ArrayList<E>

- As classes da Java Collections Framework são exemplos muito interessantes de codificação
- Como o código destas classes está escrito em Java é possível ao programador observar como é que foram implementadas

ArrayList<E>: v.i. e construtores

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer.
     */
    private transient Object[] elementData;

    /**
     * The size of the ArrayList (the number of elements it contains).
     *
     * @serial
     */
    private int size;

    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param  initialCapacity  the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        ...
        this.elementData = new Object[initialCapacity];
    }

    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this(10);
    }
```

ArrayList<E>: existe?

```
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

ArrayList<E>: inserir

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}


public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

ArrayList<E>: get e set

```
public E get(int index) {
    rangeCheck(index);

    return elementData(index);
}


public E set(int index, E element) {
    rangeCheck(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

Hierarquia de Classes e Herança

- ***(Grady Booch) The Meaning of Hierarchy:***
  - *"Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem."*
- Logo*, "Hierarchy is a ranking or ordering of abstractions."*


TOY MICE
CLOCKWORK MOTORS
COMPONENT PARTS
ATOMS
ETC. ETC

- Até agora só temos visto classes que estão ao mesmo nível hierárquico. No entanto...
- A colocação das classes numa hierarquia de especialização (do mais genérico ao mais concreto) é uma das características mais importantes da POO
- Esta hierarquia é importante:
  - ao nível da reutilização de variáveis e métodos
  - da compatibilidade de tipos

- No entanto, a tarefa de criação de uma hierarquia de conceitos (classes) é complexa, porque exige que se **classifiquem** os conceitos envolvidos

- A criação de uma hierarquia é do ponto de vista operacional um dos mecanismos que temos para criar novos conceitos a partir de conceitos existentes

  - a este nível já vimos a composição de classes

- Exemplos de composição de classes
  - um segmento de recta (exemplo da Ficha 3) é composto por duas instâncias de Ponto2D
  - um Triângulo pode ser definido como composto por três segmentos de recta ou por um segmento e um ponto central, ou ainda por três pontos

- Uma outra forma de criar classes a partir de classes já existentes é através do mecanismo de herança.

- Considere-se que se pretende criar uma classe que represente um Ponto 3D

  - quais são as alterações em relação ao Ponto2D?

    - mais uma v.i. e métodos associados

- A classe Ponto2D (incompleta):

```
public class Ponto2D {
   // Construtores
   public Ponto2D(int cx, int cy) { x = cx; y = cy; }
   public Ponto2D(){ this(0, 0); }
   // Variáveis de Instância
   private int x, y;
   // Métodos de Instância
   public int getX() { return x; }
   public int getY() { return y; }
   public void desloca(int dx, int dy) {
      x += dx; y += dy;
   }
   public void somaPonto(Ponto2D p) {
      x += p.getX(); y += p.getY();
   }
   public Ponto2D somaPonto(int dx, int dy) {
      return new Ponto2D(x += dx, y+= dy);
   }
   public String toString() {
      return new String("Pt= " +x + ", " + y);
   } }
```

- o esforço de codificação consiste em acrescentar uma v.i. (z) e getZ() e setZ()

- O mecanismo de herança proporciona um esforço de programação diferencial

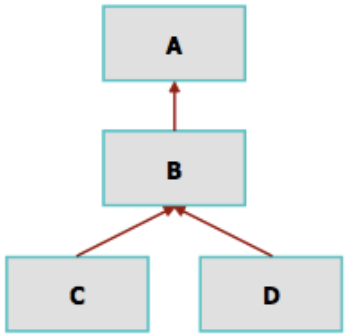
$$\text{Ponto3D} = \text{Ponto2D} + \Delta_{\text{prog}}$$
$$1\ \text{ponto3D} \Leftrightarrow 1\ \text{ponto2D} + \Delta_{\text{var}} + \Delta_{\text{met}}$$

- ou seja, para ter um Ponto3D precisamos de tudo o que existe em Ponto2D e acrescentar um *delta* que consiste nas características novas

- A classe Ponto3D aumenta, refina, detalha, especializa, a classe Ponto2D

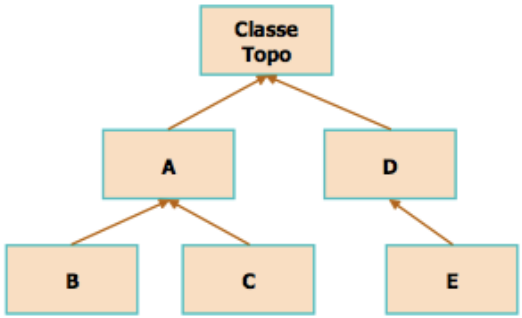
- Como se faz isto?
  - de forma ad-hoc, sem suporte, através de um mecanismo de copy&paste
  - usando composição, isto é, tendo como v.i. de Ponto3D um Ponto2D
  - através de um mecanismo existente de base nas linguagens por objectos que é a noção de hierarquia e herança

- Exemplo:



- A é superclasse de B
- B é superclasse de C e D
- C e D são subclasses de B
- B especializa A, C e D especializam B (e A!)

- Hierarquia típica em Java:

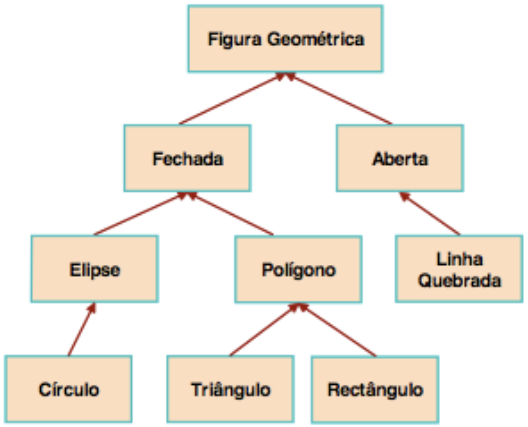


  - hierarquia de herança simples (por oposição, p.ex., a C++)

- O que significa do ponto de vista semântico dizer que duas classes estão hierarquicamente relacionadas?

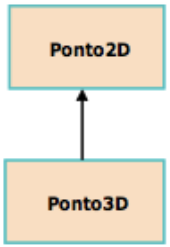
- no paradigma dos objectos a hierarquia de classes é uma hierarquia de especialização
  - uma subclasse de uma dada classe constitui uma especialização, sendo por definição mais detalhada que a classe que lhe deu origem
  - ie, possui **mais** estado e **mais** comportamento

- A exemplo de outras taxonomias, a classificação do conhecimento é realizada do geral para o particular



- a especialização pode ser feita nas duas vertentes: estrutural e comportamental

- voltando ao Ponto3D:



- ou seja, Ponto3D é subclasse de Ponto2D

```
public class B extends A { …
public class Ponto3D extends Ponto2D { …
```

- como foi dito, uma subclasse herda estrutura e comportamento da sua classe:

O mecanismo de herança

- se uma classe B é subclasse de A, então:
  - B é uma especialização de A
  - este relacionamento designa-se por "é um" ou "é do tipo", isto é, uma instância de B pode ser designada como sendo um A
    - implica que aos atributos e métodos de A acrescentou mais informação

- Se uma classe B é subclasse de A:
  - se B **pertence** ao mesmo package de A, B herda todas as variáveis e métodos de instância que não são private.
  - se B **não pertence** ao mesmo package de A, B herda todas as variáveis e métodos de instância que não são private ou package. Herda tudo o que é public ou protected

- B pode **definir** novas variáveis e métodos de instância próprios
- B pode **redefinir** variáveis e métodos de instância herdados
- variáveis e métodos de classe não são herdados. Podem ser redefinidos.
- métodos construtores não são herdados

- na definição que temos utilizado nesta unidade curricular, as nossas variáveis de instância são declaradas como **private**

- que impacto é que isto tem no mecanismo de herança?

  - total, vamos deixar de poder referir as v.i. da superclasse que herdamos pelo nome

  - vamos utilizar os métodos de acesso, *getX()*, para aceder aos seus valores

- Para percebermos a dinâmica do mecânismo de herança, vamos prestar especial atenção ao seguintes aspectos:
  - redefinição de variáveis e métodos
  - procura de métodos
  - criação de instâncias das subclasses

Redefinição variáveis e métodos

- o mecanismo de herança é automático e total, o que significa que uma classe herda obrigatoriamente da sua superclasse directa e superclasses transitivas um conjunto de variáveis e métodos
- no entanto, uma determinada subclasse pode pretender modificar localmente uma definição herdada
  - *a definição local é sempre a prioritária*

- na literatura quando um método é redefinido, é comum dizer que ele é reescrito ou *overriden*

- quando uma variável de instância é declarada na subclasse diz-se que é escondida (*hidden* ou *shadowed*)

- A questão é saber se ao redefinir estes conceitos se perdemos, ou não, o acesso ao que foi herdado!

- considere-se a classe Super

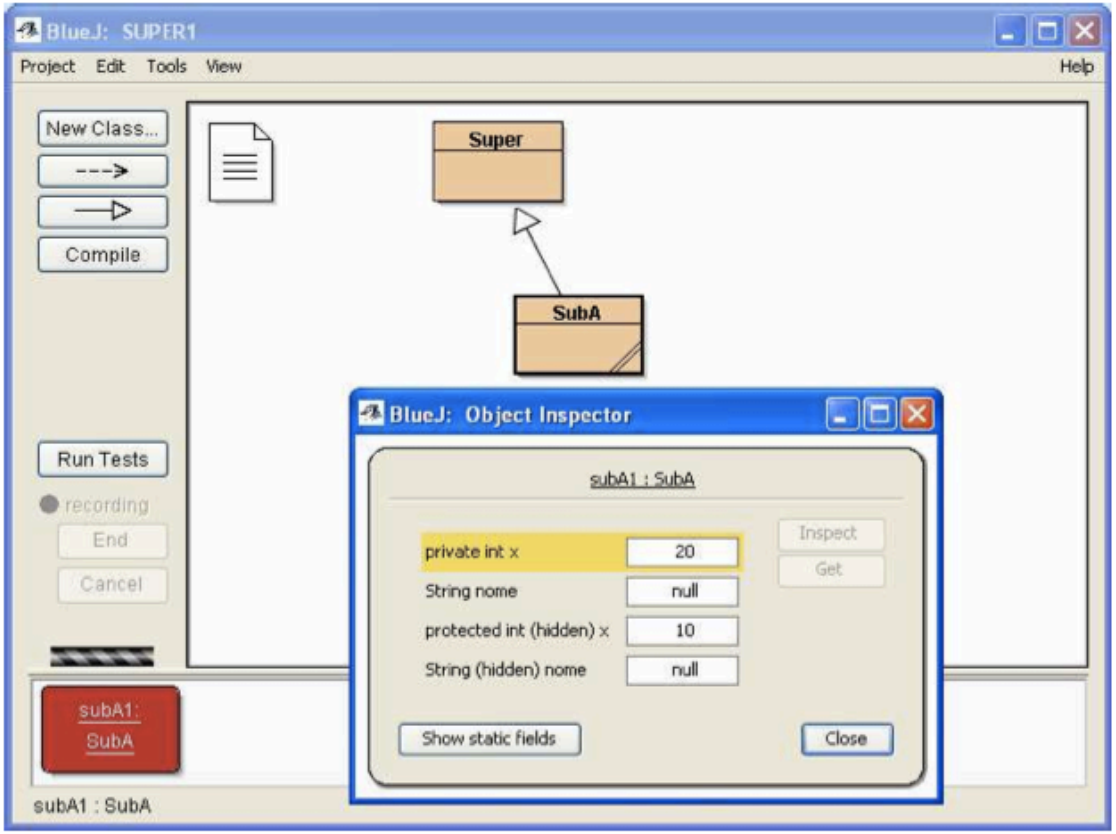
```
public class Super {
   protected int x = 10;
   String nome;
   // Métodos
   public int getX() { return x; }
   public String classe() { return "Super"; }
   public int teste() { return this.getX(); }
}
```

- e uma subclasse SubA

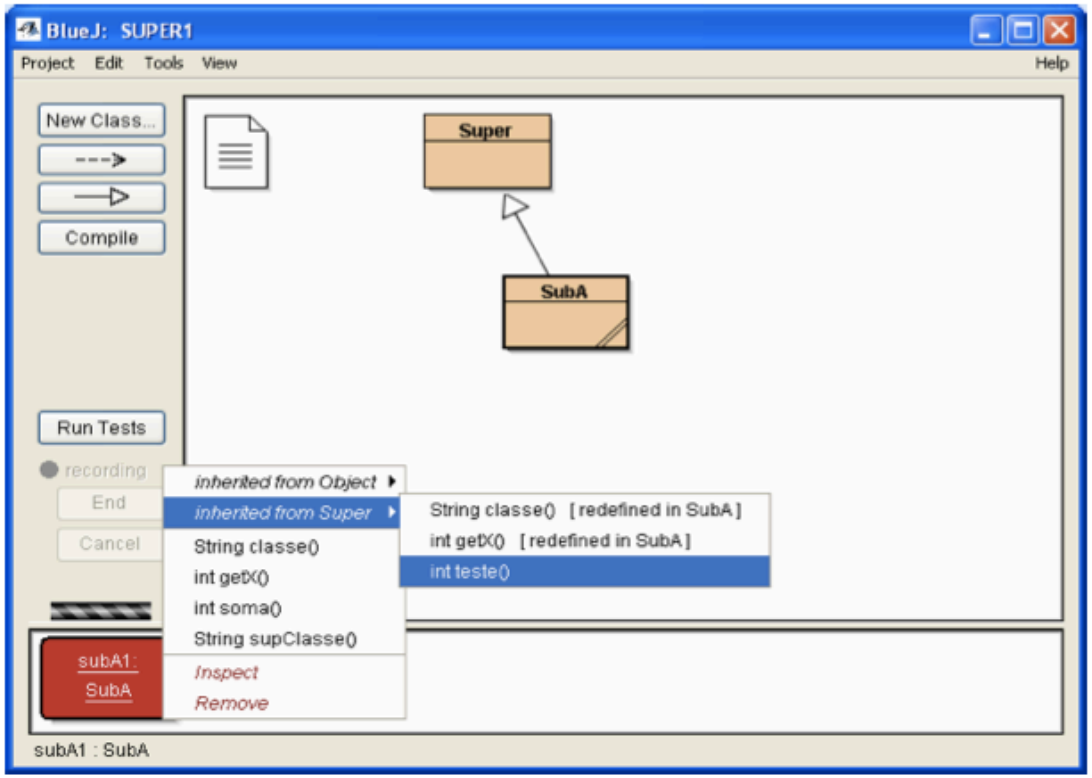
```
public class SubA extends Super {
   private int x = 20; // "shadow"
   String nome;        // "shadow"
   // Métodos
   public int getX() { return x; }
   public String classe() { return "SubA"; }
   public String supClass() { return super.classe(); }
   public int soma() { return x + super.x; }
}
```

- o que é a referência **super**?
  - um identificador que permite que a procura seja remetida para a superclasse
  - ao fazer super.m(), a procura do método m() é feita na superclasse e não na classe da instância que recebeu a mensagem
  - apesar da sobreposição (override), tanto o método local como o da superclasse estão disponíveis

- veja-se o inspector de um objecto no BlueJ



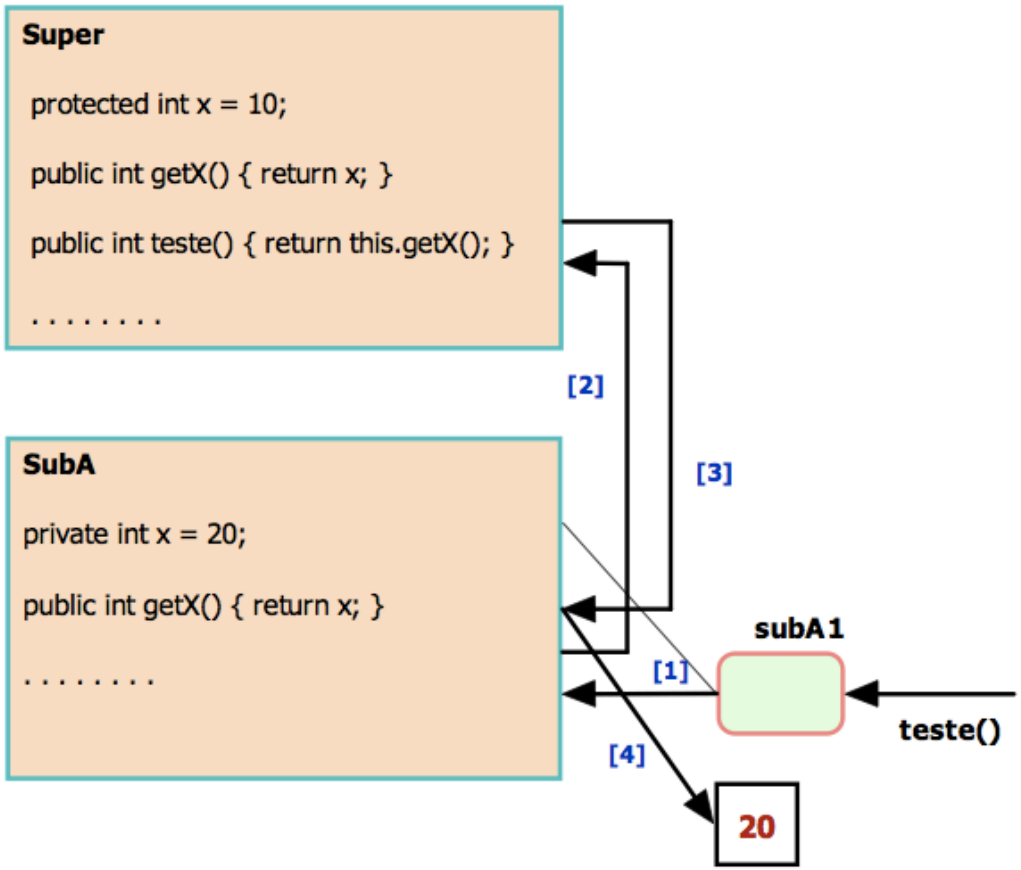
- no BlueJ é possível ver os métodos definidos na classe e os herdados



- o que acontece quando enviamos à instância subA1 (imagem anterior) a mensagem teste()?
  - teste() é uma mensagem que não foi definida na subclasse
  - o algoritmo de procura vai encontrar a definição na superclasse
  - o código a executar é `return this.getX()`

- em Super o valor de x é 10, enquanto que em SubA o valor de x é 20.
  - qual é o contexto de execução de this.getX()?
  - a que instância é que o this se refere?
- Vejamos o algoritmo de procura e execução de métodos...

- execução da invocação de teste()



- na execução do código, a referência a this corresponde sempre ao objecto que recebeu a mensagem
  - neste caso, subA1
- sendo assim, o método getX() é o método de SubA que é a classe do receptor da mensagem
  - independentemente do contexto "subir e descer", o `this` refere sempre o receptor da mensagem!

Regra para avaliaçao de this.m()

- de forma geral, a expressão **this.m()**, onde quer que seja encontrada no código de um método de uma classe (independentemente da localização na hierarquia), corresponde sempre à execução do método **m()** da classe do receptor da mensagem

Modificadores e redifinição de métodos

- a possibilidade de redefinição de métodos está condicionada pelo tipo de modificadores de acesso do método da superclasse (private, public, protected, package) e do método redefinidor
- o método redefinidor não pode diminuir o nível de acessibilidade do método redefinido

- os métodos public podem ser redefinidos por métodos public
- métodos protected por public ou protected
- métodos package por public ou protected ou package

Criação das instâncias das subclasses

- em Java é possível definir um construtor à custa de um construtor da mesma classe, ou seja, à custa de this()
- fica agora a questão de saber se é possível a um construtor de uma subclasse invocar os construtores da superclasse
    - como vimos atrás os construtores não são herdados

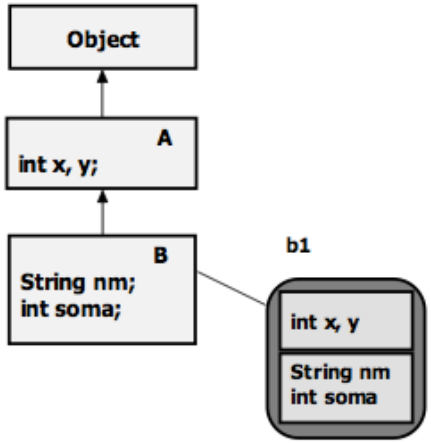
- quando temos uma subclasse B de A, sabe-se que B herda todas as v.i. de A a que tem acesso.

- assim cada instância de B é constituída pela soma das partes:

  - as v.i. declaradas em B

  - as v.i. herdadas de A

- em termos de estrutura interna, podemos dizer que temos:



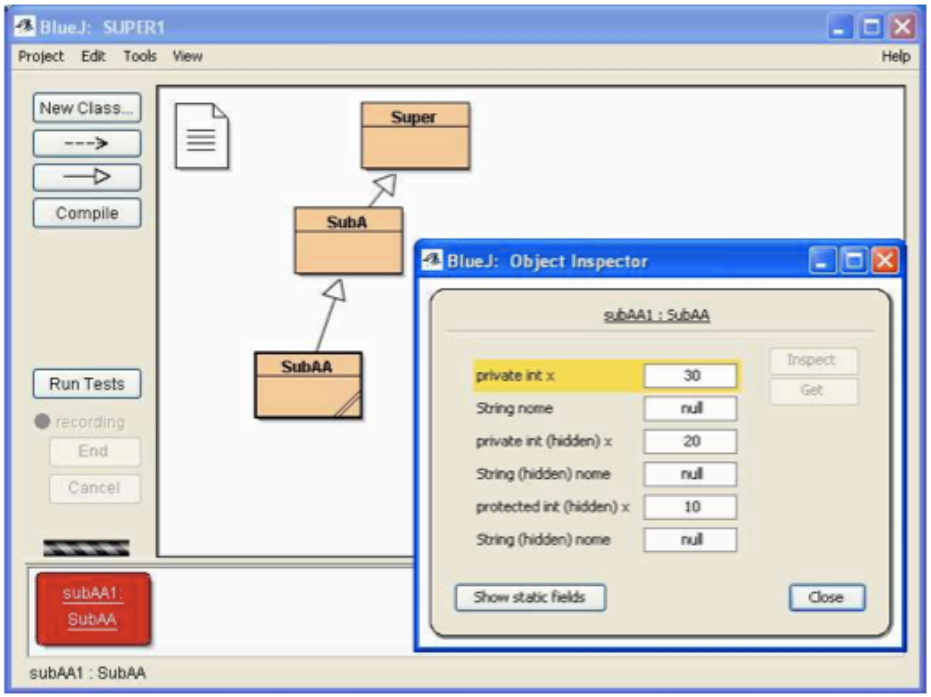
- como sabemos que B tem pelo menos um construtor definido, B(), as v.i. declaradas em B (nm e soma) são inicializadas

- ... mas quem inicializa as variáveis que foram declaradas em A?
  - a resposta evidente é: os métodos encarregues de fazer isso em A, ou sejam, os construtores de A
  - dessa forma qualquer construtor de B deve invocar os construtores de A para inicializar as v.i. declaradas em A

- em Java para que seja possível a invocação do construtor de uma superclasse esta deve ser feita logo no início do construtor da subclasse
  - recorrendo a super(...,...), em que a verificação do construtor a invocar se faz pelo matching dos parâmetros e respectivos tipos de dados
- de facto a invocação de um construtor numa subclasse, cria uma cadeia transitiva de invocações de construtores

- a cadeia de construtores é implícita e na pior das hipóteses usa os construtores que por omissão são definidos em Java.
  - por isso em Java são disponibilizados por omissão
- por aqui se percebe o que Java faz quando cria uma instância: aloca espaço e inicializa todas as v.i. que são criadas pelas diversas classe até Object

- Veja-se o exemplo:

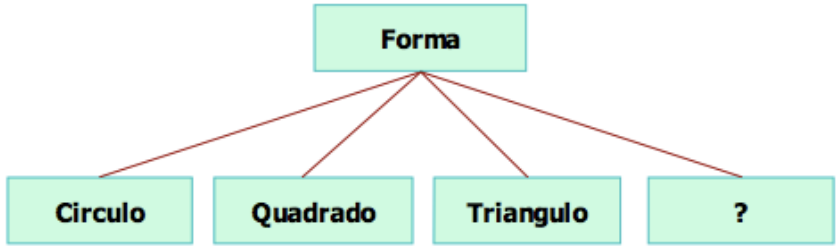


Classes Abstractas

- até ao momento todas as classes definiram completamente todo o seu estado e comportamento
- no entanto, na concepção de soluções por vezes temos situações em que o código de uma classe pode não estar completamente definido
  - esta é uma situação comum em POO e podemos tirar partido dela para criar soluções mais interessantes

- consideremos que precisamos de manipular forma geométricas (triângulos, quadrados e círculos)
  - no entanto podemos acrescentar, com o evoluir da solução, mais formas geométricas
  - torna-se necessário uniformizar a API que estas classes tem de respeitar
    - todos tem de ter `area()` e `perimetro()`

- Seja então a seguinte hierarquia:



  - conceptualmente correcta e com capacidade de extensão através da inclusão de novas subclasses de forma

  - mas qual é o estado e comportamento de Forma?

- A classe Forma pode definir algumas v.i., como um ponto central (um Ponto2D), mas se quiser definir os métodos area() e perímetro() como é que pode fazer?
  - Solução 1: não os definir deixando isso para as subclasses
    - as subclasses podem nunca definir estes métodos e aí perde-se a capacidade de dizer que todas as formas respondem a esses métodos

- Solução 2: definir os métodos area() e perimetro() com um resultado inútil, para que sejam herdados e redefinidos

- Solução 3: aceitar que nada pode ser escrito que possa ser aproveitado pelas subclasses e que a única declaração que interessa é a assinatura do método a implementar

    - a maioria das linguagens por objectos aceitam que as definições possam ser imcompletas

- em POO designam-se por **classes abstractas** as classes nas quais pelo menos um método de instância não se encontram implementados, mas apenas declarados
    - são designados por métodos abstractos ou virtuais
    - uma classe 100% abstracta tem apenas assinaturas de métodos

- no caso da classe Forma não faz sentido definir os métodos area() e perímetro, pelo que escrevemos apenas:

```
public abstract class Forma {
//
   public abstract double area();
   public abstract double perimetro();
}
```

- como os métodos não estão definidos, não é possível criar instâncias de classes abstractas

- apesar de ser uma classe abstracta, o mecanismo de herança mantém-se e dessa forma uma classe abstracta é também um (novo) tipo de dados
    - compatível com as instâncias das suas subclasses
    - torna válido que se faça `Forma f = new Triangulo()`

- uma classe abstracta ao não implementar determinados métodos, **obriga** a que as suas subclasses os implementem
  - se não o fizerem, ficam como abstractas
- para que servem métodos abstractos?
  - para garantir que as subclasses respondem àquelas mensagens de acordo com a implementação desejada

- Em resumo, as classes abstractas são um mecanismo muito importante em POO, dado que permitem:
  - escrever especificações sintácticas para as quais são possíveis múltiplas implementações
  - fazer com que futuras subclasses decidam como querem implementar esses métodos

- Classe Circulo

```
public class Circulo extends Forma {
   // variáveis de instância
   private double raio;
   // construtores
   public Circulo() {  raio = 1.0; }
   public Circulo(double r) {  raio = (r <= 0.0 ?  1.0 : r); }
   // métodos de instância
   public double area() { return PI*raio*raio; }
   public double perimetro() { return 2*PI*raio; }
   public double raio() {  return raio; }
}
```

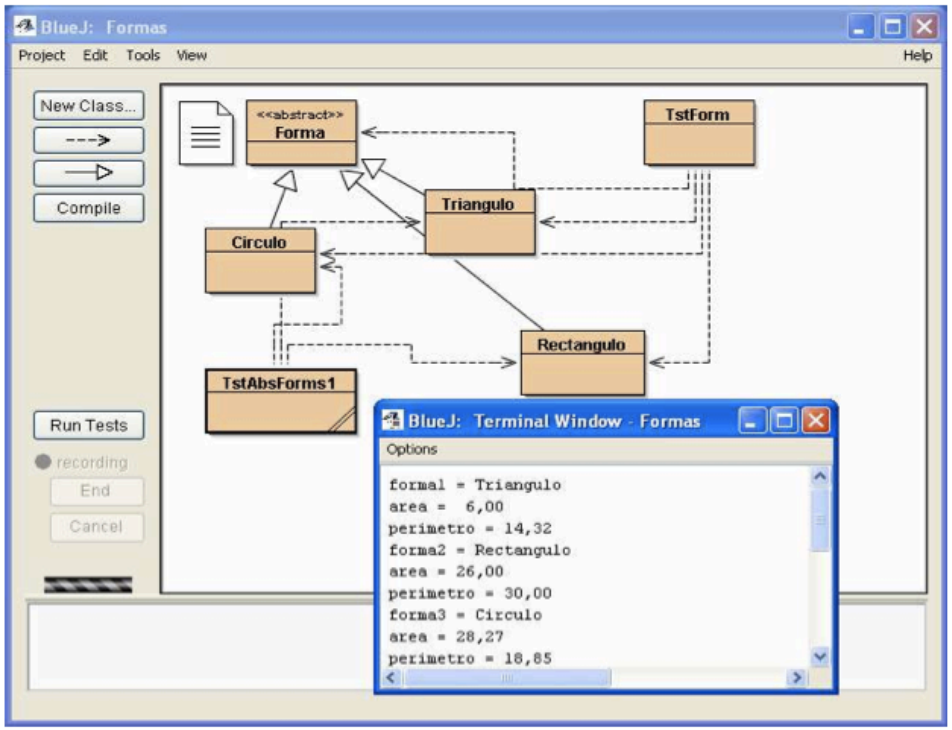
- Classe Rectangulo

```
public class Rectangulo extends Forma {
   // variáveis de instância
   private double comp, larg;
   // construtores
   public Rectangulo() {  comp = 0.0; larg = 0.0; }
   public Rectangulo(double c, double l) {  comp = c; larg = l; }
   // métodos de instância
   public double area() { return comp*larg; }
   public double perimetro() { return 2*(comp+larg); }
   public double largura() {  return larg; }
   public double comp() {  return comp; }
 }
```

- Classe Triangulo:

```
public class Triangulo extends Forma {
   /* altura tirada a meio da base */
   // variáveis de instância
   private double base, altura;
   // construtores
   public Triangulo() {  base = 0.0; altura = 0.0; }
   public Triangulo(double b, double a) {
      base = b; altura = a;
   }
   // métodos de instância
   public double area() { return base*altura/2; }
   public double perimetro() {
      return base + (2*this.hipotenusa()); }
   public double base() {  return base; }
   public double altura() {  return altura; }
   public double hipotenusa() {
      return sqrt(pow(base/2, 2.0) + pow(altura, 2.0)) ;
   }
}
```

- execução do envio dos métodos a diferentes objectos - respostas diferentes consoante o receptor: **polimorfismo**!!



Compatibilidade entre classes e subclasses

- uma das vantagens da construção de uma hierarquia é a reutilização de código, mas...
- os aspectos relacionados com a criação de tipos de dados são também não negligenciáveis
- as classes são associadas estaticamente a tipos

- é preciso saber qual a compatibilidade entre os tipos das diferentes classes (superclasses e subclasses)
  - a questão importante é saber se uma classe é compatível com as suas subclasses!
- é importante reter o princípio da substituição que diz que...

- “se uma variável é declarada como sendo de uma dada classe (tipo), é legal que lhe seja atribuído um valor (instância) dessa classe ou de qualquer das suas subclasses”
  - existe compatibilidade de tipos no sentido ascendente da hierarquia (eixo da generalização)
  - ou seja, uma instância de uma subclasse pode ser atribuída a uma instância da superclasse (`Forma f = new Triangulo()`)

- seja o código

```
A a, a1;
a = new A(); a1 = new B();
```

- ambas as declarações estão correctas, tendo em atenção a declaração de varíavel e a atribuição de valor
  - B é uma subclasse de A, pelo que está correcto
- mas o que acontece quando se executa `a1.m()`?

- o compilador tem de verificar se m() existe em A ou numa sua superclasse
  - se existir é como se estivesse declarado em B
  - a expressão é correcta do ponto de vista do compilador
- em tempo de execução terá de ser determinado qual é o método a ser invocado.

- o interpretador, em tempo de execução, faz o *dynamic binding* procurando determinar em função do valor contido qual é o método que deve invocar

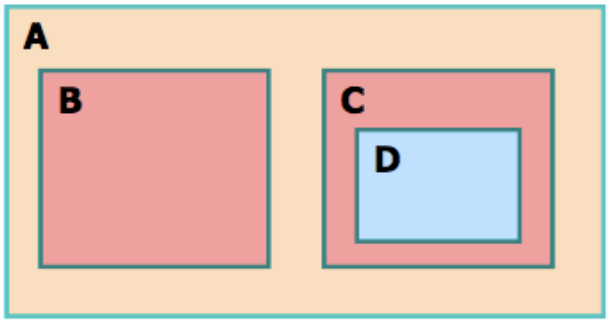
- se várias classes da hierarquia implementarem o método m(), então o interpretador executa o método associado ao tipo de dados da classe do objecto

- Na expressão f.toString(), qual é o método que é invocado? O toString da classe Triangulo!!

- Seja o seguinte código

```
public class A {
   public A() { a = 1; }
   public int daVal() { return a; }
   public void metd() { a += 10; }
}
public class B extends A {
   public B() { b = 2; }
   private int b;
   public int daVal() { return b; }
   public void metd() { b += 20 ; }
}
public class C extends A {
   public C() { c = 3; }
   private int c;
   public int daVal() { return c; }
   public void metd() { c += 30 ; }
}
public class D extends C {
   public D() { d = 33; }
   private int d;
   public int daVal() { return d; }
   public void metd() { d = d * 10 + 3 ; } }
```

- do ponto de vista dos tipos de dados especificados e da relação entre eles, podemos estabelecer as seguintes relações de inclusão:



  - ou seja, um D pode ser visto como um C ou um A. Um C pode ser visto como um A, etc...

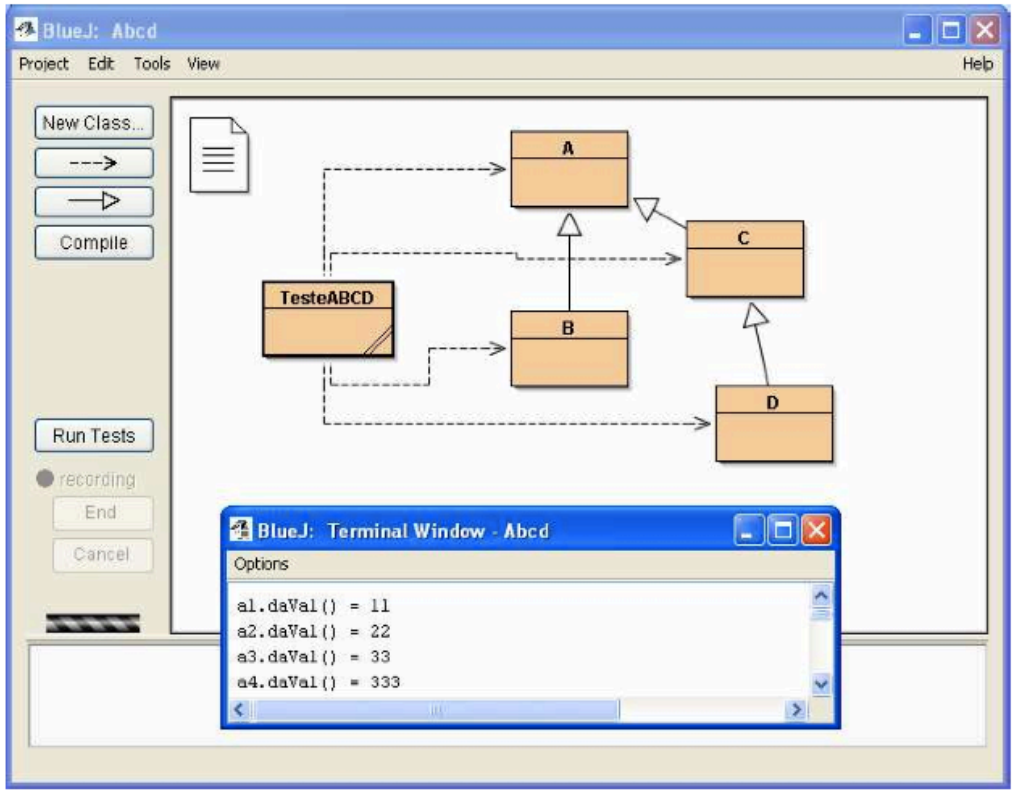
- e as seguintes inicializações de variáveis

```
import static java.lang.System.out;
public class TesteABCD {
   public static void main(String args[]) {
     A a1, a2, a3, a4;
     a1 = new A(); a2 = new B(); a3 = new C(); a4 = new D();
     a1.metd(); a2.metd(); a3.metd(); a4.metd();
     out.println("a1.metd() = " + a1.daVal());
     out.println("a2.metd() = " + a2.daVal());
     out.println("a3.metd() = " + a3.daVal());
     out.println("a4.metd() = " + a4.daVal());
   }
}
```

- qual é o resultado deste programa?

- importa agora distinguir dois conceitos muito importantes:
  - tipo *estático* da variável
    - é o tipo de dados da declaração, tal como foi aceite pelo compilador
  - tipo *dinâmico* da variável
    - corresponde ao tipo de dados associado ao construtor que criou a instância

- como o interpretador executa o algoritmo de procura dinâmica de métodos, executando metd() em cada uma das classes, então o resultado é:



O equals, novamente...

- como vimos anteriormente o método equals de uma subclasse deve invocar o método equals da superclasse, para nesse contexto comparar os valores das v.i. lá declaradas.
  - utilização de super.equals()

- seja o método equals da classe Aluno (já conhecido de todos)

```
/**
 * Implementação do método de igualdade entre dois Aluno
 *
 * @param  umAluno   aluno que é comparado com o receptor
 * ** * @return      booleano true ou false
 * ** * */
public boolean equals(Object umAluno) {
    if (this == umAluno)
        return true;

    if ((umAluno == null) || (this.getClass() != umAluno.getClass()))
        return false;
    else {
        Aluno a =  (Aluno) umAluno;
        return(this.nome.equals(a.getNome()) && this.nota == a.getNota()
               && this.numero == a.getNumero());
    }

}
```

- seja agora o método equals da classe AlunoTE, que é subclasse de Aluno:

```
/**
 * Implementação do método de igualdade entre dois Alunos do tipo T-E
 *
 * @param  umAluno   aluno que é comparado com o receptor
 * ** * @return      booleano true ou false
 * ** * */

public boolean equals(Object umAluno) {
    if (this == umAluno)
        return true;

    if ((umAluno == null) || (this.getClass() != umAluno.getClass()))
        return false;
    else {
        AlunoTE a =  (AlunoTE) umAluno;
        return(super.equals(a) & this.nomeEmpresa.equals(a.getNomeEmpresa());
    }

}
```

- considerando o que se sabe sobre os tipos de dados, a invocação this.getClass() continua a dar os resultados pretendidos?

Polimorfismo

- capacidade de tratar da mesma forma objectos de tipo diferente
  - desde que sejam compatíveis a nível de API
  - ou seja, desde que exista um tipo de dados que os inclua

- no caso do projecto dos Empregados:

```
public double totalSalarios() {
    double total = 0.0;
    for(Empregado emp : emps) total += emp.salario();
    return total;
}

public int totalGestores() {
    int total = 0;
    for(Empregado emp : emps) if(emp instanceof Gestor) total++;
    return total;
}

public int totalDe(String Tipo) {
   int total = 0;
    for(Empregado emp : emps)
      if(emp.getClass().getName().equals(Tipo)) total++;
    return total;
}

public double totalKms() {
    double totalKm = 0.0;
    for(Empregado emp : emps)
      if(emp instanceof Motorista) totalKm += ((Motorista) emp).getKms();
    return totalKm;
}
```

- todos respondem a totalSalarios(), embora com implementações diferentes

Herança vs Composição

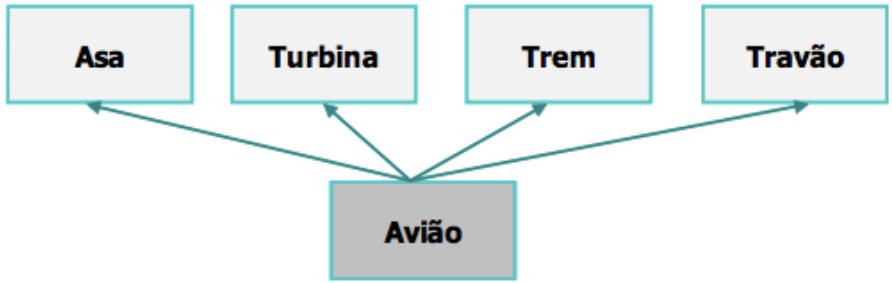
- Herança e composição são duas formas de relacionamento entre classes
  - são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas para o mesmo fim
- existe uma tendência para se confundir herança com composição

- quando uma classe é criada por composição de outras, isso implica que as instâncias das classes agregadas fazem parte da definição do contentor
  - é uma relação do tipo "parte de" (part-of)
  - qualquer instância da classe vai ser constituída por instâncias das classes agregadas
    - Exemplo: Círculo tem um ponto central (Ponto2D)

- do ponto de vista do ciclo de vida a relação é fácil de estabelecer:
  - quando a instância contentor desaparece, as instâncias agregadas também desaparecem
  - o seu tempo de vida está decisivamente ligado ao tempo de vida da instância de que fazem parte!

- esta é uma forma (e está aqui a confusão) de criar entidades mais complexas a partir de entidades mais simples:
  - Turma é composta por instâncias de Aluno
  - Automóvel é composto por Pneu, Motor, Chassis, ...
  - Empresa por instâncias de Empregado

- Por vezes em situação de herança múltipla parece tornar-se apelativa uma solução como:



- embora o que se pretende ter é composição. Na solução apresentada o avião apenas tem **uma** asa, **uma** turbina,

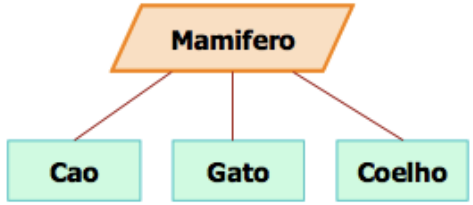
- no caso de termos herança simples (a que temos em Java) a solução de ter um Avião como subclasse de Asa é perfeitamente ridícula.
  - é errado dizer que Asa *is-a* Avião
  - é correcto dizer que Asa *part-of* Avião

- quando uma classe (apesar de ter instâncias de outras classes no seu estado interno) for uma especialização de outra, então a relação é de Herança

- quando não ocorrer esta noção de especialização então a relação deverá ser de composição

Tipos Parametrizados

- Seja a seguinte hierarquia:

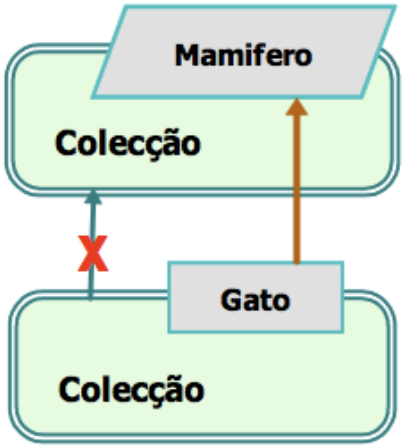


  - e considere-se uma colecção de elementos do tipo Mamífero

- Um arraylist de mamíferos, ArrayList<Mamifero> pode conter instâncias de Cão, Gato, Coelho, etc.

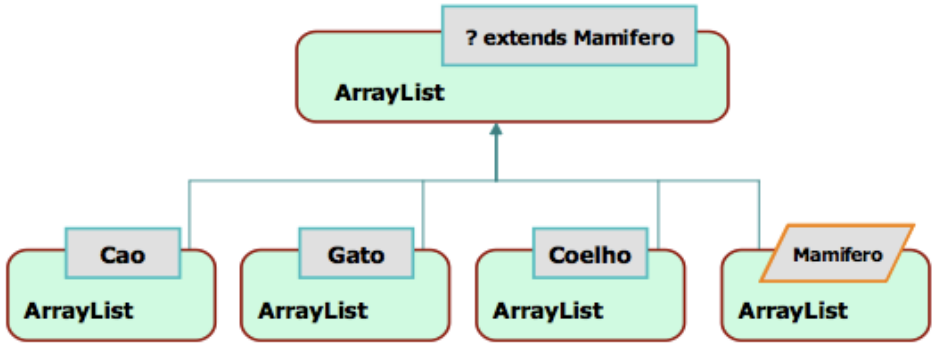
- no entanto esse arraylist não é supertipo dos arraylist de subtipos de mamíferos!!

- A hierarquia de ArrayList<E> não tem a mesma estruturação da hierarquia de E



- o arraylist de todos os arraylists de Mamifero e dos seus subtipos é o arraylist que se declara como:

- ***ArrayList<? extends Mamifero>***




```
Coleccao<? extends Mamifero> =
    Coleccao<Gato> ou
    Coleccao<Cao>  ou
    Coleccao<Coelho>
```

```
public void juntaMamif(Set<? extends Mamifero> cm) {
  ...
}
```

- Desta forma passa a ser possível ter declarações como:

```
ArrayList<Mamifero> mamifs = new ArrayList<Mamifero>();
mamifs.addAll(criaCaes());  // junta ArrayList<Cao>
mamifs.addAll(criaGatos()); // junta ArrayList<Gato>
```

  - o que era impossível no modelo anterior, na medida em que um ArrayList<Cao> não é compatível com ArrayList<Mamifero>