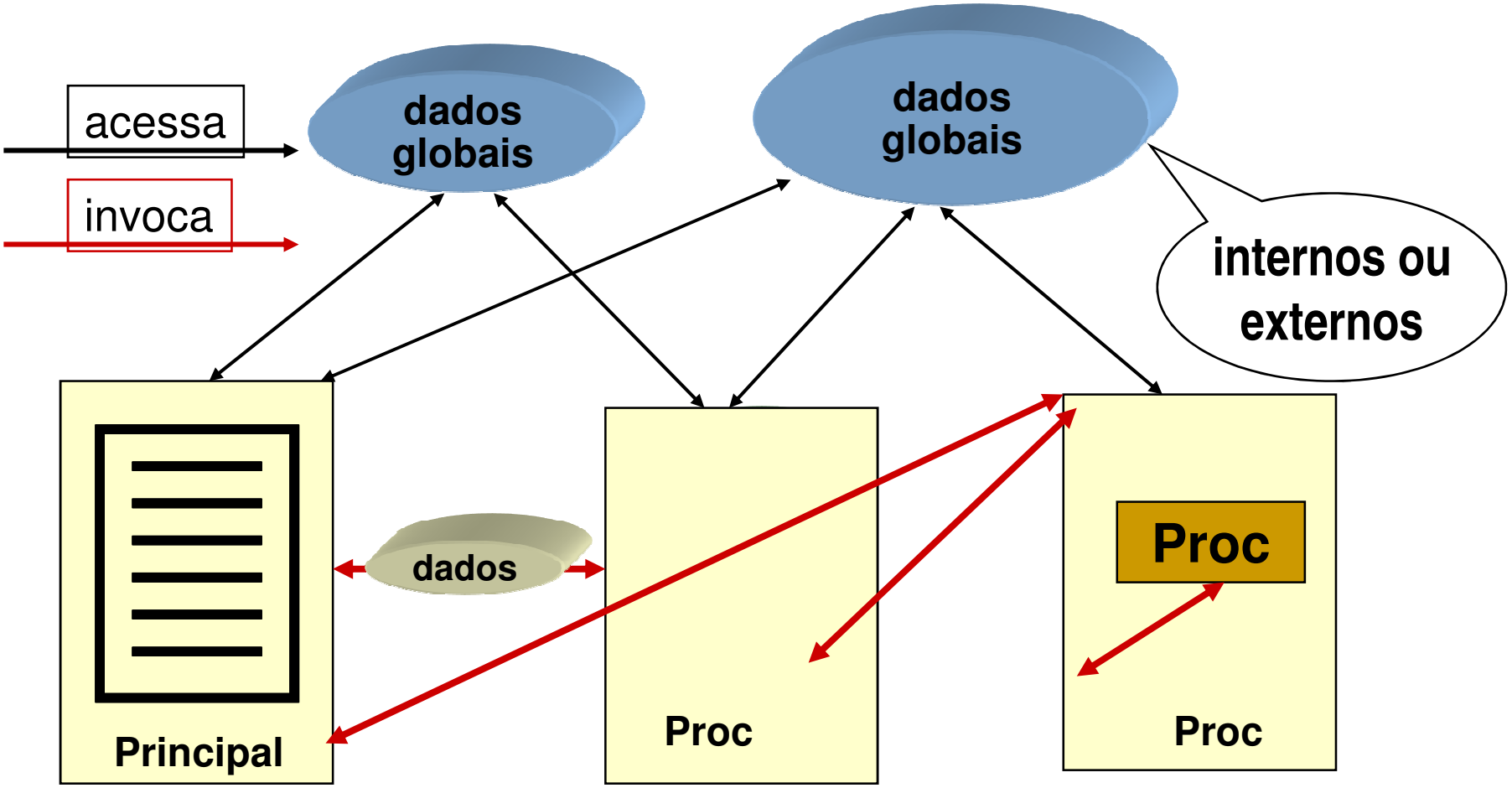
Sumário de Hoje

- **Introdução**
- Conceitos Básicos
  - Nomenclatura básica em OO
  - Variáveis e Instâncias
  - Métodos
  - Construtores
  - Herança e Polimorfismo
- Introdução a linguagem Java

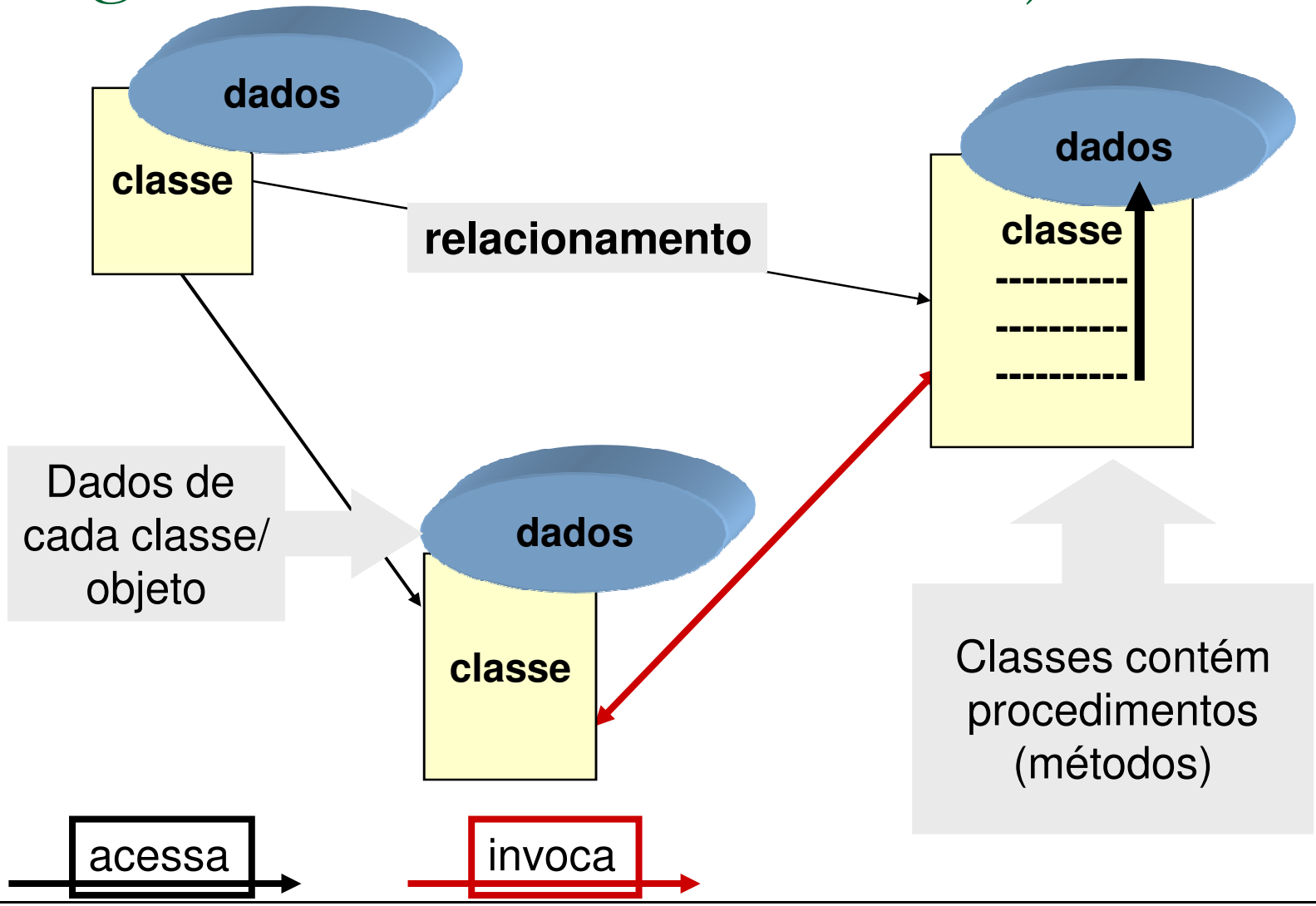
Introdução

- Programação Estruturada x Programação Orientada a Objetos
  - Modelagem com base no conceito de módulo ou sub-programa
  - Modelagem com base no conceito de classe e seus relacionamentos
- Linguagens Orientadas a Objetos
  - Simula, SmallTalk
  - C++, C#, (VB?), etc.
  - Java

Programação Estrurada



Programação Orientada a Objetos



Sumário de Hoje

- Introdução
- **Conceitos Básicos**
  - Nomenclatura básica em OO
  - Variáveis e Instâncias
  - Métodos
  - Encapsulamento
  - Herança e Polimorfismo
- Introdução a linguagem Java

Conceitos básicos de OO

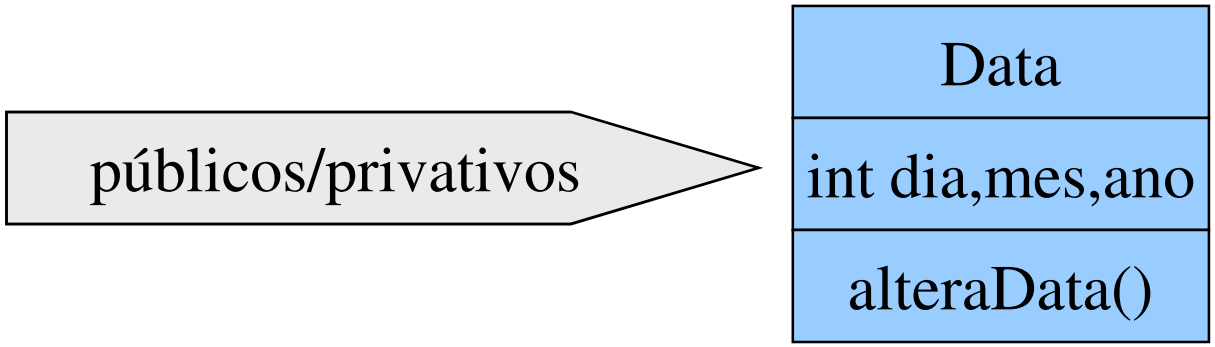
- Classe: um categoria de entidades (“coisas”)
  - Corresponde a um tipo, ou coleção, ou conjunto de entidades afins
- Objeto: Uma entidade com existência física que pertence a um determinado conjunto de entidades afins (classe)

Exemplos de Classe e Objeto

- Classes:
  - Carro, Avião, Pessoa
- Objetos:
  - Carro: Porsche 910 Placa XXXX
  - Avião: Boeing 737-300 Prefixo: PY-XXX
  - Pessoa: José da Silva CPF: XXXXXXX

Classes x Tipos de Dados

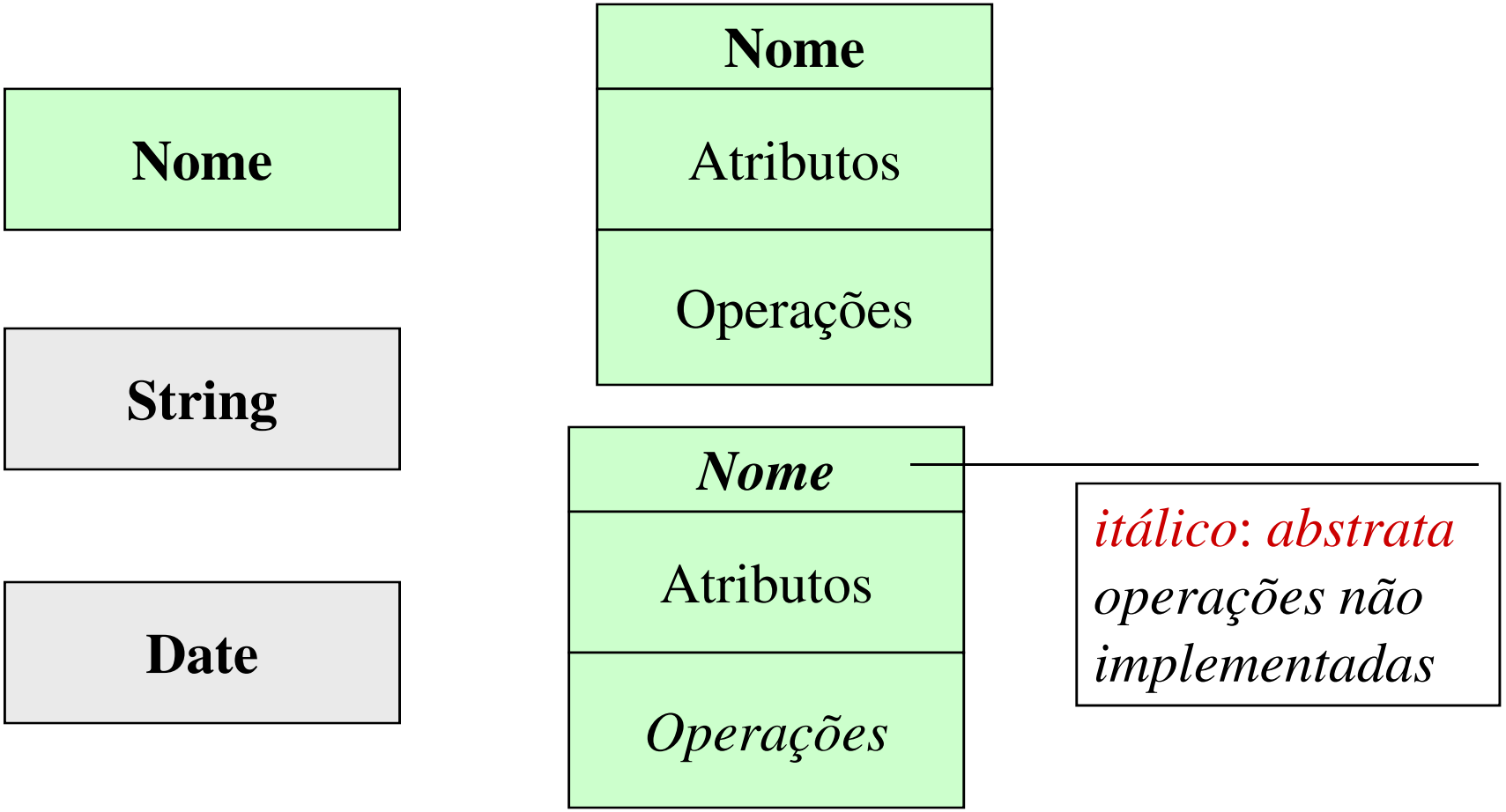
- Uma classe é um tipo definido pelo usuário que contém uma estrutura de dados e um conjunto de operações que atuam sobre estes dados
- Analogamente, é o mesmo que o tipo inteiro significa para as variáveis declaradas como inteiros: acesso a valores através de operações
- A classe encapsula dados e operações e controla o acesso a estas propriedades



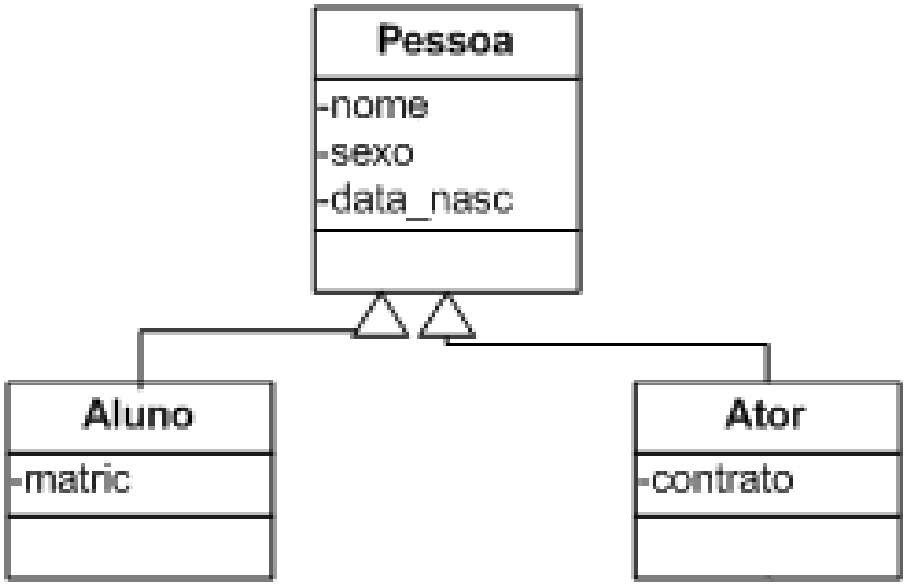
UML

- Unified Modeling Language
  - Linguagem Unificada de Modelagem
  - Linguagem Visual de Modelagem Orientada a Objetos
  - Referência: The Unified Modeling Language User Guide; G. Booch, J. Rumbaugh, I. Jacobson. ACM Press. 2000

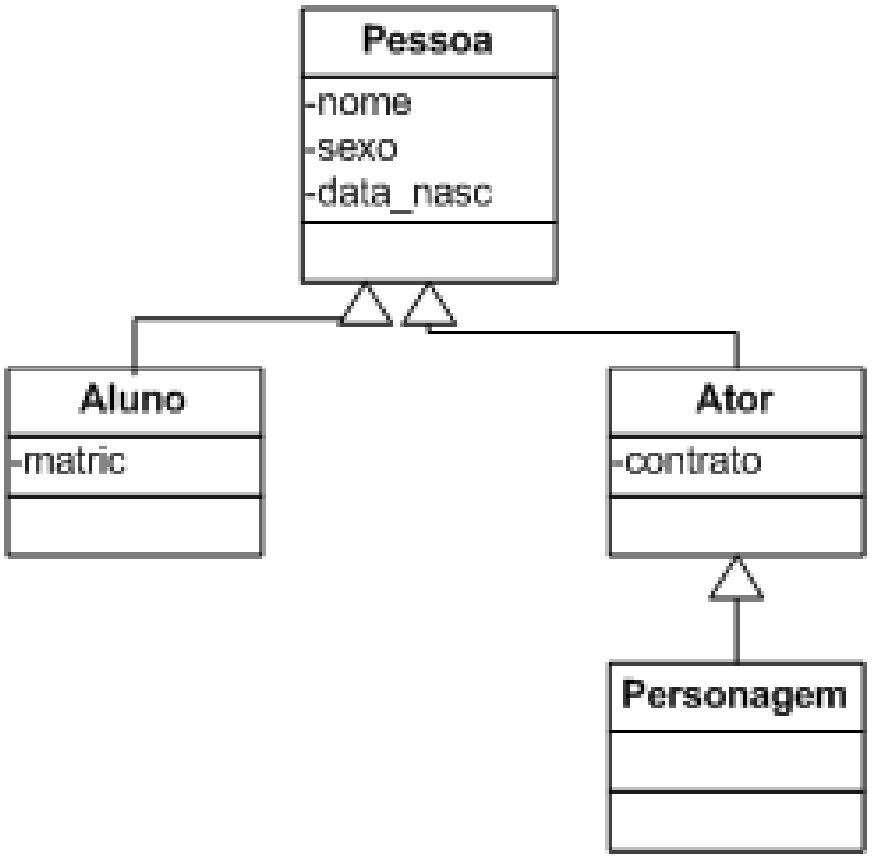
UML: notações de classes



Exemplo em UML - Herança



Exemplo em UML – Está correto ?



Exemplo de Orientação a Objetos - Java

- Classe

```
public class Pessoa {
  private int idade;
  private boolean sexo; // Verdadeiro para mulheres
 private boolean ehResponsavel() {
   if(idade>21 )
         return true;
   if(idade>18 && !sexo)
          return true;
    else
         return false;
}
public Pessoa(String nome, int id, boolean sex) {Nome=nome; idade=id; sexo=sex; }
}
}.....................................................
```

Mais em Orientação a Objetos

- Herança
  - Uma classe pode utilizar métodos e atributos de outras classes sem a necessidade de re-escrever ou copiar nada através do mecanismo de Herança
  - public class Funcionario extends Pessoa {

```
double salario;
int id;
public double getSalario() {
    return salario;
} }
```

Classes e sub-classes

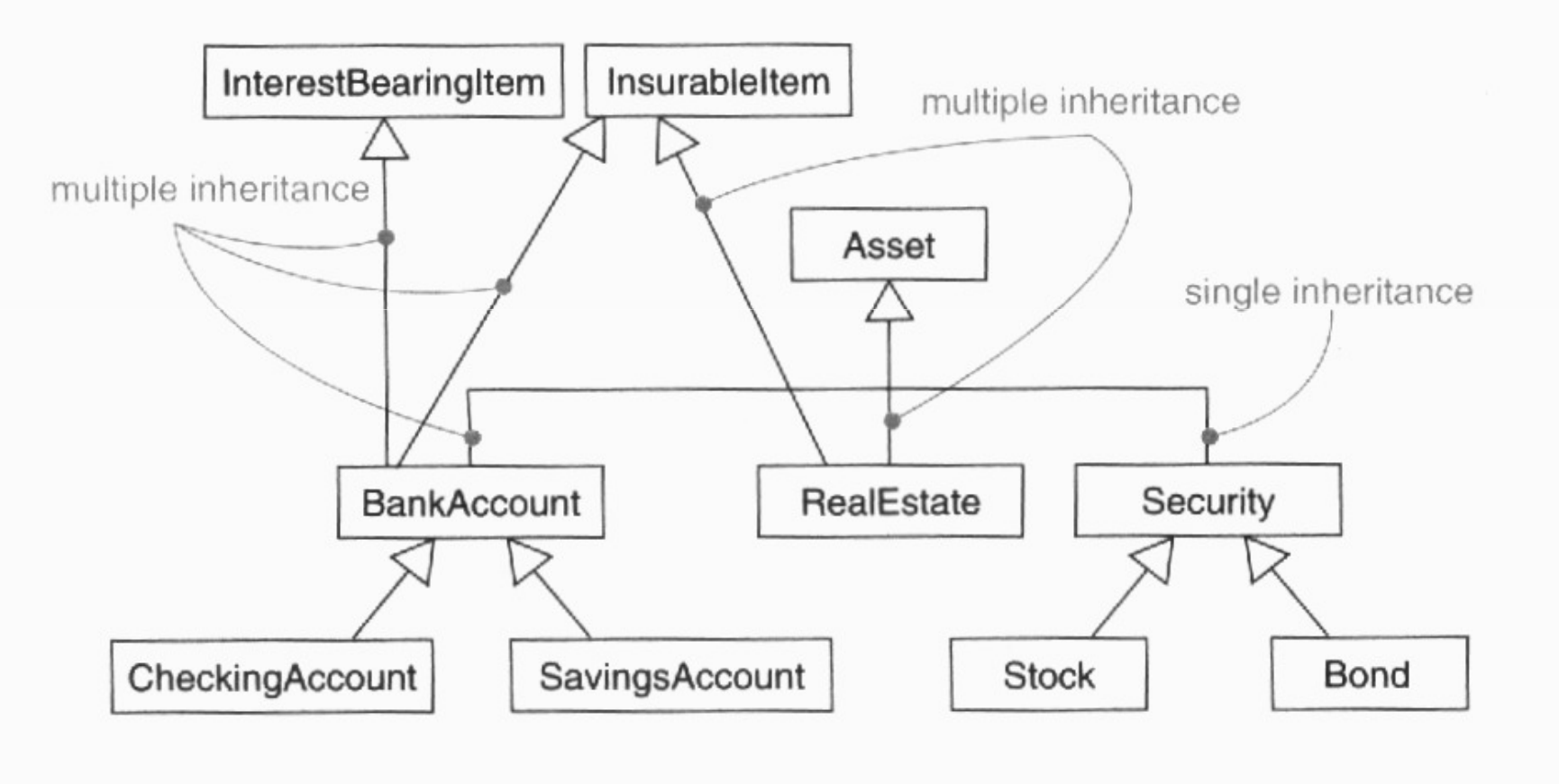
- Classe, classe-pai, super-classe, classe base:
  - Carro
  - Motor
  - Avião
- Sub-classe, classe-filha, classe derivada
  - Carro: Porsche 910
  - Motor: Ford 16V
  - Avião: Boeing 737

Relação de Herança

- O filho herda todas as características do pai
  - Comportamento: funções
  - Atributos: valores

- Em linguangens OO, geralmente há meios de restringir o que será ou não herdado

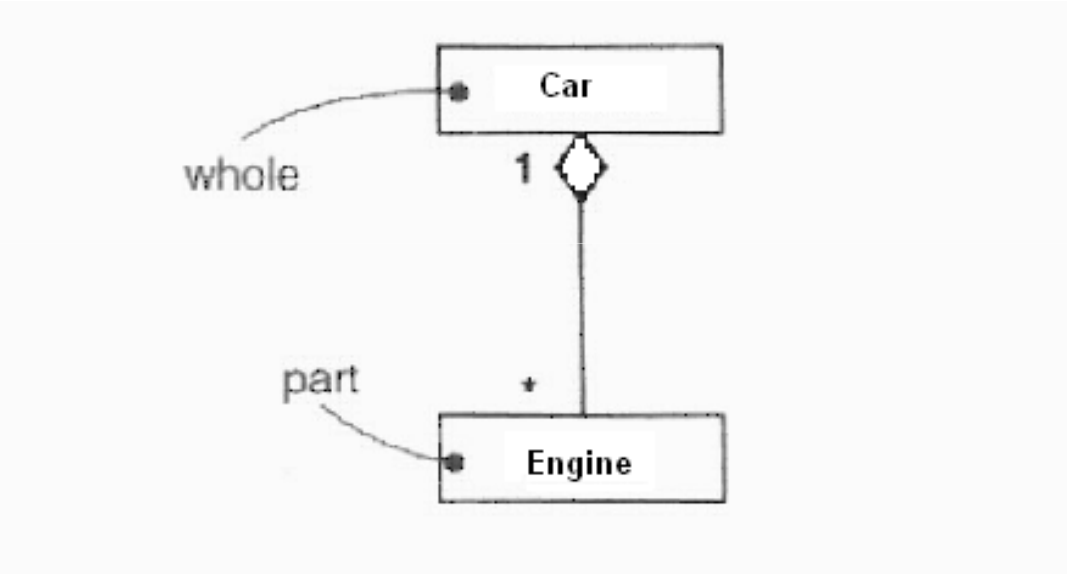
Exemplo de Herança Múltipla



Relação Agregação (Todo-Parte)

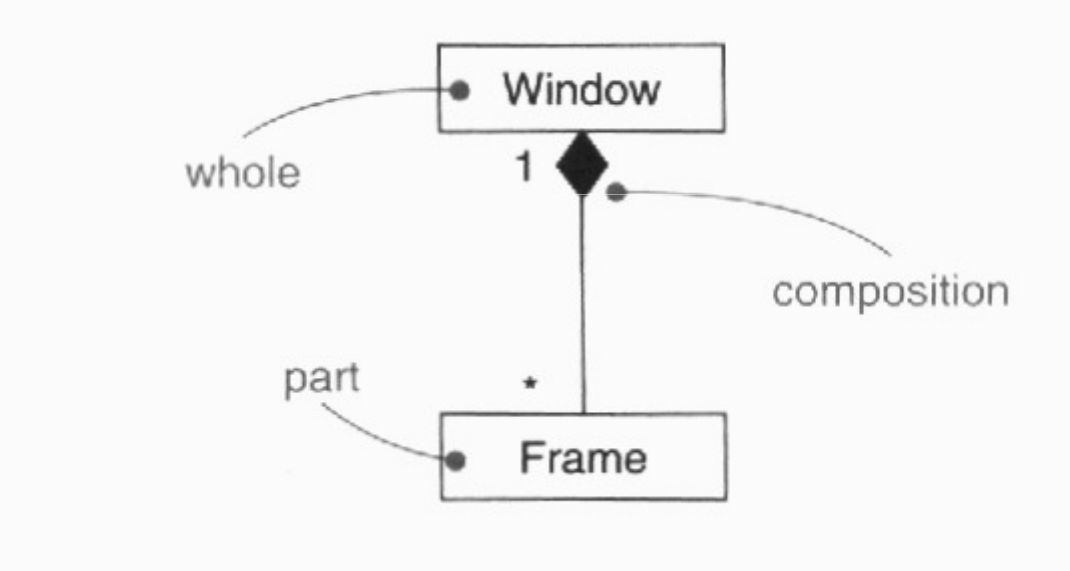
- Uma classe é composta por uma ou mais classes:
- Exemplos:
  - Carro e Motor
  - Avião e Motor
  - Pessoa e Braço

Agregação em UML

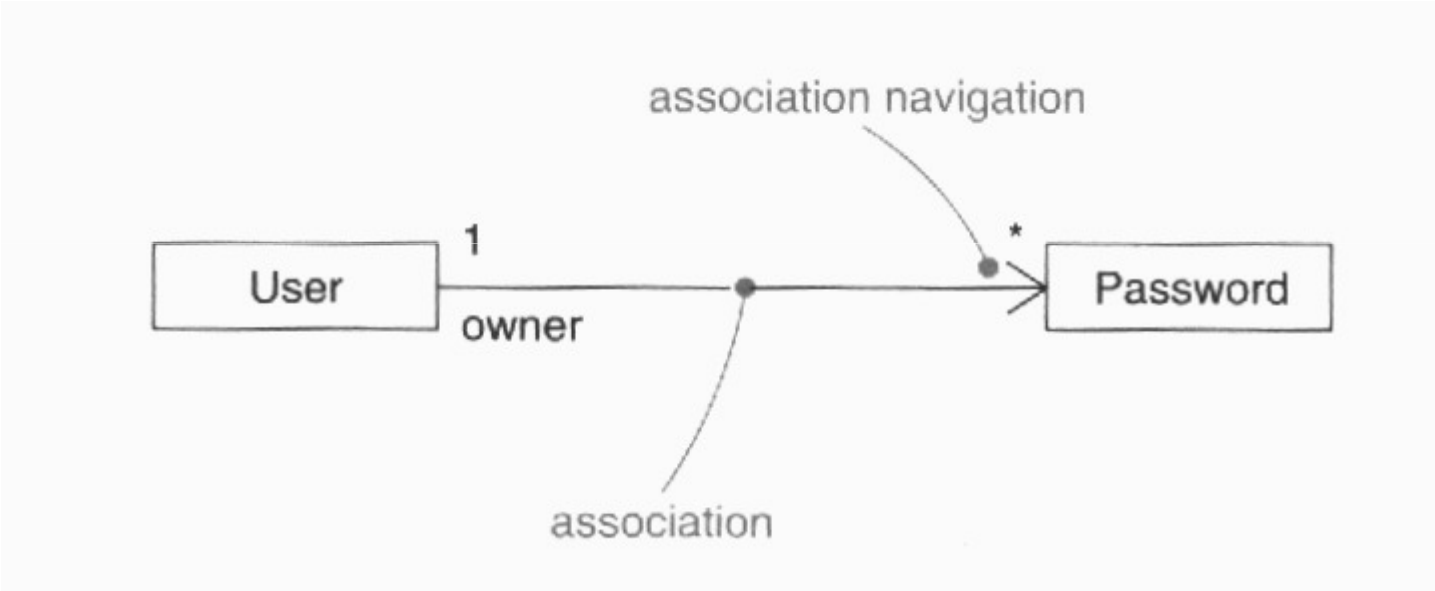


Composição: Tipo especial de Agregação

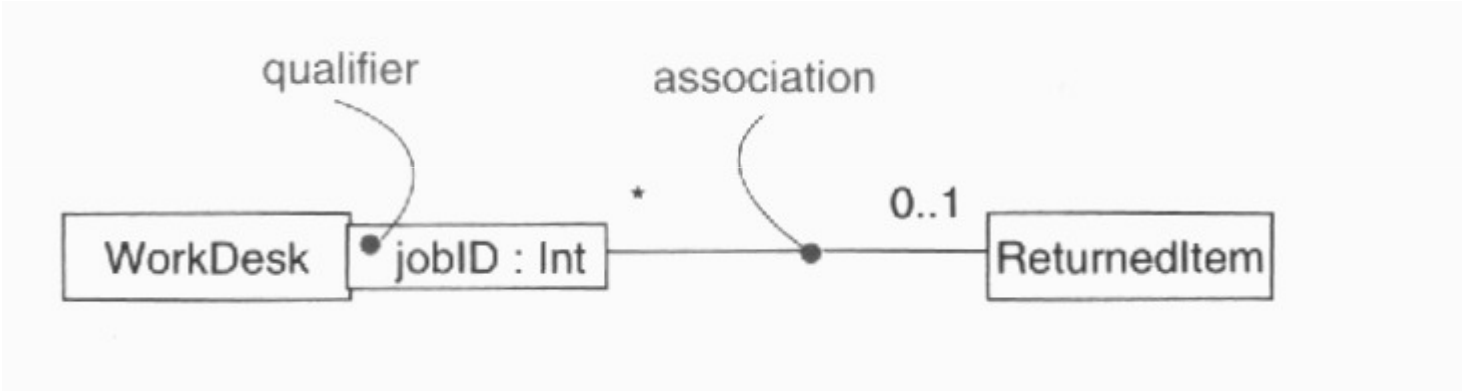
Parte pertence a apenas um todo e tem tempo de vida igual ao todo



Outras Relações: Associação



Outras Relações: Associação com Qualificação



Métodos e Atributos

- Classes podem ter vários métodos e atributos
  - Método: define um comportamento de uma classe
  - Atributo: define uma informação a ser mantida por cada instância de uma classe
- Escopo
  - Escopo de Classe:
    - Ex. Boeing 737 atributo: número de motores
  - Escopo de Objeto (“Instância”)
    - Ex. Boeing 737 atributo: número de assentos

Encapsulamento

- Encapsulamento: É a capacidade de "esconder" parte do código e dos dados do restante do programa
- Pode-se definir um grau de visibilidade aos métodos e atributos de cada Classe.
- Há vários graus de visibilidade mas todas as linguagens implementam pelo menos os seguintes:
  - Público: Todos podem acessar (ler e escrever)
  - Privado: Apenas a própria classe pode acessar.

Polimorfismo

- Um mesmo comando enviado para objetos diferentes gera (ou pode gerar) ações diferentes.
- Exemplo:
  - Comando: Mover
    - Carro
    - Avião
    - Pessoa

Exemplo de Orientação a Objetos – classe Pessoa

Arquivo: Pessoa.java

```
public class Pessoa {
  private int idade;
  private boolean sexo; // Verdadeiro para mulheres
 private boolean ehResponsavel() {
   if(idade>21 )
         return true;
   if(idade>18 && !sexo)
          return true;
    else
         return false;
}
public Pessoa(String nome, int id, boolean sex) {Nome=nome; idade=id;
     sexo=sex; }
}
}
```

Exemplo de Orientação a Objetos – subclasse Casado

Arquivo: Casado.java

```
public class Casado extends Pessoa  {
  public boolean ehResponsavel() {
    return true;
   }
  public Casado(String nome,int id,boolean sexo) {
    super(nome,idade,sexo);
  } }
```

Polimorfismo – Código Java

```
public class ExemploPessoa {
 public static void main(String[] args) {
    Casado casado=new Casado("Zé",17,false);
    Pessoa cidadao=new Pessoa("Maria",18,true);
        Pessoa[] trabalhador= new Pessoa[2];
        trabalhador[0]=casado;
        trabalhador[1]=cidadao;
        for(int i=0; i<2; i++) {
         String aux;
        if(trabalhador[i].ehResponsavel())
           aux.strcpy(trabalhador[i].Nome+"é responsável");
         else
          aux.strcpy(trabalhador[i].Nome+ "não é responsável.");
          System.out.println(aux);
        }
}
}
```

Qual o resultado da execução?

Mais em Orientação a Objetos

- Alterando comportamento nas classes filhas. Sobrescrever método.
- public class Casado extends Pessoa {

```
public boolean ehResponsavel() {
   return true;
}
public Casado(String nome,int id,boolean sexo) {
    super(nome,idade,sexo);
} }
```

Mais em Orientação a Objetos – Sobrecarga de Métodos

- Sobrecarga permite a existência de vários métodos de mesmo nome, porém com assinaturas levemente diferentes ou seja variando no número e tipo de argumentos e no valor de retorno

```
public Logaritmo {
……
  public double log(double x) {
    return Math.log(x);
    }
    public double log(double x, double b) {
    return (Math.log(x)/Math.log(b));
    }
}
```

Métodos Abstratos

- Métodos abstratos não tem implementação, porém obrigam as classes filhas a realizarem esta implementação. Útil para criar padronizações para as classes derivadas
- Se uma classe tem um (ou mais) métodos abstratos torna-se uma classe abstrata e não pode ser instanciada
- Exemplo:
  - Avião e o método Mover

Herança e Interfaces

- Herança Múltipla: quando uma classe pode herdar métodos e atributos de várias classes
- Não existe herança múltipla em Java, para evitar erros e diminuir a complexidade da programação
- Uma classe pode herdar apenas de uma outra classe
- Todas as classes herdam da classe Object
- Classe totalmente abstrata: Interface

Exemplo - Java

```
class Pessoa{
    public String nome;
    public char sexo;
    public Data dataNasc;
        ----
}
```

```
public class Ator extends Pessoa{
        public String contrato;
        /* campos herdados
        public String nome;
        public char sexo;
        public Date dataNasc; */
    -----}
```

```
public class Aluno extends Pessoa{
        public long matric;
         /* campos herdados
        public String nome;
        public char sexo;
        public Data dataNasc; */
    -----}
```

Interface

- Interface: Contrato(s) que uma classe deve respeitar. Isto é, um conjunto de métodos que a classe não pode deixar de implementar.

```
interface Veiculo {
   void mover();
    double velocidade;
}
```

Herança e Interfaces

- Uma classe em Java pode implementar várias interfaces

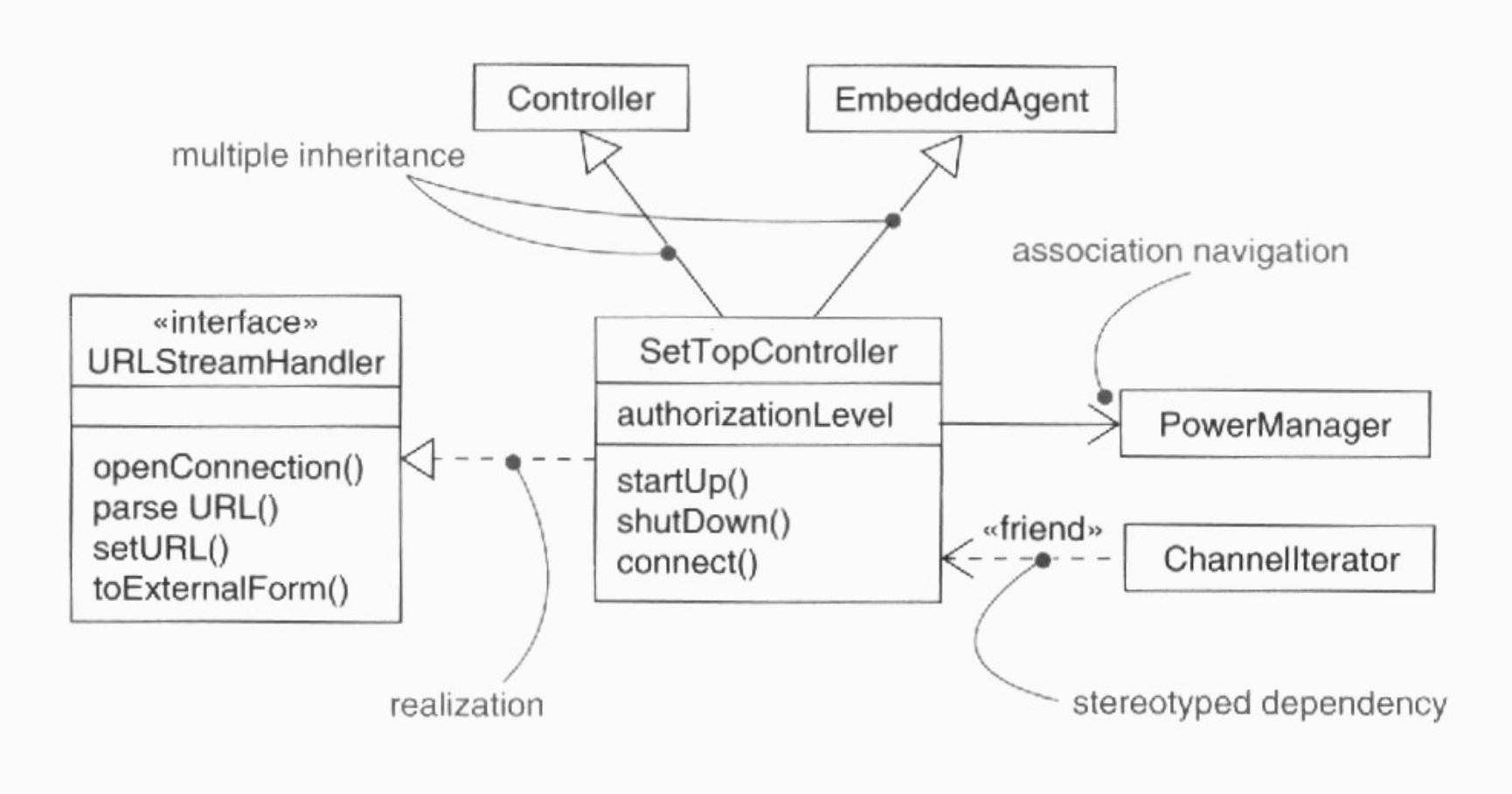
```
public class Class1 extends Class2 implements
Interface1,Interface2,Interface3 {

    ......

    ....

    }
```

Outras Relações: Realização



Resumo até agora

- Componentes de uma classe
  - Métodos
  - Atributos(variáveis)
  - Relações
- Relações entre Classes
  - Herança
  - Agregação
  - Associação
  - Realização(Classe/Inteface)

Sumário

- Introdução
- Conceitos Básicos
  - Nomenclatura básica em OO
  - Variáveis e Instâncias
  - Métodos
  - Construtores
  - Herança e Polimorfismo
- **Introdução a linguagem Java**
  - Primeiros Programas
  - Fundamentos
  - Tratamento de Erros
  - Coleções

Java x C++

- **Similarities with C++**
  - User-defined classes can be used the same way as built-in types.
  - Basic syntax
- **Differences from C++**
  - Methods (member functions) are the only function type
  - Object is the topmost ancestor for all classes
  - All methods use the run-time, not compile-time, types (i.e. all Java methods are like C++ virtual functions)
  - The types of all objects are known at run-time
  - All objects are allocated on the heap (always safe to return objects from methods)
  - Single inheritance only

Criando Programas java

- **Name of file must match name of class**
  - It is case sensitive, even on Windows
- **Processing starts in main**
  - `public static void main(String[] args)`
  - Routines usually called "methods," not "functions."
- **Printing is done with System.out**
  - System.out.println, System.out.print
- **Compile with "javac"**
  - Open DOS window; work from there
  - Supply full case-sensitive file name (with file extension)
- **Execute with "java"**
  - Supply base class name (no file extension)

Exemplo

- **File: HelloWorld.java**

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world.");
  }
}
```

- **Compiling**

```
DOS> javac HelloWorld.java
```

- **Executing**

```
DOS> java HelloWorld
Hello, world.
```

Exemplo de Orientação a Objetos - Java

- Classe

```
public class Pessoa {
  private int idade; private String nome;
  private boolean sexo; // Verdadeiro para mulheres
 private boolean ehResponsavel() {
   if(idade>21 )
         return true;
   if(idade>18 && !sexo)
          return true;
    else
         return false;
}
public Pessoa(String n, int id, boolean sex)      {
              nome=n; idade=id; sexo=sex;
}
}.....
```

Criando Classes em Java

- Convenção de Nomes em Java
  - Classes iniciam com letras maiúsculas
  - Métodos, atributos e variáveis locais iniciam com minúsculas
- Declaração de Classes

```
public class MyClasse {
.....
}
```

- Criando Objetos a partir de Classes
  - Uso do “new”

Orientação a Objetos em Java

- Instâncias
  - String teste=new String(60);
  - String teste2=“Isto é um teste”;
  - ServerSocket servidor= new ServerSocket(25);
- Construtores e destrutores
  - Os construtores são similares ao C++
  - Não há destrutores em Java.

Exemplo de Orientação a Objetos - Java

- Objetos
  - Pessoa a=new Pessoa(“Joao”,25);
  - Pessoa b= new Pessoa(“Ana”,23);
  - Pessoa c=new Pessoa(“Carla”,28);
  - d=b;
- Atributos de uma classe
  - a.idade == ?
  - d.nome = ?

Formato de uma Definição de Classe em Java

```
modifier class Classname {

  modifier data-type field1;
  modifier data-type field2;
  ...
  modifier data-type fieldN;

  modifier Return-Type methodName1(parameters) {
    //statements
  }

  ...

  modifier Return-Type methodName2(parameters) {
    //statements
  }
}
```

Acessando variáveis de instância

- Use um ponto entre o nome da variável e o campo
  - objectName.fieldName;
- Por exemplo, usando a classe Point da biblioteca Java
  - Point p=new Point(2,3); //criação de objeto p
  - int x2= p.x*p.x; // x2 é 4
  - int xPlusY=p.x+p.y; // xPlusY é 5
  - p.x=3;
  - x2=p.x* p.x; // x2 agora é 9
- Dentro de um objeto, seus métodos podem acessar as variáveis de instância (e de Classe) sem utilizar o ponto

Exemplo – Orientado a Objetos?

```
class Ship1 {
  public double x, y, speed, direction;
  public String name;
}

public class Test1 {
  public static void main(String[] args) {
    Ship1 s1 = new Ship1();
    s1.x = 0.0;
    s1.y = 0.0;
    s1.speed = 1.0;
    s1.direction = 0.0;    // East
    s1.name = "Ship1";
    Ship1 s2 = new Ship1();
    s2.x = 0.0;
    s2.y = 0.0;
    s2.speed = 2.0;
    s2.direction = 135.0; // Northwest
    s2.name = "Ship2";
```

Exemplos

```
        ...
        s1.x = s1.x + s1.speed
               * Math.cos(s1.direction * Math.PI / 180.0);
        s1.y = s1.y + s1.speed
               * Math.sin(s1.direction * Math.PI / 180.0);
        s2.x = s2.x + s2.speed
               * Math.cos(s2.direction * Math.PI / 180.0);
        s2.y = s2.y + s2.speed
               * Math.sin(s2.direction * Math.PI / 180.0);
        System.out.println(s1.name + " is at ("
                           + s1.x + "," + s1.y + ").");
        System.out.println(s2.name + " is at ("
                           + s2.x + "," + s2.y + ").");
    }
}
```

Resultado

- **Compiling and Running:**

```
javac Test1.java
java Test1
```

Output:

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

Problemas na Modelagem?

- Código específico a uma classe está escrito apenas nesta classe?

Métodos: Exemplo

```
class Ship2 {
  public double x=0.0, y=0.0, speed=1.0, direction=0.0;
  public String name = "UnnamedShip";

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }

  public void move() {
    double angle = degreesToRadians(direction);
    x = x + speed * Math.cos(angle);
    y = y + speed * Math.sin(angle);
  }

  public void printLocation() {
    System.out.println(name + " is at ("
                       + x + "," + y + ").");
  }
}
```

Exemplo (cont.)

```
public class Test2 {
  public static void main(String[] args) {
    Ship2 s1 = new Ship2();
    s1.name = "Ship1";
    Ship2 s2 = new Ship2();
    s2.direction = 135.0; // Northwest
    s2.speed = 2.0;
    s2.name = "Ship2";
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

- **Compiling and Running:**

```
javac Test2.java
java Test2
```

- **Output:**

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

Métodos Especiais: Construtores

- Construtores são métodos de uma classe que tem o mesmo nome desta classe e são chamados quando da criação de um objeto desta classe.
  - São usados para “inicializar” um objeto
  - São chamados apenas através de um comando “new”
  - Java provê automaticamente um construtor sem parâmetros, caso nenhum construtor seja declarado explicitamente
    - Por isso, podemos utilizar: Ship s1=new Ship(); Mesmo sem criar um construtor Ship()

Métodos Especiais: Construtores (2)

```
class Ship3 {
  public double x, y, speed, direction;
  public String name;

  public Ship3(double x, double y,
               double speed, double direction,
               String name) {
    this.x = x; // "this" differentiates instance vars
    this.y = y; //  from local vars.
    this.speed = speed;
    this.direction = direction;
    this.name = name;
  }

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }
```

Métodos Especiais: Construtores (3)

```
  public void move() {
    double angle = degreesToRadians(direction);
    x = x + speed * Math.cos(angle);
    y = y + speed * Math.sin(angle);
  }
  public void printLocation() {
    System.out.println(name + " is at ("
                       + x + "," + y + ").");
  }
}

public class Test3 {
  public static void main(String[] args) {
    Ship3 s1 = new Ship3(0.0, 0.0, 1.0,   0.0, "Ship1");
    Ship3 s2 = new Ship3(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

Métodos Especiais: Construtores (4)

- **Compiling and Running:**

```
javac Test3.java
java Test3
```

- **Output:**

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

A variável especial: this

- **The `this` object reference can be used inside any non-static method to refer to the current object**
- **The common uses of the `this` reference are:**

1. To pass a reference to the current object as a parameter to other methods

```
someMethod(this);
```

2. To resolve name conflicts
   - Using `this` permits the use of instance variables in methods that have local variables with the same name

– Note that it is only necessary to say `this.fieldName` when you have a local variable and a class field with the same name; otherwise just use `fieldName` with no `this`

Destrutores

- Não há destrutores em Java
  - Ao contrário de C++
- Garbage Collector
  - Coletor de lixo: Faz a limpeza (remoção da memória) de váriaveis que não serão mais utilizadas pelo programa.
  - Pode ser chamado através de :
    - System.gc();

Convenções e Boas Práticas em OO Resumo

- Código referente exclusivamente a uma classe deve ficar dentro desta classe
- Uma classe deve ter o menor número possível de métodos públicos, mas deve ter pelo menos um método público
- É uma boa prática evitar variáveis públicas. Prefira utilizar métodos para acessar as variáveis. Os métodos get e set.
- Use métodos construtores para inicializar objetos

Convenções e Boas Práticas em OO Resumo

- Classes devem iniciar com letras maiusculas, métodos, atributos e variáveis com letras minúsculas
- Métodos devem ter um tipo de retorno ou “void”
- Acesse atributos através de objectName.fieldName
- Acesse métodos através de objectName.methodName()
- Métodos estáticos não precisam de instâncias da classe
- Construtores são métodos especiais sem tipo de retorno

Convenções e características em Java Resumo

- A referência this aponta para o objeto atual
- Java faz seu próprio gerenciamento de memória e portanto não requer destrutores
- Java permite herança simples e o uso de interfaces
- As classes abstratas e interfaces não podem ser instanciadas
- Uma classe Java pode implementar várias interfaces

Fundamentos de Programação Java

- Comandos
- Sintaxe
- Estrutura
- Exemplos

Fundamentos de Programação Java

- Início e Fim de Blocos de Comandos
  - “{ “ e “}”
- comandos if, if-else, while, do-while, for e switch-case são idênticos ao C/ C++
- Comando break <label>;

Tipos básicos de Variáveis

Tipo	Tamanho
byte	1 bytes
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
char	2 bytes
boolean	1 bit

Variáveis

- Declaração
  - int inteiro; char letra;
  - byte apenasUmByte;
- Declaração e Inicialização
  - float saldoConta=0.0;
  - boolean serOuNaoSer=false;

Tipos Complexos

- int[] arrayInteiro= new int[40];
- char sTexto[]= new char[60]; // use String's

- Classe String
  - String s=new String("isto e uma string");
  - String s2="isto e outra string";
  - String c=s+s2;
  - System.out.println("S="+s+"S2="+s2+"c="+c);

Tipos Complexos

- **Arrays are accessed with []**
  - Array indices are zero-based
  - The argument to `main` is an array of strings that correspond to the command line arguments
    - args[0] returns first command-line argument
    - args[1] returns second command-line argument
    - Etc.
- **The length field gives the number of elements in an array**
  - Thus, `args.length` gives the number of command-line arguments
  - Unlike in C/C++, the name of the program is not inserted into the command-line arguments

Exemplo

- **File: ShowTwoArgs.java**

```
public class ShowTwoArgs {
  public static void main(String[] args) {
    System.out.println("First arg: " +
                       args[0]);
    System.out.println("Second arg: " +
                       args[1]);
  }
}
```

Exemplo

- **Compiling**

```
DOS> javac ShowTwoArgs.java
```

- **Executing**

```
DOS> java ShowTwoArgs Hello World
First args Hello
Second arg: Class

DOS> java ShowTwoArgs
[Error message]
```

Comandos...

- ***Comando*** := *Comando_simples* | *Comando_composto* | *Comando_condicional* | *Comando_iterativo* | *Comando_de_seleção*

- ***Comando_simples*** := *Comando_de_atribuição* | *Comando_de_entrada* | *Comando_de_saída* | *Chamada_de_subprograma* | *Comando_vazio*

- ***Comando_vazio*** := ;

- ***Comando_composto*** := { *Comando Comando . . . . . Comando* }

Comandos...

- ***Comando_condicional*** := **if** **(** *Expressão* **)** *Comando_1* | **if** **(** *Expressão* **)** *Comando_1* **else** *Comando_2*

- ***Comando_iterativo*** := *Comando_while | Comando_do | Comando_for*

Comando de Seleção em Java

```
switch ( expressão ) {
    case V11:
    case V12:
      .
      .
    case V1m: lista de comandos; break;
    case V21:
    case V22:
      .
      .
    case V2n: lista de comandos; break;
      .
      .
    case Vip: lista de comandos; break;
    default: lista de comandos;
}
```

Comandos...

- Comando atribuição condicional
  - Forma: *Expr1* **?** *Expr2* **:** *Expr3*
  - Calcula-se *Expr1*;
  - Se o valor for **Verdadeiro**, calcula-se o valor de *Expr2*, que será o valor da expressão condicional;
  - Se o valor for **Falso**, calcula-se o valor de *Expr3*, que será o valor da expressão condicional.

Comandos de repetição

- **while**

```
while (continueTest) {
  body;
}
```

- **do**

```
do {
  body;
} while (continueTest);
```

- **for**

```
for(init; continueTest; updateOp) {
  body;
}
```

Comandos Iterativos...

- **Comandos for e while versus Comando do-while**
- **Comando *break*:** Saída anormal de um comando iterativo ou comando de seleção mais interno.

```
while ( ..... ) {
  .....
  if ( ..... ) break;
  if ( ..... ) continue;
  .....
}
..... /* Proximo comando a ser executado depois do break */
.....
```

- **Comando *continue*:** encerra a iteração corrente e inicia a iteração seguinte.

Exemplo - while

```
public static void listNums1(int max) {
  int i = 0;
  while (i <= max) {
    System.out.println("Number: " + i);
    i++; // "++" means "add one"
  }
}
```

Exemplo – do/while

```
public static void listNums2(int max) {
  int i = 0;
  do {
    System.out.println("Number: " + i);
    i++;
  } while (i <= max);
                    // ^ Don't forget semicolon
}
```

Exemplo - for

```
public static void listNums3(int max) {
  for(int i=0; i<max; i++) {
    System.out.println("Number: " + i);
  }
}
```

Percorrendo todos os elementos de um array

```
public class LoopTest {
  public static void main(String[] args) {
    listNums1(5);
    listNums2(6);
    listNums3(7);
  }

  public static void listNums1(int max) {…}
  public static void listNums2(int max) {…}
  public static void listNums3(int max) {…}
}
```

Percorrendo todos os elementos de um array

```
public class ShowArgs {
  public static void main(String[] args) {
    for(int i=0; i<args.length; i++) {
      System.out.println("Arg " + i +
                         " is " +
                         args[i]);
    }
  }
}
```

Comando condicional - if

- **Single Option**

```
if (boolean-expression) {
   statement;
}
```

- **Multiple Options**

```
if (boolean-expression) {
   statement1;
} else {
   statement2;
}
```

Expressões Condicionais

- **==, !=**
  - Equality, inequality. In addition to comparing primitive types, == tests if two objects are identical (the same object), not just if they appear equal (have the same fields). More details when we introduce objects.
- **<, <=, >, >=**
  - Numeric less than, less than or equal to, greater than, greater than or equal to.
- **&&, ||**
  - Logical AND, OR. Both use short-circuit evaluation to more efficiently compute the results of complicated expressions.
- **!**
  - Logical negation.

Exemplos

```
public static int max2(int n1, int n2) {
  if (n1 >= n2)
    return(n1);
  else
    return(n2);
}
```

Comparação de Strings

```
public static void main(String[] args) {
  String match = "Test";
  if (args.length == 0) {
    System.out.println("No args");
  } else if (args[0] == match) {
    System.out.println("Match");
  } else {
    System.out.println("No match");
  }
}
```

- **Prints "No match" for *all* inputs**
  - Fix:

```
if (args[0].equals(match))
```

Criando Arrays

- **Step 1: allocate an array of references:**
  type[] var = new *type*[size];
- **Eg:**
  int[] values = new int[7];
  Point[] points = new Point[someArray.length];
- **Step 2: populate the array**
  points[0] = new Point(...);
  points[1] = new Point(...);
  ...
  Points[6] = new Point(…);
- **If you fail to populate an entry**
  - Default value is 0 for numeric arrays
  - Default value is null for object arrays

Arrays de várias dimensões

```
int[][] twoD = new int[64][32];

String[][] cats = { { "Caesar",  "blue-point" },
                    { "Heather", "seal-point" },
                    { "Ted",     "red-point"  } };
```

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

Exemplo

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

    int[][] triangle = new int[10][];

    for(int i=0; i<triangle.length; i++) {
      triangle[i] = new int[i+1];
    }

    for (int i=0; i<triangle.length; i++) {
      for(int j=0; j<triangle[i].length; j++) {
        System.out.print(triangle[i][j]);
      }
      System.out.println();
    }
  }
}
```

Resultado

```
> java TriangleArray

0
00
000
0000
00000
000000
0000000
00000000
000000000
0000000000
```

Tratamento de Erros: Tradicional

- O tratamento de erros em linguagens sem Exceções, gera um código “sujo” com código tratamento de erro:
  - ret=funcao1();
  - if(ret==ERRO)
    - //Trata erro

    ret=funcao2();
  - if(ret==ERRO)

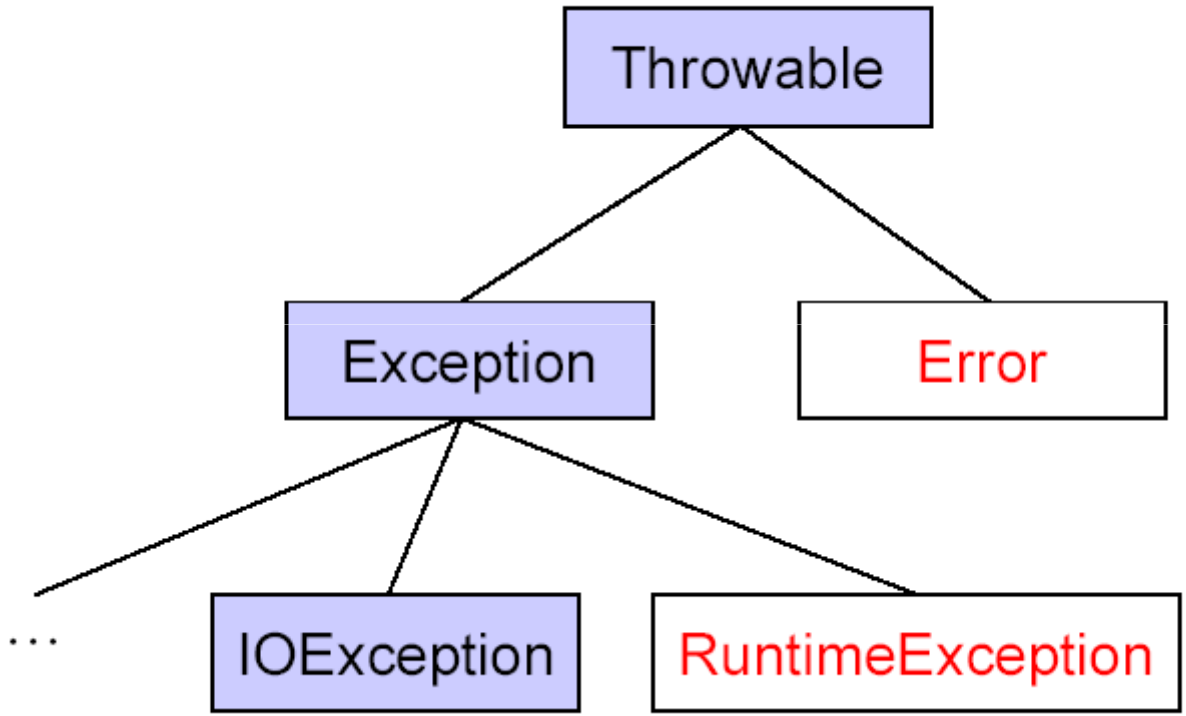
    //Trata Erro 2

Tratamento de Erros: Exceções

- Em Java, o sistema de tratamento de erros é baseado exceções
  - Exceções devem ser tratados em blocos **try/catch**
  - Quando ocorre uma exceção esta é direcionada para o correspondente **catch**
- Formato:

```
try {
  statement1;
  statement2;
  ...
} catch(SomeException someVar) {
  handleTheException(someVar);
}
```

Diagrama Simplicado de Exceções



Try-catch

- Um bloco try pode ter associados vários blocos catch

```
try {
   ...
} catch (ExceptionType1 var1) {
   // Do something
} catch (ExceptionType2 var2) {
   // Do something else
}
```

- A exceção será tratado pelo bloco catch mais específico
- Caso não seja encontrado algum apropriado, a exceção será direcionada para blocos try mais externos
  - Caso não seja encontrado nenhum try apropriado dentro do método, este irá jogar a exceção

Um exemplo de Try-catch

```
...
BufferedReader in = null;
String lineIn;
try {
  in = new BufferedReader(new FileReader("book.txt"));
  while((lineIn = in.readLine()) != null) {
    System.out.println(lineIn);
  }
  in.close();
} catch (FileNotFoundException fnfe ) {
  System.out.println("File not found.");
} catch (EOFException eofe) {
  System.out.println("Unexpected End of File.");
} catch (IOException ioe) {
  System.out.println("IOError reading input: " + ioe);
  ioe.printStackTrace(); // Show stack dump
}
```

A cláusula finally

- Ao final de um conjunto de blocos catch pode-se, opcionalmente, incluir uma cláusula **finally**. Caso nenhum bloco catch, seja executado o **finally** será sempre executado

```
try {
   ...
} catch (SomeException someVar) {
   // Do something
} finally {
   // Always executed
}
```

Jogando exceções

- **If a potential exception is not handled in the method, then the method must declare that the exception can be thrown**

```
public SomeType someMethod(...) throws SomeException {
  // Unhandled potential exception
  ...
}
```

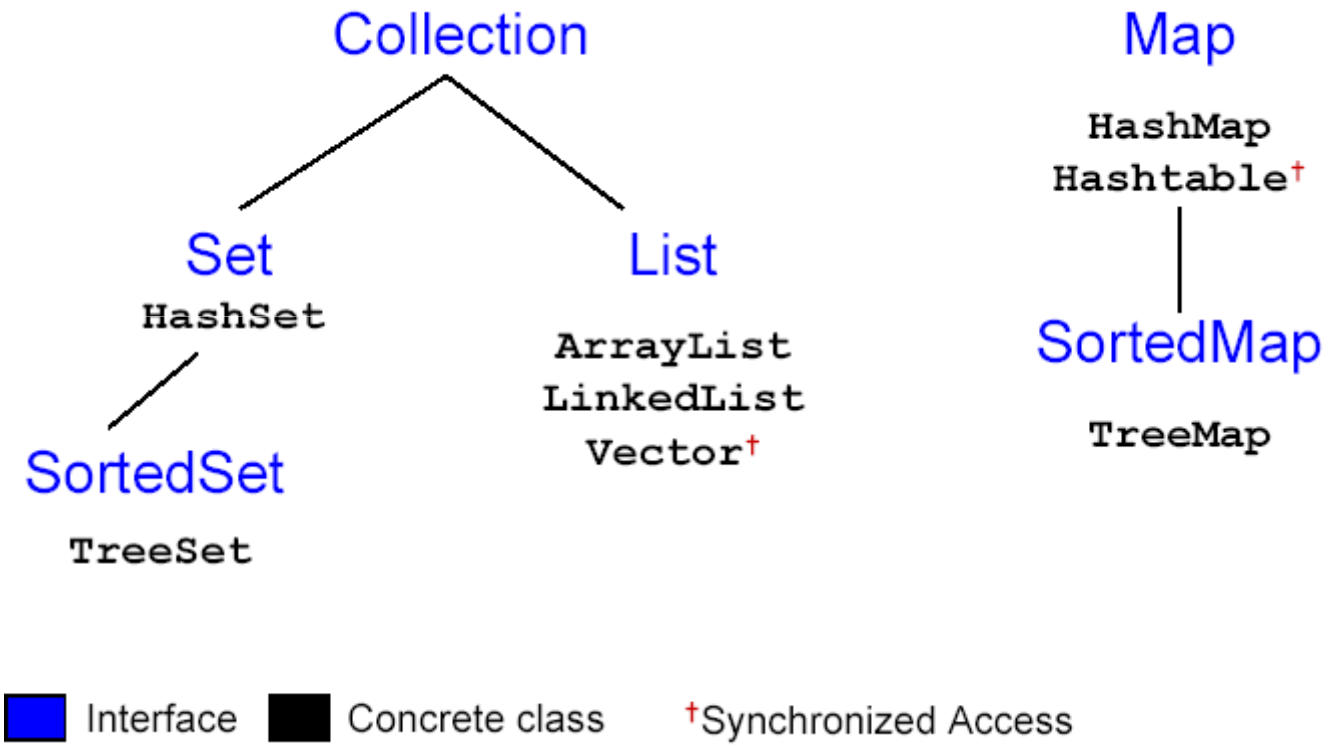
  - Note: Multiple exception types (comma separated) can be declared in the `throws` clause

- **Explicitly generating an exception**

```
throw new IOException("Blocked by firewall.");

throw new MalformedURLException("Invalid protocol");
```

Estruturas de Dados no Java 2



Collection Interfaces

- **Collection**
  - Abstract class for holding groups of objects
- **Set**
  - Group of objects containing no duplicates
- **SortedSet**
  - Set of objects (no duplicates) stored in ascending order
  - Order is determined by a `Comparator`
- **List**
  - *Physically* (versus logically) ordered sequence of objects
- **Map**
  - Stores objects (unordered) identified by unique keys
- **SortedMap**
  - Objects stored in ascending order based on their key value
  - Neither duplicate or `null` keys are permitted

Duas Estruturas de Dados Muito Úteis

- Vector
  - Um array de ***Object*** de tamanho variável
  - Tempo para acessar um objeto é independente da sua posição na lista
  - No jdk 1.2 ou superior, pode-se utilizar ArrayList
  - ArrayList não é sincronizado (thread-safe), por isso tende a ser mais rápido
- Hashtable
  - Armazena pares: nome-valor como Object
  - Valores não podem ser nulos
  - No jdk 1.2 ou superior, pode-se utilizar HashMap
  - HashMap não é sincronizado (thread-safe), por isso tende a ser mais rápido

Métodos úteis em Vector

- **addElement/insertElementAt/setElementAt**
  - Add elements to the vector
- **removeElement/removeElementAt**
  - Removes an element from the vector
- **firstElement/lastElement**
  - Returns a reference to the first and last element, respectively (without removing)
- **elementAt**
  - Returns the element at the specified index
- **indexOf**
  - Returns the index of an element that equals the object specified
- **contains**
  - Determines if the vector contains an object

Utilizando Vector

- **elements**
  - Returns an `Enumeration` of objects in the vector

```
Enumeration elements = vector.elements();
while(elements.hasMoreElements()) {
  System.out.println(elements.nextElement());
}
```

- **size**
  - The number of elements in the vector
- **capacity**
  - The number of elements the vector can hold before becoming resized

Métodos úteis em Hashtable

- **put/get**
  - Stores or retrieves a value in the hashtable
- **remove/clear**
  - Removes a particular entry or all entries from the hashtable
- **containsKey/contains**
  - Determines if the hashtable contains a particular key or element
- **keys/elements**
  - Returns an enumeration of all keys or elements, respectively
- **size**
  - Returns the number of elements in the hashtable
- **rehash**
  - Increases the capacity of the hashtable and reorganizes it

Exemplo de Uso de um Hashtable

```
import java.util.Hashtable;

public class ExemploHashtable {
    public static void main(String[] args) {
            Hashtable numbers = new Hashtable();
        numbers.put("one", new Integer(1));
        numbers.put("two", new Integer(2));
        numbers.put("three", new Integer(3));
        String key="three";
        Integer n = (Integer)numbers.get(key);
        if (n != null) {
            System.out.println(key+" = " + n);
        }
    }
}
```

Resultado

- >three = 3

Exemplo de Uso de uma Coleção Vector

```
import java.util.Iterator;
import java.util.Vector;

public class Colecoes {

      public static void main(String[] args) {
            Vector vetStrings=new Vector();
            for(int i=1;i<=5;i++)
              vetStrings.add("Linha "+i);
            //Laços de Iteração
            for (Iterator iter = vetStrings.iterator(); iter.hasNext();) {
                        String element = (String) iter.next();
                         System.out.println(element);
            }
      }
}
```

Resultado

>Linha 1
Linha 2
Linha 3
Linha 4
Linha 5

Classes Genéricas no JDK 5.0

- Classe genéricas: classes que podem ser parametrizadas para trabalharem sobre classes específicas
  - Tipos parametrizáveis: (Design Patterns, GoF)
  - Templates: C++
  - Classes genéricas: Java,C#

Avanços em Collections no JDK 5.0

- Coleções genéricas:

```
Vector<String> vetStrings=new Vector<String>();
```

- Laços de Iteração Aprimorados

```
for(String element: vetStrings) {
  System.out.println(element);
}
```

Exemplo de Uso de uma Coleção Genérica

```
import java.util.Vector;

public class ColecoesGenericas {

    public static void main(String[] args) {

        Vector<String > vetStrings=new Vector<String>();
        for(int i=1;i<=5;i++)
          vetStrings.add("Linha "+i);
        //Laços de Iteração Aprimorados
        for(String element: vetStrings) {
          System.out.println(element);
        }
    }
}
```

Resultado

>Linha 1
Linha 2
Linha 3
Linha 4
Linha 5

HashTable Genérico

```
import java.util.Hashtable;

public class ExemploHashtableGenerico {
    public static void main(String[] args) {
            Hashtable<String,Integer> numbers = new Hashtable<String,Integer>();
        numbers.put("one", new Integer(1));
        numbers.put("two", new Integer(2));
        numbers.put("three", new Integer(3));
        String key="three";
        Integer n = numbers.get(key);
        if (n != null) {
            System.out.println(key+" = " + n);
        }
    }
}
```

Resultado

- >three = 3

Classes Wrapper

- Todo tipo primitivo em Java, tem uma classe correspondente que pode encapsula-lo

Primitive Data Type	Corresponding Object Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Uso de Classes Wrapper

- Define constantes úteis, por exemplo:

```
Integer.MAX_VALUE
Float.NEGATIVE_INFINITY
```

- Conversão entre tipos de dados
  - Utilize parseXXX para fazer conversões:

```
try {
  String value = "3.14e6";
  Double d = Double.parseDouble(value);
} catch (NumberFormatException nfe) {
  System.out.println("Can't convert: " + value);
}
```

Outras conversões...

Data Type	Convert String using either ...
byte	`Byte.parseByte(string)`
	`new Byte(string).byteValue()`
short	`Short.parseShort(string)`
	`new Short(string).shortValue()`
int	`Integer.parseInteger(string)`
	`new Integer(string).intValue()`
long	`Long.parseLong(string)`
	`new Long(string).longValue()`
float	`Float.parseFloat(string)`
	`new Float(string).floatValue()`
double	`Double.parseDouble(string)`
	`new Double(string).doubleValue()`

Resumindo...

- Laços de repetição, comandos condicionais e o acesso a arrays é feito em Java da mesma forma que em C++
- String é uma classe em Java, não um array de caracteres
- Nunca compare Strings usando ==
- O tratamento de erros é feito através de exceções (blocos try-catch-finally)
- Vector, ArrayList e HashMap são estruturas de dados muito úteis disponíveis em Java
  - Podem manter um número arbitrário de elementos

Sumário de Hoje

- Introdução
- Conceitos Básicos
  - Nomenclatura básica em OO
  - Variáveis e Instâncias
  - Métodos
  - Construtores
  - Herança e Polimorfismo
- Introdução a linguagem Java