Orientação a Objeto - Reuso com Herança

Objetivos da seção

- Aprender a reusar código comum entre classes através do mecanismo de herança
- Apresentar classes e métodos abstratos
- Investigar as consequências da herança: polimorfismo, poder de substituição de um objeto por outro, upcasting e downcasting
- Apresentar como representar herança com a linguagem gráfica UML

Reuso com herança

- Nas classes de mensagens de correio eletrônico vistas (MensagemTexto, MensagemMissaoImpossivel, MensagemAudio), temos um mau cheiro terrível no código
  - Há muita repetição de código
  - Isso dificulta a manutenção de software pois uma mudança pode implicar alterações em várias partes do código, o que é "prato cheio" para introduzir bugs
- A atividade de limpar código que apresenta mau cheiro chama-se refatoramento
- Vamos refatorar as três classes de mensagens eletrônicas

Fatorando o que há de comum

- Primeiro juntamos tudo que tem de comum entre as três classes e criamos uma nova classe que chamaremos MensagemAbstrata
  - O porquê da palavra "Abstrata" ficará claro logo adiante
- O resultado segue abaixo:

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal da Paraíba
 *
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 * Não redistribuir sem permissão.
 */
package p1.aplic.correio;

import p1.aplic.geral.*;
import java.io.*;

/**
 * Classe abstrata que representa uma mensagem de correio eletronico.
 * Uma mensagem contém um remetente, um assunto uma data de envio e algum conte
 * O conteúdo depende do tipo exato de mensagem (textual, áudio).
 * Uma mensagem pode ser exibida (lida) e marcada para exclusão.
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 */

public class MensagemAbstrata implements Mensagem {
  protected static final int LIDA = 0x1;
  protected static final int EXCLUÍDA = 0x2;
  protected static final int NOVA = ~(LIDA | EXCLUÍDA);

  protected String  remetente;
```

```
    protected String  assunto;
    protected Data    dataEnvio;
    protected int     estado;

    /**
     * Recupera o remetente da mensagem
     * @return O remetente da mensagem
     */
    public String getRemetente() {
      return remetente;
    }

    /**
     * Recupera o assunto da mensagem
     * @return O assunto da mensagem
     */
    public String getAssunto() {
      return assunto;
    }

    /**
     * Recupera a data de envio da mensagem
     * @return A data de envio da mensagem
     */
    public Data getDataEnvio() {
      return dataEnvio;
    }

    /**
     * Informa se a mensagem foi lida ou não
     * @return true se a mensagem foi lida, false caso contrário
     */
    public boolean isLida() {
      return (estado & LIDA) == LIDA;
    }

    /**
     * Informa se a mensagem foi excluída ou não
     * @return true se a mensagem foi excluída, false caso contrário
     */
    public boolean isExcluída() {
      return (estado & EXCLUÍDA) == EXCLUÍDA;
    }

    /**
     * Marcar a mensagem como excluída.
     * A exclusão deve ser feita pela coleção que armazena as mensagens.
     * Um exemplo de tal coleção é CaixaPostal.
     */
    public void excluir() {
      estado |= EXCLUÍDA;
    }

    /**
     * Marcar a mensagem como não excluída.
     */
    public void marcarNãoExcluída() {
      estado &= ~EXCLUÍDA;
    }
```

```
  /**
   * Marcar a mensagem como não lida.
   */
  public void marcarNãoLida() {
    estado &= ~LIDA;
  }
}
```

- A palavra "protected" será explicada adiante
- O que colocamos no código acima é apenas o que tem de comum entre as três classes de Mensagens
    - Observe que não incluímos atributos que não sejam comuns às 3 classes (conteúdo, arquivoÁudio) nem os métodos que manipulam esses atributos

Adicionando métodos ao núcleo comum

- Não temos uma classe completa ainda, por dois motivos
    - Primeiro, não temos construtor, porque os construtores das 3 classes não são iguais e não foram portanto incluídos aqui
    - Segundo, a interface Mensagem exige os métodos equals(...), exibir() e toString() e não temos uma versão comum para esses métodos
- Podemos resolver, pelo menos parcialmente, o primeiro problema introduzindo um construtor parcial que faça "o possível" para inicializar os atributos
    - Veremos como completar o trabalho logo em seguida

```
protected MensagemAbstrata(String remetente, String assunto) {
  this.remetente = remetente;
  this.assunto = assunto;
  dataEnvio = new Data();
  estado = NOVA;
}
```

- Podemos fazer algo semelhante ("o possível", mesmo que parcial) com o método equals()

```
/**
 * Testa a igualdade de um objeto com esta mensagem.
 * @param objeto O objeto a comparar com esta mensagem.
 * @return true se o objeto for igual a esta mensagem, false caso contrário.
 */
public boolean equals(Object objeto) {
  if(! (objeto instanceof Mensagem)) {
    return false;
  }
  Mensagem outra = (Mensagem)objeto;
  return getRemetente().equals(outra.getRemetente())
         && getAssunto().equals(outra.getAssunto());
}
```

Adicionando métodos abstratos

- Por outro lado, precisamos ainda fornecer uma implementação dos métodos exibir() e toString() para satisfazer à interface que prometemos implementar
- Como não podemos fornecer uma versão desses métodos mas eles *devem* existir, fornecemos uma versão "abstrata" dos mesmos
    - "Abstrato" significa "conhecemos o alto nível (o que fazer) mas não o baixo nível (como fazer)"
    - Intuitivamente, você concorda que todas as classes de mensagens terão esses dois métodos e com as mesmas assinaturas, embora como fazer a implementação seja diferente em cada caso

- Devido a esses métodos abstratos, a classe em si é abstrata
  - Uma classe abstrata não pode ser instanciada (pois faltam detalhes de implementação)

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal da Paraíba
 *
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 * Não redistribuir sem permissão.
 */
package p1.aplic.correio;

import p1.aplic.geral.*;
import java.io.*;

/**
 * Classe abstrata que representa uma mensagem de correio eletronico.
 * Uma mensagem contém um remetente, um assunto uma data de envio e algum conte
 * O conteúdo depende do tipo exato de mensagem (textual, áudio).
 * Uma mensagem pode ser exibida (lida) e marcada para exclusão.
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 */

public abstract class MensagemAbstrata implements Mensagem {
  protected static final int LIDA = 0x1;
  protected static final int EXCLUÍDA = 0x2;
  protected static final int NOVA = ~(LIDA | EXCLUÍDA);

  protected String  remetente;
  protected String  assunto;
  protected Data    dataEnvio;
  protected int     estado;

  // ...

  /**
   * Exibir a mensagem. Isso poderá imprimir algo na saída
   * ou provocar outras saídas relacionadas com a leitura da mensagem.
   * Após este método, a mensagem é considerada "lida".
   */
  public abstract void exibir();

  /**
   * Forneça uma representação da mensagem como String
   * @return A representação da mensagem como String.
   */
  public abstract String toString();
}
```

Completando o trabalho com herança

- Muito bem. Ainda temos 3 coisas a resolver até termos uma solução:
  - Cadê os atributos que faltam?
  - O que fazer com o construtor que é diferente dos originais?
  - Como completar os métodos ausentes?
- As três coisas são resolvidas pela extensão da classe MensagemAbstrata através do conceito de herança
- Vejamos um primeiro exemplo: a classe MensagemTexto (ver MensagemTexto.java)

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal da Paraíba
 *
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 * Não redistribuir sem permissão.
 */
package p1.aplic.correio;

import p1.aplic.geral.*;
import java.io.*;

/**
 * Classe que representa uma mensagem normal de correio eletronico.
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 */

public class MensagemTexto extends MensagemAbstrata {
  protected String  conteúdo;

  /**
   * Cria uma mensagem textual de correio eletrônico
   * @param remetente O remetente da mensagem
   * @param assunto O assunto da mensagem
   * @param contéudo O conteúdo da mensagem, podendo conter várias linhas
   */
  public MensagemTexto(String remetente, String assunto, String conteúdo) {
    super(remetente, assunto);
    this.conteúdo = conteúdo;
  }

  /**
   * Recupera o conteúdo da mensagem.
   * O conteúdo é um String podendo conter várias linhas.
   * @return O conteúdo da mensagem
   */
  public String getConteúdo() {
    return conteúdo;
  }

  /**
   * Exibir a mensagem. Os dados da mensagem são apresentados na saída padrão.
   * Após este método, a mensagem é considerada "lida".
   */
  public void exibir() {
    System.out.println("De: " + remetente);
```

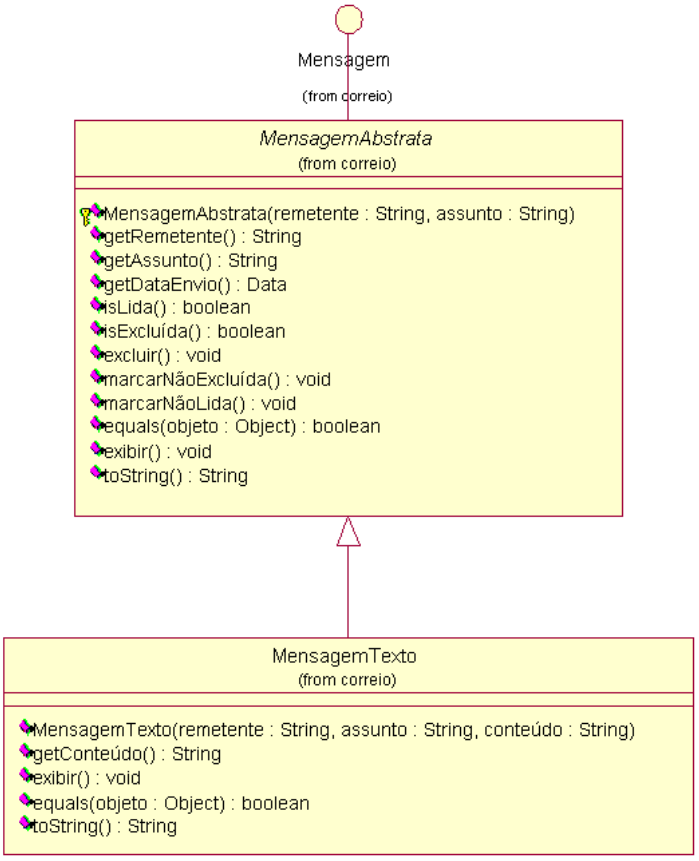
```
      System.out.println("Data: " + dataEnvio.DDMMAAAAHHMM());
      System.out.println("Assunto: " + assunto);
      System.out.println(conteúdo);
      estado |= LIDA;
    }

    /**
     * Testa a igualdade de um objeto com esta mensagem.
     * @param objeto O objeto a comparar com esta mensagem.
     * @return true se o objeto for igual a esta mensagem, false caso contrário.
     */
    public boolean equals(Object objeto) {
      if(! (objeto instanceof MensagemTexto)) {
        return false;
      }
      MensagemTexto outra = (MensagemTexto)objeto;
      return super.equals(objeto) &&
             getConteúdo().equals(outra.getConteúdo());
    }

    /**
     * Forneça uma representação da mensagem como String
     * @return A representação da mensagem como String.
     */
    public String toString() {
      return "Remetente: " + remetente +
             ", Data: " + dataEnvio.DDMMAAAAHHMM() +
             ", Assunto: " + assunto +
             ", Conteúdo: " + conteúdo;
    }
  }
```

- Seguem algumas observações sobre esta classe
  - O conceito mais importante é que a classe MensagemTexto estende a classe MensagemAbstrata
    - Isso significa que tudo que MensagemAbstrata é ou tem, MensagemTexto também é ou tem (o reverso não é verdade)
    - MensagemAbstrata é a classe mãe e MensagemTexto é a classe filha (ou subclasse)
    - Por definição, a classe filha implementa todas as interfaces que a classe mãe implementa (e pode implementar mais interfaces se quiser)
  - O atributo que faltava (conteúdo) e o método associado (getConteúdo()) foram acrescentados à classe
    - Objetos dessa classe têm todos os atributos da classe mãe mais os atributos da classe filha
  - Um construtor apropriado foi declarado
    - O corpo do construtor chama o construtor da classe mãe com a linha: super(remetente, assunto);
  - Os métodos que eram abstratos na classe mãe (exibir() e toString()) foram implementados
  - O método equals() foi implementado do zero
    - Ele chama equals() da classe mãe para ajudar a fazer o trabalho
    - Quando a classe filha substitui um método da classe mãe por outra implementação, dizemos que está havendo override (sobreposição)
  - Observe que os atributos não são "private": são "protected"
    - Isso permite que possam ser acessados em subclasses

Representando herança em UML

- A relação de herança que existe entre classes pode ser representada como mostramos

abaixo usando UML
- Observe que uma classe abstrata tem seu nome em itálico
    - Métodos abstratos deveriam estar em itálico também mas meu programa (Rational Rose) está com um bugzinho, acho



A classe MensagemAudio

- Com o conceito de herança, a classe MensagemAudio pode facilmente ser criada
- Ver MensagemAudio.java

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal da Paraíba
 *
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 * Não redistribuir sem permissão.
 */
package p1.aplic.correio;

import p1.aplic.geral.*;
import java.io.*;
import java.net.*;
import java.applet.*;

/**
 * Classe que representa uma mensagem de áudio de correio eletronico.
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal da Paraíba.
```

```
 */

public class MensagemAudio extends MensagemAbstrata {
  protected String  arquivoÁudio;

  /**
   * Cria uma mensagem de áudio de correio eletrônico
   * @param remetente O remetente da mensagem
   * @param assunto O assunto da mensagem
   * @param arquivoÁudio O arquivo contendo o áudio da mensagem
   */
  public MensagemAudio(String remetente, String assunto, String arquivoÁudio) {
    super(remetente, assunto);
    this.arquivoÁudio = arquivoÁudio;
  }

  /**
   * Recupera o arquivo de áudio da mensagem.
   * @return O arquivo de áudio da mensagem.
   */
  public String getarquivoÁudio() {
    return arquivoÁudio;
  }

  /**
   * Exibir a mensagem. O arquivo de áudio é tocado.
   * Após este método, a mensagem é considerada "lida".
   */
  public void exibir() {
    try {
      URL u = new URL("file", "localhost", arquivoÁudio);
      AudioClip clip = Applet.newAudioClip(u);
      System.out.println("Se tiver multimidia no computador, o clip deve estar
      clip.play();
    } catch(Exception e) {
      System.out.println("Nao pode abrir Audio Clip: " + arquivoÁudio);
    }
    estado |= LIDA;
  }

  /**
   * Testa a igualdade de um objeto com esta mensagem.
   * @param objeto O objeto a comparar com esta mensagem.
   * @return true se o objeto for igual a esta mensagem, false caso contrário.
   */
  public boolean equals(Object objeto) {
    if(! (objeto instanceof MensagemAudio)) {
      return false;
    }
    MensagemAudio outra = (MensagemAudio)objeto;
    return super.equals(objeto) &&
           getarquivoÁudio().equals(outra.getarquivoÁudio());
  }

  /**
   * Forneça uma representação da mensagem como String
   * @return A representação da mensagem como String.
   */
  public String toString() {
    return "Remetente: " + remetente +
```

```
            ", Data: " + dataEnvio.DDMMAAAAHHMM() +
            ", Assunto: " + assunto +
            ", Arquivo de áudio: " + arquivoÁudio;
  }
}
```

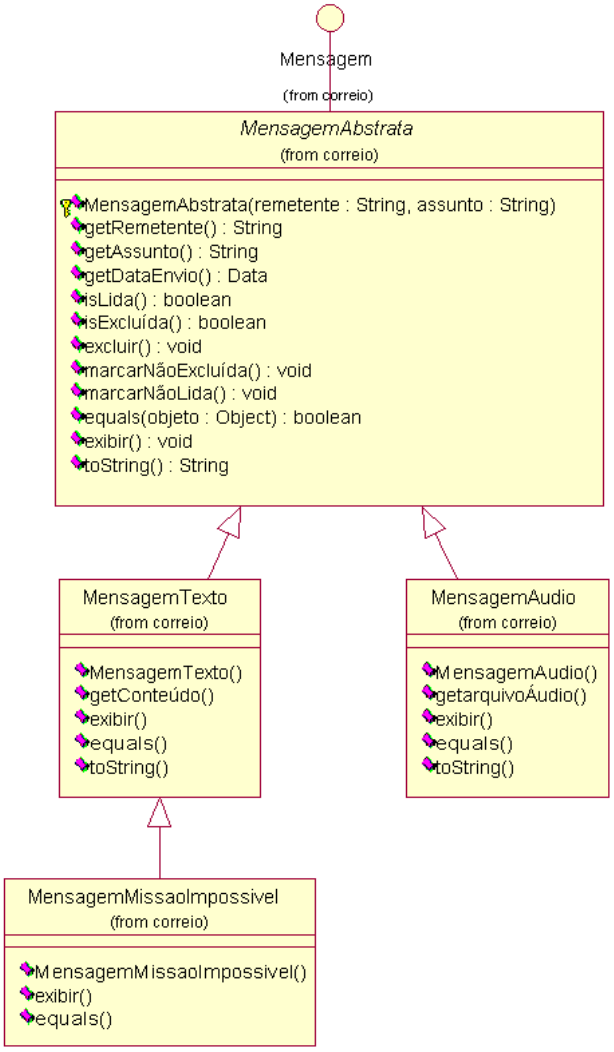
A classe MensagemMissaoImpossivel

- Esta classe é em tudo igual a Mensagemtexto, com exceção de:
  - O construtor: toda classe tem que ter seu próprio construtor
  - O método exibir(): usamos override para implementar a funcionalidade que queremos
  - O método equals(): as classes a testar são diferentes
- Eis o resultado (ver MensagemMissaoImpossivel.java)

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal da Paraíba
 *
 * Copyright (C) 2001 Universidade Federal da Paraíba.
 * Não redistribuir sem permissão.
 */
package p1.aplic.correio;

import p1.aplic.geral.*;
import java.io.*;

/**
 * Classe que representa uma mensagem textual de correio eletronico mas que se
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 2001 Universidade Federal da Paraíba.
 */

public class MensagemMissaoImpossivel extends MensagemTexto {
  /**
   * Cria uma mensagem textual de correio eletrônico
   * @param remetente O remetente da mensagem
   * @param assunto O assunto da mensagem
   * @param contéudo O conteúdo da mensagem, podendo conter várias linhas
   */
  public MensagemMissaoImpossivel(String remetente, String assunto, String cont
    super(remetente, assunto, conteúdo);
  }

  /**
   * Exibir a mensagem. Os dados da mensagem são apresentados na saída padrão.
   * Após este método, a mensagem se auto-destroi.
   */
  public void exibir() {
    super.exibir();
    excluir();
  }

  /**
   * Testa a igualdade de um objeto com esta mensagem.
```

```
     * @param objeto O objeto a comparar com esta mensagem.
     * @return true se o objeto for igual a esta mensagem, false caso contrário.
     */
    public boolean equals(Object objeto) {
      if(! (objeto instanceof MensagemMissaoImpossivel)) {
        return false;
      }
      MensagemMissaoImpossivel outra = (MensagemMissaoImpossivel)objeto;
      return super.equals(objeto);
    }
}
```

- Observe de quem a classe MensagemMissaoImpossivel herda: MensagemTexto
  - Estamos fazendo "Programming by Difference"
  - Isto é: colocamos numa classe a forma com a qual ela é diferente da classe mãe
- O resultado do que fizemos até agora em UML segue abaixo
- Fica óbvio nesta figura que criamos uma hierarquia de classes



Consequências da herança

Poder de substituição de um objeto por outro

- Criamos 4 classes formando uma hierarquia
- Examinando a hierarquia, podemos dizer que uma subclasse é um tipo de classe mãe
  - MensagemTexto "é um tipo de" MensagemAbstrata
  - MensagemAudio "é um tipo de" MensagemAbstrata

    - MensagemMissaoImpossivel "é um tipo de" MensagemTexto
- A relação "É um tipo de" implica que tudo que um objeto da classe mãe faz pode ser feito por um objeto da subclasse
    - Portanto, onde um objeto da classe mãe aparece, podemos colocar um objeto de uma subclasse
    - Chamamos este princípio de "Substitutabilidade"
        - Inventei essa palavra
        - Significa "habilidade de permitir a subtituição"
- Exemplo: na linha seguinte ...

```
MensagemTexto m = new MensagemTexto(remetente, assunto, conteúdo);
m.exibir();
m.excluir();
```

- ... m é uma MensagemTexto
    - Mas tudo funcionaria perfeitamente se fizéssemos m = new MensagemMissaoImpossivel(...)

Polimorfismo

- Como vimos antes, o polimorfismo ocorre quando fazemos chamadas a métodos através de uma interface (ou tipo) genérica
- Mas, ao herdar, uma subclasse automaticamente implementa as interfaces que a classe mãe implementa
- Portanto, uma classe define um tipo, que nem uma interface
    - A diferença sendo que a classe também implementa a interface
    - Talvez seja um implementação parcial, se houver métodos abstratos
- Portanto, em Java, podemos fazer polimorfismo com classes e não só com o uso de "interface"
- O seguinte exemplo

```
Mensagem m = ...;
m.exibir();
```

- ... também poderia ser escrito assim ...

```
MensagemAbstrata m = ...;
m.exibir();
```

- Finalmente, vale a pena observar que podemos fazer hierarquias de interfaces também:

```
public interface MensagemMultimidia extends Mensagem {
   public boolean isÁudio();
   public boolean isVídeo();
   public boolean isAnimação();
}
```

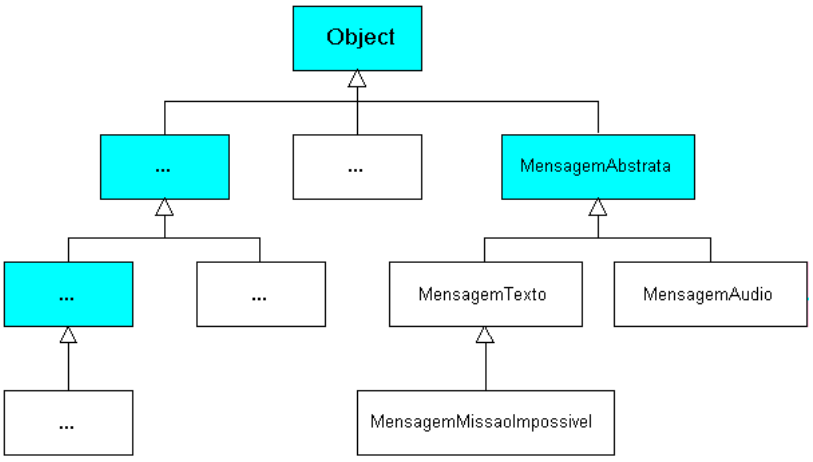
Mais sobre hierarquias de classes

- Em Java, todas as classe pertencem a uma mesma hierarquia com a classe Object no topo
- Se você não falar de que classe mãe uma classe herda, ela herda de Object
    - As duas linhas abaixo são equivalentes:

```
public class CaixaPostal {

public class CaixaPostal extends Object {
```

- É por isso, por exemplo, que um ArrayList, digamos, pode armazenar qualquer coisa

- Ele é feito para armazenar objetos da classe Object o que significa que pode armazenar objetos de qualquer classe
  - Devido ao princípio de Substitutabilidade
- Segue uma representação da hierarquia de classes em Java



- Nesta figura, as classe em azul são abstratas
  - Como se vê, a herança pode ser feita tanto de classe abstratas quanto de classes concretas

Upcasting e downcasting

- Vamos examinar a classe CaixaPostal novamente
  - Apenas parte do código é mostrado abaixo

```
// ...
public class CaixaPostal {
  private List  mensagens;
  private int     índiceMensagemCorrente;

  // ...

  public void inserir(Mensagem m) {
    mensagens.add(m);
    índiceMensagemCorrente = Math.max(índiceMensagemCorrente, 0);
  }

  public Mensagem mensagemCorrente() {
    return índiceMensagemCorrente >= 0 ? (Mensagem)mensagens.get(índiceMensagem
  }

  public void salvar() {
    // primeiro, remover as mensagens excluídas
    Iterator it = iterator();
    while(it.hasNext()) {
      Mensagem m = (Mensagem)it.next();
      if(m.isExcluída()) {
        it.remove();
      }
    }
    // ... salva em disco
  }
}
```

- Observe que, no método inserir(), inserimos objetos do tipo Mensagem
  - Pode ser qualquer objeto das classes concretas MensagemTexto, MensagemAudio, MensagemMissaoImpossivel

- Com que tipo o método add() de List recebe os objetos?
  - Examine a documentação de List e verá que é add(Object objeto)
- Portanto, estamos transformando um tipo de baixo para cima na hierarquia
  - Isso se chama upcasting e é sempre possível, devido ao princípio de Substitutabilidade
- No método salvar(), estamos vendo o contrário: downcasting
  - O método next() do iterador retorna um Object (verifique a documentação!)
  - Nós queremos tratar esse objeto como algo mais específico: uma Mensagem
- O downcasting nem sempre funciona!
  - Você tem que ter certeza que só colocou objetos do tipo Mensagem dentro do ArrayList!
  - Se sair um objeto da classe CachorroMorto, vai dar ClassCastException
- Exercício: Nas linhas abaixo:
  - Haverá ClassCastException na primeira linha? Por que ou por que não?
  - Haverá ClassCastException na segunda linha? Por que ou por que não?

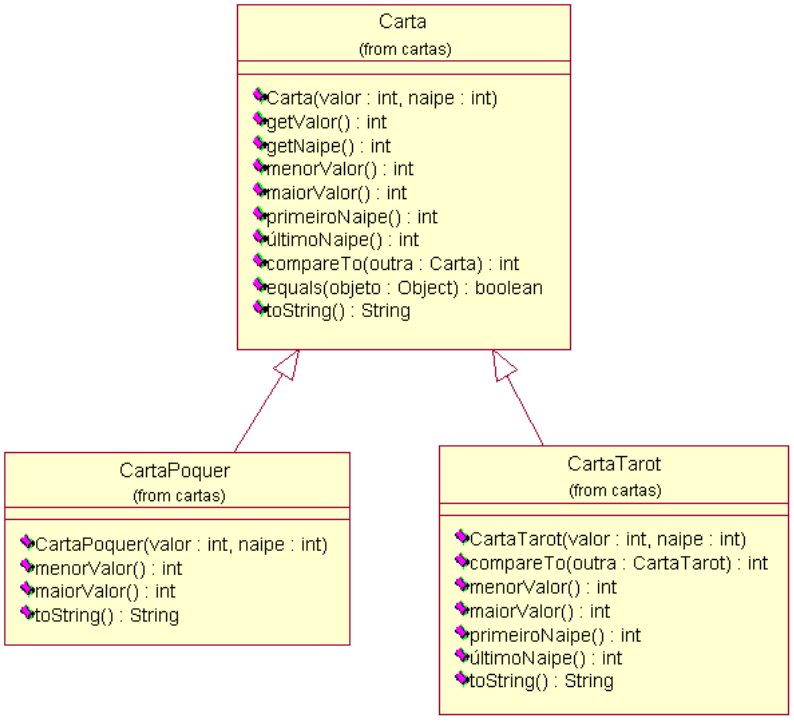
```
Object x = new MensagemTexto(remetente, assunto, conteúdo);
System.out.println((String)x);
```

Exemplos do uso de herança

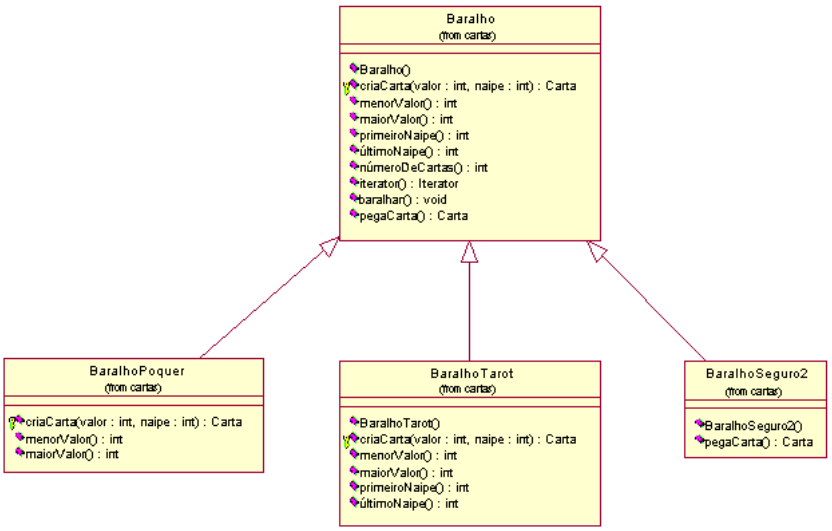
- No package p1 usado na disciplina, temos vários exemplos do uso de herança
- Usamos UML abaixo para mostrar as situações do uso de herança
  - O aluno é responsável por estudar todos os exemplos aqui
  - Haverá avaliação baseada nesses exemplos

No mundo das cartas

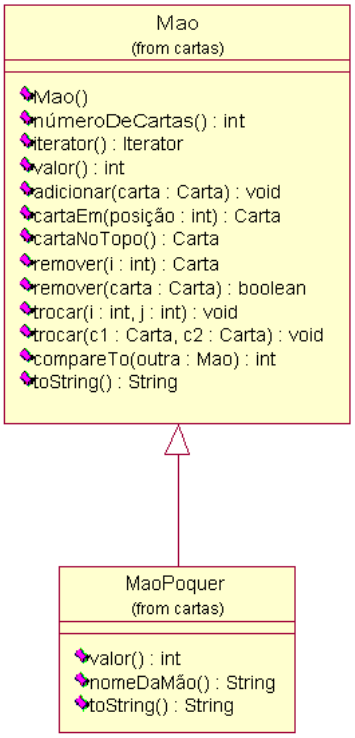
- Exemplo 1: Carta, CartaPoquer, CartaTarot
  - Ver Carta.java, CartaPoquer.java, CartaTarot.java
  - Observe que todas as classes são concretas (podem ser instanciadas)



- Exemplo 2: Baralho, BaralhoPoker, BaralhoSeguro, BaralhoSeguro2, BaralhoTarot
  - Ver Baralho.java, BaralhoPoker.java, BaralhoSeguro.java, BaralhoSeguro2.java, BaralhoTarot.java



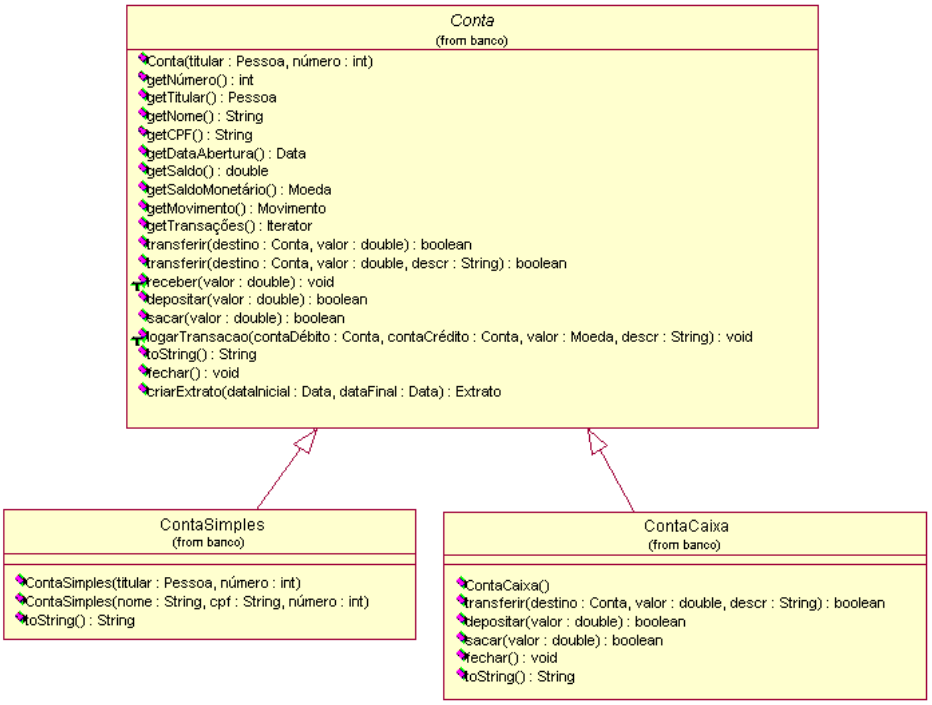
- Exemplo 3: Mao, MaoPoquer
  - Ver Mao.java, MaoPoquer.java



- Tem mais exemplos no package p1.aplic.cartas
  - Descubra-os!

No mundo bancário

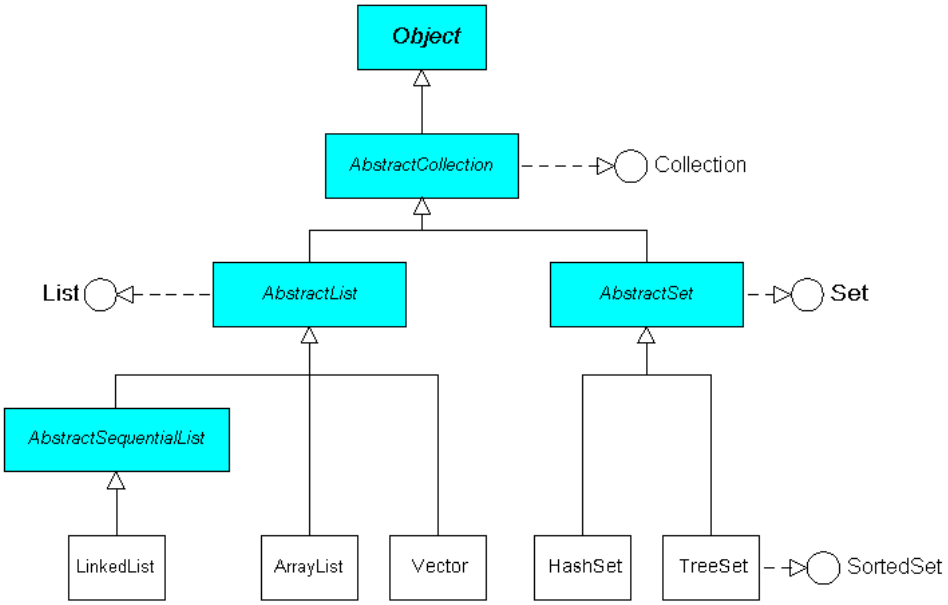
- Exemplo 1: Conta, ContaSimples, ContaCaixa
  - Ver Conta.java, ContaSimples.java, ContaCaixa.java



- Tem mais exemplos no package p1.aplic.banco
  - Descubra-os!

No mundo Java

- Tem dezenas de exemplos
- Um deles: o mundo das coleções



- Observe que as 4 interfaces em si formam uma hierarquia:

```
public interface List extends Collection { ... }
public interface Set extends Collection { ... }
public interface SortedSet extends Collection, Set { ... }
```

- Veja que SortedSet herda de *duas* interfaces
  - Java permite herança múltipla, mas apenas de interfaces, não de classes
- Fuce a documentação do Java para ver como são essas hierarquias

Exercício

- As classes CorreioIU1, CorreioIU2 e CorreioIU3 têm muita coisa em comum
    - Ver CorreioIU1.java, CorreioIU2.java e CorreioIU3.java
- Use UI herança para refatorar o código

oo-5 programa anterior próxima