Nome: _____ Data: ____/____/____

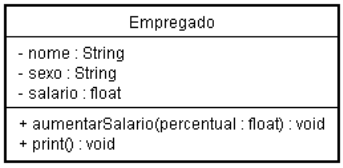
Classes e objetos

Classes:

Java, por ser uma linguagem completamente orientada a objetos, necessita que sempre se trabalhe com pelo menos uma classe. **Não importa o quão simples seja o seu programa, você sempre vai utilizar no mínimo uma classe.**

Obs.: Um arquivo .java pode ter várias classes, porém **somente uma pode ser pública**. A classe pública **necessariamente** deve ter o mesmo nome do arquivo .java.

```
3  public class Empregado {
4
5      private String nome;
6      private String sexo;
7      private float salario;
8
9      public Empregado(String n, String s, float sal) {
10         nome = n;
11         sexo = s;
12         salario = sal;
13     }
14
15     public void aumentarSalario(float percentual) {
16         salario = salario * (1 + percentual / 100);
17     }
18
19     public void print() {
20         System.out.println(nome + " - " + sexo + " - " + salario);
21     }
22
23 }
```



No exemplo acima temos a classe Empregado, a qual possui os atributos: **nome**, **sexo** e **salario** e os métodos: **aumentarSalario** e **print**. A classe Empregado possui ainda um outro método especial. Esse método é o construtor, ou seja, é o método utilizado para inicializar objetos da classe.

Construtores:

Construtores são utilizados para inicializar objetos de uma classe. Todas as classes possuem pelo menos um construtor. Quando não especificamos um construtor o compilador Java inclui o construtor default.

Obs.: O construtor default é um construtor que não possui nenhum argumento.

Os construtores são métodos especiais e por isso seguem algumas regras:

- Tem o mesmo nome da classe.
- Pode ter nenhum, ou vários parâmetros.
- Sempre é chamado com o operador new.
- Não retorna valor.

A função do construtor é inicializar os objetos da classe. Quando não especificamos um construtor, ou seja, quando o construtor default é utilizado, ele inicializa os atributos (variáveis de instância) da seguinte forma: Objetos com **null**, números com **0** e valores lógicos com **false**.

Encapsulamento:

O encapsulamento é o resultado (ou o ato) de ocultar do usuário os detalhes da implementação de um objeto. Para isso, sempre se deve declarar uma variável de instância (atributo) como sendo privado (**private**), ou seja, somente a classe tem acesso a ele. Para a manipulação (leitura e escrita) dos dados desses atributos, são usados os métodos acessadores e modificadores. Métodos acessadores são aqueles que simplesmente lêem os dados do atributo e métodos modificadores são aqueles que podem alterar esses dados. Cada atributo possui o seu método acessador e seu método modificador. (Veja nas convenções para codificação como devem ser os nomes desses métodos).

- Os métodos acessadores (get) devem retornar o mesmo tipo de dado do seu atributo, não recebendo nenhum argumento.
- Os métodos modificadores (set) não retornam valor e devem receber um argumento do mesmo tipo do seu atributo.

```
3  public class Encapsulamento {
4
5      private String nome = "";
6      private int idade   = 0;
7
8      public String getNome() {
9          return this.nome;
10     }
11
12     public int getIdade() {
13         return this.idade;
14     }
15
16     public void setNome(String nome) {
17         this.nome = nome;
18     }
19
20     public void setIdade(int idade) {
21         this.idade = idade;
22     }
23 }
```

Métodos de classe e de instância:

Assim como podemos ter variáveis de classe e de instância, também podemos ter métodos de classe e métodos de instância. Os métodos de classe são aqueles que pertencem à classe e não a um objeto criado a partir da classe. Em contrapartida, os métodos de instância pertencem aos objetos (instâncias) e não às classes. Métodos de classe (ou métodos estáticos) são declarados usando-se a palavra reservada **static**. Quando vamos usar um método de classe, não precisamos (e nem devemos) criar um objeto, acessamos o método através da classe:

```
47      /* Métodos de instância */
48      public void aumentarSalario(float percentual) {
49          salario = salario * (1 + percentual / 100);
50      }
51
52      public void print() {
53          System.out.println(nome + " - " + sexo + " - " + salario);
54      }
55
56      /* Métodos de classe */
57      public static float consultaAumento(float salario, float percentual) {
58          float retorno = salario * (1 + percentual / 100);
59          return retorno;
60      }
```

```
3  public class TesteEmpregado {
4
5      public static void main(String[] args) {
6
7          Empregado empregado = new Empregado("Fulano", "M", 500.00f);
8          empregado.aumentarSalario(10);
9
10         float consulta =  Empregado.consultaAumento(500.00f, 10);
11     }
12 }
```

Sobrecarga de métodos (overload):

A sobrecarga ocorre quando temos vários métodos com o mesmo nome, porém assinaturas diferentes. Como os nomes dos métodos são iguais, o interpretador Java precisa diferenciar um método do outro, para saber qual método chamar. Os métodos que tem o mesmo nome são diferenciados por outros dois detalhes:

- Quantidade de argumentos
- Tipo dos argumentos

Esses dois detalhes, somados ao nome do método compõem a assinatura do método.

Obs.: O mecanismo usado pelo Java para diferenciar os métodos é muitas vezes chamado de resolução de sobrecarga (overloading resolution).

```
49      public void aumentarSalario(float percentual) {
50          salario = salario * (1 + percentual / 100);
51      }
52
53      public void aumentarSalario(float percentual, float comissao) {
54          salario = salario * (1 + percentual / 100) + comissao;
55      }
```

A sobrecarga de métodos pode eliminar a necessidade de métodos inteiramente diferentes que fazem quase o mesmo. Ela também torna possível que os métodos se comportem de maneiras diferentes, de acordo com os argumentos que receber.

O objeto this:

Ocasionalmente é necessário acessar o objeto atual em sua totalidade e não somente uma variável de instância. A linguagem Java tem um atalho para isso – a palavra chave **this**. Em um método, a palavra chave **this** faz referência ao objeto ao qual o método pertence. Muitas classes Java possuem um método chamado **toString**(), que retorna uma string que descreve o objeto. Se você passar qualquer objeto para o método **System.out.println**, esse método vai chamar o toString no objeto e imprimir a string resultante. Dessa forma pode-se imprimir o estado atual de um objeto fazendo: System.out.println(this);.

Pacotes:

A linguagem Java permite agrupar classes em uma coleção chamada pacote (package). Os pacotes são convenientes para organizar o trabalho e separar suas classes de classes fornecidas por terceiros. Pacotes nada mais são, do que a estrutura de diretórios onde a classe se encontra. A biblioteca padrão Java é distribuída em inúmeros pacotes como **java.lang**, **java.util**, **java.net**, e muitos outros. Os pacotes padrão Java são um bom exemplo de pacotes hierárquicos, ou seja, da mesma forma que se aninha vários níveis de diretórios em um disco, podemos aninhar os pacotes. Um motivo para se aninhar pacotes é garantir a exclusividade. Por exemplo, nós poderíamos criar várias classes utilitárias e distribuí-las em um pacote chamado util. Poderíamos ter uma classe chamada Data, dentro de um pacote chamado util. Porém outra pessoa também poderia ter essa idéia. Conclusão: Conflito de nomes. Para tentar solucionar isso, a Sun recomenda que as empresas usem o próprio nome de domínio na Internet (o qual, presumivelmente é único) escritos com a ordem invertida como prefixo para seus nomes de pacotes. Supondo que a nossa empresa tenha o seguinte domínio: senac.com.br, o prefixo dos nossos pacotes seria: br.com.senac.

Para colocar uma classe dentro de um pacote, basta colocar no topo do arquivo antes de qualquer outra linha de código a palavra reservada package seguida do nome do pacote. No nosso exemplo, no topo do arquivo Data.java teríamos:

package util;

Usando pacotes:

Supondo que tenhamos a nossa classe Data dentro do pacote util, teríamos a seguinte estrutura para chegar até ela: br.com.senac.util.Data. Esse é o nome qualificado da nossa classe, ou seja o nome completo dela, contendo o nome do pacote onde ela se encontra e o próprio nome da classe. Para usarmos a nossa classe, poderíamos fazer da seguinte forma:

```
br.com.senac.util.Data data = new br.com.senac.util.Data();
```

Isso além de cansativo, é incomodo, pois aumenta e muito o tamanho da nossa linha de código. Uma solução muito mais amigável é o uso da palavra reservada **import**. Podemos importar somente uma classe em específico, ou todo o conteúdo do pacote:

```
import br.com.senac.util.Data; // importa a classe Data
import br.com.senac.util.*;    // importa todo o conteúdo do pacote
```

Obs.: Mesmo quando importamos todo o pacote, o compilador Java só importa efetivamente aquelas classes que realmente são utilizadas.

Javadoc:

O utilitário javadoc analisa arquivos fonte procurando por classes, métodos e comentários /** ... */. Ele gera documentos HTML no mesmo formato da documentação da API. Na verdade a documentação da API é o resultado do javadoc sobre os arquivos fonte da linguagem Java.

Como inserir comentários:

O javadoc extrai informações sobre:

- Pacote
- Classe pública
- Interface pública
- Método público ou protegido
- Variável ou constante pública ou protegida

Para inserir comentários de documentação, basta colocá-los entre /** ... */. Os comentários contém texto livre, seguidos por comandos (quando existirem). Um comando começa com @, como @author ou @param.

A primeira sentença do texto livre deve ser um enunciado resumido. O javadoc gera automaticamente páginas de resumo extraídas dessas sentenças. Pode-se também usar marcas HTML no texto livre, como ... para itálico, ou ... para negrito. Deve-se evitar cabeçalhos <h1> ou réguas <hr>, pois esses podem interferir na formatação do documento.

Comentários gerais:

Os seguintes comandos podem ser usados em todos os comentários de documentação:

Comando	Descrição
@since <texto>	Esse comando cria uma entrada "since" (desde). <texto> pode ser qualquer descrição da versão que introduziu esse recurso.
@deprecated <texto>	Esse comando adiciona um comentário de que a classe, o método ou variável não deveria mais ser usada.
@see <link>	Esse comando adiciona um link à seção "veja também" da documentação. Ele pode ser usado com classes e métodos. <link>

Comentários de Classes e Interfaces:

O comentário de uma classe deve ser colocado depois de todas as intruções **import** (se houverem) e imediatamente antes da declaração da classe.

Os seguintes comandos podem ser usados em comentários de Classes e Interfaces:

Comando	Descrição
@author <nome>	Esse comando cria uma entrada "autor". Podem haver vários comandos desses, mas eles devem estar todos juntos.
@version <texto>	Esse comando cria uma entrada "versão". O <texto> pode ser qualquer descrição sobre a versão atual.

Exemplo:

```
3 /**
4  * Essa classe &eacute; usada para testar a classe Empregado
5  *
6  * @version 1.00 07 de Outubro de 2004
7  * @author Alexandre Falcão Viriato
8  * @see "Documentação sobre javadoc"
9  */
```

Comentários de métodos:

O comentário de um método deve ser colocado imediatamente antes da declaração do método.

Os seguintes comandos podem ser usados em comentários de métodos:

Comando	Descrição
@param <descrição>	Esse comando adiciona uma entrada à seção "parâmetros" do método atual. A <descrição> pode ocupar várias linhas e pode usar marcas HTML. Todas os comandos @param de um método precisam estar juntos.
@return <descrição>	Esse comando adiciona uma seção "Returns" ao método atual. A <descrição> pode ocupar várias linhas e usar marcas HTML
@throws <classe> <descrição>	Esse comando adiciona uma entrada à seção "Throws" do método atual. A <descrição> pode ocupar várias linhas e pode ter marcas HTML. Todos os comandos @throws de um método precisam estar juntos. <classe> é a classe da exceção que o método pode lançar.

Exemplo:

```
3 /**
4  * Formata um double em uma string
5  *
6  * @param x o número a ser formatado
7  * @return a string formatada
8  * @throws IllegalArgumentException caso o argumento seja inválido
9  */
```

Como extrair comentários:

Vá ao diretório que contém os arquivos-fonte que você quer documentar e execute o seguinte comando:

javadoc –d <caminho> <arquivo>

Onde: <caminho> é o caminho onde a documentação será gerada.
<arquivo> é o arquivo que será documentado

O utilitário javadoc pode ser ajustado com algumas opções de linha de comando. Por exemplo, pode-se usar as opções **–author** e **–version** para incluir os comandos **@author** e **@version** na documentação. (Por padrão eles são omitidos).

Memória e referencia à memória:

Quando estamos trabalhando com objetos, estamos na verdade trabalhando com referências a endereços de memória (ponteiros). Quando utilizamos o operador new para criar um novo objeto, a máquina virtual Java:

- Aloca memória para o objeto
- Constrói o objeto (através do construtor)
- Guarda esse objeto no espaço de memória reservado
- Cria uma referência para o objeto e guarda essa referência na variável.

Só podemos utilizar um objeto, após a construção do mesmo. Da mesma forma não podemos utilizar um objeto após atribuirmos **null** a ele.

```
11          /* Cria uma variável do tipo Empregado
12           * A VARIÁVEL AINDA NÃO PODE SER USADA, POIS ELA
13           * NÃO FAZ REFERÊNCIA A NENHUM OBJETO (ENDEREÇO DE MEMÓRIA)
14           */
15          Empregado empregado;
16
17          /* Constrói um novo objeto e guarda a
18           * referência na variável empregado
19           */
20          empregado = new Empregado("Fulano", "M", 500.00f);
21
22
23          /* Agora a variável objeto pode ser usada */
24          empregado.print();
25
26          /* Quebra a referência entre a variável e a memória
27           * A VARIÁVEL NÃO PODE MAIS SER USADA, POIS ELA NÃO FAZ
28           * MAIS REFERÊNCIA A NENHUM ENDEREÇO DE MEMÓRIA
29           * */
30          empregado = null;
```

Quando atribuímos um objeto a outro, na verdade estamos fazendo com que as duas variáveis objeto façam referência (apontem) para o mesmo endereço de memória.

Exemplo:

```
1  package classes;
2
3  import java.awt.Point;
4
5  public class TesteReferencia {
6
7      public static void main(String[] args) {
8
9          Point ponto1;
10         Point ponto2;
11
12         ponto1 = new Point(100, 100);
13         ponto2 = ponto1;
14
15         ponto1.x = 200;
16         ponto1.y = 200;
17
18         System.out.println("Ponto1 : x=" + ponto1.x + " y=" + ponto1.y);
19         System.out.println("Ponto2 : x=" + ponto2.x + " y=" + ponto2.y);
20
21     }
22 }
```

Garbage Collector:

O Garbage Collector (coletor de lixo), libera a memória utilizada por um objeto se não houver mais nenhuma referência para ela. Dependendo do algoritmo de coleta de lixo usado na máquina virtual Java em que o programa está sendo executado, nem todo objeto não referenciado é coletado. Um objeto mais antigo tem uma probabilidade menor de não ter referência do que um objeto novo, de modo de um algoritmo comum para a coleta de lixo analisa objetos mais antigos com menos freqüência do que objetos mais novos.

Liberando um objeto da memória:

Uma maneira de tornar um objeto candidato a ser retirado da memória pelo coletor de lixo, é atribuir um valor **null** à sua referência. Com isso o objeto torna-se não referenciado e pode ser retirado da memória pelo coletor de lixo.