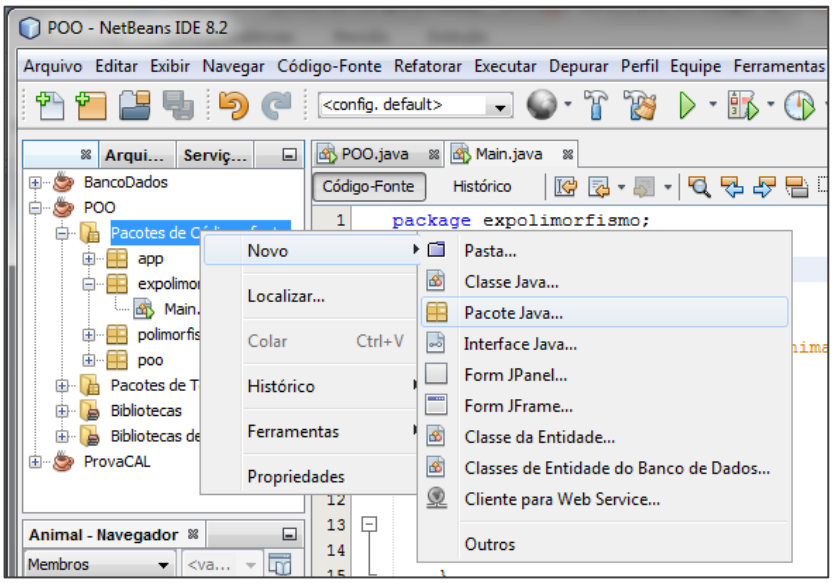
UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 05 – Pacotes, Herança, Polimorfismo

1. PACOTES

Muitas linguagens de programação que permitem o uso do paradigma orientado a objetos usam um conceito de subdivisão ou agrupamento de códigos chamado de pacotes. Os pacotes são como pastas/diretórios e tem como principal intuito a organização do código fonte do programa e o encapsulamento. As IDEs de Java costumam materializar os pacotes efetivamente como diretórios e geralmente permitem a criação de novos pacotes de maneira bem simples e óbvia. É possível inclusive gerar uma hierarquia de pacotes, onde pacotes podem ser alocados dentro de outros pacotes.



Cada arquivo fonte Java deve indicar na sua primeira linha de código em qual pacote se encontra. Esta indicação deve ser feita com a palavra-chave *package* seguida do nome do pacote e ponto-e-vírgula. Na figura acima, pode-se ver que o arquivo Main.java está aberto e a sua primeira linha de código indica que este arquivo se encontra no pacote expolimorfismo. O esquecimento da indicação do pacote pode gerar um erro no início da execução, acusando que a classe (do código fonte sem a especificação do pacote) não foi encontrada.

Já discutimos alguns aspectos sobre o importante componente da orientação a objetos que é o conceito de classes. É importante relembrar que classes também possuem um escopo de encapsulamento, podendo ser públicas – acessíveis por classes definidas em outros arquivos – ou escopo local – acessíveis apenas pelas classes definidas no mesmo arquivo. Lembre-se que classes públicas precisam residir em um arquivo com o mesmo nome da classe, logo, apenas uma classe pública pode ser definida por arquivo. Dessa forma, é comum a definição de várias classes públicas e arquivos em um projeto Java.

> **O escopo *package* ou *default*:** Agora que já falamos sobre o conceito de pacotes é possível apresentar mais um tipo de encapsulamento utilizado no Java: o encapsulamento *package* ou *default*. Ele recebe este nome porque se nenhum encapsulamento for explicitado, por padrão o atributo ou método pertencerá ao pacote, sendo acessível a todas as classes que estejam no mesmo pacote.

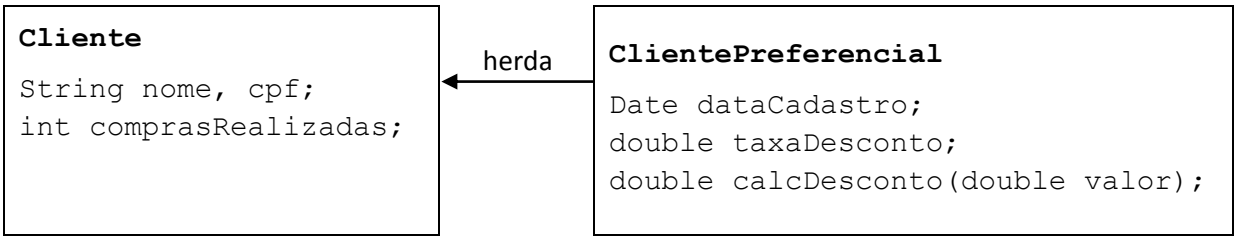
2. HERANÇA

A herança é um conceito da orientação a objetos que permite o relacionamento de classes entre si de maneira hierárquica. Por meio desta propriedade, uma classe pode ser descendente de outra classe, herdando todos os atributos e métodos de sua(s) classe(s) ancestral(is), e ainda pode definir seus próprios atributos e métodos, especializando uma entidade. Uma classe descendente costuma ser chamada de subclasse (especialistas), já a classe pai de uma determinada subclasse é chamada de superclasse (generalista).

O conceito de herança costuma ser empregado em situações em que existem classes semelhantes – com atributos e métodos iguais ou parecidos – e deseja-se reaproveitar estas semelhanças, especializando a subclasse no que a difere. Dessa maneira é possível reaproveitar o que já foi definido pela superclasse e adicionar apenas os atributos e métodos novos, que são característicos da subclasse.

Perceba o intuito de reaproveitamento de código nesta propriedade da O.O.

Para exemplificar, considere uma situação em que se deseja representar os clientes regulares de um estabelecimento e também os clientes preferenciais que recebem um desconto de 5% nas suas compras. Seria possível utilizar uma variável como analogia para indicar se um determinado cliente é regular ou preferencial e implementar o método de compra com uma condicional de desconto para os clientes preferenciais. Entretanto, quanto mais condicionais específicas a cada tipo de cliente forem introduzidas, mas complicado será de alterá-las, ou até mesmo encontrá-las, posteriormente. Assim, nessa situação, uma abordagem com uso de herança seria mais vantajosa:



Note que na tentativa de criar duas classes distintas, sem herança, adicionando todos os atributos e métodos da classe Cliente na classe ClientePreferencial, seriam adicionadas muitas repetições de código, impactando no tempo de implementação e de possíveis manutenções.

Além de herança simples, a orientação a objetos ainda prevê a herança composta e a herança múltipla. Uma herança composta é quando há múltiplos níveis de herança. Considerando o exemplo acima, se houvesse uma classe Vendedor que também possui os atributos nome e CPF, poderia ser criada uma nova classe Pessoa, por exemplo, que é superclasse de Cliente e Vendedor, e possui os atributos e métodos em comum entre eles. Dessa forma, a classe ClientePreferencial herdaria atributos e métodos da classe Cliente, e esta, por sua vez, herdaria atributos e métodos da classe Pessoa, criando a herança indireta entre ClientePreferencial e Pessoa. Já a herança múltipla é quando uma classe herda atributos e métodos de diversas superclasses. Um exemplo clássico seria a representação de CarroAnfíbio, que herda as características de uma classe Carro e de uma classe Barco, ou ainda a representação de um animal onívoro que é tanto carnívoro como herbívoro.

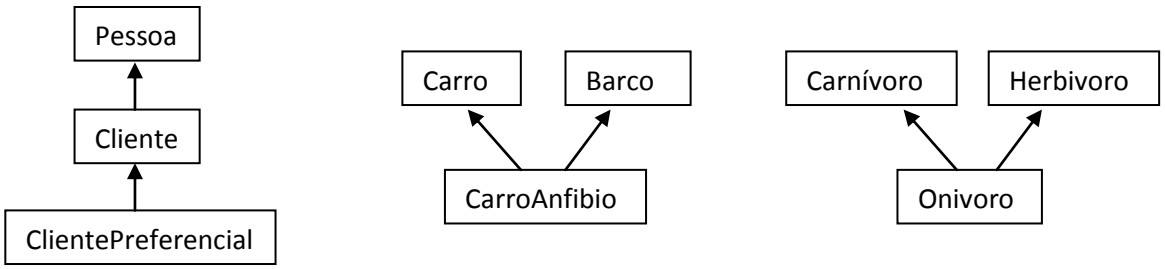


FIGURA 1. Representação de diferentes tipos de herança (setas representam herança). A figura da esquerda representa herança composta (indireta) e as figuras do centro e da direita representam herança múltipla.

A linguagem de programação Java implementa o conceito de herança simples e composta, mas não aceita o uso de herança múltipla. A palavra chave utilizada para representar uma situação de herança é *extends*, que é utilizada após o nome da classe.

Java permite que uma determinada classe seja superclasse de qualquer número de subclasses, mas uma subclasse pode herdar apenas de uma superclasse de maneira direta (i.e. não é permitido o uso de herança múltipla).

Em Java, uma subclasse herda todos os métodos e atributos da superclasse, exceto pelos métodos construtores, que não são herdados mas, podem ser utilizados pelas subclasses nos seus métodos construtores. Para invocar um método construtor de uma superclasse, deve-se utiliza a palavra-chave *super*, a qual invoca o método construtor apropriado da superclasse considerando a quantidade e os tipos de parâmetros passados como argumento. Se a chamada ao construtor da superclasse não for efetuada explicitamente no programa, o Java irá executar o construtor *default* da superclasse (se ele existir).

OBS: a chamada a um dos construtores da superclasse deve ser o primeiro comando a ser executado por um construtor de uma subclasse – considerando o exemplo do ClientePreferencial, podemos pensar que uma pessoa tem que ser um cliente (superclasse) antes de ser um ClientePreferencial (subclasse)

Uma postura no mínimo interessante – no que tange aos conceitos de orientação a objetos – que o Java adota em relação a este conceito de herança é que toda classe que não possui um relacionamento explícito de herança é implicitamente subclasse de uma classe suprema (superclasse de maior nível) chamada de Object, ou java.lang.Object. Esta abordagem é interessante, porque desta forma, tudo efetivamente é considerado um objeto, conforme a proposta do paradigma orientado a objetos. Todas as classes ou herdam implicitamente da superclasse Object, ou herdam de forma indireta através da sua própria superclasse.

2.1. Herança e peculiaridades de escopo

Esse processo de herança de atributos e métodos passa a ter relação estreita com as propriedades de encapsulamento. Até o momento já vimos três formas diferentes de encapsulamento existentes que podemos usar: o público, acessível a todas as outras classes, o padrão ou do pacote, acessível a todas as classes do pacote, e o encapsulamento privado, acessível apenas internamente à classe. Com relação a estas formas de encapsulamento, quando uma superclasse define atributos e métodos públicos ou do pacote, estes ficam acessíveis a todas as classes ou a todas as classes do pacote, respectivamente, incluindo as subclasses. Por outro lado,

aqueles atributos e métodos que forem declarados como privados nas superclasses ficam inacessíveis para todas as classes, incluindo as subclasses.

Em determinadas situações é interessante que um atributo ou método não seja aberto a todas as classes, mas seja acessível a suas subclasses que residam em outros pacotes. Para estas situações existe o especificador de escopo protegido, ou *protected*, que faz com que atributos e métodos sejam acessíveis apenas a todas as classes do pacote, e também as suas subclasses. A Tabela 1 resume de forma enxuta e precisa o nível de acesso de cada forma de encapsulamento.

TABELA 1 – Encapsulamentos e sua acessibilidade

	Classe	Pacote	Subclasse	Mundo
public	✔	✔	✔	✔
protected	✔	✔	✔	✘
default | package	✔	✔	✘	✘
private	✔	✘	✘	✘

2.2. Redefinição (*Overriding*)

Um dos mecanismos fundamentais na programação orientada a objetos é o conceito de redefinição ou sobrescrita de métodos (*overriding*) em subclasses. A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição (ou seja, um novo corpo) em uma classe derivada.

Tanto atributos quanto métodos podem ser sobrescritos, mas a redefinição de métodos provavelmente é de longe a mais utilizada para fins práticos durante a implementação de sistemas computacionais. A redefinição de atributos ocorre quando um atributo é declarado em uma subclasse com o mesmo nome de um atributo declarado na superclasse. Já a redefinição de métodos ocorre quando um método é implementado em uma subclasse com a mesma assinatura (i.e. mesmo nome e, mesma quantidade e ordem de tipos de parâmetros) de um método da superclasse. É considerado boa prática, adicionar a expressão: *@Override* uma linha acima de um método ou variável que realiza a redefinição.

2.3. Exemplo de Herança e Redefinição de Método

Arquivo: Pessoa.java

```
 1.  package heranca;
 2.  class Pessoa{
 3.      public String nome;
 4.      private String cpf = "-";
 5.      //-----------------------------------------
 6.      public String getCPF(){ return cpf; }
 7.      //-----------------------------------------
 8.      public void setCPF(String cpf){
 9.          cpf = cpf.replace(".", "");
10.          cpf = cpf.replace("-", "");
11.          if( cpf.length() == 11 )
12.              this.cpf = cpf;
13.      }
```

```
14.      /** ----------------------------------------
15.       * Cria uma instância de Pessoa. Se um cpf
16.       * inválido for passado este será redefinido
17.       * como "-".
18.       * @param cpf O cpf do novo indivíduo
19.       * @param nome O nome do novo indivíduo
20.       */
21.      public Pessoa(String cpf, String nome){
22.          setCPF( cpf );
23.          this.nome = nome;
24.      }
25.  }
```

Arquivo: Vendedor.java

```
 1.  package heranca;
 2.  class Vendedor extends Pessoa{
 3.      private int vendas;
 4.      //----------------------------------------
 5.      public Vendedor(String cpf, String nome){
 6.          super( cpf, nome );
 7.          vendas = 0;
 8.      }
 9.      //----------------------------------------
10.      public void vendaRealizada(){ vendas++; }
11.  }
```

Arquivo: Cliente.java

```
 1.  package heranca;
 2.  class Cliente extends Pessoa{
 3.      private int comprasRealizadas;
 4.      protected int pontuacao;
 5.      //----------------------------------------
 6.      public Cliente(String cpf){
 7.          super( cpf, "Cliente Regular");
 8.          comprasRealizadas = 0;
 9.      }
10.      //----------------------------------------
11.      public Cliente(String cpf, String nome){
12.          super( cpf, nome );
13.          comprasRealizadas = 0;
14.      }
15.      //----------------------------------------
16.      public void realizarCompra(){ comprasRealizadas += 1; }
17.      /** ----------------------------------------
18.       * Atualiza a quantidade de pontos de um cliente baseado
19.       * no valor da compra realizada.
20.       * @param valor O valor da compra realizada em centavos.
21.       */
22.      public void acumularPontos(int valor){
23.          pontuacao += valor / 100;
24.      }
25.      //----------------------------------------
26.      public void usarPontos(int pontos){
27.          pontuacao -= pontos;
28.      }
29.  }
```

Arquivo: ClientePreferencial.java

```
 1.  package heranca;
 2.  import java.util.Date;
 3.
 4.  class ClientePreferencial extends Cliente{
 5.      private float desconto;
 6.      private Date dataCadastro;
 7.
 8.      //-----------------------------------------
 9.      public ClientePreferencial(String cpf, String nome){
10.          super(cpf, nome);
11.          desconto = 0.05f; // desconto padrão de 5%
12.          dataCadastro = new Date(); // gera a data atual
13.      }
14.      //-----------------------------------------
15.      // Exemplo de sobrescrita de método
16.      /**
17.       * Um cliente preferencial tem um acúmulo de pontos
18.       * bônus de 25% em relação a quantidade
19.       * @param valor O valor da compra realizada em centavos
20.       */
21.      @Override
22.      public void acumularPontos(int valor){
23.          pontuacao += (int) ((valor / 100) * 0.25f);
24.      }
25.      //-----------------------------------------
26.      public Date getDataCadastro(){ return dataCadastro; }
27.      //-----------------------------------------
28.      public int getPrecoPreferencial(int valor){
29.          return (int) (valor * (1-desconto));
30.      }
     }
```

3. POLIMORFISMO

Polimorfismo é o princípio pelo qual uma subclasse pode se comportar hora como uma entidade da subclasse, hora como uma entidade da superclasse. A decisão sobre como o objeto será tratado, ou qual método deverá ser executado é tomado em tempo de execução, através do mecanismo de ligação tardia (*late binding*), também conhecido pelo termo ligação dinâmica (*dynamic binding*) ou ligação em tempo de execução (*run-time binding*). Este mecanismo de ligação tardia é fundamental para a viabilidade de uso de polimorfismo, que pode ocorrer no contexto de escolha de um método de mesmo nome, mas com assinatura diferente (sobrecarga), ou na generalização de um objeto especializado (*upcasting*) e especialização de um objeto que foi generalizado (*downcasting*). Provavelmente estes termos ainda estão um pouco confusos para você neste primeiro momento – isso é normal. Vamos nos aprofundar um pouco mais nestes usos corriqueiros de polimorfismo.

3.1. Sobrecarga de Métodos

Em algumas linguagens os métodos são diferenciados apenas pelo seu nome. Dessa maneira, não pode haver dois métodos com um mesmo nome, mesmo que eles tenham tipo de retorno e lista de parâmetros totalmente diferentes. Com o mecanismo de ligação tardia, contudo, é possível que dois métodos sejam diferenciados por mais do que apenas o seu nome. Tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método. De fato, a linguagem Java efetivamente compara métodos pela sua assinatura – nome do método e, quantidade e ordem dos tipos de parâmetros – desconsiderando apenas o tipo de retorno. Para exemplificar considere as seguintes assinaturas de métodos abaixo:

```
a) void usarPontos(int pontos);
b) void usarPontos(char tipo, int pontos);
c) void usarPontos(int pontos, char tipo);
d) int usarPontos(int pontos);
```

Quais métodos você acha que são considerados como iguais pelo Java e, portanto, não poderiam coexistir em uma mesma classe? Você acertou se respondeu que (a) e (d) são consideradas iguais, pois elas diferem apenas no tipo de retorno, que não é considerado pela linguagem para diferenciar a assinatura de métodos. Essa característica de poder implementar diversos métodos com o mesmo nome, mas com assinatura diferente é denominada de sobrecarga (*overloading*), um conceito muito mencionado na orientação a objetos. Outro exemplo clássico e que inclusive já foi mencionado em aulas anteriores, é a possibilidade de escrever vários métodos construtores para uma classe, desde que sua assinatura seja diferente.

> **Curiosidade:** Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia, exceto em dois casos: (1) métodos declarados como *final* (métodos constantes) não podem ser redefinidos e, portanto, não são passíveis de invocação polimórfica da parte de seus descendentes, e (2) métodos declarados como privados, que são implicitamente constantes (*final*).

Os conceitos de redefinição e sobrecarga compartilham algumas semelhanças, mas não podem ser confundidos. As principais diferenças são:

- Redefinição ocorre entre duas classes que possuem uma relação de herança;
- Sobrecarga ocorre em uma única classe;
- Na redefinição os parâmetros devem ser os mesmos;
- Na sobrecarga, a lista de parâmetros (quantidade e ordem) deve ser diferente;

2.2. *Upcasting* e *Downcasting*

Upcasting e *Downcasting* são dois conceitos provenientes da relação entre herança e polimorfismo. O primeiro, *upcasting*, diz respeito a um *casting* de uma instância de uma subclasse a uma instância da superclasse (está elevando – *up* – o grau de generalização), enquanto que o segundo, *downcasting*, diz respeito a um *casting* de uma instância de uma superclasse a uma instância de subclasse (está reduzindo – *down* – o grau de generalização). O exemplo apresentado na Figura 1 ilustra bem, visualmente, a ideia de *upcasting* e *downcasting*.

Considere um exemplo similar ao da Figura 1, implementado no código mostrado a seguir, no qual há uma superclasse animal e três subclasses: cachorro, gato e cavalo, contendo - cada uma -

um atributo próprio. Note que cada uma dessas classes possui um método *saySomething()* que "reproduz" um dizer do animal, fazendo uso da propriedade de redefinição de métodos.

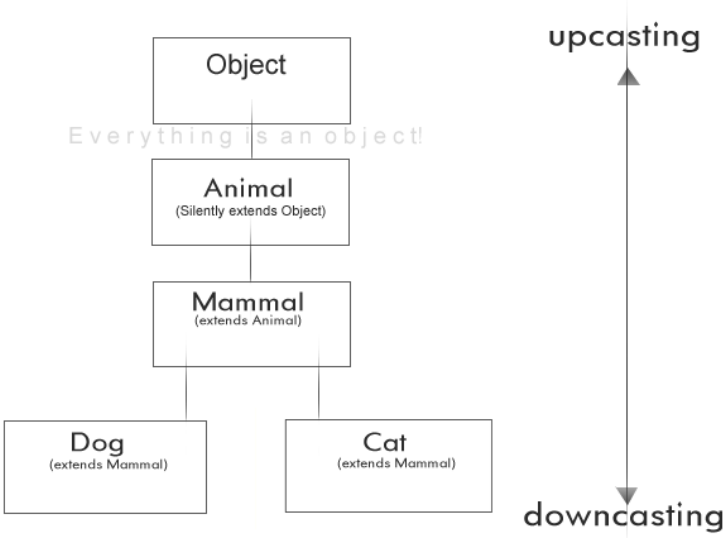


FIGURA 1 – Exemplo visual de *upcasting* e *downcasting (Sinipull – codecall.net)*

Arquivo: Animal.java

```
 1.  package expolimorfismo;
 2.  class Animal {
 3.      public void saySomething(){
 4.          System.out.println("[Some Animal sound]");
 5.      }
 6.  }
 7.
 8.  class Dog extends Animal {
 9.      int idTag;
10.      public Dog(int id){ idTag = id; }
11.      @Override
12.      public void saySomething() {
13.          System.out.println("Dog #"+idTag+": au au, woof woof\"");
14.      }
15.  }
16.
17.  class Cat extends Animal {
18.      int idTag;
19.      public Cat(int id){ idTag = id; }
20.      @Override
21.      public void saySomething() {
22.          System.out.println("Cat #"+idTag+": mii ... miiau\"");
23.      }
24.  }
25.
26.  class Horse extends Animal {
27.      String name;
28.      public Horse(String name){ this.name = name; }
29.      @Override
30.      public void saySomething() {
31.          System.out.println(name+" says: \"Neigh Neigh\"");
32.      }
33.  }
```

OBS: alguns métodos não estão devidamente identados a fim de economizar espaço.
Não faça isso em casa! [=D]

Agora considere que em uma aplicação queremos implementar um método *speach()* que mostra uma frase padronizada e faz uso do método *saySomething()*. Em uma abordagem sem uso de *upcasting* teríamos que usar sobrecarga de método para cada um dos animais existentes. No exemplo ilustrado são "apenas" três classes, o que já ficaria bastante redundante, mas poderiam ser mais classes. Observe como ficaria uma implementação que define e faz uso dos métodos *speach()*.

Arquivo: SemUpcasting.java

```
 1.  package expolimorfismo;
 2.
 3.  public class SemUpcasting {
 4.      static void speach(Dog animal){
 5.          System.out.println("Silence now fo the speach:");
 6.          animal.saySomething();
 7.      }
 8.      static void speach(Cat animal){
 9.          System.out.println("Silence now fo the speach:");
10.          animal.saySomething();
11.      }
12.      static void speach(Horse animal){
13.          System.out.println("Silence now fo the speach:");
14.          animal.saySomething();
15.      }
16.      public static void main( String[] args ){
17.          Dog dog = new Dog(1);
18.          Cat cat = new Cat(2);
19.          Horse horse = new Horse("Rocket");
20.
21.          SemUpcasting.speach( dog );
22.          SemUpcasting.speach( cat );
23.          SemUpcasting.speach( horse );
24.      }
25.  }
```

Uma melhor abordagem para esta intenção de funcionalidade seria fazer uso de polimorfismo com *upcasting* escrevendo apenas um único método para todos os animais, uma vez que o método *speach()* faz a mesma coisa para cada animal. Observe no exemplo ilustrado no quadro a seguir que mesmo o parâmetro da função sendo um objeto Animal, ao invocar o método *saySomething()*, o programa sabe que o método a chamar deve ser o especializado, redefinido por cada subclasse. O *upcasting* ocorre de forma implícita e geralmente é utilizado através de chamadas de métodos, na qual o método generaliza o seu parâmetro servindo a mais de um propósito de uma só vez.

Os exemplos tratados até o momento trataram de *upcasting*, mas e o *downcasting*? É comum que os dois conceitos estejam interligados, pois usualmente um objeto especialista (instancia de subclasse) recebe *upcasting* para a superclasse, mas em algum momento poderá ser associado à sua classe especialista novamente por meio de um *downcasting*. Considerando o exemplo envolvendo os animais, imagine que esta abstração está sendo usada para o gerenciamento de animais e de sua eventual adoção. O exemplo apresentado no arquivo Downcasting.java mostra uma situação em que quatro novos animais são recebidos (um cachorro, um gato e dois cavalos) e depois são realizadas três adoções de animais. Para cada adoção é utilizado o método *saySomething()* e em seguida, tenta-se transformar o animal em uma instância de cachorro (*downcasting*) e utiliza-se o método *getBall()*, exclusivo da classe Dog.

Arquivo: ComUpcasting.java

```
 1.  package expolimorfismo;
 2.  public class ComUpcasting {
 3.      static void speach(Animal animal){
 4.          System.out.println("Silence now for the speach:");
 5.          animal.saySomething();
 6.      }
 7.
 8.      public static void main( String[] args ){
 9.          Dog dog = new Dog(1);
10.          Cat cat = new Cat(2);
11.          Horse horse = new Horse("Rocket");
12.
13.          ComUpcasting.speach( dog );
14.          ComUpcasting.speach( cat );
15.          ComUpcasting.speach( horse );
16.      }
17.  }
```

Duas observações merecem atenção. Primeiro, perceba que o processo de *downcasting* precisa ser explícito – diferente do *upcasting* que é implícito. Segundo, durante a execução do programa Downcasting.java um erro ocorrerá, pois o primeiro animal é efetivamente um cachorro, mas o segundo e o terceiro não são. Quando um *downcasting* é realizado de forma imprópria (tentar transformar um animal cavalo em um animal cachorro, por exemplo) uma exceção é lançada.

Para contornar este problema devemos, sempre que preciso, verificar se um determinado objeto é uma instancia de uma determinada classe. Isso pode ser através de uma condição utilizando o operador representado pela palavra-chave *instanceof*. Corrigindo o exemplo de Downcasting apresentado, é apresentado um novo quadro que atualiza o método *main()* do programa, fazendo uso do *instanceof* para verificar se o animal adotado é um cachorro. Desta maneira, o método getBall() só será invocado quando efetivamente o animal adotado for um cachorro.

Arquivo: Downcasting.java

```
 1.  package expolimorfismo;
 2.  public class Downcasting {
 3.      static final int MAX_ANIMALS = 100;
 4.      static Animal[] animals = new Animal[MAX_ANIMALS];
 5.      static int animalsLastIndex = 0; // gerencia o índice
 6.
 7.      /**-------------------------------------------------------
 8.       * Receives a new animal that can be adopted
 9.       * @param animal The new animal
10.       * @return True if there is space left, else return false
11.       */
12.      static boolean receivesAnimal( Animal animal ){
13.          if( animalsLastIndex+1 >= MAX_ANIMALS )
14.              return false;
15.          animalsLastIndex += 1;
16.          animals[ animalsLastIndex ] = animal;
17.          return true;
18.      }
19.
```

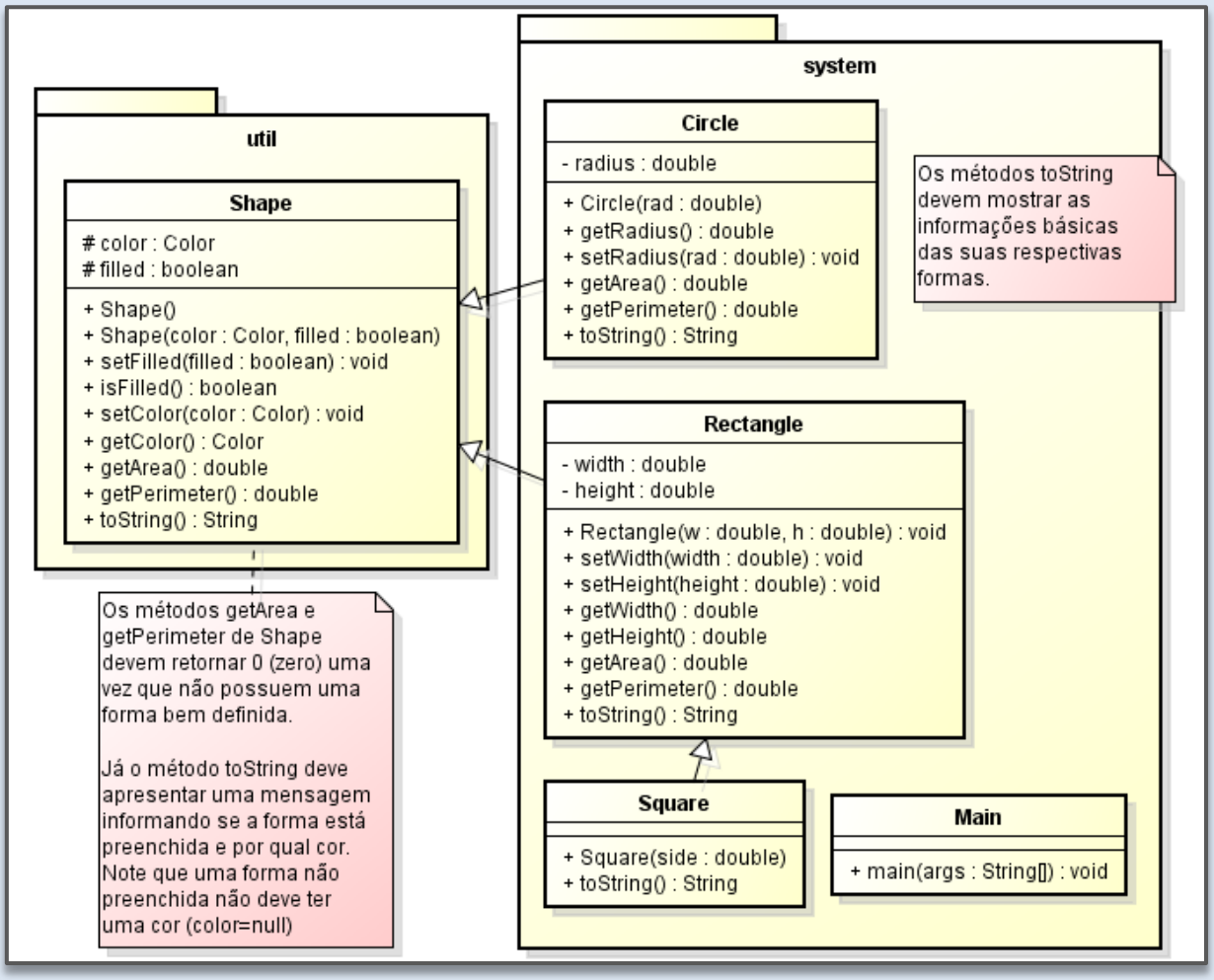
```
20.      /**-------------------------------------------------------
21.       * Adopts the most recent (last) animal
22.       * @return the adopted animal
23.       */
24.      static Animal adoptAnimal(){
25.          if( animalsLastIndex == -1)
26.              return null;   // no animal left ='(
27.          animalsLastIndex -= 1;
28.          return animals[ animalsLastIndex+1 ];
29.      }
30.
31.      //-------------------------------------------------------
32.      public static void main(String[] args){
33.          receivesAnimal( new Horse("Alazan") );
34.          receivesAnimal( new Cat(12) );
35.          receivesAnimal( new Horse("Chocobo") );
36.          receivesAnimal( new Dog(121) );
37.
38.          Animal adopted;
39.          for(int i=0; i<3; i++){
40.              adopted = adoptAnimal();
41.              if( adopted != null ){
42.                  System.out.println("\nNew Animal Adopted: ");
43.                  adopted.saySomething();
44.                  Dog myDog = (Dog) adopted;
45.                  myDog.getBall();
46.                  // ((Dog) adopted).getBall();
47.              }
48.          }
49.      }
50.  }
```

Arquivo: Downcasting.java (atualizado)

```
 2.  public class Downcasting {
         //...
32.      public static void main( String[] args ){
             //...
38.          Animal adopted;
39.          for(int i=0; i<3; i++){
40.              adopted = adoptAnimal();
41.              if( adopted != null ){
42.                  System.out.println("\nNew Animal Adopted: ");
43.                  adopted.saySomething();
44.                  if( adopted instanceof Dog ){
45.                      Dog myDog = (Dog) adopted;
46.                      myDog.getBall();
47.                      // ((Dog) adopted).getBall();
48.                  }
49.              }
50.          }
51.      }
52.  }
```

Exercícios

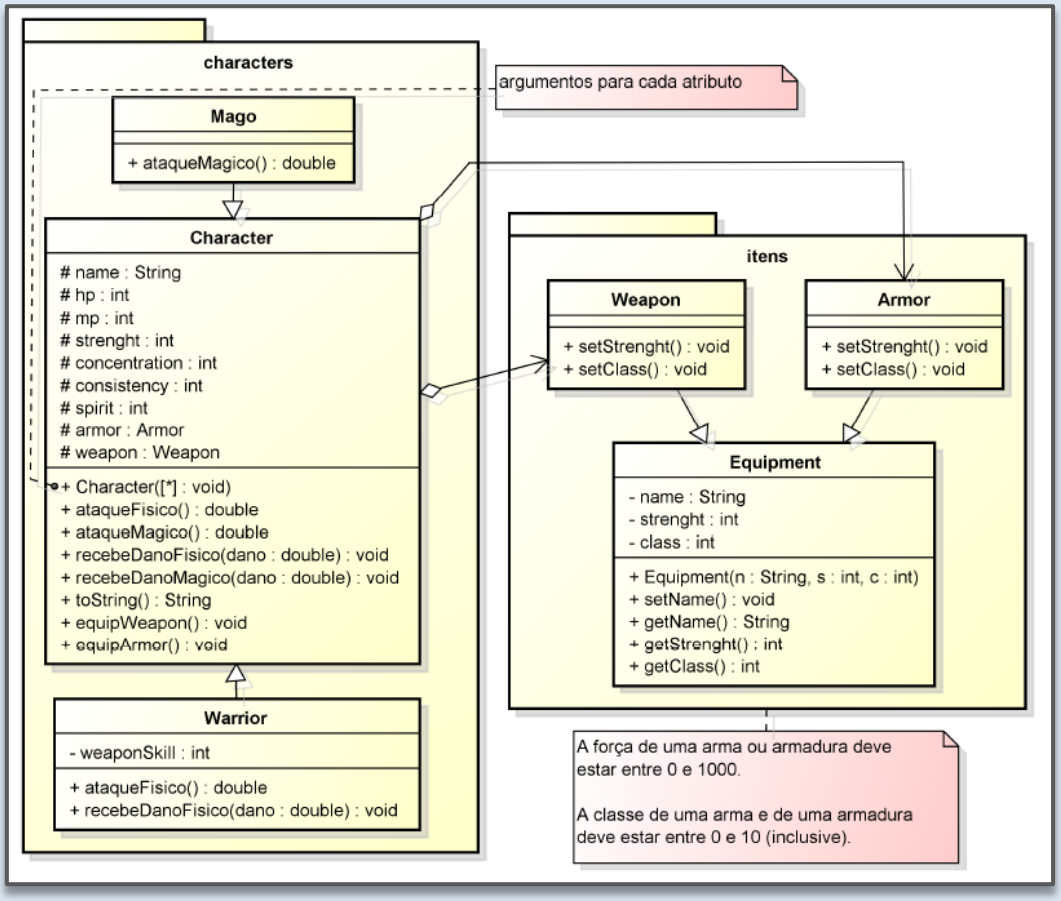
1) Construindo uma aplicação de formas geométricas com herança.
A aplicação a ser construída deverá seguir o modelo apresentado no seguinte diagrama de classes. O símbolo + indica encapsulamento público, - indica privado, # indica protegido e ~ indica encapsulamento do pacote. A seta entre uma classe e outra indica um relacionamento de herança onde a classe apontada é a superclasse.



No método main da classe Main, crie quatro instâncias de formas: (1) um círculo de raio 7 com preenchimento azul, (2) um quadrado de lado 5 sem preenchimento, (3) um retângulo de lados 3x6 com preenchimento preto, e (4) uma forma (Shape) com preenchimento verde. Em seguida remova o preenchimento do retângulo.

Ainda, implemente um método adicional polimórfico denominado *showInfo(Shape)*, que recebe uma forma (Shape) e mostra alguma informação prévia na tela seguida das informações da forma passada por parâmetro usando seu método toString().

2) Construindo um sistema básico de batalhas entre personagens de RPG.



Neste jogo de RPG um jogador pode ser um personagem comum, um guerreiro ou um mago. Os ataques de um personagem normal devem respeitar as fórmulas:

$$ataque\_fisico = força\_arma \times (1.0 + classe\_arma \times 0.05 + força \times 0.0125)$$
$$ataque\_magico = força\_arma \times (1.0 + classe\_arma \times 0.05 + concentração \times 0.0125)$$

Lembrando que um ataque mágico gasta 5 pontos de MP e só pode ser utilizado caso o personagem tenha essa quantidade de pontos. As fórmulas de dano são as seguintes:

$$dano\_fisico = dano \times (1.0 - classe\_armadura \times 0.05 - consistência \times 0.005)$$
$$dano\_magico = dano \times (1.0 - classe\_armadura \times 0.03 - espírito \times 0.015)$$

Entretanto, magos possuem uma fórmula própria para seu ataque mágico e guerreiros possuem uma fórmula própria para ataque físico e redução de dano:

$$ataque\_magico\_mago = força\_arma \times (1.0 + classe\_arma \times 0.07 + concentração \times 0.0175)$$
$$ataque\_fisico\_guerreiro = força\_arma \times (1.0 + classe\_arma \times 0.06 + força \times 0.02)$$
$$dano\_fisico\_guerreiro = dano \times (1.0 - classe\_armadura \times 0.05 - consistência \times 0.008)$$

Por fim instancia alguns personagens, guerreiros e magos, faça uso do método toString para exibir informações dos personagens e crie um método polimórfico:

```
Atack(Character attacker, Character defensor, TipoAtaque tipoAtaque);
```

Este método deverá calcular o valor de dano causado pelo ataque de um personagem e o dano recebido pelo seu oponente (defensor). Note que o tipo de ataque – físico ou mágico – deve ser identificado através de um enumerador.