UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 02 – IDE Netbeans e Conceitos Básicos de Programação Orientada a Objetos

1. INTERFACES DE DESENVOLVIMENTO PARA JAVA

Existem diversas Interfaces de Desenvolvimento (IDEs) que suportam a linguagem de programação Java. As IDEs simplificam e muito algumas tarefas da programação de aplicativos, oferecendo opções de preenchimento automático, ajuste de identação de código, identificação de erros e possíveis correções, gerenciamento das propriedades do projeto, simplificação de compilação e execução, apresentação da documentação de funções, entre muitas outras coisas. Algumas das mais conhecidas IDEs para programar em Java são: Netbeans, Eclipse e IntelliJ. Qualquer uma destas ferramentas poderá ser utilizada na disciplina, mas este material fará uso da IDE Netbeans, que pode ser adquirida gratuitamente no website: https://netbeans.org/downloads/. Note que esta IDE consome uma boa porção de memória ao ser executada.

Para se familiarizar com a ferramenta é interessante conhecer as principais regiões da sua interface. Para tanto, é apresentada na Figura 1 uma imagem da tela principal do Netbeans após o seu carregamento inicial. Foram demarcadas na Figura 1 algumas regiões importantes; elas são: (A) área de listagem dos projetos e seus respectivos arquivos de código fonte. Esta área pode ser usada para visualizar ou configurar informações dos projetos; (B) ferramentas de compilação e execução do projeto; (C) região destinada à edição e visualização do conteúdo dos arquivos de código fonte (na figura é mostrada apenas a página inicial de boas vindas do Netbeans); e (D) região de console que exibe as mensagens escritas em tela e permite a leitura de entrada pelo teclado.

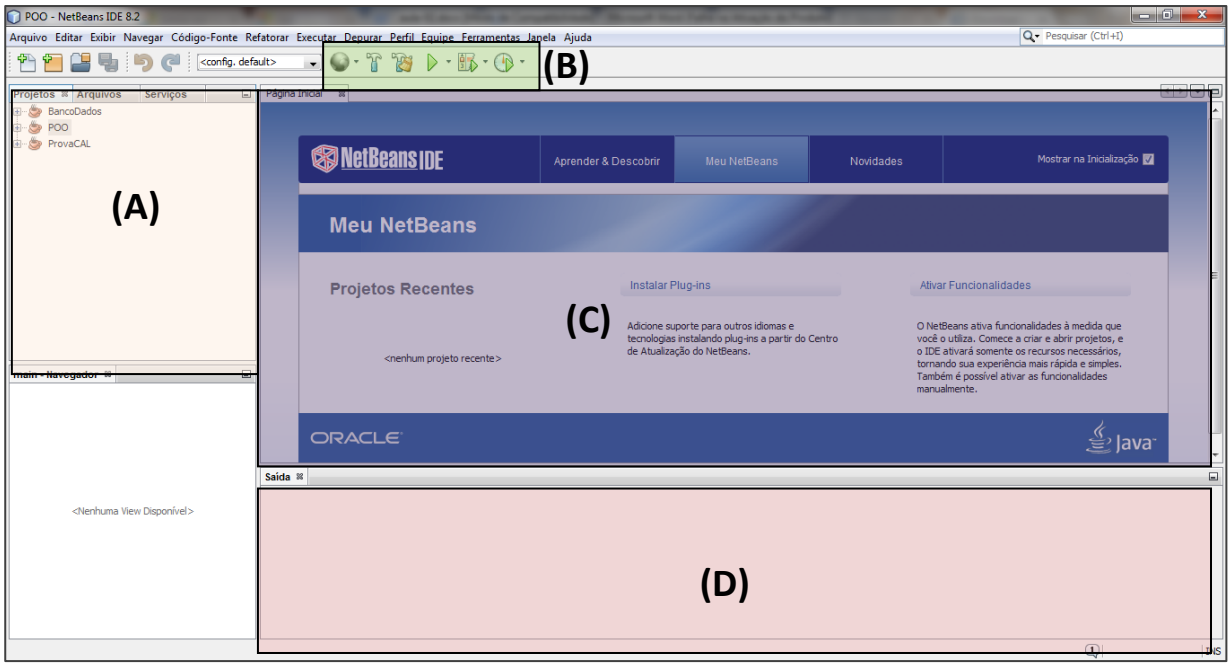

FIGURA 1 – Interface principal da IDE Netbeans

1.1. Criando e executando o primeiro projeto no Netbeans

Vamos criar o nosso primeiro projeto acessando a opção de menu "Arquivo" e então "Novo Projeto". Vamos escolher a categoria "Java", o tipo de projeto "Aplicação Java" e então pressionamos o botão "Próximo >". A próxima tela permite a edição de algumas configurações básicas do projeto. No campo nome do projeto vamos definir o nome do projeto como: POO; o campo de localização do projeto pode ser deixado como está (os projetos são salvos por padrão no diretório "Documentos\NetBeansProjects" na pasta do usuário do sistema). Vamos deixar a opção "criar classe principal" marcada e vamos deixar o nome da classe principal como está. Para finalizar o processo devemos pressionar o botão Finalizar e... pronto, o projeto foi criado e já podemos escrever o código da nossa aplicação Java.

Para realizarmos um teste vamos usar o código fonte abaixo e executar o programa utilizando a setinha verde da região de ferramentas de execução e compilação (região B da Figura 1). O programa deve pedir por um número inteiro e verifica e informa se tal número é primo ou não (OBS: não foi utilizada a abordagem mais eficiente para resolver este problema).

Arquivo: POO.java

```
 1.  package poo;
 2.  import java.util.Scanner;
 3.
 4.  /**
 5.   * @author udesc
 6.   */
 7.  public class POO {
 8.      /**
 9.       * @param args the command line arguments
10.       */
11.      public static void main(String[] args) {
12.          int n;
13.          Scanner scan = new Scanner(System.in);
14.          System.out.print("Digite um número inteiro: ");
15.          n = scan.nextInt();
16.
17.          boolean primo = true;
18.          for(int i=2; i<n; i++){
19.              if( n%i == 0){ // se n for divisível por i
20.                  primo = false;
21.                  break; // se for divisível já pode parar
22.              }
23.          }
24.          if( primo )
25.              System.out.println("O número "+n+" é primo!");
26.          else
27.              System.out.println("O número "+n+" não é primo!");
28.      }
29.  }
```

2. CONCEITOS BÁSICOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Como já mencionado, o paradigma orientado a objetos é uma forma de entender e representar sistemas complexos como estruturas hierárquicas de objetos. Dois conceitos primordiais na orientação a objetos são os conceitos de CLASSE e OBJETOS.

Uma CLASSE é um componente que tenta modelar objetos tangíveis ou intangíveis de forma computacional, associando-lhes um nome, atributos (variáveis) e métodos (comportamento ou funções). De certa forma, classes podem ser entendidas como uma extensão do conceito de STRUCT da linguagem de programação C, a qual tem como objetivo agrupar variáveis comuns que representam algo. Se precisarmos representar carros em um determinado sistema usando orientação a objetos, devemos pensar em um nome que identifica esse tipo de objeto, quais valores/atributos são necessários para caracterizá-lo e quais são as operações vinculadas a ele que são importantes para o sistema. A seguir é apresentada uma sugestão dos elementos mencionados:

Classe: Carros	
Atributos	**Métodos**
cor : String, ano : int, quilometragem : int valor : int proprietario : String	pintar(novaCor) lavar( ) alterarValor(novoValor)

Para representar esta classe em um programa Java, poderíamos criar uma nova classe com as definições especificadas acima. Uma possível codificação para a classe sugerida é apresentada no quadro abaixo.

> Uma recomendação de boa prática de programação diz que devemos nomear classes com a primeira letra em maiúsculo. Se for uma palavra composta todas as primeiras letras de cada palavra devem aparecer em maiúsculo.

Arquivo: Carro.java

```
 1.  package poo;
 2.
 3.  /**
 4.   * @author udesc
 5.   */
 6.  public class Carro {
 7.      String cor, proprietario;
 8.      int ano, quilometragem, valor;
 9.
10.      void pintar( String novaCor ){
11.          cor = novaCor;
12.      }
13.
14.      void lavar(){
15.          System.out.println("O carro está brilhando!");
16.      }
17.
18.      void alterarValor( int novoValor ){
19.          valor = novoValor;
20.      }
21.  }
```

Um OBJETO, por sua vez, é uma instância de uma classe. Isto é, uma representação de um elemento com valores para os atributos de uma classe e que pode utilizar métodos implementados nesta classe. Pode-se imaginar um OBJETO como uma variável de um tipo não primitivo ou então como uma porção de memória destinada a representar um elemento. Podemos instanciar (criar) um ou *n* OBJETOS da CLASSE (tipo) Carro, sendo que cada objeto terá os seus próprios valores de cor, proprietario, ano, quilometragem e valor, e os métodos (funções) pintar, lavar e alterarValor irão afetar apenas os atributos do objeto em questão.

A primeira questão importante a ser respondida é: como instanciar/criar um OBJETO referente a uma CLASSE? Para a criação de novos objetos usamos o operador (e palavra reservada) new. O operador new pode ser entendido como um malloc ou calloc da linguagem C, realizando a alocação de memória para o novo objeto que está sendo criado. A sua sintaxe é a seguinte:

```
Carro meu_carro = new Carro();
```

Após a instanciação de um objeto podemos utilizá-lo acessando os seus atributos e os seus métodos com o uso do operador ponto (.), semelhante à forma de acessar variáveis em uma STRUCT na linguagem C. Vale reforçar que se um novo OBJETO da classe Carro viesse a ser criado para uma nova variável, os seus valores não teriam vínculo com a instância do objeto/variável meu_carro.

```
meu_carro.valor = 50000;
meu_carro.pintar( "Preto Fosco" );
```

Exemplo: Criar três objetos da classe Carro, alterar seus atributos diretamente ou através dos métodos e mostrar seus valores na tela.

Nomenclatura: Quando uma variável é declarada dentro de um método/função, a denominamos simplesmente de variável. Entretanto, quando uma variável é declarada fora de qualquer função, ela pertence a classe, e por isso recebe o nome de atributo.

2.1. Atributos e Métodos Estáticos

Mas e nos casos nos quais é de interesse ter um atributo compartilhado entre cada objeto? Ou em outras palavras, e se for de interesse ter um atributo que pertence à classe?

Nestes casos utilizamos a propriedade static que faz com que um atributo ou método seja pertencente à CLASSE e compartilhado entre todos os OBJETOS desta determinada classe. No exemplo acima poderíamos especificar, por exemplo, o atributo proprietário como static caso o proprietário dos carros seja o mesmo para todas as instâncias (o proprietário é a revendedora, por exemplo). Desta forma ao alterar o proprietário em um determinado OBJETO da CLASSE carro, estaremos alterando o valor do proprietário para todos os OBJETOS da CLASSE, pois o atributo agora se refere à CLASSE e não mais a cada OBJETO individualmente.

Apesar de ser permitido alterar o valor de um atributo estático utilizando um objeto, esta ação não é uma boa prática. Aconselha-se neste caso, a utilizar o nome da CLASSE para acessar o atributo estático. As duas maneiras de alteração mencionadas são ilustradas abaixo:

```
meu_carro.proprietario = "Marques Carros"; (desaconselhável)
Carro.proprietario = "Marques Carros"; (boa prática)
```

Arquivo: Carro.java (atualização)

```
    ...
6.  public class Carro {
7.      String cor;
8.      static String proprietario;
9.      int ano, quilometragem, valor;
        ...
```

Além dos atributos estáticos também existem os métodos estáticos, e de forma semelhante, estes métodos são referentes às CLASSES e não aos OBJETOS. De fato, métodos estáticos não podem acessar ou alterar atributos não estáticos ou fazer uso de métodos não estáticos, pois estes pertencem a OBJETOS – a menos que estes atributos e métodos pertençam a uma variável que possa ser usada dentro do método (uma variável local, por exemplo). Por consequência, métodos estáticos só podem fazer uso de atributos e métodos estáticos ou variáveis locais (declaradas dentro do método estático). A declaração de um método estático é igual a declaração de um método comum, mas deve receber a palavra-chave static antes do tipo de retorno do método. Já a invocação de um método estático deve ser feita através do nome da CLASSE.

```
static boolean alterarProprietario( String nome ){ (…) }
Carro.alterarProprietario("Eu");
```

Com este novo conceito já é possível entender o que significa o static no método principal do programa (main) e o porquê não é possível invocar métodos não estáticos declarados na mesma CLASSE ou em outras CLASSES. Vale reforçar que atributos e métodos não estáticos podem ser usados dentro da definição de um método estático, desde que sejam acessados por um objeto, como uma variável local ou um parâmetro recebido pela função estática.

Desafio: é possível acessar o método renomear a partir do método main sem tornar o atributo nome estático? O que deve ser feito?

```
    ...
6.  public class POO {
7.      String nome;
8.      void renomear( String nome ){
9.          if( nome.length() > 3 )
10.             this.nome = nome;
11.     }
12.     public static void main(String[] args){
13.         // ???
14.     }
```

length é um método da Classe String que retorna o tamanho da String do objeto

Quando há um conflito de nome entre um atributo e uma variável com mesmo nome, deve-se utilizar a expressão **this.** seguida do nome do atributo para acessá-lo.

2.2 Encapsulamento (parte I)

Um problema muito usual no paradigma imperativo tem relação ao livre acesso as variáveis, principalmente em relação as variáveis de uma estrutura. Muitas vezes é comum ser necessário o uso de uma variável de controle interno da estrutura que não deveria ser acessada ou alterada em outras partes do programa (até porque isso poderia gerar um erro), ou então existir a necessidade de

garantir uma certa restrição de consistência para o valor que a variável pode assumir. Esse tipo de isolamento não é possível no paradigma imperativo/estruturado.

Considerando esta problemática do livre acesso a conteúdo que não deveriam ser acessados livremente, o paradigma de orientação a objetos propõe o conceito de encapsulamento que permite ocultar do mundo externo ao objeto os detalhes de implementação e restringir o acesso às propriedades e métodos. Assim, é possível especificar se um atributo ou método de uma classe é aberto (público) ou privativo à classe (privado), por exemplo, permitindo a criação de programas com menos erros e mais clareza. Ao todo existem quatro tipos de encapsulamento: público (*public*), privado (*private*), protegido (*protected*) e do pacote (*package*), mas por enquanto vamos nos ater e utilizar apenas aos dois primeiros:

- *public*: permite que um atributo ou método seja acessado ou modificado em qualquer parte do código fonte. Contudo, lembre-se que para acessar um método ou atributo não estático, que pertence aos objetos de uma determinada classe (e não a classe em si), deve-se utilizar um objeto (variável ou atributo) para utilizar o método.
- *private:* permite que um atributo ou método seja utilizado apenas dentro dos métodos da classe onde for declarado. Note que mesmo sendo um método estático, que pertence a classe, este só poderá ser acessado dentro da própria classe.

No Java, a identificação do encapsulamento de um método ou atributo deve ser escrita antes de qualquer elemento. Isto é, o encapsulamento aparece antes da indicação do tipo e antes da palavra reservada static na declaração de um atributo, e aparece antes da indicação do tipo de retorno e da palavra reservada static na declaração de um método. E quando uma variável ou atributo é declarado sem um encapsulamento, o que isso significa? Na linguagem de programação Java, todos os métodos e atributos que não recebem explicitamente um encapsulamento são do pacote (*package*), mas vamos falar das propriedades deste encapsulamento mais para frente.

2.3. *Getters* & *Setters*

Uma vez que existe encapsulamento, uma boa prática em orientação a objetos é definir como privado os atributos de controle interno da classe ou aqueles que possuem alguma restrição específica sobre os seus valores. Para estes atributos que possuem restrições é aconselhável o uso de métodos chamados de *getters* & *setters* que nada mais são do que métodos que retornam o valor atual do atributo (já que não pode ser acessado diretamente) e métodos que definem alguma regra para atribuição de valor ao atributo em questão.

Para exemplificar esta questão vamos considerar o nosso programa que contém a CLASSE Carro. Da forma que o código foi escrito, o atributo ano pode ser acessado e modificado de qualquer forma em qualquer parte do código, permitindo inclusive atribuição de valores não desejáveis ou incorretos para a lógica da aplicação, por exemplo:

```
Carro meu_carro = new Carro();
meu_carro.ano = -5000;
```

Para evitar que um uso errôneo seja feito e centralizar o controle de valores admissíveis e não admissíveis para um atributo, define-se o atributo como privado e implementasse um método público que altera este atributo seguindo as regras desejadas. Este método de alteração é chamado de *setter* e, como boa prática de programação, recebe o nome do atributo a ser alterado antecedido

pela palavra *set* (*setNomeAtributo).* Todavia, uma vez que o atributo ano agora é privado, ele não pode ser acessado diretamente através de seu objeto e é justamente neste cenário que entra os métodos *getters* que retornam o valor do atributo. No exemplo da CLASSE Carro, poderíamos fazer o seguinte para melhorar o código:

Arquivo: Carro.java (atualização)

```
    ...
 6. public class Carro {
        ...
10.     private int ano;
        ...
16.     public void setAno( int ano ){
17.         if( ano > 1700 && ano < 2500 )
18.             this.ano = ano;
19.         else
20.             System.out.println("@Carro: ano inválido!");
21.     }
22.     public int getAno(){
23.         return ano;
24.     }
        ...
42. }
```

Assinaturas usuais dos getters and setters:
getters: `public tipo getAtributo(){…}`
setters: `public void setAtributo(tipo atr){…}`

É comum os *getters* terem apenas uma instrução de retorno do valor do atributo, mas eles também podem conter alguma implementação.

Dica: No Netbeans é possível criar os métodos getters & setters automaticamente. Basta clicar com o botão direito em qualquer área do código fonte da classe em questão, escolher a opção "Refatorar" e a subopção "Encapsular Campos" (atalho: Ctrl+Alt+Shift+E). Deve-se escolher então quais *getters* & *setters* deseja-se adicionar automaticamente e clcar no botão "Refatorar".

Existe certa discussão em relação a quais métodos devem receber os métodos *getters* & *setters*. Por um lado, alguns dizem que deve-se utilizar este padrão para todos os atributos, inclusive para aqueles que não possuem restrições, a fim de manter um padronização única para o código e garantir que se alguma regra seja introduzida, que ela possa ser verificada em toda parte do código sem requerer muito esforço. Por outro lado, outros clamam que esta abordagem deve ser utilizada apenas para os atributos que possuem alguma restrição na definição de seu valor, e que definir tais métodos para todos os atributos é algo desnecessário que ocupa tempo, espaço e processamento.

2.4. Método Construtor

Lembra do operador new que cria uma instância de um objeto? Vamos entender melhor como ele funciona! Toda classe possui um método implícito (ou seja, não precisamos escrevê-lo) chamado de construtor básico ou trivial que basicamente cria um espaço na memória para o objeto. Para invocar este método especial usamos a sintaxe que você já conhece: new NomeClasse(). Mas e se quisermos realizar alguma operação específica ao criar/instanciar um novo objeto, ou garantir uma certa restrição ao criar o objeto, ou então receber um ou mais valores de entrada? Neste contexto entram os métodos construtores explícitos, que são escritos no código fonte da classe.

Métodos construtores possuem uma sintaxe específica apenas com a definição de escopo + nome da classe + lista de parâmetros. Não é utilizado o tipo de retorno nem o nome do método pois todo método construtor tem como retorno uma instância do objeto da classe que se está implementando e o nome do método construtor, por definição, é o próprio nome da classe. Dentro

do corpo do método construtor é possível especificar o que deve ser feito sempre que uma nova instância daquela classe (um novo objeto) é criada. Um exemplo é apresentado no quadro a seguir.

Arquivo: Carro.java (atualização)

```
    ...
 6. public class Carro {
        ...
12.     /**
13.      * Construtor da classe Carro
14.      * @param ano O ano de fabricação do carro
15.      * @param cor A cor atual do carro
16.      * @param prop O nome do proprietário do carro
17.      */
18.     public Carro( int ano, String cor, String prop ){
19.         this.cor = cor;
20.         this.ano = ano;
21.         proprietario = prop;
22.     }
23. }
```

A partir de agora lembre-se de documentar todos os seus códigos. Isto é muito útil e o Netbeans mostra a sua documentação sempre que o método for sugerido na caixa de "auto-completar" (Ctrl+Espaço).

Pronto, agora a classe Carro só pode ser instanciada caso seja passado um inteiro e duas Strings. Alguém pode perceber que o nosso código na CLASSE POO.java começou a acusar algum erro, sendo que nem alteramos esta CLASSE. O problema ocorreu porque uma vez que definimos um método construtor explícito qualquer, o método construtor implícito é descartado. Contudo, não se preocupe, este método construtor pode ser escrito novamente de forma explícita – e sim, é possível ter dois ou mais métodos construtores desde que a assinatura (quantidade e/ou tipos de dados) seja diferente entre cada um deles. Este último detalhe discutiremos mais adiante quando falarmos sobre sobrecarga de métodos. Enfim, teremos que alterar o código que instancia objetos do tipo Carro para algo similar a:

```
Carro meu_carro = new Carro( 2010, "Branco", "Revendedora" );
```

CURIOSIDADE: Ok, agora você já está apto a entender um pouco melhor o porquê da forma que escrevemos as funções de escrita e leitura do Java. Vamos lá:

System.out.println => acessamos a classe System (própria do Java), dentro desta classe temos um atributo público chamado out, que é uma instância da classe Stream. Este atributo, por ser um objeto da classe Stream, possui os métodos print, println e printf.

Scanner aux = new Scanner(System.in) => para fazer a leitura instanciamos um objeto da classe Scanner. Para instanciá-lo, porém, temos que passar um argumento para o método construtor – este argumento é o fluxo de onde os dados de entrada devem ser capturados. Para usarmos o fluxo de entrada padrão (teclado) usamos o System.in

Você sabia? É possível definir mais de uma CLASSE em um mesmo arquivo Java, mas toda CLASSE pública (que possui o escopo *public*) só pode aparecer no arquivo que possui o seu nome. Por exemplo, a CLASSE `public class Carro{ ... }` só pode ser declarada / implementada no arquivo Carro.java. Contudo, uma CLASSE que não tem o escopo público (ex: `class Moto{ ... }`) pode ser declarada e implementada em qualquer arquivo, mas ela só será visível / acessível pelas CLASSES que estão no mesmo pacote (pasta)!

2.5. Casting

A propriedade de *casting*, transformar tipos primitivos em outros tipos primitivos, não é uma exclusividade do paradigma orientado a objetos, mas merece atenção. Na linguagem de programação Java, nem sempre é possível atribuir valores de tipos primitivos diferentes a variáveis ou atributos – existem certas regras. A regra básica é sempre exigir *casting* explícito quando se poderá haver perda de informação no processo e *casting* implícito quando isto não ocorre. Para exemplificar, considere a atribuição de um valor real a uma variável inteira. A parte decimal do valor será perdida, portanto, Java exige que seja escrito um *casting* explícito, alertando ao programador que poderá haver perda de informação naquele trecho. Agora pense na situação contrária, uma atribuição de um valor inteiro a uma variável real. Se o número inteiro for muito grande até pode haver certa perda de precisão, mas ela é pequena em relação ao outro caso apresentado. Assim, o *casting* fica implícito na atribuição e não precisa ser escrito. A tabela abaixo ilustra os *castings* possíveis, sendo que a sigla (Impl.) indica um *casting* implícito.

PARA: / **DE:**	**byte**	**short**	**char**	**int**	**long**	**float**	**double**
byte	-	(Impl.)	(char)	(Impl.)	(Impl.)	(Impl.)	(Impl.)
short	(byte)	-	(char)	(Impl.)	(Impl.)	(Impl.)	(Impl.)
char	(byte)	(short)	-	(Impl.)	(Impl.)	(Impl.)	(Impl.)
int	(byte)	(short)	(char)	-	(Impl.)	(Impl.)	(Impl.)
long	(byte)	(short)	(char)	(int)	-	(Impl.)	(Impl.)
float	(byte)	(short)	(char)	(int)	(long)	-	(Impl.)
double	(byte)	(short)	(char)	(int)	(long)	(float)	-

Um caso que merece atenção é a atribuição entre variáveis do tipo real. Em Java, um valor do tipo primitivo *double* é representado por um número com ponto, ao passo que, para representar um valor do tipo *float* deve-se acrescentar um *f* ao final do número (sem espaço, conforme exemplo abaixo). Como o tipo *float* apresenta menos precisão que o tipo *double*, é necessário fazer o *casting* explícito conforme o exemplo apresentado a seguir:

```
float v1 = 3.14f;
float v2 = (float) 134.25;
```

Exercícios

1) Altere a classe Carro de modo a encapsular todos os atributos desta classe utilizando os métodos *getters* & *setters*. Previna que a cor e nome do proprietário tenham menos do que três caracteres, e que os atributos quilometragem e valor recebam valores negativos.

2) Crie três instâncias de carros no seu programa:
Proprietário: DogãoCar
Carro #1: cor azul escuro metálico, ano 2013, 32.055 km, avaliado em R$ 43.200
Carro #2: cor prata, ano 2006, 89.233 km, avaliado em R$ 23.900
Carro #3: cor preto fosco, ano 2013, 19.631 km, avaliado em R$ 139.800

Em seguida crie um novo método *show()* na classe Carro, que exibe todas as informações do carro na tela. Utilize então este novo método para mostrar as informações das instâncias de carros criadas.

3) Crie uma classe para abstrair personagens de um determinado jogo, onde cada personagem tem um nome e alguns atributos: força, constituição, destreza e agilidade. A seguir são definidas algumas restrições a serem seguidas:

- Um personagem só pode ser criado caso seu nome e atributos sejam informados, além disso, deve-se garantir que o valor mínimo para cada atributo seja 0 (zero), isto é, não se deve aceitar valores negativos.
- O nome do personagem não pode ser alterado depois de sua criação.
- Atributos não podem ser alterados livremente; estes só podem ser alterados de forma incremental por um valor positivo informado [ex: aumentaForca(5)].
- Deve haver uma função que retorna a quantidade de pontos de vida do personagem (HP) que é calculada pela fórmula: ($constiuição \times 100 + força \times 5$).
- A classe deve ter um contador do número de personagens criados durante a execução que pode ser acessado, mas não pode ser alterado externamente.
- Deve existir um método que exibe todas as informações do personagem

Por fim, crie três instâncias de personagens, utilize os métodos criados, mostre em tela o número de personagens criados (usando o contador da classe especificado) e os dados de cada um deles.

4) Crie um novo projeto com o seguinte método *main()* e implemente as classes Retangulo e Circulo (utilizadas no código abaixo), fazendo o programa executar conforme esperado.

```
public static void main(String[] args) {
    double largura = 2.5;
    double altura = 6.0;
    Retangulo ret = new Retangulo( largura , altura );
    ret.showInfo(); //imprimir medidas do retangulo
    System.out.println("area = " + ret.area() +
                       "perimetro = " + ret.perimetro());
    double raio = 3.75;
    Circulo cir = new Circulo( raio );
    cir.showInfo(); // imprimir valor do raio do círculo
    System.out.println("area = " + cir.area() +
                       "perimetro = " + cir.perimetro());
}
```