# Aula 2

# Introdução à Orientação a Objetos

## Objetos

Um **objeto** é uma entidade que encapsula informações de estado ou dados e possui um conjunto de operações associadas que manipulam estes dados. Uma **operação** é definida como sendo uma ação que um objeto realiza sobre outro para elicitar uma reação. Em geral, o estado de um objeto é completamente escondido e protegido de outros objetos e a única maneira de examiná-lo é através da invocação de uma operação (isto é, o envio de uma mensagem) para este fim. Objetos apresentam um comportamento bem definido e uma identidade que é única. **Comportamento** define o modo como um objeto age e reage em termos das suas mudanças de estado e envio de mensagens e é completamente definido pelas suas operações. **Identidade** é a propriedade de um objeto que o distingüe de outros objetos.

Um objeto comunica-se com outro através de mensagens que identificam operações a serem realizados no objeto receptor da mensagem. O objeto responde a uma mensagem mudando possivelmente seu estado e/ou retornando um resultado. O conjunto de operações de um objeto que podem ser invocados por outros objetos compreende a **interface pública** do objeto. Portanto, a visão externa de um objeto nada mais é do que a sua interface pública.

No contexto de programação, um objeto é um conjunto de variáveis e métodos relacionados, que representam o estado do objeto e modificam este estado, respectivamente. Os **métodos** são implementações das operações. Objetos em um software podem representar entidades do mundo real. Por exemplo, poderíamos representar um cachorro como um objeto de software em um programa de animação ou uma bicicleta em um programa que controla exercícios em bicicletas ergométricas. Entretanto, pode-

mos também representar entidades imaginárias, tais como um evento de pressionar o mouse no ambiente Windows.

Tudo que um objeto de software sabe (estado) ou pode fazer (comportamento) é expressado pelas variáveis e métodos dentro daquele objeto. Um objeto de software que modela uma bicicleta do mundo real, por exemplo, teria variáveis que indicariam o estado atual da bicicleta: sua velocidade é 10 Km/h, seu pedal gira a 90 rpm e sua marcha atual é a quinta. Estas variáveis são conhecidas como **variáveis de instância**. A bicicleta de software poderia possuir métodos para frear, mudar a rotação do pedal, mudar a marcha e assim por diante. Tudo que um objeto não conhece ou não pode fazer é excluído dele. Por exemplo, a nossa bicicleta não possui um nome e nem sabe latir. Logo, não há variáveis nem métodos para aquele estado e comportamento na bicicleta.

# Classes

Uma **classe** é a descrição de um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares. Todo objeto é **instância** de apenas uma classe. Toda classe possui um nome e um corpo que define o conjunto de atributos e operações possuídas pelas suas instâncias. É importante distingüirmos objetos de suas classes; o termo classe é usado para identificar um grupo de objetos e o termo objeto é usado para identificar uma instância particular de uma classe.

No modelo de objetos, atributos e operações são parte da definição de uma classe. Os **atributos** são propriedades nomeadas de um objeto e armazenam o estado abstrato de cada objeto. Operações caracterizam o comportamento de um objeto e são o único meio para fazer acesso, manipular e modificar os valores dos atributos de um objeto. A classe define a interface pública e contém a implementação das operações (isto é, os métodos) de seus objetos.

No contexto de programação, uma classe de objetos define um tipo de objeto, isto é, as variáveis de instância e os métodos de um tipo de objeto. Por exemplo, poderíamos criar uma classe bicicleta que declara várias variáveis de instância para conter a marcha atual, a rotação atual do pedal e assim por diante, para cada objeto bicicleta. A classe também declararia e forneceria implementações para os métodos que permitem o ciclista mudar as marchas e a rotação do pedal.

## Classes e Métodos em Java

Vamos, através de um problema, estudar como podemos aplicar os conceitos vistos antes e gerar um pequeno trecho de programa que define as classes envolvidas com o problema. Consideremos, primeiro, o problema: *suponha que você seja solicitado para construir um programa orientado a objetos que simula o jogo de cartas "paciência" usando uma interface gráfica*. Sem precisar fazer uma análise detalhada do problema, podemos afirmar que uma "carta" é um elemento fundamental do problema. Então, apenas com o intuito de exemplificar os conceitos vistos antes, vamos nos concentrar na representação desse elemento como um objeto.

Uma carta de baralho, por si mesma, possui informações e comportamento bem definidos e independentes do jogo em que ela está sendo utilizada. Isto é, toda carta possui uma cor, um naipe (ouro, copas, paus ou espada) e um valor, que indica sua posição entre as demais cartas do mesmo naipe. Portanto, podemos descrever uma classe `Carta` para representar o objeto carta do mundo real, contendo os atributos `cor`, `naipe` e `valor`. `Carta` também poderia conter os métodos `cor`, `naipe` e `valor`, que retornam a cor, o naipe e o valor dos objetos da classe `Carta`, e os métodos `apagar`, `desenhar`, `paracima` e `vira`, que desfaz a imagem de uma carta na interface, desenha uma carta, verifica se a face da carta está virada para cima e vira uma carta, respectivamente.

Em Java, a classe `Carta` poderia ser definida como:

```
class Carta {

   // valores constantes para cores e naipes
   final public int vermelha = 0;
   final public int preta = 1;
   final public int espada = 0;
   final public int copas = 1;
   final public int ouro = 2;
   final public int paus = 3;

   // atributos
   private boolean faceparacima;
   private int v;
   private int n;

   // construtor
   public Carta(int vv, int vn) {v = vv; n = vn;
```

```
                                faceparacima = false;}

   // metodos para acessar os atributos
   public int valor() {return v;}
   public int naipe() {return n;}
   public int cor() {
     if (naipe() == ouro || naipe() == copas)
        return vermelha;
     return preta;
   }

   public boolean faceParaCima() {return faceparacima;}

   // desenha uma carta
   public void draw (Graphics g, int x, int y) {
     // codigo omitido
   }

   public void vira() {faceparacima = !faceparacima;}
}
```

Como pode ser visto acima, descrições de classe em Java iniciam com a palavra-chave `class`. A palavra-chave `public` precede aqueles elementos da classe que fazem parte da sua interface pública, como é o caso dos métodos `valor` e `naipe`. Já a palavra-chave `private` precede aqueles elementos da classe que não fazem parte da interface pública da classe e podem apenas ser acessados pelos métodos da classe e por nada mais.

O método `Carta(int, int)` na descrição da classe é único em muitos aspectos. Ele além de possuir o mesmo nome da classe à qual ele pertence, também não há tipo de retorno, nem mesmo `void`. Este método é denominado **construtor** e serve para inicializar objetos da classe quando estes são criados. A inicialização de objetos será estudada com detalhes mais adiante. As palavras-chaves `void`, `int` e `boolean` antes dos nomes dos métodos da classe `Carta` indicam que o método não retorna valor, que o valor de retorno do método é um inteiro e que o método retorna um valor lógico (`true` ou `false`), respectivamente.

A implementação dos métodos deve ser fornecida obrigatoriamente na definição da classe, ao invés de opcionalmente separado da classe, como em C++. O protótipo de um método e seu corpo são descritos de forma muito semelhante àquela usada na linguagem C. Com exceção do construtor, todos os métodos devem especificar um

valor de retorno, o qual pode ser `void`.

Em Java, não há pré-processador, variáveis globais ou tipos de dados enumerados (`enum`). Constantes simbólicas podem ser criadas através da declaração e inicialização de uma variável com o uso da palavra-chave `final`. Tais constantes não podem ter seus nomes associados a outros valores subsequentemente à criação e inicialização delas. Java não possui apontadores, referências, comando `goto` e sobrecarga de operadores. Não há também o conceito de passagem de parâmetros por referência. Veremos, no decorrer do curso, como lidaremos com isso.

# Envio de Mensagens

Nós usamos o termo **envio de mensagem** para designar o processo *dinâmico* de pedir a um objeto para executar uma ação específica. O envio de uma mensagem lembra a chamada a um procedimento, mas possui algumas diferenças fundamentais:

- Uma mensagem é sempre enviada *para* algum objeto, denominado receptor.
- A ação executada em resposta a uma mensagem não é sempre a mesma. Ao invés disso, ela é dependente da classe do receptor. Isto é, objetos distintos podem aceitar a mesma mensagem e ainda executar ações distintas.

Há três componentes em qualquer expressão de envio de mensagens. Eles são o **receptor**, que é o objeto para o qual a mensagem está sendo enviada, o **seletor da mensagem**, que é um identificador que indica a mensagem sendo enviada, e os **argumentos** usados como parâmetros da mensagem. Embora o conceito de envio de mensagens seja fundamental em programação orientada a objetos, os termos utilizados nas várias linguagens e a sintaxe empregada para representar as idéias variam enormemente.

## Envio de Mensagens em Java

A sintaxe usada para envio de mensagens em Java é quase idêntica aquela utilizada pela linguagem C++. Seja `umaCarta` uma instância (objeto) da classe `Carta` definida antes, então o comando

```
umaCarta.draw(win,25,37);
```

consiste no envio de uma mensagem que seleciona o método `draw` com os argumentos `win`, 25 e 37 ao objeto `umaCarta`. Como veremos mais adiante, esta mensagem pede ao objeto `umaCarta` para que ele se desenhe em uma dada janela (`win`) nas coordenadas 25 e 37 da janela.

Mesmo quando nenhum argumento é requerido na mensagem, os parênteses são ainda necessários. Por exemplo, na expressão

```
umaCarta.faceParaCima();
```

a mensagem não possui quaisquer argumentos, mas ainda assim devemos usar os parênteses após o nome do método `faceParaCima()`. Esta exigência deve-se ao fato de que o compilador precisa saber se o nome `faceParaCima` se refere a um método ou uma variável de instância do objeto `umaCarta`. Quando usamos os parênteses, o compilador entende que estamos nos referindo a um método e não a uma variável de instância do objeto.

# Criação e Inicialização de Objetos

Antes de examinar em detalhes o mecanismo de criação e inicialização de objetos em Java, examinaremos duas questões associadas com essas ações: alocação de espaço de pilha versus alocação de espaço em *heap* e recuperação de memória alocada.

## Pilha versus Heap

A questão de armazenamento em pilha versus armazenamento em *heap* diz respeito a como o espaço para variáveis é alocado e liberado e quais passos explícitos o programador deve executar nestes processos. Neste contexto, distingüimos variáveis que são automáticas daquelas que são dinâmicas. A diferença entre tais variáveis é que o espaço para uma variável automática é alocado quando o fluxo de execução do programa entra no procedimento (bloco) contendo a declaração da variável, e é liberado quando o fluxo de execução do programa sai do procedimento (bloco). Tanto a alocação quanto a liberação são, neste caso, realizadas automaticamente.

Agora, consideremos o caso das variáveis dinâmicas. Em muitas linguagens estruturadas de programação, tais como Pascal, uma variável dinâmica é criada por um procedimento `new(x)` fornecido pelo sistema, que recebe como seu argumento uma variável declarada como sendo um apontador. Segue abaixo um exemplo:

```
type
    figura: record
        forma : (triangulo, quadrado);
        lado:   integer;
    end;

var
    umaForma : ^figura;

begin
    new(umaForma);
    ...
end.
```

Neste caso, o espaço recém criado é alocado e, como conseqüência, o valor do argumento variável é modificado para apontar para este novo espaço. Logo, os processos de alocação e de dar um nome à variável estão associados.

Em outras linguagens, tais como C, os processos de alocação e atribuição de nome não estão associados. Em C, por exemplo, a memória é alocada por uma função do sistema denominada `malloc`, que tem como argumento a quantidade de memória a ser alocada. A chamada a `malloc` retorna um apontador para o bloco de memória, que é então associado a uma variável por um operador de atribuição. O seguinte trecho de código serve como exemplo:

```
struct figura {
   enum {triangulo, quadrado} forma;
   int                        lado;
};

figura *umaForma;
...

umaForma = (figura *) malloc (sizeof(figura));
```

A diferença essencial entre alocação de espaço em pilha e em *heap* é que alocação de espaço para uma variável automática (residente na pilha) é realizada sem qualquer diretiva explícita do usuário, enquanto alocação de espaço para variáveis dinâmicas é realizada apenas quando requisitada. A forma como a variável é liberada também é diferente nas duas técnicas. O programador quase nunca necessita considerar a

liberação de variáveis automáticas, enquanto ele pode necessitar considerar a liberação de variáveis baseadas em *heap*.

## Recuperação de Memória

Quando técnicas de alocação de espaço em *heap* são empregadas, alguns meios devem ser fornecidos para recuperar a memória que não está sendo mais utilizada. Geralmente, as linguagens fazem parte de uma de duas grandes categorias. Pascal, C e C++ requerem que o programador mantenha o controle das variáveis para determinar quando elas não são mais úteis e, explicitamente, liberar o espaço através de uma chamada a uma rotina do sistema. Esta rotina é chamada `free` em C e `dispose` em Pascal.

Outras linguagens, tais como Java e Smalltalk, podem automaticamente detectar quando valores não são mais acessíveis e, portanto, não podem mais contribuir para quaisquer computações futuras. Tais valores são então coletados automaticamente e o espaço ocupado por eles é recuperado e reciclado em futuras alocações de memória. Este processo é conhecido como **garbage collection**. Vários algoritmos bem conhecidos podem ser usados para realizar tal processo.

Há argumentos favoráveis e desfavoráveis para a adoção de qualquer uma das duas técnicas. Estes argumentos tendem a confrontar eficiência com flexibilidade. O processo de *garbage collection* automático pode ser caro além de necessitar de um sistema em tempo de execução para gerenciar a memória. Por outro lado, em linguagens nas quais o programador é solicitado para gerenciar alocação dinâmica de memória, os seguintes erros são muito comuns:

- tentativa de usar uma área de memória que ainda não foi alocada;
- memória é alocada dinamicamente, mas nunca é liberada;
- tentativa de usar memória que já foi liberada;
- tentativa de liberar área de memória mais de uma vez.

Para evitar estes problemas é freqüentemente necessário garantir que todo objeto com memória alocada dinamicamente tenha um *proprietário*. O proprietário da memória é responsável por garantir que a posição de memória seja utilizada apropriadamente e seja liberada quando ela não é mais necessária. Em programas grandes, como na vida real, disputas pela propriedade de recursos compartilhados podem ser uma fonte de dificuldades.

# Criação e Inicialização em Java

Uma diferença crucial entre Java e C++ é que Java utiliza o processo automático de *garbage collection*, portanto o programador Java não precisa lidar com gerenciamento de memória. Todos os valores são automaticamente recuperados quando eles não são mais acessíveis no ambiente do programa em execução.

Todas as variáveis de tipo de objeto em Java são inicialmente associadas ao valor `null`. Valores de objetos são criados com o operador `new`, como abaixo:

```
Carta umaCarta = new Carta(Carta.espada, 5);
```

Neste caso, `umaCarta` é uma variável que representa o nome do objeto. O objeto, na verdade, é criado com o operador `new` e, através do operador de atribuição (`=`), o nome é associado ao objeto. Observe que o operador `new` foi usado juntamente com o construtor `Carta`, que recebeu os argumentos `Carta.espada` e `5` e inicializou as variáveis de instância do objeto com estes valores. Portanto, em Java, a criação de um objeto é realizada sempre de forma dinâmica.

Um objeto cujo estado uma vez inicializado nunca poderá ser modificado também pode ser criado em Java. Para tal, basta usarmos a palavra-chave `final`:

```
final Carta umaCarta = new Carta(Carta.espada, 5);
```

O conceito de destrutor em Java também é ligeiramente diferente daquele em C++. O conceito de destrutor em Java é representado por uma função chamada `finalize`, que não possui argumentos e nem retorna valor algum. `finalize` é automaticamente chamada pelo sistema depois que a memória foi recuperada pelo processo de *garbage collection* e antes que tal memória seja reciclada para um novo uso. O programador não possui nenhuma garantia de quando um método `finalize` será chamado, se é que ele será.