CENTRO PAULA SOUZA

GOVERNO DO ESTADO
SÃO PAULO

Apostila de

Programação Orientada a Objeto (POO)

Prof. Francesco Artur Perrotti

Fatec Americana

Revisão: Janeiro de 2014

Programação Orientada a Objeto (POO)

Prof. Francesco Artur Perrotti – Fatec Americana

1) Classes e Objetos

Classe pode ser definida como uma categoria de objetos. No nosso dia-a-dia, usamos as classes para facilitar o entendimento dos objetos que nos cercam.

Esse é o modo como nosso cérebro funciona. Estamos sempre colocando os objetos dentro de categorias. Fazemos isso para reconhecer os objetos e determinar suas características e comportamento. Toda vez que olhamos para qualquer objeto, automaticamente nosso cérebro o classifica, ou seja, o coloca dentro de alguma classe de objetos que já conhecemos. Nós só podemos entender e reconhecer um objeto qualquer depois que ele foi devidamente classificado. Quando não conseguimos classificar um objeto, não podemos saber o que ele é e nem entendê-lo. Este processo de classificação é automático e inconsciente, normalmente não o percebemos, a não ser quando ele falha.

Note que classe é um conceito, um modelo, um conjunto de definições. Não tem existência física. Já os objetos existem fisicamente no universo.

Por exemplo, ao vermos um carro estacionado na rua, reconhecemos esse objeto porque automaticamente classificamos esse objeto como pertencente à classe Carro. Uma vez que ele foi classificado, podemos inferir suas características e seu comportamento, mesmo que elas não sejam aparentes. Sabemos por exemplo que esse carro é capaz de acelerar, de frear, de virar para a esquerda ou para a direita, que é capaz de acender os faróis e muitas outras ações que ele é capaz de realizar, mesmo não vendo essas ações sendo executadas naquele momento. Sabemos disso porque sabemos que todos os objetos que pertencem à categoria Carro são capazes destas ações, portanto se aquele objeto que vimos é um carro, então ele também é capaz destas ações.

Da mesma forma que a classe determina as ações que um objeto é capaz de realizar (seu comportamento), também determina o conjunto de atributos dos objetos. Ainda usando o exemplo do carro, sabemos que o carro que vimos estacionado tem um modelo, ano de fabricação, cor, número de chassi, número de Renavam, proprietário, placa e muitos outros atributos que embora não sejam aparentes, sabemos que eles existem, porque todos os carros têm estes atributos. Embora cada carro possa ter um valor diferente para cada atributo, todos têm o mesmo conjunto de atributos.

Em resumo, objetos têm atributos (características) e comportamento (ações que são capazes de realizar). Objetos da mesma classe têm o mesmo conjunto de atributos e o mesmo comportamento.

Instância

A palavra instância tem muito significados na língua portuguesa. Pode significar "insistência" ou "pedido insistente", também é usada para designar o nível de autoridade em uma instituição jurídica. Na programação orientada a objeto, instância tem o sentido de "manifestação". A classe, que é algo abstrato, se manifesta concretamente através dos objetos criados a partir dela. A classe é tida como um molde, um modelo a partir do qual os objetos são criados. Portanto o termo "instanciar uma classe" significa criar um objeto daquela classe. Como na programação orientada a objeto todos os objetos pertencem a alguma classe, então todos os objetos são instâncias de suas respectivas classes.

Generalização/Especialização

Classes podem ser divididas em subclasses de modo a formar uma hierarquia onde no topo estão as classes mais gerais e na base as classes mais especializadas.

Como citado antes, um carro qualquer pertence à classe Carros, entretanto essa não é a única classe a que pertence. Podemos imaginar uma classe mais geral que inclua não só os carros, mas também outros veículos, como motocicletas, caminhões, trens, etc. Note que um carro continua pertencendo à classe Carros, mas além disso também pode pertencer à classe Veículos. Podemos pensar na classe Carros como uma subclasse da classe Veículos. Neste caso, Veículos é uma classe mais geral e Carros uma classe mais especializada. Veja o exemplo na figura 1:

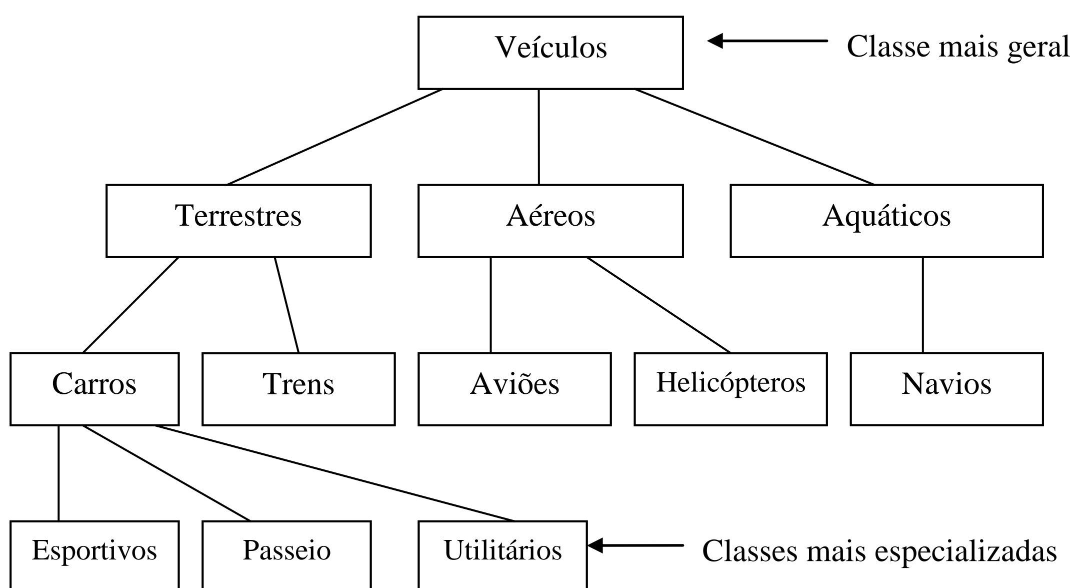

Figura 1 – Exemplo de hierarquia de classes

Processo de generalização – Consiste em detectar características e comportamentos comuns entre duas classes para criar outra classe que contenha apenas o que é comum às duas classes. Esta nova classe será ancestral das outras.

Processo de Especialização – Consiste em detectar características e comportamentos diferentes entre objetos da mesma classe e criar novas classes descendentes mais especializadas.

Encapsulamento

O termo encapsulamento vem do verbo encapsular, que significa literalmente colocar dentro de uma cápsula, um recipiente. Na POO, este termo se refere ao fato que é possível esconder os detalhes da implementação dos objetos dos usuários destes. Neste caso, o objeto passa a funcionar como uma caixa preta, onde os mecanismos internos ficam escondidos e o acesso é apenas aos **métodos** que ativam estes mecanismos.

Como exemplo, vamos considerar um telefone celular. Todos nós sabemos como fazer ligações em um celular. Sabemos que basta digitar o número e apertar o botão *Send*, entretanto poucos conhecem os mecanismos envolvidos em uma ligação telefônica. Não sabemos como funcionam os circuitos internos de um celular e nem como ele faz a conexão para estabelecer a ligação. Mas isso não nos impede de fazer ligações, porque conhecemos o **método** oferecido pelo celular, ou seja, o teclado numérico e o botão *Send*. Celulares diferentes usam circuitos e mecanismos diferentes internamente, mas todos oferecem o mesmo método, ou seja, todos têm um teclado numérico e um botão Send. Portanto somos capazes de fazer ligações em qualquer celular, não importando como são construídos internamente.

Neste exemplo, o celular pode ser considerado uma caixa preta, porque não temos acesso aos mecanismos internos dele, nosso acesso é limitado à sua **interface**, ou seja, o conjunto de métodos que ele oferece para ativar e controlar seus mecanismos internos.

Voltando à POO, como um objeto guarda na mesma estrutura os dados (atributos) e as funções que os processam (ações – métodos) pode-se pensar em atributos e métodos privados, ou seja, dados e funções que só podem ser manipulados pelas funções que estão dentro da estrutura. Desta maneira é possível formar uma camada protetora nos dados e evitar atribuições desastradas que comprometeriam o funcionamento do programa. Os defensores mais ortodoxos da POO dizem que todos os dados de um objeto deveriam ser privados e o número de funções públicas deve ser o menor possível. O Java implementa este conceito e oferece atributos e métodos públicos (*public*), privados (*private*) e protegidos (*protected*).

Herança

Como citado, uma classe geral tem atributos e métodos que são comuns a todas as suas subclasses mais especializadas. Ao especializar uma classe, a nova classe **herda** automaticamente todos os recursos públicos (métodos e atributos) da classe original. Essa capacidade é chamada de ***Herança***.

A herança permite que o código já escrito para uma classe seja facilmente reaproveitado quando é necessário criar uma nova classe mais especializada.

Uma subclasse sempre herda todos os recursos da classe original. Neste caso a subclasse é chamada de **classe descendente** e a classe original de **classe ancestral**. No exemplo da figura 1, a classe Veículos é ancestral de todas as outras classes. A classe Terrestres tem como descendentes as classes Carros, Trens e todas as classes descendentes delas, ou seja: Esportivos, Passeio e Utilitários. Já a classe Esportivos tem como ancestrais as classes Carros, Terrestres e Veículos, portanto herda todos os recursos presentes nestas três classes.

Observação: No java, a classe ancestral é chamada de super-classe, ou simplesmente **super**.

Polimorfismo

Refere-se à capacidade de objetos de classes diferentes responderem de formas diferentes a um mesmo método, conforme sua classe, a fim de obter o mesmo resultado ou resultado equivalente. Por exemplo, vamos considerar a hierarquia da figura 1. Sabemos que todos os veículos devem ser capazes de frear e de acelerar. Na verdade, estas ações fazem parte da própria definição do conceito de veículo. Um objeto que não seja capaz de acelerar e frear, não é um veículo. Então todo e qualquer objeto classificado como veículo tem que ser capaz de realizar essas ações, portanto deve oferecer métodos para desencadeá-las. Entretanto, sabemos que frear um trem é um processo completamente diferente de frear um navio ou um avião, os mecanismos envolvidos para executar esta ação são completamente diferentes, mas o resultado da ação será sempre o mesmo: o veículo diminui sua velocidade ou para completamente.

Uma consequência deste conceito é a existência de classes abstratas. Uma classe abstrata pode conter métodos que estão declarados, mas não estão implementados. A implementação é feita em alguma classe descendente, conforme as características específicas daquela classe. Então, a classe ancestral define uma linguagem comum (conjunto de métodos) para todas as classes descendentes e estas por sua vez implementam estes métodos de acordo com seus recursos e características. Classes abstratas não podem ser instanciadas. Isso significa que não é possível criar objetos destas classes, elas podem apenas ser ancestrais de outras classes.

2) Tipos e variáveis

Tipos primitivos

São tipos de dados que permitem criar variáveis que não são objetos.

int - inteiro de 4 bytes com sinal.

short - inteiro de 2 bytes com sinal.

long - inteiro de 8 bytes com sinal.

byte - inteiro de 1 byte com sinal. (armazena valores de -127 a +128)

char - 2 bytes - Pode trabalhar com a tabela de caracteres UNICODE ou ASCII.
Também pode ser usado como um inteiro sem sinal.

float - ponto flutuante de precisão simples (4 bytes).

double - ponto flutuante de precisão dupla (8 bytes).

boolean - 1 byte para armazenar "true" ou "false".

Importante: Strings em Java não são variáveis de um tipo primitivo como no Pascal, e nem são tratadas como um array de caracteres como no C. No java, strings são instâncias da classe String.

Variáveis de valor e referência

Em relação ao conteúdo de uma variável, podemos dividir as variáveis do Java em dois grupos: **Variáveis de valor** e **variáveis de referência**.

As variáveis de valor estão sempre associadas a algum tipo primitivo e podem armazenar internamente um valor desse tipo. Estas variáveis já nascem prontas para uso, uma vez que foram declaradas já podem receber valores e participar de expressões ou atribuições.

Já as variáveis de referência (também chamadas de **variáveis de instância**) armazenam uma referência a um objeto de alguma classe, então estão sempre associadas a uma classe. O que é importante notar é que os objetos não ficam armazenados dentro das variáveis, eles estão na memória, o que estas variáveis armazenam é uma referência à posição de memória que o objeto ocupa, ou seja, estas variáveis são na verdade ponteiros e armazenam endereços de memória. Por conseqüência, uma variável de referência precisa receber uma referência válida antes que possa ser efetivamente usada como uma referência a um objeto, não basta apenas ser declarada.

É fácil saber ser uma variável é de valor ou referência. Variáveis de valor só podem ser de tipos primitivos (listados acima). Qualquer outro tipo é variável de referência.

3) Estrutura das classes

Todas as classes no Java seguem a mesma estrutura básica mostrada abaixo:

```
<modificador de acesso> class <nome da classe>{

          // corpo da classe

           <atributos>

           <construtores>

           <métodos>
}
```

Modificadores de acesso para a classe

O modificador de acesso define a visibilidade da classe em relação ao resto do programa. Para as classes existem duas opções possíveis:

public: é o único modificador permitido para classes. Se a classe é pública, ela é visível para todas as outras classes. Uma classe pública deve ser definida dentro de um arquivo fonte com o mesmo nome que a classe. **Um arquivo fonte pode conter muitas classes, mas apenas UMA classe pode ser pública.**

sem modificador (omitido): Neste caso a classe fica com o acesso padrão, então só pode ser acessada pelas classes dentro do mesmo pacote.

Modificadores de acesso para atributos e métodos

Atributos, métodos e construtores também podem ter a sua visibilidade afetada por um modificador, mas para estes elementos o java define quatro opções:

public: Sem restrição. Pode ser acessado por qualquer classe.

protected: pode ser acessado pelas classes do mesmo pacote e pelas classes descendentes.

private: visível apenas para a própria classe.

sem modificador (omitido): visível apenas para as classes do mesmo pacote.

Convenções para os nomes das classes, métodos e atributos

O Java utiliza algumas convenções para os nomes de classes e métodos, que embora não sejam obrigatórias, é altamente recomendado segui-las a fim de facilitar o entendimento do programa.

A primeira e mais importante é que todo nome de classe inicia com uma letra maiúscula seguida por letras minúsculas (exemplo: **Funcionario**). Se o nome da classe for formado por duas ou mais palavras juntas, então a primeira letra de cada palavra fica em maiúscula (Exemplo: **FuncionarioProducao**).

Os nomes de métodos iniciam com letras minúsculas (exemplo: **get**), mas se forem formados por duas ou mais palavras, as outras palavras iniciam com letras maiúsculas (exemplo: **getNome**).

Já os atributos são sempre escritos com minúsculas.

Estas convenções foram sugeridas pelos projetistas da linguagem Java e podem ser encontradas no documento Java Code Conventions.

4) Getters e Setters

Os atributos de uma classe muito raramente são públicos. Um atributo público pode ser modificado a partir de qualquer ponto do programa e a classe não tem nenhum controle sobre as alterações em seu conteúdo, portanto não pode garantir sua validade. Tornar um atributo público significa abrir mão do controle sobre esse atributo e isso vai completamente contra toda a filosofia e o objetivo da POO. Então o acesso direto aos atributos deve sempre ser restrito associando a eles os modificadores **private** ou **protected**. Isso impede que o conteúdo dos atributos seja alterado sem o controle da classe.

Por outro lado, atributos não são constantes e pode haver a necessidade de modificar seu conteúdo. Então deve existir uma maneira permitir a alteração do conteúdo dos atributos por entidades externas à classe, desde que a classe tenha controle suficiente sobre essa alteração para poder garantir a validade do atributo.

A maneira de resolver essa situação é criar métodos especializados em dar um acesso controlado aos atributos. Estes métodos são chamados de **getters** e **setters**. O **get** de um atributo retorna o seu conteúdo e o **set** permite sua alteração. Note que se o **set** de um atributo for o único modo de uma entidade externa à classe modificar o conteúdo dele, a classe pode implementar no setter qualquer validação que seja necessária e decidir se aceita ou não a alteração que está sendo feita. Veja o exemplo abaixo, usando a classe Pessoa.

Exemplo:

```
public class Pessoa {
    private String nome;
    private int idade;

    // Construtor
    public Pessoa(String vNome, int vIdade) {
        nome  = vNome;
        idade = vIdade;
    }
```

```
    // Getters
    public String getNome() {
        return nome;
    }

    public int getIdade() {
        return idade;
    }

    // Setters
    public void setNome(String value){
        if(value != null) nome = value;
    }

    public void setIdade(int value){
        if(value >= 0) idade = value;
    }

} // classe Pessoa
```

No exemplo acima, os dois atributos da classe são privados, portanto não é possível ter acesso direto a eles de fora da classe. Apesar disso é possível que outras classes possam obter o valor dos atributos através dos métodos **getNome** e **getIdade**. Caso outra classe precise alterar o valor de algum atributo, isso é possível usando os métodos **setNome** ou **setIdade**, ***desde que as condições impostas pelos métodos sejam satisfeitas.*** O método **setNome** não aceita uma string nula (vazia) e o método **setIdade** não aceita um valor negativo.

Dicas e casos especiais

Use o netbeans: O netbeans sabe como programar getters e setters básicos, então deixe-o fazer isso. No arquivo fonte da classe, clique com o botão direito em qualquer linha dentro da classe, mas fora de qualquer método (por exemplo entre dois métodos), escolha a opção "Inserir código" e depois "Getter e setter". O netbeans irá mostrar uma janela com todos os atributos da classe e então basta selecionar os atributos que o netbeans se encarrega de escrever o código básico dos getters e setters para os selecionados.

Mas cuidado!!! Não vá enfiando getters e setters públicos no programa sem necessidade. Só faça se for realmente necessário e depois de avaliar o impacto sobre o funcionamento da classe. Existem casos em que o atributo não pode ter getters ou setters (pelo menos não públicos), por que isso acaba com os controles internos da classe. Um exemplo típico é uma classe que gerencia uma conta corrente e tem como um de seus atributos o saldo da conta. A classe oferece métodos para depositar e para sacar dinheiro da conta e estes métodos atualizam o valor do saldo. Se o atributo para o saldo

tiver um setter, todo esse mecanismo de depósitos e saques perde o sentido. Neste caso o saldo não pode ter um setter.

Uso interno: Alguns atributos podem ser necessários apenas para controle interno da classe e não há motivo para que sejam acessados externamente. Neste caso não é necessário (e nem aconselhável) implementar getters e setters para eles, ou caso sejam implementados, devem ficar com acesso privado ou protegido. Só ofereça getters e/ou setters públicos se isso for realmente necessário.

Read only: Se quiser que algum atributo seja apenas para leitura, basta tornar o setter dele privado e deixar o getter público.

Atributos "virtuais": Getters e setters não precisam necessariamente ter correspondência com os atributos declarados na classe. Considere o getter abaixo para a classe Pessoa:

```
public class Pessoa {

    ...

    public String getFaixaEtaria() {
        if(idade < 12) return "Criança";
        else
            if(idade < 18) return "Adolescente";
            else
                return "Adulto";
    }

} // classe Pessoa
```

Neste exemplo, não existe realmente um atributo para a faixa etária da pessoa, essa informação é calculada a partir da idade. Mas as outras classes não sabem e não precisam saber disso.

Armadilha: Falando em informações calculadas, um erro bastante comum para quem está iniciando na POO é declarar atributos cujo conteúdo é resultado de um cálculo ou processo feito a partir de outros atributos. Se uma informação é resultado de um cálculo ou processo, então deve ser implementado um método que retorne essa informação e não armazená-la em um atributo, assim como foi feito no exemplo anterior para a faixa etária.

5) Construtores

São métodos especiais que servem para construir e inicializar instâncias de uma classe. Em geral estes métodos são responsáveis por atribuir valores iniciais para os atributos do novo objeto que está sendo criado, mas freqüentemente é necessário um processamento adicional que vai além da inicialização de atributos.

Toda classe **precisa** ter pelo menos um construtor para criar objetos. Se não for declarado um construtor explicitamente na classe, o Java vai criar um construtor padrão automaticamente. Nesse caso, o que o construtor padrão faz é atribuir o valor nulo correspondente a cada tipo de atributo (zero para os atributos numéricos, false para os booleanos e nulo para strings e outros objetos). É importante notar que o Java só criará o construtor padrão automaticamente se não for definido nenhum construtor para a classe.

Quando um construtor é implementado, tenha em mente que sua principal função é atribuir valores válidos para todos os atributos garantindo assim que o objeto já inicie com valores apropriados para seu correto funcionamento.

Importante:

- Todo construtor tem exatamente o mesmo nome que a classe.
- Faça construtores públicos. É possível a existência de construtores privados ou protegidos, mas são utilizados raramente em situações especiais.
- **Não tem retorno explícito (nem mesmo void).** Se for declarado algum tipo de retorno para o construtor, ele deixa de ser considerado um construtor pelo java.
- Para construir objetos, deve ser chamado através do operador **new**.

Exemplo de construção de um novo objeto:

```
UmaClasse vr; // vr é uma variável de referência

vr = new UmaClasse(); /* UmaClasse() é o construtor da classe UmaClasse*/
// A partir daqui, vr contém uma referência válida ao novo objeto.
```

No exemplo acima estamos construindo um novo objeto da classe **`UmaClasse`**. A primeira linha declara uma variável de referência (**`vr`**) para conter o novo objeto que será criado. Note que **`UmaClasse`** é o tipo dessa variável e também a classe que será instanciada.

A construção do objeto acontece realmente na segunda linha, na seguinte seqüência:

1) O operador **new** aloca memória suficiente para conter o objeto.
2) O construtor da classe (**UmaClasse()**) é chamado para inicializar os atributos do novo objeto.
3) O endereço inicial da memória alocada para o objeto é retornado para a variável de referência (**vr**).

Não confunda a variável de referência com o objeto em si. O objeto existe na memória, não está dentro da variável, a variável apenas faz referência ao objeto, ou seja, essa variável é na verdade um ponteiro para o objeto, não o próprio objeto.

Somente depois que o objeto foi construído é que é possível utilizar os métodos que ele tem disponíveis.

Um objeto sem referência, que não tem nenhuma variável apontando para ele, será automaticamente destruído pelo Java.

Exemplo da declaração do construtor na classe:

```
public class UmaClasse {
     //
     // declaração dos atributos
     //
     public UmaClasse (){  // este é o construtor
          //
          // inicialização dos atributos
          //
      }
     public void UmMetodo () {
          //
          // aqui um método da classe
          //
     }
}
```

Operador new

É o operador de instanciação do java, deve ser usado sempre que um novo objeto é criado. Exige um operando que é um construtor da classe que está sendo instanciada. Na pratica o que este operador faz é alocar memória para o novo objeto e chamar o construtor da classe. O resultado retornado é uma referência para o novo objeto que foi criado.

Exemplo:

```
Ponto pt; // variável de referência

pt = new Ponto (10, 20);
```

Sobrecarga de construtores

Sobrecarregar um método é criar duas ou mais versões do mesmo método. Isso também é possível com construtores. Uma classe pode ter dois ou mais construtores, desde que eles tenham listas de parâmetros diferentes. Neste caso, qual construtor será usado depende dos parâmetros usados na construção do objeto. Note que todos os construtores sempre têm o mesmo nome, que é o nome da classe.

Exemplo de classe com dois construtores

```
public class Pessoa {
    private String Nome;
    private int Idade;

    // Construtor padrão da classe
    public Pessoa() {
        Nome  = null;
        Idade = 0;
    }


     // Construtor com parâmetros da classe
    public Pessoa(String vNome, int vIdade) {
        Nome  = vNome;
        Idade = vIdade;
    }
}
```

Exemplo de construção com os dois construtores.

```
Pessoa ps1, ps2;

ps1 = new Pessoa();
ps2 = new Pessoa("Emerson",21);
```

Observações:

1. Se a classe só precisa do construtor padrão, não é necessário declara-lo. Se só é necessário zerar os atributos, deixe o Java fazer isso. Mas se a classe tiver algum outro construtor, então o Java não irá criar o construtor

padrão. O construtor padrão só é criado automaticamente pelo Java se não for definido nenhum construtor para a classe.

2. Por convenção os construtores são declarados antes de todos os outros métodos, logo depois da declaração dos atributos. Acredite, seguir as convenções vai facilitar muito tua vida como programador.
3. **Não declare nenhum tipo de retorno para um construtor nem mesmo o void.**

Copy constructor (construtor de cópia)

É bastante comum que as classes definam um ***copy constructor*** (construtor de copia) como opção para a construção de objetos. Um copy constructor serve para construir um objeto copiando os valores dos atributos de outro objeto da mesma classe. Portanto, um copy constructor sempre tem como único parâmetro outro objeto da mesma classe.

Por exemplo, para a classe Pessoa usada no exemplo acima, o copy constructor seria:

```
public Pessoa(Pessoa ps) {
    Nome  = ps.Nome;
    Idade = ps.Idade;
}
```

Observação: Neste exemplo específico, o copy constructor poderia aproveitar o código já implementado no construtor parametrizado da classe chamando-o através da referência **this**, como mostrado abaixo:

```
public Pessoa(Pessoa ps) {
    this(ps.nome, ps.sobrenome);
}
```

Referência this

As palavras reservadas **super** e **this** são referências que podem ser usadas dentro da classe para acessar métodos, atributos e construtores da classe ancestral (super) ou da própria classe (this). As duas palavras só podem ser usadas dentro dos métodos e construtores das classes.

this é uma referência à própria classe. Pode ser usada para chamar um construtor dentro de outro construtor (quando a classe tem mais de um) ou para fazer referência a métodos e atributos da própria classe quando existe algum tipo de conflito de nomes. Também serve como uma variável de referência interna à instância em processo (o objeto).

Para acesso de métodos e atributos

Para fazer referência a métodos ou atributos da própria classe a sintaxe é:

this.metodo(<lista de parâmetros>)
ou
this.atributo

Esta forma de fazer referência a atributos e métodos só é necessária quando existe algum conflito de nomes. Uma situação muito comum ocorre em métodos que são feitos para atribuir valores a atributos a partir dos parâmetros recebidos como é o caso dos construtores e setters. Nestes métodos é comum que o nome dos parâmetros seja igual ao nome dos atributos que eles irão inicializar. Isso provoca um conflito de nomes entre o nome dos parâmetros e o nome dos atributos. Pela regra de prioridade do Java, os parâmetros têm preferência sobre os atributos porque são "mais locais". O exemplo abaixo ajuda a entender melhor:

```
public class Ponto
  protected double x, y;

  // Construtor - note o nome dos parâmetros
  public Ponto(double x, double y) {
     this.x= x;
     this.y= y;
  }

  // Setters
  public void setX(double x){
     this.x = x;
  }

  public void setY(double y){
     this.y = y;
  }

} // classe Ponto
```

No exemplo acima, vamos considerar o método setX, mas o mesmo princípio vale para o construtor e o método setY. O parâmetro x do método setX tem o mesmo nome que o atributo x da classe, portanto há um conflito de nomes. Dentro do método, a preferência é do parâmetro, porque é a declaração mais local, então a única forma de acessar o atributo x dentro do método é usando a referência this.

Para acesso a outros construtores da mesma classe

No caso de uma classe ter mais de um construtor é bastante comum que um construtor aproveite o código de outro construtor. A palavra **this** pode ser usada para chamar um construtor da classe dentro de outro construtor. Para isso a sintaxe é:

```
this( <lista de parâmetros> );
```

A lista de argumentos na chamada deve combinar com a lista de parâmetros do construtor que será chamado. No exemplo abaixo foi acrescentado um copy constructor para a classe que aproveita o código do outro construtor:

```
public class Ponto
  protected double x, y;  // coordenadas do ponto

  // Construtor parametrizado
  //
  public Ponto(double x, double y) {
     this.x= x;
     this.y= y;
  }

  // Copy constructor
  //
  public Ponto(Ponto pt){
    this(pt.x, pt.y); // chama o construtor acima
  }

  // Setters
  //
  public void setX(double x){
     this.x = x;
  }

  public void setY(double y){
     this.y = y;
  }

} // classe Ponto
```

6) Métodos

Assinatura

Nas linguagens mais antigas, cada função é chamada através de seu identificador (nome). Dentro do contexto onde ela é válida o identificador da função precisa ser único para que não haja ambigüidades no momento da chamada.

Já nas linguagens mais modernas, existe o conceito de ***assinatura de funções***, ou no caso da programação orientada a objeto, de métodos. Nesse caso, não é apenas o nome que identifica um método, o que torna um método único é sua assinatura. A assinatura do método é formada pelo seu nome e pelos tipos dos parâmetros na ordem em que aparecem. Desta forma, é possível existirem na mesma classe, métodos com o mesmo nome, desde que tenham listas de parâmetros diferentes. Note que o nome dos parâmetros não faz parte da assinatura, apenas o tipo, quantidade e ordem deles.

Exemplo:

Declaração: `public void facaAlgo(String st)`
Assinatura: `facaAlgo(String)`

Declaração: `public void facaAlgo (int Valor, String st)`
Assinatura: `facaAlgo(int, String)`

Sobrecarga

É a capacidade que uma classe tem de conter várias versões do mesmo método, usando o mesmo identificador (nome), desde que estas versões tenham assinaturas diferentes. A sobrecarga pode ocorrer na mesma classe ou em classes descendentes. Esse importante recurso permite que a mesma tarefa seja executada de várias maneiras diferentes dependendo das informações que estão disponíveis no momento.

Sobreposição

Ocorre quando uma classe descendente implementa uma nova versão de um método que existe em alguma classe ancestral, usando exatamente a mesma assinatura. Nesse caso, a nova versão se sobrepõe à versão antiga. Um objeto criado pela classe descendente só poderá acessar a última versão do método. Dentro da classe descendente ainda é possível acessar a versão anterior através da referência super.

Exemplo:

```
public class Ancestral {
      . . .
      public void umMetodo(int par){
        . . .
      }
}

public class Descendente extends Ancestral {
      . . .
      public void umMetodo(int par){
           super.umMetodo(par); // acessando a versão da classe
                                // ancestral
           // Aqui o código acrescentado pela classe descendente
      }
}
```

7) Herança

É o processo pelo qual uma classe herda atributos e métodos de outra classe. A classe original é chamada de **ancestral** ou super-classe e a nova classe (herdeira) é a classe **descendente** ou sub-classe. Se uma classe é descendente de outra, então todos os métodos e atributos públicos ou protegidos da classe ancestral estão automaticamente disponíveis na classe descendente, sem ser necessária nenhuma implementação adicional. A herança é indicada no Java usando a palavra reservada **extends**.

Construtores não são herdados. A classe descendente precisa implementar seu próprio conjunto de construtores e dentro deles chamar algum construtor da classe ancestral para que o objeto seja construído corretamente. A maneira de chamar o construtor da classe ancestral é usando a palavra reservada **super** no lugar do nome do construtor.

```
public class Descendente extends Ancestral {
      // Atributos da classe descendente

      // Novo construtor
      public Descendente( <lista de parâmetros> )
      {
          super( <parâmetros> ); // chama o construtor ancestral
          // inicialização dos novos atributos
      }

      // Métodos da classe descendente
}
```

A palavra reservada super (referência)

As palavras reservadas **super** e **this** são referências que podem ser usadas dentro da classe para acessar métodos, atributos e construtores da classe ancestral (super) ou da própria classe (this). As duas palavras só podem ser usadas dentro dos métodos e construtores das classes.

super faz sempre referência à classe ancestral, então só pode ser usada dentro de uma classe descendente para acessar construtores, métodos e atributos da classe ancestral.

Para acesso de construtores da classe imediatamente ancestral

Para entender o processo de construção de objetos, basta lembrar que do ponto de vista do Java uma classe descendente é uma extensão da classe ancestral. Então, antes de construir a parte do objeto que é uma extensão (um acréscimo), primeiro temos que construir o que vai ser estendido (o núcleo - a parte ancestral do objeto). Para que este mecanismo de construção funcione, é responsabilidade dos construtores da classe descendente chamar algum construtor da classe imediatamente ancestral antes de executar qualquer código e para isso é usada a referência **super**. Neste caso com a seguinte sintaxe:

super(<lista de argumentos>);

A lista de argumentos na chamada deve combinar com a lista de parâmetros de algum construtor da classe ancestral, é assim que o compilador sabe qual construtor chamar. Será aquele cuja assinatura combina com a chamada. Considere o exemplo abaixo:

```
public class Ponto
  protected double x, y;

  public Ponto(double vX, double vY) {
     x= vX;
     y= vY;
  }
}
```

No exemplo, a classe **Ponto** define apenas um construtor que exige dois parâmetros do tipo double. Como a classe definiu um construtor, o Java não criou um construtor padrão para a classe, então a única maneira de construir objetos da classe Ponto é chamando esse construtor.

Portanto, qualquer classe descendente direta da classe **Ponto**, será obrigada a implementar um construtor que chame o construtor de Ponto antes de executar qualquer código e é para isso que a referência **super** é usada. Veja o exemplo abaixo de uma classe descendente de Ponto:

```
public class Circulo extends Ponto
  protected double radius;

  public Circulo(double vX, double vY, double vRadius) {
     super(vX, vY); // Aqui é chamado o construtor da classe Ponto
     radius= vRadius;
  }
}
```

Para acesso de métodos e atributos ancestrais

Normalmente não é preciso usar super para acessar métodos e atributos da classe ancestral. Se forem públicos ou protegidos eles estarão disponíveis na classe descendente sem ser necessária uma referência à classe ancestral. Entretanto, métodos e atributos da classe ancestral podem ser sobrepostos (overriding) pela classe descendente e nesse caso é necessário usar a palavra super quando se quer acessar a implementação da classe ancestral. Para isso usa-se a seguinte sintaxe:

super.metodo(<lista de argumentos>)
ou
super.atributo

Veja o exemplo abaixo:

```
public class Ponto
  protected double x, y;

  public Ponto(double vX, double vY) {
     x= vX;
     y= vY;
  }

  // Faz o escalonamento do ponto pelo fator informado
  public void escale(double factor){
    x *= factor;
    y *= factor;
  }
}
```

O método escale agora está implementado pela classe Ponto. Sendo público, qualquer classe descendente de Ponto herda o método e pode chamá-lo sem nenhuma referência adicional. Mas vejamos o que acontece se este método for reescrito por uma classe descendente como no exemplo abaixo:

```
public class Circulo extends Ponto
  protected double radius;

  public Circulo(double vX, double vY, double vRadius) {
     super(vX, vY); // Aqui é chamado o construtor da classe Ponto
     radius= vRadius;
  }
```

```
  // Faz o escalonamento do círculo pelo fator informado
  public void escale(double factor){

    // chama o método escale da classe ancestral
    super.escale(factor);
    // Acrescenta o escalonamento do raio
    radius *= factor;
  }
}
```

Note que apesar do método escale estar disponível para a classe Circulo, a implementação da classe Ponto não é suficiente para fazer o escalamento de um círculo porque o raio também precisa ser considerado. Então a classe Circulo precisa reescrever o método escale para escalar também o raio. Para aproveitar o código já implementado pela classe Ponto, a versão ancestral do método escale é chamada usando a palavra super.

Importante: Note que neste exemplo o uso de super é obrigatório. Se a chamada ao método escale for feita sem a referência super, então o método escale da classe Circulo é que vai ser chamado. Isso provocaria um loop infinito que termina com uma sonora exceção, já que teríamos um método chamando ele mesmo (recursivo) sem nenhum critério de parada.

8) Classes Abstratas

Algumas classes não agrupam objetos concretos, mas servem para agrupar tipos ou classes de objetos. Para entender melhor voltemos ao exemplo dos veículos e carros da figura 1. Sabemos que carros podem ser realmente construídos, são objetos concretos que existem fisicamente no universo. Mas a classe Veículos, não representa um grupo de objetos físicos apenas por ela mesma, representa um agrupamento de classes de objetos, e estas sim são classes de objetos concretos. Se tentarmos construir um veículo, obrigatoriamente teremos que escolher que tipo de veículo será construído. Vai ser um carro, uma moto, um avião, um barco ou o que? Não é possível construir um veículo genérico que não esteja associado a algum tipo (subclasse) de veículo. Então a classe Veículos depende das suas subclasses, ou classes descendentes para criar objetos. Apenas por ela mesma não é possível construir objetos. Neste caso dizemos que a classe Veículos, é uma **classe abstrata**.

As classes abstratas são usadas na POO para serem ancestrais de outras classes e não para serem instanciadas. Ao tornar uma classe abstrata, o programador impede que sejam criadas instâncias dela. Frequentemente é criada uma classe ancestral para conter o código que é comum a todas suas classes descendentes. Essa classe ancestral é extremamente útil porque evita a replicação de código nas classes descendentes, mas não faz sentido a criação de objetos dessa classe no sistema. Por exemplo, suponha que um sistema acadêmico precise implementar classes para professores e alunos.

Parte dos atributos e métodos pode ser comum a essas duas classes e essa parte pode perfeitamente ser colocada em uma classe que seja ancestral para as duas classes, por exemplo, a classe Pessoa. Então a classe Pessoa contém todo o código que é comum às classes Professor e Aluno e isso é extremamente útil, mas não faz nenhum sentido criar instâncias da classe Pessoa. Neste caso tornar a classe Pessoa uma classe abstrata, garante que nenhuma instância dessa classe poderá ser criada.

Métodos abstratos

Classes abstratas podem conter métodos abstratos. Um método abstrato é declarado na classe, mas não tem seu código implementado, ele será implementado em alguma classe descendente. Uma classe abstrata pode ou não ter métodos abstratos, mas se uma classe contém métodos abstratos, então obrigatoriamente a classe deve ser declarada abstrata. Uma classe descendente de uma classe abstrata precisa implementar o código de todos os métodos abstratos para se tornar concreta e poder ser instanciada. Caso contrário, a classe descendente continua sendo abstrata e precisa ser declarada como tal.

Apesar de um método abstrato não ter seu código implementado na classe que o declara, esta classe pode usar e fazer chamadas a ele. Isso acontece porque só vai existir um objeto se a classe abstrata tiver pelo menos uma classe descendente que implemente todos os métodos abstratos. Então durante a execução, o método vai ter sua implementação disponível.

Em java, usa-se a palavra reservada **abstract** para indicar que uma classe ou um método é abstrato.

Exemplo:

```
public abstract class NomeClasse {
     .........
     public abstract tipo NomeMétodo (lista de parâmetros);
}
```

Note que após a lista de parâmetros é colocado um ponto-e-vírgula encerrando a declaração em vez de abrir chaves para iniciar o corpo do método.

Apesar de um método abstrato não ter seu código implementado na classe que o declara, esta classe pode usar e fazer chamadas a ele. Isso acontece porque só vai existir um objeto (uma instância) se existir pelo menos uma classe descendente que implemente todos os métodos abstratos. Então durante a execução é garantido que o método vai ter sua implementação disponível.

Veja o exemplo abaixo:

```
public abstract class Shape {
    protected double centerX, centerY;

    public Shape(double centerX, double centerY) {
        this.centerX = centerX;
        this.centerY = centerY;
    }

    public abstract double getArea();

    public void printArea(){
        System.out.format("Area: %.1f\n", getArea());
    }
}
```

Note que o método **printArea()** faz uma chamada ao método **getArea()** que é um método abstrato. A classe **Shape** não tem uma implementação para o método **getArea**, mas como é uma classe abstrata, nunca vai instanciar objetos diretamente. Para que exista um objeto executando o código da classe **Shape** tem que existir uma classe descendente que implemente o método **getArea()**.

9) Coleções de Dados (Arrays)

Arrays são estruturas indexadas capazes de armazenar uma coleção de dados do mesmo tipo. Em Java os Arrays são sempre objetos e portanto, precisam ser construídos antes de serem utilizados.

Declaração

`tipobase variável[];` ou `tipobase[] variável;`

O tipobase pode ser um tipo primitivo ou uma classe de objetos.

Exemplo:

```
int vetor[];
```

Construção

```
variavel= new Tipobase[quantidade];
```

A construção pode ser feita já na declaração:

```
int vetor[]= new int[10];
```

O primeiro elemento do array tem sempre o índice 0. Para acessar um elemento qualquer basta indicar a variável seguida do índice entre colchetes.

Exemplo:

```
vetor[0]=15;
vetor[1]=vetor[0]*2;
```

Todo array tem o atributo lenght que indica o tamanho dele (quantidade de elementos).

Exemplo:

```
int qtd;
qtd=vetor.lenght;
```

Importante: Um array tem sempre um tamanho fixo. Não é possível alterar o tamanho depois que foi construído.

Arrays de objetos

Um array também pode armazenar uma coleção de objetos. Usa-se da mesma forma que um array de tipos primitivos, mas é importante notar que a construção do array não implica na construção dos seus elementos. Eles também precisam ser construídos.

Exemplo:

```
Pessoa ps[];          // Declaração
Pessoa umaPessoa;

ps=new Pessoa()[10]; // Construção de um array com 10 elementos.
ps[0]=new Pessoa();  // Contrução de um objeto para o 1° elemento
umaPessoa=new Pessoa();
ps[1]=umaPessoa;     // Atribuição de um objeto já construído para o
                     // 2° elemento
```

10) Interfaces

Assim como a maioria das linguagens orientadas a objeto, o java não permite herança múltipla, ou seja, cada classe só pode herdar de uma única classe. Entretanto, o java fornece a opção de implementar interfaces, quantas forem necessárias, o que na maioria das vezes resolve essa limitação. Uma interface é uma estrutura parecida com uma classe, mas com várias restrições. Seus métodos são sempre abstratos e públicos. Pode ter atributos, mas serão sempre públicos, estáticos e finais (não podem ter seu valor modificado, ou seja, são constantes).

Assim como as classes abstratas, a função das interfaces é criar uma linguagem padrão de comunicação com as classes, definindo métodos que serão implementados nas classes. A diferença é que a interface não usa o mecanismo de herança das classes, portanto uma classe pode herdar de outra classe e também implementar uma ou mais interfaces.

O uso de interface proporciona um padrão para a comunicação entre duas classes e permite também a subtipagem de classes. Assim como uma classe descendente é um subtipo da classe ancestral, a classe que implementa uma interface se torna um subtipo dela.

A grande vantagem de usar interfaces é que uma classe pode ter acesso e usar outras classes que não são conhecidas por ela. Como isso é possível? Como dito antes, quando uma classe implementa uma interface, se torna um subtipo dela. Então é possível que um método receba um objeto de uma classe desconhecida, mas se a classe desse objeto implementa uma interface conhecida, o método pode identificar a interface através do operador **`instanceof`** e utilizar os métodos da interface que estão no objeto. Isso é extremamente útil para fazer a integração de sistemas diferentes, ou para criar sistemas que podem ser facilmente expandidos com novos módulos.

IMPORTANTE: uma interface não tem construtores.

Sintaxe:

```
public interface NomeInterface {
     Lista de atributos
     Lista de métodos
}
```

Exemplo de declaração de interface:

```
interface NomeInterface{
   int Atrib = 10;  //este atributo é estático e final
   void Metodo1();  //Estes métodos são abstratos e públicos
   double Metodo2();
   int Metodo3();
}
```

Associação com classe e implementação:

```
public class NomeClasse implements NomeInterface{
   // atributos
   // construtores
   // outros métodos
```

```
    public void Metodo1(){
        //código do método 1
    }
    public double Metodo2(){
        //código do método 2
    }
    public int Metodo3(){
        //código do método 3
    }
}
```

Se uma classe precisa implementar duas ou mais interfaces, basta separar por vírgulas na declaração da classe:

```
public class NomeClasse implements Interface1, Interface2{
...
}
```

11) Exceções

Exceções são situações inválidas que ocorrem durante o processamento e impedem que o programa continue seu fluxo normal até que a situação seja de algum modo resolvida.

Nas linguagens mais antigas, os programas tentavam evitar que estas situações ocorressem através de testes de validação e fazendo o processamento apenas quando todas as condições fossem válidas. Esse modo de programar funciona, mas deixa o código inchado e menos legível já que, conforme o caso, uma grande quantidade de testes pode ser necessária antes de qualquer processamento.

Na programação moderna, o código é escrito sem preocupação com as condições de erro. Quando ocorre uma situação inválida, o sistema emite (lança) uma exceção e então o programa tem a opção de tratar essa ocorrência.

Mecanismo das exceções

Como tudo no java, exceções são instâncias de classes. Quando um método ou o sistema de detecta uma situação de exceção, uma instância da classe de exceção correspondente é criada e "lançada" no programa.

Para entender melhor o mecanismo usado, é preciso entender o conceito de **pilha de execução**. A qualquer momento do processamento existe sempre uma "pilha" de métodos ou funções em execução.

Por exemplo, um programa Java começa executando o método main da classe principal. Dentro do método main serão chamados outros métodos de objetos

criados ali ou de classes já existentes no Java. Estes métodos chamarão outros métodos, que chamarão outros, e assim por diante, constituindo uma pilha.

Sempre que o método A chama o método B, o método A fica parado esperando a execução do método B para poder continuar. O método B, por sua vez, também pode chamar outros métodos e também vai ter que ficar esperando a execução destes para continuar. Se uma exceção acontecer, vai acontecer sempre no método que está no topo da pilha, porque é sempre este método que está sendo executado, os outros estão em estado de espera.

Nesse caso, o método pode ou não "capturar" essa exceção. Se não capturar, o método é abortado e a execução vai para o método seguinte da pilha. Ali também a exceção pode ser capturada ou não e se não for capturada, o método também é encerrado e a execução vai para o método seguinte. Se nenhum método capturar a exceção ela chegará ao método main que, se também não capturar a exceção será encerrado, finalizando o programa.

Essa propagação da exceção é encerrada quando ela é capturada por algum método. Portanto, ou o método captura a exceção ou a exceção encerra o método.

Ciclo de vida da exceção

Quando a situação ocorre ela é:

1) **Criada:** é feita uma instância da classe de exceção apropriada;
2) **Lançada:** É propagada para todos os métodos em execução no momento, iniciando pelo que estiver no topo da pilha de execução;
3) **Capturada:** Algum dos métodos da pilha pode capturar a exceção, geralmente para tratar a situação. Uma vez capturada a exceção não se propaga mais;
4) **Tratada:** Quando é possível resolver a situação inválida, o método que capturou a exceção também pode tratá-la.

Captura de exceções no Java

Usa a estrutura **try-catch-finally**. Cada uma destas palavras reservadas tem um bloco de comandos associado:

```
try{
  // bloco de comandos sujeito a exceções
}
catch (ClasseExceção ObjExc){
  //Código de tratamento para a exceção especificada
}
finally {
  // Este bloco sempre será executando ocorrendo ou
  // não alguma exceção. Use-o para finalizar uma
  // operação que não pode ficar pendente.
```

```
}
```

Pode haver vários blocos catch, cada um com uma exceção diferente, caso ocorra uma exceção, a classe da exceção será comparada com as classes listadas pelos blocos catch, na ordem que aparecem no código. O primeiro que combinar será executado e os de baixo não serão comparados.

A classe **Exception** é a classe ancestral de todas as exceções, portanto sempre vai combinar com qualquer exceção. Se aparecer em um bloco catch, deve ser o último bloco, caso contrário os de baixo nunca serão comparados.

A estrutura **try-catch-finally** permite que ou o bloco **catch** ou o bloco **finally** seja omitido, mas nunca os dois. Portanto é possível usar **try-finally** ou **try-catch**.

Exemplo:

```
public double DivAtrib(double value){
      try{
            return Atrib/value;
      }
      catch(Exception e){
            System.out.println("Operação invalida");
      }
}
```

Método toString()

O método **toString()** é definido na classe **Object**, que é ancestral de todas as classes no Java. Este método é usado para gerar uma representação do objeto na forma de string. Idealmente, todas as classes deveriam reescrever o método **toString()** para manter a representação válida.

As exceções também são descendentes de **Object** e também tem o método **toString()**. Nas exceções, este método é usado para armazenar uma string explicativa sobre o que gerou a exceção.

Exemplo:

```
      try{
         \\ código
      }
      catch(Exception e){
         System.out.println(e.toString());
      }
```

Resumo try - catch – finally:

- **try:** bloco com código sujeito a exceção. O bloco será executado até o ponto onde a exceção ocorreu, se ocorrer. Neste caso a execução passa para o bloco **catch** (se existir) e depois para o bloco **finally**.
- **catch** Podem haver vários blocos catch, cada um associado a uma exceção diferente. A exceção será comparada com as classes listadas nos blocos catch e o primeiro que combinar terá seu código executado. Apenas um bloco catch será executado. A classe **Exception** é ancestral de todas as exceções, portanto combina com qualquer exceção.
- **finally** Os comandos nesse bloco serão sempre executados, em qualquer situação, ocorrendo ou não alguma exceção.

Os blocos **catch** ou **finally** podem ser omitidos desde que um deles esteja presente.

Para criar exceções

Exceções são objetos, instâncias de uma classe. Toda classe de exceção é descendente da classe Exception. O Java já define uma grande quantidade de classes para exceções, entretanto, é possível criar novas classes sempre que necessário.

Depois de criada, a exceção precisa ser lançada na pilha de execução. O comando para lançar a exceção é o comando **throw**.

É comum criar e lançar a exceção na mesma linha sem utilizar uma variável de objeto para a exceção:

```
throw new Exception();
```

Uma vez lançada a exceção, nenhum outro código será executado, seja no método que lançou, seja em qualquer outro método na pilha de execução até que a exceção seja capturada.

IMPORTANTE: Um método que lança uma ou mais exceções precisa informar a exceção na sua assinatura usando o comando **throws**.

MAIS IMPORTANTE: Não faz sentido um método tentar capturar a exceção que ele mesmo criou. O método lança a exceção quando não é capaz de resolver a situação inválida e captura a exceção quando é capaz de tratar a situação.

Exemplo:

```
public void UmMetodo() throws Exception{
    if (condição) throw new Exception();
    //codigo do metodo
}
```

Para criar uma nova classe de exceção

A definição de uma nova classe de exceção é extremamente simples, basta criar um descendente da classe Exception e reescrever o método toString() para retornar a string explicativa.

Exemplo:

```
public class NovaExceção extends Exception{
   public String toString(){
      return "Aconteceu a nova exceção";
   }
}
```

Apêndice: Hello word no Netbeans

Iniciando no NetBeans

Para começar a usar os recursos do NetBeans vamos iniciar criando o clássico Hello World. Siga os passos abaixo para criar o programa. Mais adiante, os termos usados no programa serão explicados.

1) Depois de ativar o NetBeans selecione a opção do menu: **Arquivo / Novo projeto** ou use o atalho **Ctrl-Shift-N**
2) Uma nova janela será aberta. Nesta janela localize a caixa **Categorias** e selecione a opção **Java** (default). Na caixa **Projetos** selecione a opção **Aplicativo Java** (também default) e clique no botão **Next >**.
3) No campo Nome do projeto escreva **Hello**. A seguir clique no botão **Procurar** e selecione a pasta onde o projeto será salvo.
4) No campo **Criar classe principal**, deixe o checkbox ligado (default). Deixe os outros campos com os valores default e clique no botão **Finish**.
5) Neste ponto, o NetBeans já criou a estrutura da aplicação, com a classe principal do programa (**Main**), incluindo o método **main** da classe. Dentro do método **main** (entre as chaves) escreva o seguinte comando:

   System.out.println(“Oi mundo!!!!!”);
6) Agora a aplicação está pronta para ser executada. Para isso Você pode usar a tecla **F6**, ou clicar na setinha verde que está na barra de atalhos, ou ainda usar a opção do menu **Executar / Executar projeto principal**. A saída do programa é mostrada na caixa de saída (Output) que fica na parte inferior da tela. Se houver algum erro no programa também é ali que as mensagens de erro são mostradas. Se tudo deu certo, a string “Oi mundo!!!!!” aparecerá na caixa de saída.

   Obs: se você copiou e colou a linha acima, é bem provável que tenha ocorrido um erro de compilação, porque este texto foi gerado no Word e o Word usa caracteres especiais para representar as aspas. Apague e digite de novo as aspas que envolvem a string que o programa deve compilar corretamente.

Elementos do programa criado

O programa inicia com algumas linhas de comentário que informam como alterar o template (modelo) usado para criar o programa com o assistente. Assim como na linguagem C, os caracteres **/*** iniciam comentário e os caracteres ***/** encerram os comentários. Também é possível usar os caracteres **//** para criar uma linha de comentários. Neste caso, tudo o que for escrito após os caracteres **//** até o final da linha é considerado comentário.

Em seguida, vemos uma linha com:

package hello;

package (pacote) é uma palavra reservada na linguagem que indica que o arquivo em questão faz parte do pacote **hello**. Um **"package"** é basicamente um agrupamento de classes que estão relacionadas por algum critério. No nosso exemplo, criamos uma aplicação extremamente simples, que só usou uma classe e um arquivo, mas aplicações mais complexas podem exigir várias classes que serão definidas em vários arquivos. Também é comum usar os pacotes como bibliotecas de classes, não necessariamente associadas a uma aplicação específica.

Uma vez criado um pacote, ele pode ser incluído em qualquer aplicação ou implementação de uma classe usando a palavra reservada **import**, mais adiante voltaremos a esse assunto.

A linha seguinte (que não é comentário) aparece como segue:

public class Main {

Main é o nome da classe que estamos definindo neste arquivo. Note que esse também é o nome do arquivo que tem a extensão .java. Por convenção (não é obrigado), os nomes de classes começam sempre com a primeira letra maiúscula e as outras letras minúsculas. Se o nome for formado por duas ou mais palavras como no nosso exemplo, a primeira letra de cada palavra é escrita com maiúsculas.

class também é uma palavra reservada que serve para indicar que o que vem depois é a definição de uma classe. Todo arquivo de aplicação, como o do exemplo, deve ter uma e apenas uma classe pública. É possível ter várias classes no mesmo arquivo, mas apenas uma pode ser pública. A palavra reservada **public** no início da linha indica que a classe **Main** é pública.

Finalmente a linha termina abrindo chaves o que significa que tudo o que vier a seguir e até o fechamento correspondente das chaves é o corpo da classe, onde serão declarados seus atributos e implementados seus métodos. No caso específico desta classe, não há nenhum atributo e o único método é o método **main**.

Mais abaixo encontramos a implementação do método **main** como segue:

```
public static void main(String[ ] args) {
    System.out.println("Oi mundo!!!!!");
}
```

Se estivermos criando uma aplicação, é obrigatório que a classe principal do programa, tenha um método chamado **main**. Esta classe, na verdade é a própria aplicação. Essa classe nunca deve ser instanciada explicitamente pelo programa, na verdade ela nem precisa ser instanciada porque o único método que ela contém, o método main, é um método de classe e não de instância (veja abaixo). O método **main** será chamado automaticamente e ele é o ponto de partida do programa. É análogo à função main de um programa em C. Note que nem toda classe é uma aplicação e nem sempre as classes são desenvolvidas para uma aplicação específica. Este método só existe na classe principal da aplicação, em outros casos ele não existe.

IMPORTANTE: No início pode ficar meio confuso que a classe principal da aplicação e o método **main** tenham, por default, o mesmo nome, mas são coisas completamente diferentes. A classe **Main** (escrito com a inicial em maiúscula) representa a aplicação. Esse nome é só uma sugestão, você pode escolher outro nome quando cria a aplicação no assistente. Já o método **main** é o ponto de partida do programa, por onde a execução é iniciada. A classe principal da aplicação, não importa que nome ela tenha, tem que ter um método chamado **main**, o nome desse método não pode ser mudado, precisa ser **main** (com a inicial em minúscula).

A palavra reservada **void** imediatamente antes do nome do método indica que este método não vai retornar nenhum valor depois que for executado. A palavra reservada **static** indica que este é um método de classe e não de instância. Métodos de classe são chamados a partir da classe, sem ser necessário criar um objeto daquela classe. Mais adiante voltaremos a esse assunto.

Finalmente, o modificador **public** indica que é um método público, ou seja, ele é visível e pode ser chamado por qualquer objeto do sistema.

Dentro dos parêntesis vemos o parâmetro **String [ ] args**. Este parâmetro é um vetor de strings e armazena os argumentos passados na linha de comando quando o programa é chamado via linha de comando.

Depois dos parâmetros, novamente encontramos a abertura de chaves iniciando o corpo do método **main** e que é encerrado no fechamento de chaves correspondente.

Para finalizar, temos o comando que realmente imprime a string dentro do método **main**. **System** é uma classe do Java que representa os recursos do sistema no programa. Está disponível em qualquer programa feito em Java. Ela oferece atributos e métodos bastante úteis e que serão usados frequentemente. Um de seus atributos é o objeto **out** que representa a stream de saída padrão do sistema. **println** é um método de **out** que permite imprimir uma string na saída padrão (console).

Note que mesmo programa o mais básico possível, o clássico Hello World, envolve uma grande quantidade de conceitos para ser implementado. Mas não é preciso entrar em pânico, estes conceitos serão retomados durante o curso e com o tempo se tornarão familiares conforme sejam usados com mais freqüência.

Melhorando o programa

Vamos agora acrescentar alguns elementos a este programa e aproveitar para conhecer mais alguns conceitos.

Veja que o método **main** não declara nenhuma variável e tem apenas um comando para imprimir o texto. Acrescentemos então uma variável local ao método para armazenar o texto que será impresso. Sendo um texto, a variável precisa ser da classe **String**. Note que em Java, **String** não é um tipo básico de dados como o **int** ou **double**, mas uma classe que contém seus próprios métodos. Ao atribuir um valor para uma variável String, estamos na verdade criando uma instância para a classe, ou seja, criando um objeto.

Modifique o método main da seguinte forma:

```
public static void main(String[ ] args) {
    String st;

    st = "Oi mundo!!!!!";
    System.out.println(st);
}
```

Ao executar o programa o resultado será exatamente o mesmo de antes, mas agora o texto é armazenado em um objeto String antes de ser impresso. Isso permite alguns recursos adicionais que antes não eram possíveis, por exemplo, é possível usar métodos da classe String para processar o texto antes de imprimir.

Altere a linha que imprime o texto como sugerido abaixo:

```
    System.out.println( st.toUpperCase() );
```

Veja que executando o programa agora, o texto aparecerá com todas as letras em maiúsculas porque esse é resultado do método **toUpperCase** da classe **String**. Coloque o cursor logo após o ponto depois de st e pressione Ctrl-Espaço. O NetBeans mostrará uma lista de todos os métodos disponíveis para um objeto da classe **String**.