Ainda sobre classes

- *"The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are important differences between these two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essence" of an object, as it were.*
  - a definição de classe que temos vindo a utilizar está incompleta
    - quer as instâncias, quer as classes são **objectos**

- as classes não deixam de ser objectos
  - objectos que guardam o que é comum a todas as instâncias
  - **apenas um** objecto-classe por classe
- se objectos possuem estado e comportamento, então podemos extrapolar e dizer que a classe também tem:
  - variáveis e métodos de classe

- os métodos de classe são activados a partir de mensagens que são enviados para o objecto classe.
  - exemplo: Ponto2D.metodo()
- se uma classe possui variáveis de classe o acesso a essas variáveis deverá ser feito através dos métodos de classe
  - métodos de instância => v. instância
  - métodos de classe => v. classe

- o que é que se pode guardar como variável de classe?
  - valores que sejam comuns a todas os objectos instância
  - não faz sentido colocar estes valores em todos os objectos (repetição)

- Imagine-se que na classe Conta Bancária se pretende:

  a) saber quantas contas foram criadas

  b) saber qual é o somatório dos saldos das contas existentes

- se para b) podemos ter outras soluções (quais?), como é que poderemos satisfazer o requisito expresso em a)?

- uma variável que guarde o número de contas não é certamente uma variável de instância
    - actualização do contador em todas as instâncias?
    - redundância?
- teriam de se ter implementados mecanismos de comunicação entre todas as instâncias!!

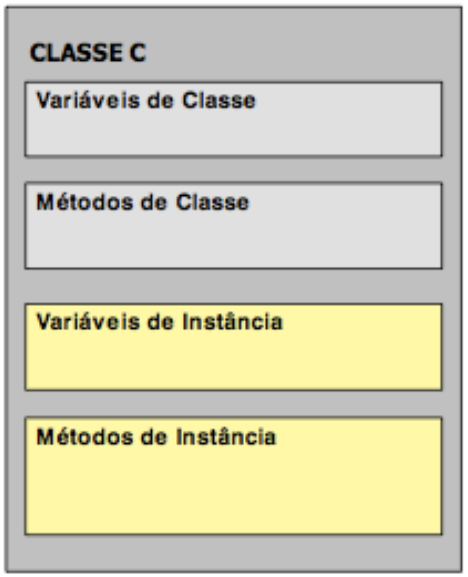
- as variáveis de classe servem para guardar *informação global* a todas as instâncias
- podem também ser utilizadas para guardar constantes que são utilizadas pelos diversos objectos instância
  - exemplo: Math.PI

- os métodos de classe fazem o acesso às variáveis de classe
- aos métodos de classe aplicam-se as mesmas regras de visibilidade que se aplicam aos métodos de instância
- os métodos de classe são sempre acessíveis às instâncias, mas métodos de classe **não** tem acesso aos métodos de instância

- como se declaram métodos e variáveis de classe?
    - utilizando o prefixo **static**
- a definição de classe passa a ter:
    - declaração de variáveis de classe
    - declaração de métodos de classe
    - declaração de varíaveis de instância
    - declaração de métodos de instância

Estrutura de uma classe

- estrutura tipo de uma classe



Classe PMMB

- seja uma classe PMMB (porta moedas multibanco)
  - queremos acrescentar informação sobre o número total de PMMB's criados e sobre o valor total de saldo existente em todos os cartões

- duas variáveis de classe e respectivos métodos de acesso

```
public class PMMB {
   // Variáveis de Classe
   public static int numPMMB = 0;
   public static double saldoTotal = 0.0;

   // Métodos de Classe
   public static int getNumPMMB() {
      return numPMMB;
   }
   public static double getSaldoTotal() {
      return saldoTotal;
   }
   public static void incNumPMMB() {
      numPMMB++;
   }
   public static void actSaldoTotal(double valor) {
      saldoTotal += valor;
   }
```

```
// Variáveis de Instância
private String codigo;
private String titular;
private double saldo;
private int numMovs;  // total de movimentos
```

- onde é que se actualiza a informação do número de cartões existentes?
  - no construtor de PMMB

```
public PMMB() {
   codigo = ""; titular = "";
   saldo = 0.0; numMovs = 0;
   this.actSaldoTotal(0);
   this.incNumPMMB();
}
```

  - no construtor, *this* representa a classe, sempre que o método invocado seja de classe.

```
// Métodos de Instância
. . . .
public void carregaPM(double valor) {
   saldo = saldo + valor;
   numMovs++; actSaldoTotal(valor);
}
// pré-condição
public boolean prePaga(double valor) {
   return saldo >= valor ;
}

public void pagamento(double valor) {
   saldo = saldo – valor;
   numMovs++; actSaldoTotal(-valor);
}
```

- para não confundir quem lê, as invocações anteriores poderiam ter sido feitas na forma:
  - PMMB.actSaldoValorTotal(valor)

- os métodos de classe anteriormente definidos devem ser utilizados como:

```
int pMoedas = PMMB.getNumPMMB();
double saldos = PMMB.getSaldoTotal();

PMMB.incNumPMMB();       //
PMMB.actSaldoTotal(valor);
```

- para que se consiga perceber o código é necessário que se siga a convenção que diz que as classes começam por letra maiúscula

Estruturas de Dados

- Como já vimos atrás, podemos criar conceitos mais complexos através do mecanismo de composição/agregação de objectos de classes mais simples:
  - a Turma a partir de Aluno
  - o Stand a partir de Veículo
  - etc.
- Mas para isso precisamos de ter colecções de objectos...

- Até ao momento apenas temos disponível a utilização de arrays

```
Aluno[] alunos = new Aluno[30];
Veiculo[] carros = new Veiculo[10];

for (int i=0; i<alunos.length && !encontrado; i++)
  if (alunos[i].getNota() == 20)
    encontrado = true;


for(Veiculo v: carros)
  System.out.println(v.toString());
```

- Esta é uma solução simples e testada, com a inconveniente de o tamanho da estrutura de dados ser estaticamente definido.

- Será que conseguimos ter uma estrutura de dados, baseada em arrays, que pudesse crescer de forma transparente para o utilizador?

  - Sim, bastando para tal criarmos uma classe com esse comportamento

- Seja essa classe chamada de GrowingArray e, por comodidade, vamos utilizar instâncias de Circulo

- Que operações necessitamos:

  - adicionar um circulo (no fim e numa posição)
  - remover um círculo
  - ver se um circulo existe
  - dar a posição de um circulo na estrutura
  - dar número de elementos existentes

- Documentação com os métodos necessários:

Constructor Summary

`GrowingArray()`
`GrowingArray(int capacidade)`

Method Summary

void	`add(Circulo c)` Adiciona o elemento passado como parâmetro ao fim do array
void	`add(int indice, Circulo c)` Método que adiciona um elemento numa determinada posição, forçando a que os elementos à direita no array façam shift.
boolean	`contains(Circulo c)` Método que determina se um elemento está no array.
Circulo	`get(int indice)` Devolve o elemento que está na posição indicada.
int	`indexOf(Circulo c)` Método que determina o índice do array onde está localizada a primeira ocorrência de um objecto.
boolean	`isEmpty()` Método que determina se o array contém elementos, ou se está vazio.
boolean	`remove(Circulo c)` Remove a primeira ocorrência do elemento que é passado como parâmetro.
Circulo	`remove(int indice)` Remove do array o elemento que está na posição indicada no parâmetro.
void	`set(int indice, Circulo c)` Método que actualiza o valor de uma determinada posição do array.
int	`size()` Método que determina o tamanho do array de elementos.

- Declarações iniciais:

```
public class GrowingArray {

  private Circulo[] elementos;
  private int size;

  /**
   * variável que determina o tamanho inicial do array,
   * se for utilizado o construtor vazio.
   */
  private static final int capacidade_inicial = 20;

  public GrowingArray(int capacidade) {
    this.elementos = new Circulo[capacidade];
    this.size = 0;
  }

  public GrowingArray() {
    this(capacidade_inicial);
  }
```

- get de um elemento da estrutura de dados

```
/**
 * Devolve o elemento que está na posição indicada.
 *
 * @param  indice posição do elemento a devolver
 * @return o objecto que está na posição indicada no parâmetro
 * (deveremos ter atenção às situações em que a posição não existe)
 */

public Circulo get(int indice) {
   if (indice <= this.size)
       return this.elementos[indice];
   else
      return null; // ATENÇÃO!

}
```

- set de uma posição da estrutura

```
/**
 * Método que actualiza o valor de uma determinada posição do array.
 *
 * @param indice a posição que se pretende actualizar
 * @param c o circulo que se pretende colocar na estrutura de dados
 *
 */
public void set(int indice, Circulo c) {
  if (indice <= this.size) //não se permitem "espaços vazios"
    this.elementos[indice] = c;
}
```

- adicionar um elemento à estrutura de dados

```
/**
 * Adiciona o elemento passado como parâmetro ao fim do array
 *
 * @param c circulo que é adicionado ao array
 *
 */

public void add(Circulo c) {
  aumentaCapacidade(this.size + 1);
  this.elementos[this.size++] = c;
}
```

- método auxiliar que aumenta espaço

```
/**
 * Método auxiliar que verifica se o array alocado tem capacidade
 * para guardar mais elementos.
 * Por cada nova inserção, verifica se estamos a mais de metade
 * do espaço
 * alocado e,caso se verifique, aloca mais 1.5 de capacidade.
 *
 */

private void aumentaCapacidade(int capacidade) {
   if (capacidade > 0.5 * this.elementos.length) {
      int nova_capacidade = (int)(this.elementos.length * 1.5);
      this.elementos = Arrays.copyOf(this.elementos,nova_capacidade);
   }
}
```

```java
/**
 * Método que adiciona um elemento numa determinada posição,
 * forçando a
 * que os elementos à direita no array façam shift.
 * Tal como no método de set não são permitidos espaços.
 *
 * @param indice indice onde se insere o elemento
 * @param c circulo que será inserido no array
 *
 */

public void add(int indice, Circulo c) {
  if (indice <= this.size) {
    aumentaCapacidade(this.size+1);
    System.arraycopy(this.elementos, indice, this.elementos,
                     indice + 1, this.size - indice);
    this.elementos[indice] = c;
    this.size++;
  }
}
```

```java
/**
 * Remove do array o elemento que está na posição indicada no parâmetro.
 * Todos os elementos à direita do índice sofrem um deslocamento
 * para a esquerda.
 * @param indice índice do elemento a ser removido
 * @return o elemento que é removido do array. No caso do índice não
 * existir devolver-se-á null.
 */
public Circulo remove(int indice) {
    if (indice <= this.size) {
      Circulo c = this.elementos[indice];

      int deslocamento = this.size - indice - 1;
      if (deslocamento > 0)
        System.arraycopy(this.elementos, indice+1, this.elementos,
                         indice, deslocamento);
      this.elementos[--this.size] = null;
      return c;
    }
    else
      return null;
}
```

```java
 * Remove a primeira ocorrência do elemento que é passado como parâmetro.
 * Devolve true caso o array contenha o elemento, falso caso contrário.
 *
 * @param c círculo a ser removido do array (caso exista)
 * @return true, caso o círculo exista no array
*/
public boolean remove(Circulo c) {
   if (c != null) {
      boolean encontrado = false;
      for (int indice = 0; indice < this.size && !encontrado; indice++)
              if (c.equals(this.elementos[indice])) {
                  encontrado = true;
                  int deslocamento = this.size - indice - 1;
                  if (deslocamento > 0)
                    System.arraycopy(this.elementos, indice+1,
                        this.elementos, indice, deslocamento);
                  this.elementos[--this.size] = null;
              }
      return encontrado;
   }
   else
     return false;
}
```

```java
/**
 * Método que determina o índice do array onde está localizada a
 * primeira ocorrência de um objecto.
 *
 * @param c círculo de que se pretende determinar a posição
 * @return a posição onde o círculo se encontra. -1 caso não esteja no
 * array ou o círculo passado como parâmetro seja null.
 */

public int indexOf(Circulo c) {
   int posicao = -1;
   if (c != null) {
          boolean encontrado = false;
          for (int i = 0; i < this.size && !encontrado; i++)
              if (c.equals(this.elementos[i])) {
                  encontrado = true;
                  posicao = i;
              }
   }
   return posicao;
}
```

```java
/**
 * Método que determina se um elemento está no array.
 *
 * @param c círculo a determinar se está no array
 * @return true se o objecto estiver inserido na estrutura de dados,
 * false caso contrário.
 */
public boolean contains(Circulo c) {
  return indexOf(c) >= 0;
}
```

```java
/**
 * Método que determina se o array contém elementos, ou se está vazio.
 *
 * @return true se o array estiver vazio, false caso contrário.
 */

public boolean isEmpty() {
  return this.size == 0;
}
```

```java
public class TesteGA {
  public static void main(String[] args) {

    Circulo c1 = new Circulo(2,4,4.5);
    Circulo c2 = new Circulo(1,4,1.5);
    Circulo c3 = new Circulo(2,7,2.0);
    Circulo c4 = new Circulo(3,3,2.0);
    Circulo c5 = new Circulo(2,6,7.5);

    GrowingArray ga = new GrowingArray(10);
    ga.add(c1.clone());
    ga.add(c2.clone());
    ga.add(c3.clone());

    System.out.println("Num elementos = " + ga.size());
    System.out.println("Posição do c2 = " + ga.indexOf(c2));

  }
}
```

Colecções Java

- O Java oferece um conjunto de classes que implementam as estruturas de dados mais utilizadas
  - oferecem uma API consistente entre si
  - permitem que sejam utilizadas com qualquer tipo de objecto - são parametrizadas por tipo

- Poderemos representar:
  - ArrayList<Aluno> alunos
  - HashSet<Aluno> alunos;
  - HashMap<String, Aluno> turmaAlunos;
  - TreeMap<String, Docente> docentes;
  - Stack<Pedido> pedidosTransferência;

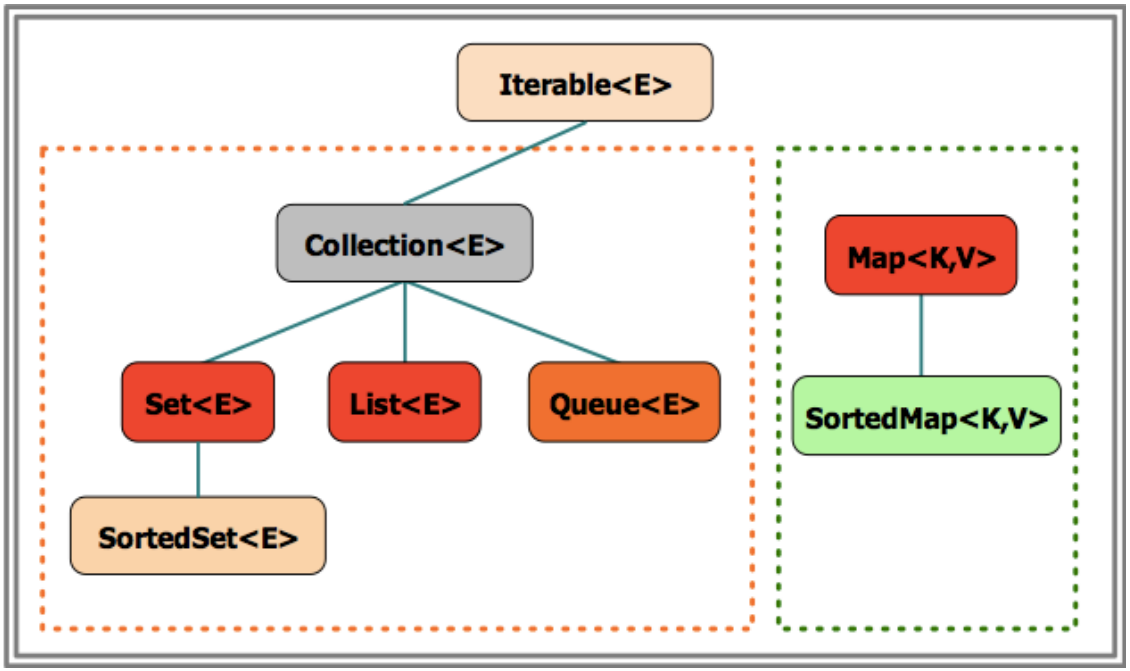
- Ao fazer-se ArrayList<Circulo> passa a ser o compilador a testar, e validar, que só são utilizados objectos do tipo Circulo no arraylist.
    - isto dá uma segurança adicional aos programas, pois em tempo de execução não teremos erros de compatibilidade de tipos
    - os tipos de dados são verificados em tempo de compilação

JCF

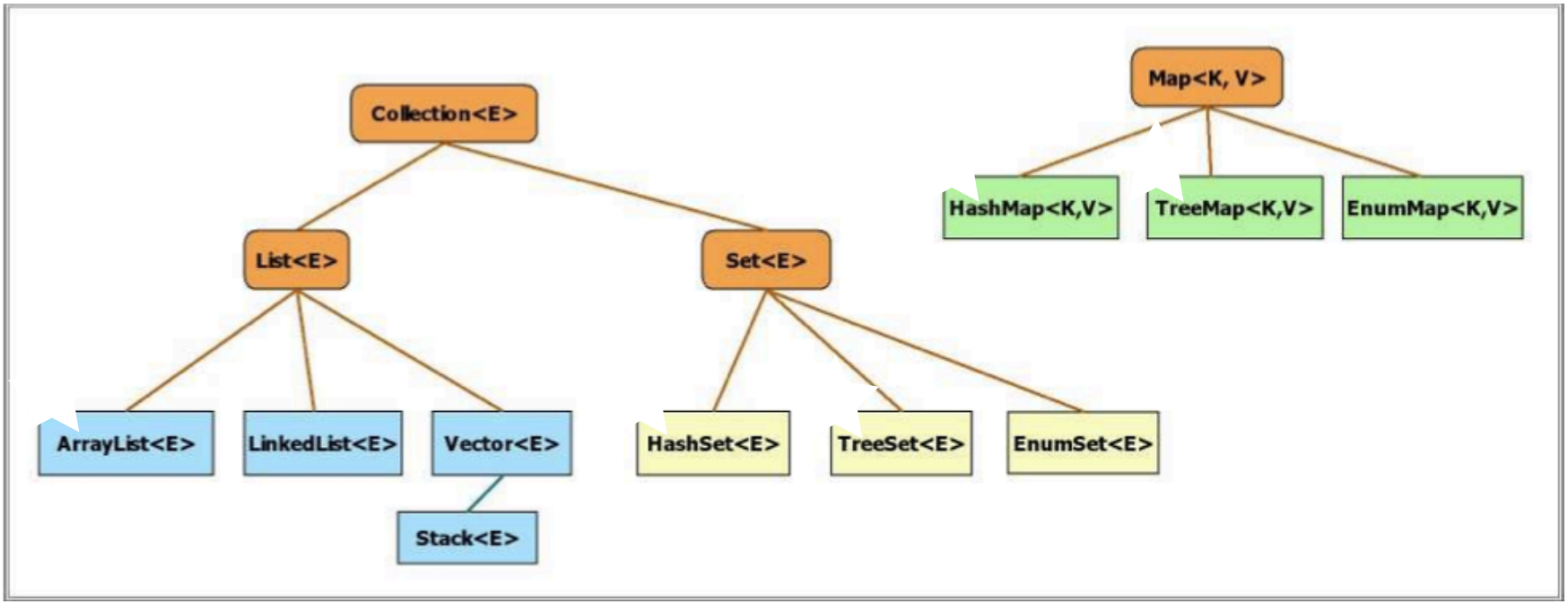
- O JCF (Java Collections Framework) agrupa as várias classes genéricas que correspondem às implementações de referência de:
  - Listas: API de **List<E>**
  - Conjuntos: API de **Set<E>**
  - Correspondências unívocas: API de **Map<K,V>**

Estrutura da JCF

- Existe uma arrumação por API (interfaces)



- Para cada API (interface) existem diversas implementações (a escolher consoante critérios do programador)



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;
}
```

Colecções Java

- Tipos de colecções disponíveis:
  - listas (definição em List<E>)
  - conjuntos (definição em Set<E>)
  - queues (definição em Queue<E>)
- noção de família (muito evidente) nas APIs de cada um destes tipos de colecções.

List<E>

- Utilizar sempre que precise de manter ordem
- O método add não testa se o objecto existe na colecção
- O método contains verifica sempre o resultado de equals
- Implementação utilizada: **ArrayList<E>**

ArrayList<E>

Categoria de Métodos	API de ArrayList<E>

Construtores	`new ArrayList<E>()`
	`new ArrayList<E>(int dim)`
	`new ArrayList<E>(`*`Collection`*`)`

Inserção de elementos	`add(E o); add(int index, E o);` `addAll(`*`Collection`*`); addAll(int i, `*`Collection`*`);`
Remoção de elementos	`remove(Object o); remove(int index);` `removeAll(`*`Collection`*`); retainAll(`*`Collection`*`)`

Consulta e comparação de conteúdos	`E get(int index); int indexOf(Object o);`
	`int lastIndexOf(Object o);`
	`boolean contains(Object o); boolean isEmpty();`
	`boolean containsAll(`*`Collection`*`); int size();`

Criação de Iteradores	`Iterator<E> iterator();`
	`ListIterator<E> listIterator();`
	`ListIterator<E> listIterator(int index);`

Modificação	`set(int index, E elem); clear();`
Subgrupo	`List<E> sublist(int de, int ate);`
Conversão	`Object[] toArray();`
Outros	`boolean equals(Object o); boolean isEmpty();`

```java
import java.util.ArrayList;
public class TesteArrayList {
  public static void main(String[] args) {

    Circulo c1 = new Circulo(2,4,4.5);
    Circulo c2 = new Circulo(1,4,1.5);
    Circulo c3 = new Circulo(2,7,2.0);
    Circulo c4 = new Circulo(3,3,2.0);
    Circulo c5 = new Circulo(2,6,7.5);

    ArrayList<Circulo> circs = new ArrayList<Circulo>();
    circs.add(c1.clone());
    circs.add(c2.clone());
    circs.add(c3.clone());

    System.out.println("Num elementos = " + circs.size());
    System.out.println("Posição do c2 = " + circs.indexOf(c2));

    for(Circulo c: circs)
      System.out.println(c.toString());

  }
}
```

Set<E>

- Utilizar sempre que se quer garantir ausência de elementos repetidos
- O método add testa se o objecto existe
- O método contains utiliza a lógica do equals, mas não só...
- Duas implementações: **HashSet<E>** e **TreeSet<E>**

Set<E>

Categoria de Métodos	API de Set<E>
Inserção de elementos	`add(E o);` `addAll(Collection); addAll(int i, Collection);`
Remoção de elementos	`remove(Object o); remove(int index);` `removeAll(Collection); retainAll(Collection)`
Consulta e comparação de conteúdos	`boolean contains(Object o); boolean isEmpty();` `boolean containsAll(Collection); int size();`
Criação de Iteradores	`Iterator<E> iterator();`
Modificação	`clear();`
Subgrupo	`List<E> sublist(int de, int ate);`
Conversão	`Object[] toArray();`
Outros	`boolean equals(Object o); boolean isEmpty();`

HashSet<E>

- O método add calcula o valor do hash de E para determinar a posição do objecto na estrutura de dados
- O método contains necessita de saber o hash do objecto para determinar a posição em que o encontra
- Logo, não chega ter o equals definido
  - é necessário ter o método **hashcode()**

TreeSet<E>

- O método add determina a posição na árvore em que deve colocar o objecto
- É necessário fornecer um método de comparação dos objectos - **compare()** ou **compareTo()**
  - sem este método de comparação não é possível utilizar o TreeSet, a não ser para tipos de dados simples (String, Integer, etc.)

Map<K,V>

- Quando se pretende ter uma associação de um objecto chave a um objecto valor
- Na dimensão das chaves não existem elementos repetidos (é um conjunto!)
- Duas implementações disponíveis: HashMap<K,V> e TreeMap<K,V>
  - aplicam-se à dimensão das chaves as considerações anteriores sobre conjuntos

Percorrer uma colecção

- Podemos utilizar o foreach para percorrer a colecção toda, mas...
  - podemos querer parar antes do fim
  - podemos não ter o acesso à posição do elemento na colecção (caso dos conjuntos)
  - logo, é necessário um mecanismo comum para percorrer colecções

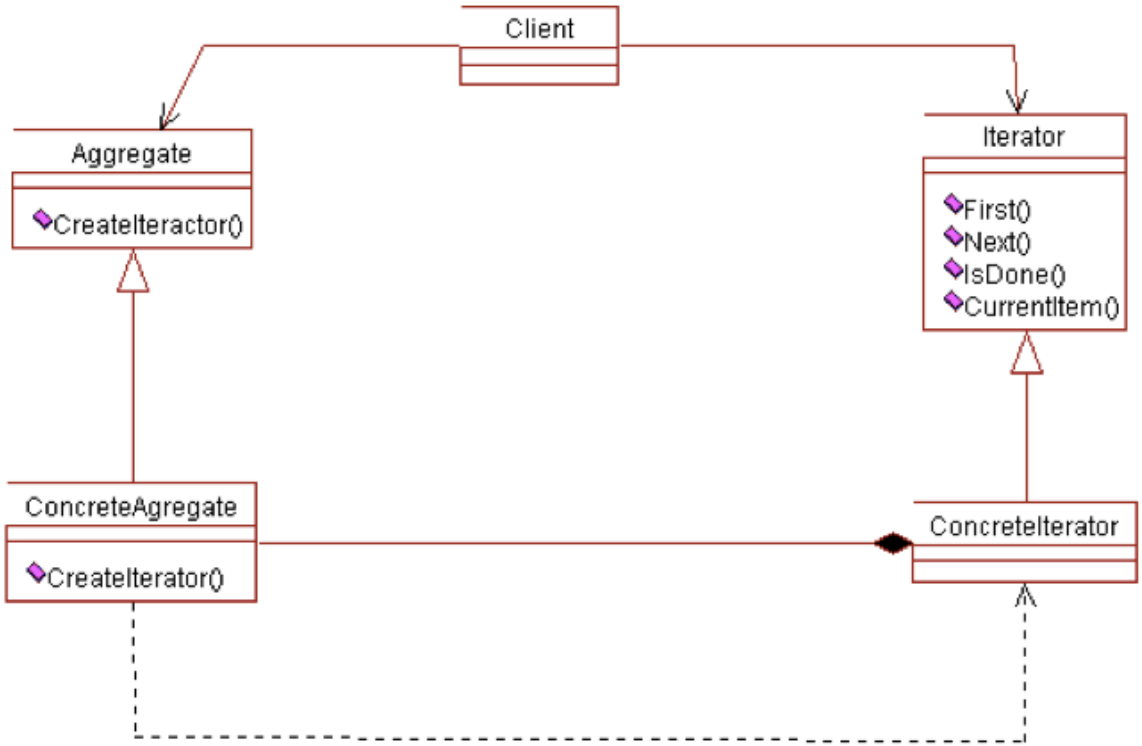
Iterator

- O Iterator é um padrão de concepção identificado e que permite providenciar uma forma de aceder aos elementos de uma colecção de objectos, sem que seja necessário saber qual a sua representação interna
  - basta para tal, que todas as colecções saibam criar um iterator

- Um iterador de uma lista poderia ser:

- o iterator precisa de ter mecanismos para:
- aceder ao objecto apontado
- avançar
- determinar se chegou ao fim

- A estrutura geral do iterator será:



OUTROS ITERADORES DE COLECÇÕES

```
Iterator<E> iterator();  // método que cria um Iterador sobre a colecção
                         // que recebeu a mensagem. Se a colecção tem
                         // objectos de tipo E o Iterador é de tipo E
```

- UM `Iterator<E>` É UMA ESTRUTURA COMPUTACIONAL QUE IMPLEMENTA UM ITERADOR SOBRE TODOS OS ELEMENTOS DA COLECÇÃO ORIGEM (SEM ORDEM PREDEFINIDA), USANDO OS MÉTODOS `hasNext()`, `next()` E `remove()`.

EXEMPLO 1: USO DE `iterator()` COM `while() { ... }`

```
Iterator<E> it = colecção.iterator();
E elem; // tipo dos objectos que estão na colecção
while(it.hasNext()) {
   elem = it.next();
     // fazer qq. coisa com elem
}
```

EQUIVALENTE A:

```
for(E elem : colecção) { … }
```

NOTA: Mas *foreach* não permite parar procuras!!

Assim, em algoritmos típicos de procura tem que se usar a construção baseada em `iterator()`, cf.

```
Iterator<E> it = colecção.iterator();   // criar o Iterator<E>
E elem ;  // variável do tipo dos objectos da colecção
boolean encontrado = false;
while(it.hasNext() && !encontrado) {
    elem = it.next();
    if(elem possui certa propriedade) encontrado = true;
}
```

Exemplo:

```
// encontrar o 1º ponto com coordenada X igual a Y
Ponto2D pontoCopia = null;
Iterator<Ponto2D> it = linha.iterator();   // criar o Iterator<E>
Ponto2D ponto ;  // variável do tipo dos objectos da colecção
boolean encontrado = false;
while(it.hasNext() && !encontrado) {
    ponto = it.next();
    if(ponto.getX() == ponto.getY()) encontrado = true;
}
if(encontrado) pontoCopia = ponto.clone();
```

Map<K,V>

Categoria de Métodos	API de Map<K,V>
Inserção de elementos	`put(K k, V v);` `putAll(Map<? extends K, ? extends V> m);`
Remoção de elementos	`remove(Object k);`

Consulta e comparação de conteúdos	`V get(Object k);`
	`boolean containsKey(Object k);`
	`boolean isEmpty();`
	`boolean containsValue(Object v); int size();`

Criação de Iteradores	`Set<K> keySet();`
	`Collection<V> values();`
	`Set<Map.Entry<K, V>> entrySet();`

Outros	`boolean equals(Object o); Object clone()`

TreeMap<K, V> métodos adicionais
`TreeMap<K, V>()`
`TreeMap<K, V>(Comparator<? super K> c)`
`TreeMap<K, V>(Map<? extends K, ? extends V> m)`
`K firstKey()`
`SortedMap<K, V> headMap(K toKey)`
`K lastKey()`
`SortedMap<K, V> subMap(K fromKey, K toKey)`
`SortedMap<K, V> tailMap(K fromKey)`

Regras para utilização de colecções

- Escolher com critério se a colecção a criar deve ser uma lista ou um conjunto (duplicados ou não) ou então uma correspondência entre chaves e valores
- Escolher para sets e maps uma classe de implementação adequada, cf. Hash (sem ordem especial) ou Tree (com comparação pré-definida ou definindo uma ordem de comparação)

Regras para utilização de colecções

- Nunca usar os métodos pré-definidos **addAll()** ou **putAll()**. Em vez destes, usar antes o iterador for e fazer clone() dos objectos a copiar para a colecção cópia
- Sempre que possível, os resultados dos métodos devem ser generalizados para os tipos List<E>, Set<E> ou Map<K,V> em vez de devolverem classes específicas como ArrayList<E>, HashSet<E> ou TreeSet<E> ou HashMap<K,V>.
  - aumentando-se assim a abstracção

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

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

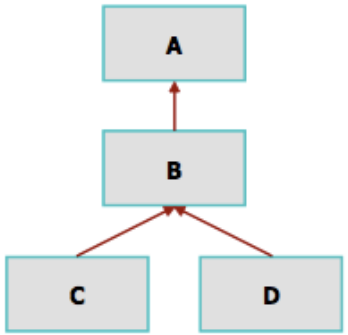
- ou seja, para ter um Ponto3D precisamos de tudo o que existe em Ponto2D e acrescentar um *delta* que consiste nas características novas

- 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

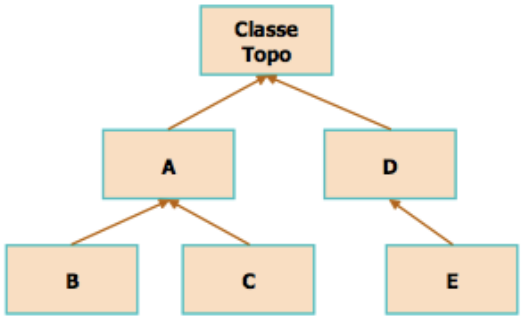
```java
/**
 * Ponto3D através de composição.
 */

public class Ponto3D {
  private Ponto2D p;
  private int z;

  public Ponto3D() {
    this.p = new Ponto2D();
    this.z = 0;
  }

  public Ponto3D(int x, int y, int z) {
    this.p = new Ponto2D(x,y);
    this.z = z;
  }

  //...
  public int getX() {return this.p.getX();}
  public int getY() {return this.p.getY();}
  public int getZ() {return this.z;}
```

- 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:

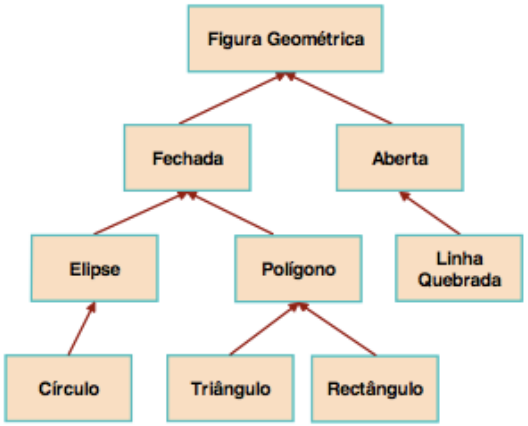


  - hierarquia de herança simples (por oposição, p.ex., a C++)

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

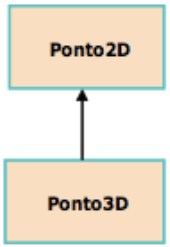
- no paradigma dos objectos a hierarquia de classes é uma hierarquia de especialização
  - uma subclasse de uma dada classe constitui uma especialização, sendo por definição mais detalhada que a classe que lhe deu origem
  - isto é, possui **mais** estado e **mais** comportamento

- 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 se 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

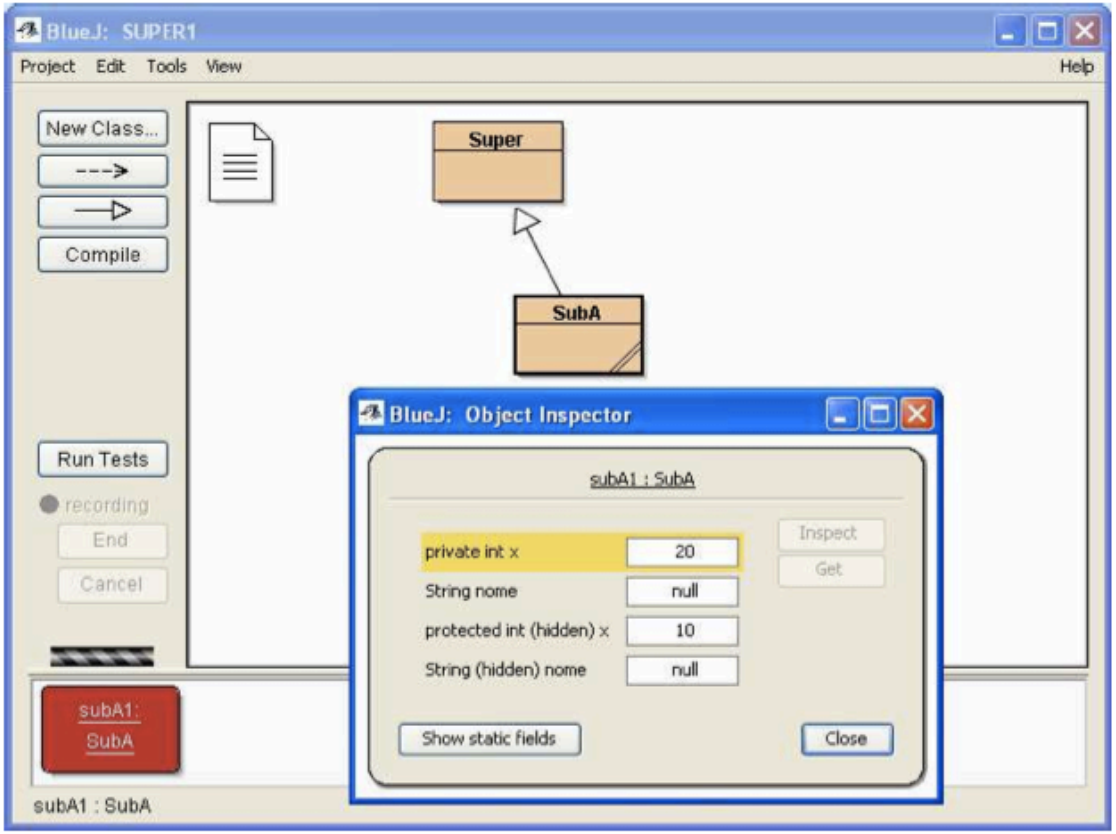
```
public class Super {
   protected int x = 10;
   String nome;
   // Métodos
   public int getX() { return x; }
   public String classe() { return "Super"; }
   public int teste() { return this.getX(); }
}
```

- e uma subclasse SubA

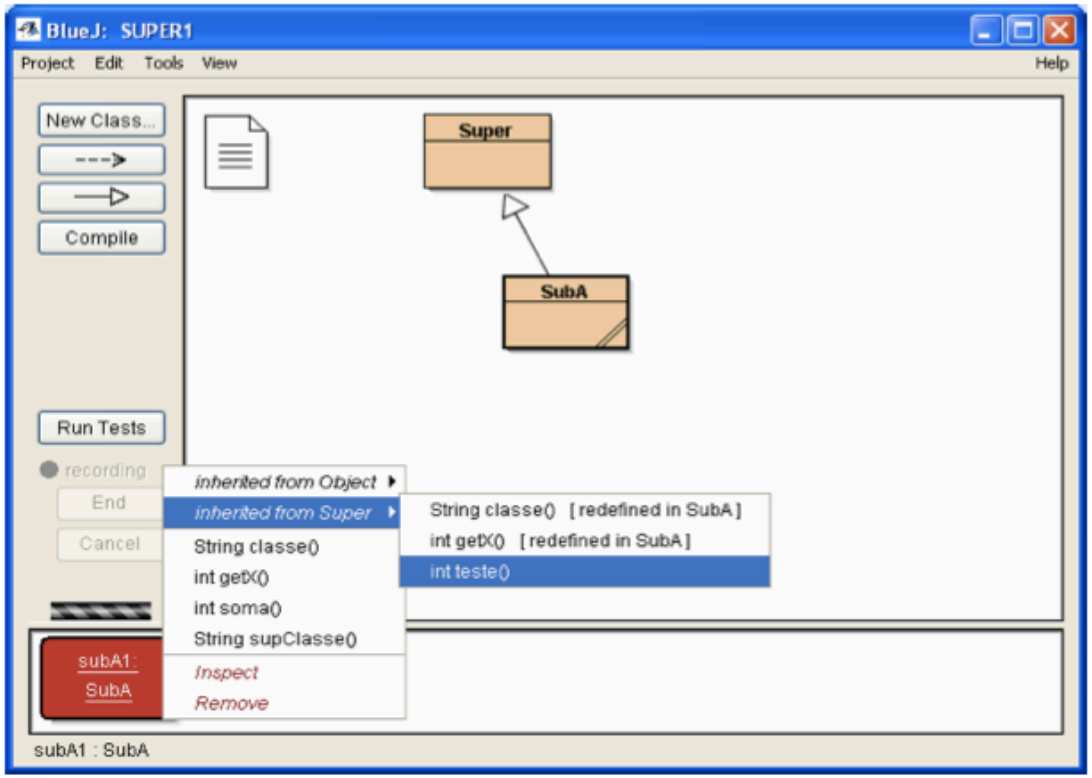
```
public class SubA extends Super {
   private int x = 20; // "shadow"
   String nome;        // "shadow"
   // Métodos
   public int getX() { return x; }
   public String classe() { return "SubA"; }
   public String supClass() { return super.classe(); }
   public int soma() { return x + super.x; }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

- 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



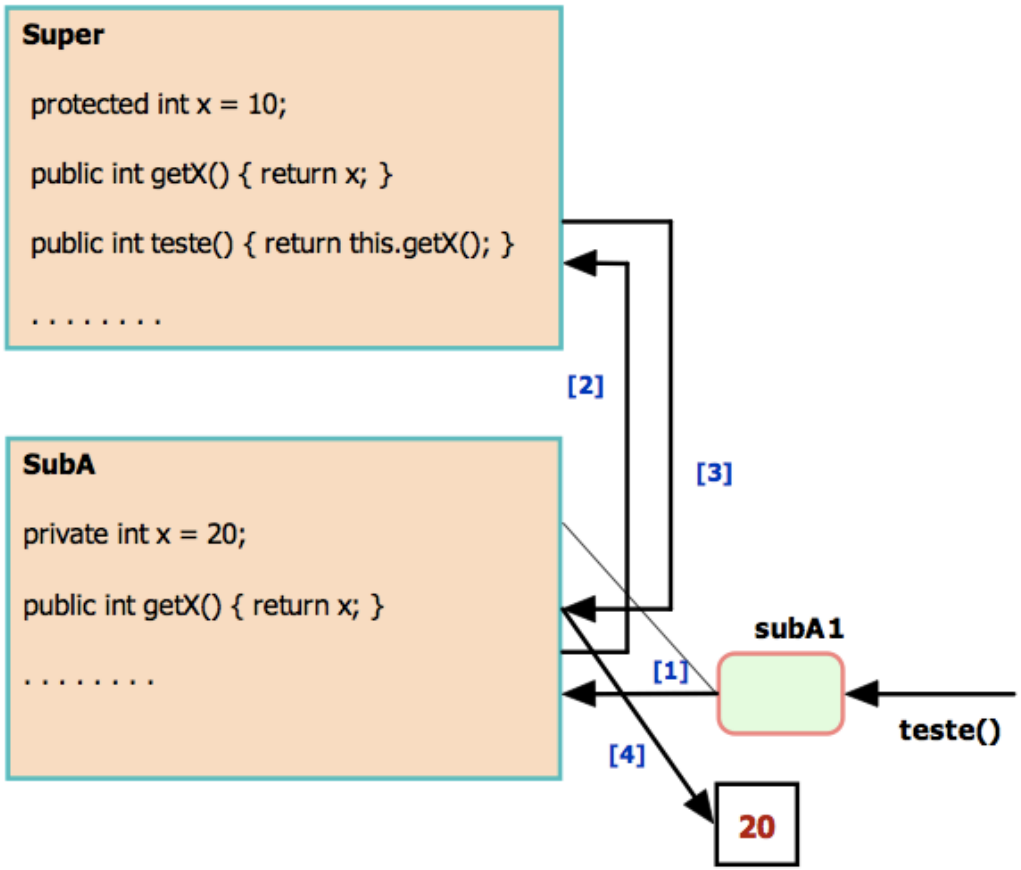
- no BlueJ é possível ver os métodos definidos na classe e os herdados



- 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!