Capítulo 1 - Introdução a Programação Orientação a Objetos

1. Introdução à programação orientada a objetos

1.1 A base da Orientação a Objetos

Há três pilares para a Orientação a Objeto:

1. Encapsulamento,
2. Herança e
3. Polimorfismo.

Inicialmente vamos relacionar os conceitos de cada "pilar" com as frases abaixo.

4. Encapsulamento – "Não mostre as cartas de seu baralho".
5. Herança – "Filho de peixe, peixe é".
6. Polimorfismo – "Vamos nos adaptar".

Antes de partimos para o estudo de cada "pilar" da OO (Orientação a Objetos) vamos seguir com alguns conceitos básicos neste capítulo.

1.2 Porque Orientação a Objetos?

Inicialmente vamos olhar para os precursores da programação orientada a objetos (POO). Atualmente, quando usa um computador, você está tirando proveito de mais de 50 anos de refinamento. Antigamente, a programação era engenhosa: os programadores introduziam os programas diretamente na memória principal do computador, através de bancos de chaves (switches). Os programadores escreviam seus programas em linguagem binária, tal programação era extremamente propensa a erros e, a falta de estrutura, tornou a manutenção do código praticamente impossível. Além disso, o código da linguagem binária não era muito acessível.

Quando os computadores tornaram-se mais comuns, linguagens de nível mais alto e procedurais começaram; a primeira foi FORTRAN. Entretanto, linguagens procedurais posteriores como ALGOL, tiveram mais influência sobre a orientação a objetos (OO). As linguagens procedurais permitem ao programador reduzir um programa em procedimentos refinados para processar dados. Esses procedimentos geram a execução de um programa procedural. O programa termina quando acaba de chamar sua lista de procedimentos. Esse paradigma apresentou diversas melhorias em relação à linguagem binária, incluindo a adição de uma estrutura de apoio: o procedimento. As funções menores não são apenas mais fáceis de entender, mas também são mais fáceis de depurar. Por outro lado, a programação procedural limita a reutilização de código. Isso leva os programadores, com muita freqüência, a produzir "código de espaguete" – código cujo caminho de execução se assemelhava a uma tigela de espaguete. Dessa forma, a natureza voltada aos dados da programação procedural causou alguns problemas próprios.

Como os dados e o procedimento são separados, não existe nenhum encapsulamento desses dados, isso exige que cada procedimento saiba como

manipulá-los corretamente. Infelizmente, um procedimento com comportamento errôneo poderia introduzir erros se não manipulasse os dados corretamente. Uma mudança na representação dos dados exigiria alterações em cada lugar que os acessassem. Assim, mesmo uma pequena alteração poderia levar a uma cascata de alterações, por todo o programa – em outras palavras, um pesadelo de manutenção.

A programação modular, com uma linguagem como MODULA2, tenta melhorar algumas das deficiências encontradas na programação procedural. A programação modular divide os programas em vários componentes ou módulos constituintes. Ao contrário da programação procedural, que separa dados e procedimentos, os módulos combinam os dois. Um módulo consiste em dados e procedimentos para manipular esses dados. Quando outras partes do programa precisam usar um módulo, elas simplesmente exercitam a interface do módulo. Como os módulos ocultam todos os dados internos do restante do programa, é fácil introduzir a idéia de estado: um módulo contém informações de estado que podem mudar a qualquer momento. Mas a programação modular possui as seguintes deficiências:

1. Os módulos não são extensíveis, significando que você não pode fazer alterações incrementais em um módulo sem abrir o código fonte "a força" e fazer as alterações diretamente;
2. Os módulos não podem se basear em um outro, a não ser através de delegação;
3. E embora um módulo possa definir um tipo, um módulo não pode compartilhar o tipo de outro módulo.

Finalmente, a programação modular também é um híbrido procedural que ainda divide um programa em vários procedimentos. Porem, em vez de atuar em dados brutos, esses procedimentos manipulam módulos.

A Programação Orientada a Objetos (POO) dá o próximo passo lógico após a programação modular, adicionando herança e polimorfismo ao módulo. A POO estrutura um programa, dividindo-o em vários objetos de alto nível. Cada objeto modela algum aspecto do problema a ser resolvido. Escrever listas seqüenciais de chamadas de procedimento para dirigir o fluxo do programa não é mais o foco da programação sob a OO. Em vez disso, os objetos interagem entre si, para orientar o fluxo global do programa. De certa forma, um programa OO se torna uma simulação viva do problema.

Os objetos nos obrigam verificar, em nível conceitual, tudo o que eles fazem, ou seja, nos fazem listar todos os seus comportamentos. Ver um objeto a partir do nível conceitual não é definir sua *implementação* (em termos de programação, implementação é o código). Essa mentalidade nos obriga a pensar em programas em termos naturais e reais. Ou seja, os objetos permitem que você modele programas nos substantivos, verbos e adjetivos do *domínio* de seu problema. Sendo o domínio, o espaço onde um problema reside, o conjunto de conceitos que representam os aspectos importantes do problema.

Quando recua e pensa nos termos do problema que está resolvendo, você evita se emaranhar nos detalhes da implementação. Provavelmente alguns de seus objetos de alto nível precisarão interagir com o computador. Entretanto, o objeto isolará essas interações do restante do sistema.

Vamos pensar em um objeto "carrinho de compras". Ocultação da implementação significa que o caixa não vê dados brutos ao totalizar um

pedido. O caixa não sabe procurar, em certas posições de memória, números de item e outra variável para um cupom. Em vez disso, o caixa interage com objetos item. Ele sabe perguntar quando custa o item.

Antes de prosseguir, como exercício, tente imaginar os objetos *item, carrinho de compras e caixa*. A seguir você pode ver uma tabela de uma possível representação simplificada dos objetos.

Objeto	**Descrição**
Item	Representa um produto, pode fornecer dados sobre ele, como preço, foto, descrições, dimensão, peso, quantidade...
Carrinho de compras	Armazena itens. Um item pode ser incluído ou retirado.
Caixa	Contabiliza o valor dos itens do carrinho de compras e efetua a compra.

Agora já podemos definir formalmente um objeto: **Um *objeto* é uma construção de software que encapsula *estado* e *comportamento*. Os objetos permitem que você modele seu software em termos reais e abstração.**

Rigorosamente um objeto é uma instância de uma classe. Sobre o que é uma classe veremos mais adiante, mas já podemos dizer que uma instância é o objeto propriamente dito alocado em memória.

Assim como o mundo real é constituído de objetos, o software orientado a objetos também o é. Em uma programação OO pura, tudo é um objeto, desde os tipos mais básicos, como inteiros e lógicos, até as instâncias de classes mais complexas; nem todas as linguagens orientadas a objeto chegam a esse ponto. Em algumas (como o C# e Java), primitivas "int" e "float" não são tratadas como objetos, assim como no Flash também há limitações.

1.3 O que é uma classe?

Assim como os objetos do mundo real, o mundo da Programação Orientada a Objetos (POO) agrupa os objetos pelos comportamentos e atributos comuns.

A biologia classifica todos os cães, gatos, elefantes e seres humanos como mamíferos. Características compartilhadas dão a essas criaturas separadas um senso de comunidade. No mundo do software, as classes agrupam objetos relacionados da mesma maneira.

Uma classe define todas as **características comuns a um tipo de objeto**. Especificamente, a classe **define todos os atributos e comportamentos** expostos pelo objeto. A classe define às quais mensagens o seu objeto responde. Quando um objeto quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Freqüentemente, isso é referido como "envio de mensagem".

Uma *classe* define os atributos e comportamentos comuns compartilhados por um tipo de objeto. Imagine a classe como a forma do objeto. Os objetos de certo tipo ou classificação compartilham os mesmos comportamentos e

atributos. As classes atuam de forma muito parecida com um cortador de molde ou biscoito, no sentido de que você usa uma classe para criar ou *instanciar* objetos.

Os *atributos* são as características de uma classe visíveis externamente. A cor dos olhos e a cor dos cabelos são exemplos de atributos. Um objeto pode expor um atributo fornecendo um link direto a alguma variável interna ou retornando o valor através de um método.

Comportamento é uma ação executada por um objeto quando passada uma mensagem ou em resposta a uma mudança de estado: é algo que um objeto faz. Um objeto pode executar o *comportamento* de outro, executando uma operação sobre esse objeto. Você pode ver os termos: *chamada de método, chamada de função* ou *passar uma mensagem*; usados em vez de executar uma *operação*. O que é importante é que cada uma dessas ações omite o comportamento de um objeto.

1.4 Classes e objetos

Pegue um objeto item, por exemplo. Um item tem uma descrição, id, preço unitário, quantidade e um desconto opcional. Um item saberá calcular seu preço descontado.

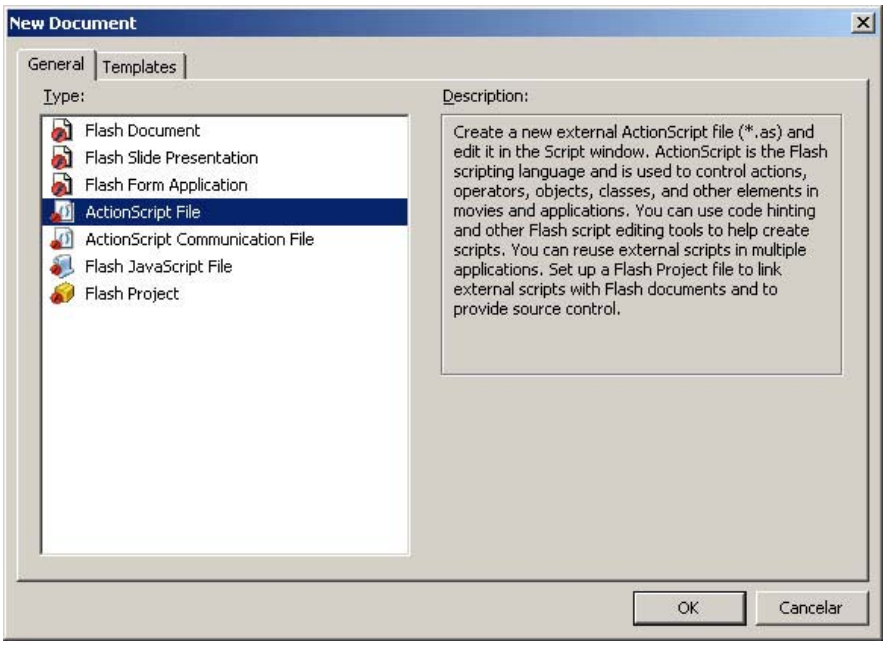
No mundo POO, você diria que todos os objetos item são instâncias da classe Item. Uma classe Item poderia ser definida, como segue:

Antes de analisar os códigos abaixo perceba que:

> *this*. é uma referência que aponta para a instância do objeto. Cada objeto possui sua própria referência para si mesmo. A instância usa essa referência para acessar suas próprias variáveis e métodos.

Obs: Todos os exemplos da apostila serão feitos em Actionscript e Java, assim temos como fazer um comparativo entre as linguagens.

Crie um novo Documento do tipo Actionscript (AS)



Insira o código abaixo e salve o arquivo com o mesmo nome da Classe (“Item”)

No momento, não se foque em entender perfeitamente o código, mas sim em perceber a estrutura de um código orientado a objetos.

Em Actionscript

```
class Item
{
	//ATRIBUTOS - INICIO
	private var unit_price :Number = 0;
	//uma porcentagem de discount
	//que se aplica ao preço
	private var discount:Number = 0;
	private var quantity:Number = 0;
	private var description:String;
	private var id :String;
	//ATRIBUTOS - FIM

	//CONSTRUTORES - INICIO
	public function Item (id:String , description:String ,
	                                  quantity:Number , price: Number)
	{
		this.id = id;
		this.description = description;
		this.unit_price  = price;
		setQuantity(quantity);
	}
	//CONSTRUTORES - FIM

	//PROPRIEDADES - INICIO
	public function setQuantity (valor:Number)//recebe o valor
	{//de fora para dentro
		if (valor >= 0 )
		{
			this.quantity = valor;
		}
	}

	public  function getQuantity():Number//envia o valor
	{//de dentro para fora
		return quantity;
	}


	public  function setDiscont(valor:Number)
	{
		if ( (valor <= 1) && (valor >= 0) )
		{
			this.discount = discount;
		}
	}

	public function getDiscont():Number
	{
		return discount;
	}

	public function getProductID():String
	{
		return id;
	}

	public  function getDescription():String
```

```
        {
                return description;
        }

        public function getAdjustedTotal():Number
        {
                var total:Number = unit_price  * quantity;
                var total_discount :Number = total * discount;
                var adjusted_total :Number = total - total_discount ;

                return adjusted_total;
        }
        //PROPRIEDADES - INICIO
}
```

Em Java

```
public class Item
{
        //ATRIBUTOS - INICIO
        private double unit_price = 0;
        //uma porcentagem de desconto
        //que se aplica ao preço
        private double discount = 0;
        private int quantity = 0;
        private String description;
        private String id;
        //ATRIBUTOS - FIM

        //CONSTRUTORES - INICIO
        public Item (String id, String description,
                            int quantity, double price)
        {
                this.id = id;
                this.description = description;
                this.unit_price = price;
                setQuantity(quantity);
        }
        //CONSTRUTORES - FIM

        //PROPRIEDADES - INICIO
        public void setQuantity (int value)//recebe o valor
        {//de fora para dentro
                if (value >= 0 )
                {
                        this.quantity = value;
```

```
		}
	}
	public int getQuantity()//envia o valor
	{//de dentro para fora
		return quantity;
	}
	public void setDiscont (double value)
	{
		if ( (value <= 1) && (value >= 0) )
		{
			this.discount = discount;
		}
	}
	public double getDiscont()
	{
		return discount;
	}
	public String getProductID()
	{
		return id;
	}
	public String getDescription()
	{
		return description;
	}
	public double getAdjustedTotal()
	{
		double total = unit_price * quantity;
		double total_discount = total * discount;
		double adjusted_total = total - total_discount;

		return adjusted_total;
	}
	//PROPRIEDADES - INICIO
}
```

Observem o método a seguir:

Actionscript

```
public function Item (id:String , description:String , quantity:Number , price: Number)
```

Java

```
public Item (String id, String description, int quantity, double price)
```

Esse tipo de método é chamado de *constructor*. Os construtores inicializam um objeto durante sua criação.

Métodos como:

- setDiscount()
- getDescription()
- getAdjustedTotal()

São todos comportamentos da classe Item que retornam ou configuram atributos. Quando um caixa quer totalizar o carrinho, ele simplesmente pega cada item e envia ao objeto a mensagem getAdjustedTotal()

Já unit_price, discount, quantity, description e id são todas variáveis internas da classe Item. Esses valores compreendem o *estado* do objeto. O estado de um objeto pode variar com o tempo. Por exemplo, ao fazer compras, um consumidor pode aplicar um cupom ao item. Aplicar um cupom ao item mudará o estado do item, pois isso mudará o valor de discount. Podemos dizer também que essas variáveis são *atributos da classe*, pois mesmo sendo privadas o conteúdo delas pode ser visível exteriormente através de uma *propriedade*.

As *propriedades* são métodos identificados pelas palavras-chave "get" (identificam métodos *acessores*) e "set" (identificam métodos *mutantes*). **Os *acessores* dão acesso aos dados internos de um objeto.** Entretanto, os *acessores* ocultam o fato de os dados estarem em uma variável, em uma combinação de variáveis ou serem calculados. Os *acessores* permitem que você mude ou recupere o valor e têm 'efeitos colaterais' sobre o estado do objeto**. Os *mutantes* permitem que você altere o estado interno do objeto.** Um mutante pode processar sua entrada como quiser, antes de alterar o estado interno do objeto. Observe o método setDiscount(), esse método garante que o desconto não seja maior que 100%, antes de o aplicar.

Ao serem executados, seus programas usam classes como a Item para criar ou instanciar os objetos que compõem o aplicativo. Cada nova instância é uma duplicata da última. Entretanto, uma vez instanciada, a instância transporta comportamentos e controla seu estado. Então, o que inicia sua vida como clone poderia se comportar de maneira muito diferente durante sua existência.

Por exemplo, se você criar dois objetos item a partir da mesma classe Item, um objeto item poderá ter um desconto de 11%, enquanto o segundo pode não ter desconto. Alguns itens podem ser mais caros que outros. Assim, embora o estado de um item possa variar com o passar do tempo, a instância ainda é um objeto Item. Considere o exemplo da biologia; um mamífero de cor cinza é tão mamífero quanto outro de cor marrom.

Vamos ver a seguir, como utilizar em um programa a classe item para criar (instanciar) diferentes objetos.

Crie um novo documento Flash (FLA) com nome “compra”.

Podemos instanciar uma classe a partir de outra classe ou um arquivo FLA, contanto que os arquivos de código da aplicação façam parte do mesmo *projeto* (inicialmente você pode imaginar um projeto como uma pasta que contém todos seus arquivos).

Actionscript

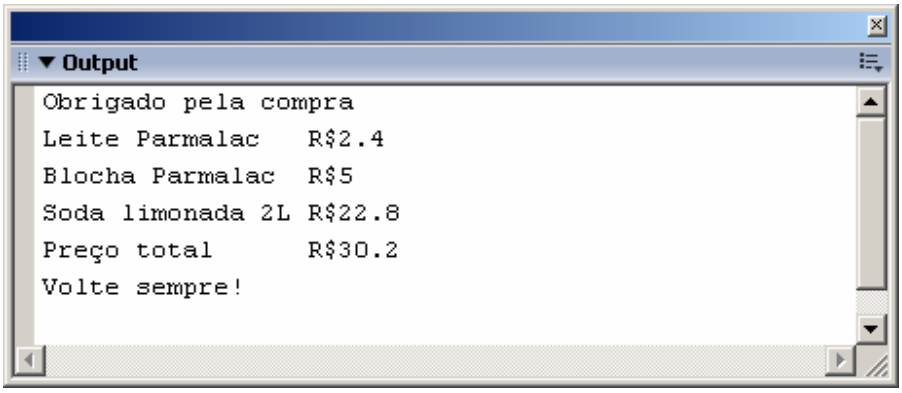
```
var leite   = new Item("parmalac-069", "Leite Parmalac", 2, 1.20);
var bolacha = new Item("parmalac-051", "Blocha Parmalac", 5, 1.00);
var soda  = new Item("antarquica-033", "Soda limonada 2L", 12, 1.90);

//aplica descontos
leite.setDiscont(0.15);

//obtém preços ajustados
var p_leite   = leite.getAdjustedTotal();
var p_bolacha = bolacha.getAdjustedTotal();
var p_soda    = soda.getAdjustedTotal();

//imprime recibo
trace("Obrigado pela compra ");
trace( leite.getDescription() + "\t R$" + p_leite);
trace( bolacha.getDescription() + "\t R$" + p_bolacha);
trace( soda.getDescription() + " R$" + p_soda);
//calcula e imprime o total
var total = p_leite+p_bolacha+p_soda;
trace( "Preço total \t R$" + total);
trace("Volte sempre!");
```

Java

```
public class Executor {
	/*
	 * Ponto de entrada da Aplicação
	 * */
	public static void main(String[] args) {
				//cria os itens
	Item leite = new Item("parmalac-069","Leite Parmalac",2,1.20);
	Item bolacha = new Item("parmalac-051","Blocha Parmalac", 5, 1.00);
	Item soda = new Item("antarquica-033","Soda limonada 2L",12, 1.90);

	//aplica descontos
	leite.setDiscont(0.15);

	//obtém preços ajustados
	double p_leite   = leite.getAdjustedTotal();
	double p_bolacha = bolacha.getAdjustedTotal();
	double p_soda    = soda.getAdjustedTotal();

	//imprime recibo
	System.out.println("Obrigado pela compra ;)");
	System.out.print("\n");//pula uma linha
	System.out.println( leite.getDescription() + "\tR$" + p_leite);
	System.out.println( bolacha.getDescription() + "\tR$" + p_bolacha);
	System.out.println( soda.getDescription() + "\tR$" + p_soda);
	System.out.print("\n");//pula uma linha
	//calcula e imprime o total
	double total = p_leite+p_bolacha+p_soda;
	System.out.println( "Preço total \t R$" + total);
	System.out.print("\n");//pula uma linha
	System.out.println("Volte sempre!");
```

```
}
}
```

Veja o exemplo da impressão do recibo:



1.5 Relacionamentos de objeto

O como os objetos se relacionam é muito importante para a programação orientada a objetos. Os objetos podem se relacionar de duas maneiras importantes.

1. Os objetos podem existir independentemente uns dos outros. Dois objetos de Item podem aparecer no carrinho de compras simultaneamente. Se esses objetos separados precisarem interagir, eles interagirão passando mensagens um para o outro (‘Passar uma mensagem’ é o mesmo que chamar um método para mudar o estado do objeto ou para exercer um comportamento).
2. Um objeto poderia conter outros objetos. Assim, como os objetos compõem um programa em POO, eles podem compor outros objetos através da agregação. A partir do exemplo Item, você poderia notar que o objeto item contém muitos outros objetos. Por exemplo, o objeto item também contém uma descrição e uma id (identidade). A descrição e a id são objetos de String. Cada um desses objetos tem uma interface que oferece métodos e atributos. Lembre-se de que, na POO, tudo é um objeto, mesmo as partes que compõem um objeto.

A comunicação funciona da mesma maneira entre um objeto e os objetos que ele contém. Quando os objetos precisarem interagir, eles trocam mensagens.

Mensagens é um importante conceito da Orientação a Objetos. Elas permitem que os objetos permaneçam independentes. Quando um objeto envia uma mensagem para outro, geralmente ele não se preocupa com a maneira como o objeto escolhe transportar o comportamento solicitado. O objeto solicitante se preocupa apenas que o comportamento aconteça.

1.6 Vantagens e Objetivos da Orientação a Objetos

A POO (programação orientada a objetos) se esmera em produzir software que tenha as seguintes características:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível
6. Oportunos

Natural

> Programar utilizando a terminologia de seu problema em particular. Não há a necessidade de se aprofundar nos detalhes técnicos enquanto se projeta o programa. Mas é importante "pensar com os pés no chão", saber que a funcionalidade que se determina é possível de se implementar.

Confiável

> Os objetos isolam o conhecimento e a responsabilidade de onde pertencem.

Reutilizável

> Devemos reinventar a roda? Criar classes bem feitas é uma tarefa difícil que exige concentração e atenção à abstração. Mas classes bem feitas podem ser reutilizadas. Uma vez que o problema esteja resolvido, você deve reutilizar a solução.

Manutenível

> Corrigir o erro em um único lugar.

Extensível

> O software não é estático. Ele deve crescer e mudar com o passar do tempo, para permanecer útil. A POO apresenta ao programador vários recursos para estender o código, como herança, polimorfismo, sobreposição, delegação e uma variedade de padrões de projetos.

Oportunos

> O ciclo de vida do projeto de software moderno é freqüentemente medido em semanas. A POO diminui o tempo de desenvolvimento, fornecendo software confiável, reutilizável e facilmente extensível.
>
> Quando se divide um programa em vários objetos, o desenvolvimento de cada parte pode ocorrer em paralelo. Vários desenvolvedores podem trabalhar em classes paralelas.

1.7 Armadilhas

Inicialmente em Orientação a Objetos existem quatro armadilhas a serem evitadas.

1. Pensar na POO simplesmente como uma linguagem

   Você não programa OO só porque utiliza uma linguagem que implementa a OO. POO é um estado da mente que exige que você veja seus problemas como um grupo de objetos e use encapsulamento, herança e polimorfismo corretamente.

2. Medo da reutilização

   Aprender a reutilizar sem culpa freqüentemente é uma das lições mais difíceis de aprender e há basicamente duas dificuldades que passamos aqui:

   a. Os programadores gostam de criar;

   b. E aqueles que não confiam no software que não escrevem.

3. Pensar em OO como uma solução para tudo

   Existem ocasiões que você não deve utilizar a OO. Há a necessidade do uso do bom senso para a escolha da ferramenta correta. O sucesso aparece somente com planejamento, projeto e codificação cuidadosos.

4. Programação egoísta

   Lembre-se dos outros desenvolvedores quando programar. Faça interfaces limpas e inteligíveis. O mais importante: escreva documentação. Documente suposições, parâmetros de métodos, documente o máximo que você puder.

1.8 Resumo

Você fez um passeio pela programação orientada a objetos. Você começou vendo a evolução dos principais paradigmas de programação e aprendeu alguns dos fundamentos da POO. Agora, você entenderá as idéias conceituais da OO, como, por exemplo, o que é uma classe e como os objetos se comunicam.

Definições são importantes, mas nunca devemos perder o rumo do que estamos tentando fazer usando OO, nos atendo ao 'como' do que estivermos fazendo. As seis vantagens e objetivos resumem o que a programação orientada a objetos espera cumprir:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível

5. Extensível
6. Oportunos

Você nunca pode perder esses objetivos de vista.

1.9 Perguntas – Exercícios

1) O que é programação orientada a objetos?

2) Quais são as seis vantagens e objetivos da programação orientada a objetos?

3) Explique um dos objetivos da programação orientada a objetos?

4) Defina os seguintes termos:
   i. Classe
   ii. Objeto
   iii. Comportamento

5) Como os objetos se comunicam entre si?

6) O que é um construtor?

7) O que é um acessor?

8) O que é um mutante?

9) O que é "this"?

10) O que são atributos?

11) O que são propriedades?

Ao terminar de responder as questões acima, envie-as em um arquivo texto ao seu tutor.