

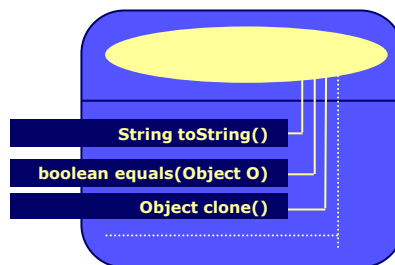
PG II

Programação Orientada aos Objectos em Java

Clonagem:

- Classe Object
- Object.clone
- CloneNotSupportedException
- Marker Interface Cloneable
- Conclusões

No capítulo “**Definição de Classes**” foi explicado que segundo as boas regras de programação em Java, na implementação de uma Classe deveriam ser sempre definidos os métodos `toString`, `equals` e `clone`.



De seguida foi explicado porquê:

Resposta - Uma vez que:

- estes métodos são nativos da Classe Object;
- todas as classes em Java derivam “inerentemente” da classe Object,

então a implementação dos métodos `toString`, `equals` e `clone` corresponde, **não a uma definição**, mas sim, mais correctamente, a uma **redefinição** dos mesmos (**overriding**).

Esta redefinição deve ser feita porque na maioria das situações a implementação existente ao nível destes métodos e que é herdada por todas as classes, pode não corresponder à funcionalidade desejada.

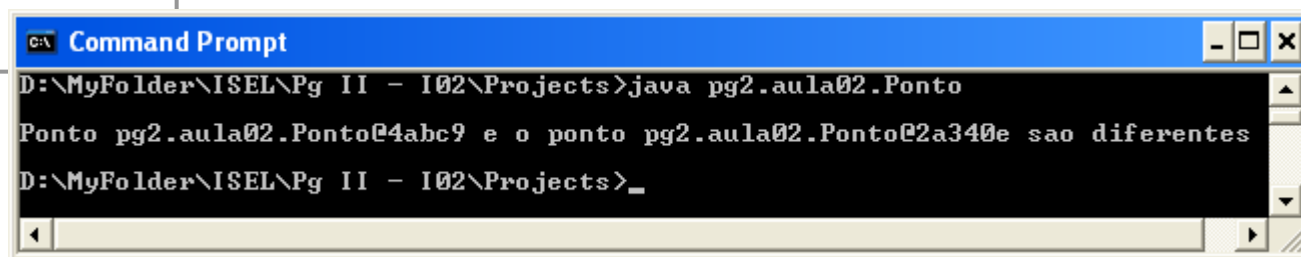
Vejamos mais detalhadamente, no exemplo da classe Ponto...

... Classe Object

Se não tivessem sido redefinidos os métodos `toString` e `equals` na classe `Ponto` o resultado do seguinte programa seria:

```
public static void main (String [] args){
    Ponto p1 = new Ponto(3,7);
    Ponto p2 = new Ponto(p1.getX(),p1.getY());

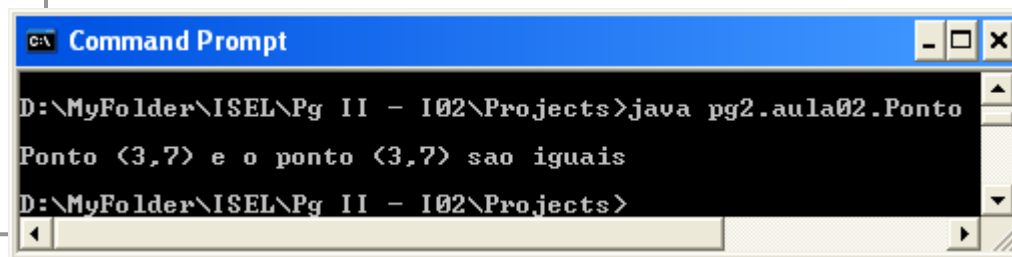
    IO.cout.writeln("Ponto " + p1 + " e o ponto " + p2 +
        " são " + (p1.equals(p2)? "iguais":"diferentes"));
}
```



```
C:\> Command Prompt
D:\MyFolder\ISEL\Pg II - I02\Projects>java pg2.aula02.Ponto
Ponto pg2.aula02.Ponto@4abc9 e o ponto pg2.aula02.Ponto@2a340e sao diferentes
D:\MyFolder\ISEL\Pg II - I02\Projects>
```

Depois de redefinidos os métodos o resultado passou a ser:

```
public String toString () {
    return new String ("(" + this.getX() + "," +
        this.getY() + ")");
}
public boolean equals (Object obj) {
    if ((obj != null) && (obj instanceof Ponto))
        return this.getX() == ((Ponto) obj).getX() &&
            this.getY() == ((Ponto) obj).getY();
    return false;
}
```



```
C:\> Command Prompt
D:\MyFolder\ISEL\Pg II - I02\Projects>java pg2.aula02.Ponto
Ponto (3,7) e o ponto (3,7) sao iguais
D:\MyFolder\ISEL\Pg II - I02\Projects>
```

Object.clone

E no caso do método `clone`, será que a sua implementação original não serviria os interesses de qualquer Classe ????

A funcionalidade nativa do método `clone` na classe `Object`, tal como indicada na especificação da sua API é:

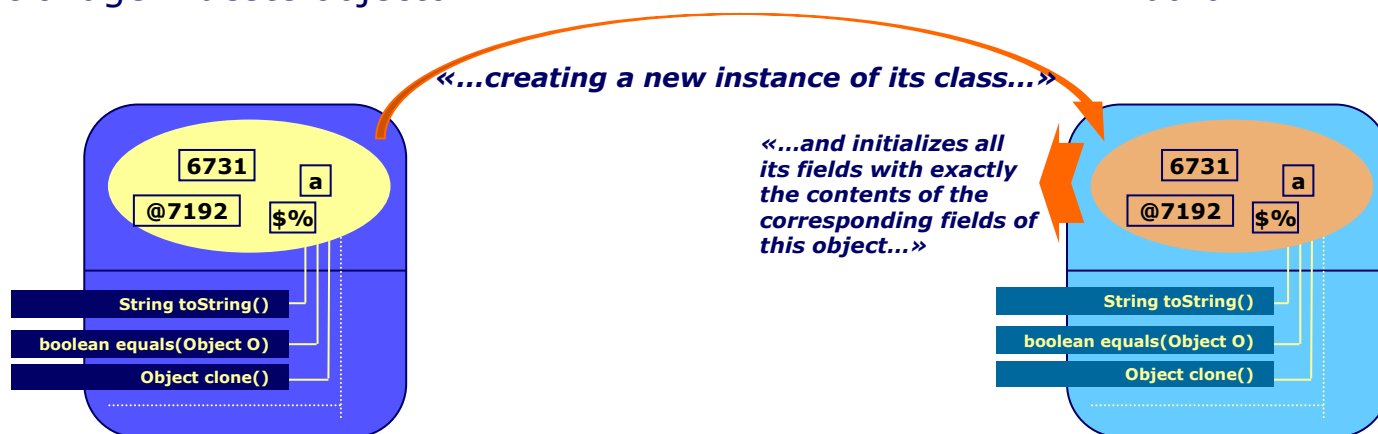
«Copying an object will typically entail creating a new instance of its class, but it also may require copying of internal data structures as well. ...»

The method clone for class Object performs a specific cloning operation. ...

Otherwise, this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment;»

Ou seja, a clonagem deste objecto:

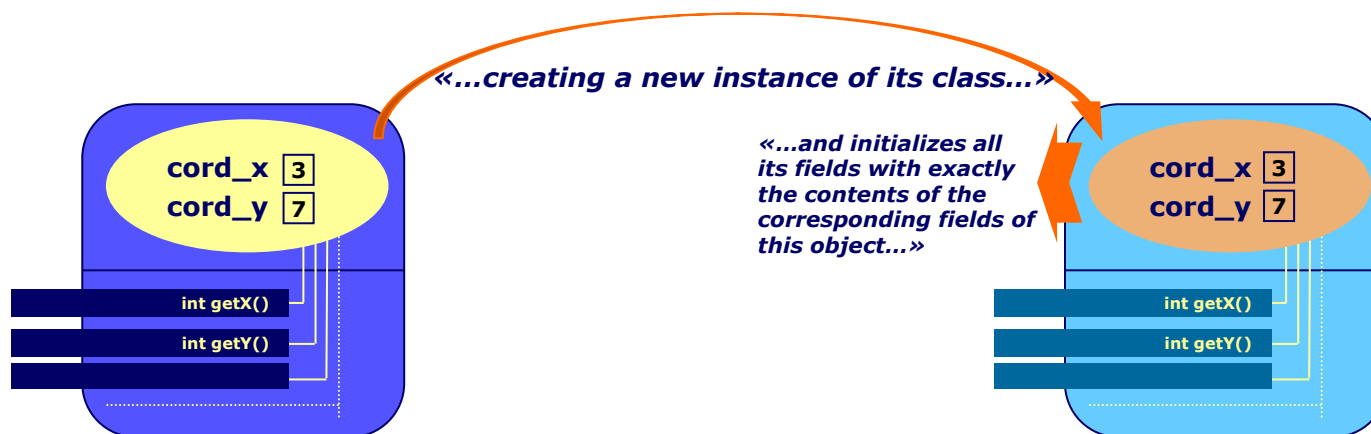
dará:



... Object.clone

No caso da Classe Ponto esta forma de implementar a clonagem serviria?

Aplicando o mesmo processo de clonagem que anteriormente, teríamos:



Então neste caso não seria necessário redefinir o método `clone`.

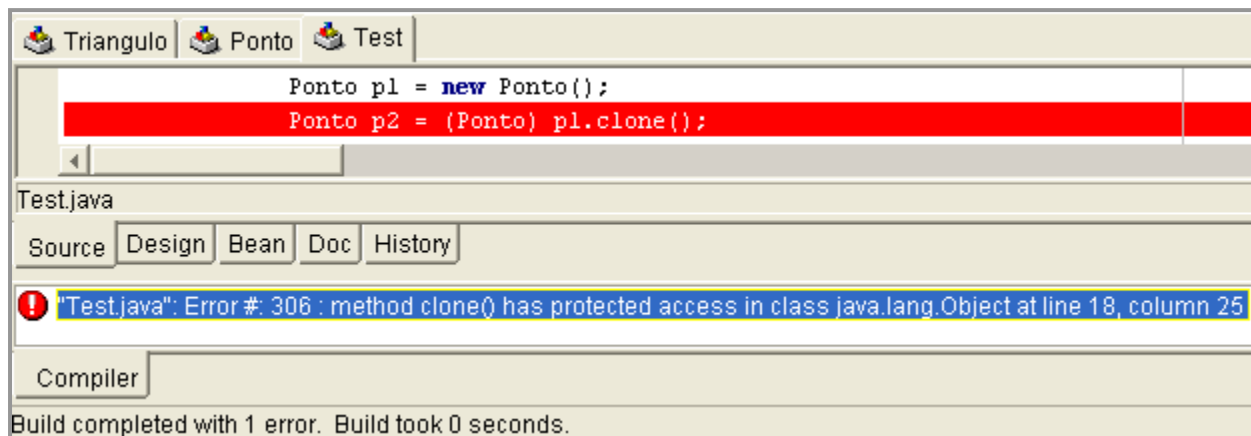
Contudo existe uma restrição que nos obriga a fazer essa redefinição. Analisando com detalhe a API da classe `Object` verificamos o seguinte:

Method Summary	
protected <u>Object</u>	<u>clone</u> () Creates and returns a copy of this object.
boolean	<u>equals</u> (<u>Object</u> obj) Indicates whether some other object is "equal to" this one.
.....	

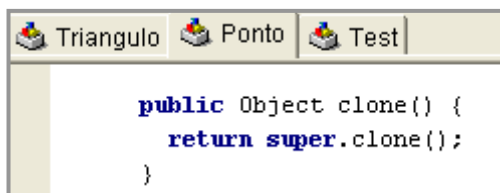
... Object.clone

Ou seja, sendo o método declarado como *protected* então não lhe é dado acesso do exterior da classe Ponto.

Isto é, se tentarmos aceder externamente ao método `clone` duma instância da Classe Ponto obtemos os seguinte erro:



Então, para que as instâncias da classe Ponto, tenham na sua interface a implementação nativa do método `clone` disponível a qualquer outro objecto, é necessário fazer a seguinte redefinição:



E basta fazer só isto?

CloneNotSupportedException

Analizando ainda com maior detalhe a API da classe Object verificamos que:

clone

```
protected Object clone()
                throws CloneNotSupportedException
```

Creates and returns a copy of this object. The precise meaning of "copy" may vary from implementation to implementation.

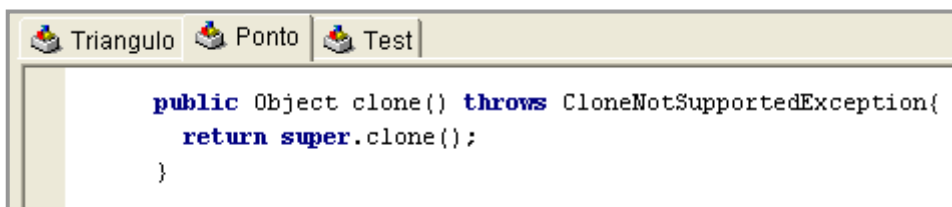
E por sua vez:

java.lang

Class CloneNotSupportedException

```
java.lang.Object
|
+-- java.lang.Throwable
    |
    +-- java.lang.Exception
        |
        +-- java.lang.CloneNotSupportedException
```

Então, uma das formas de redefinir correctamente o método `clone` seria:



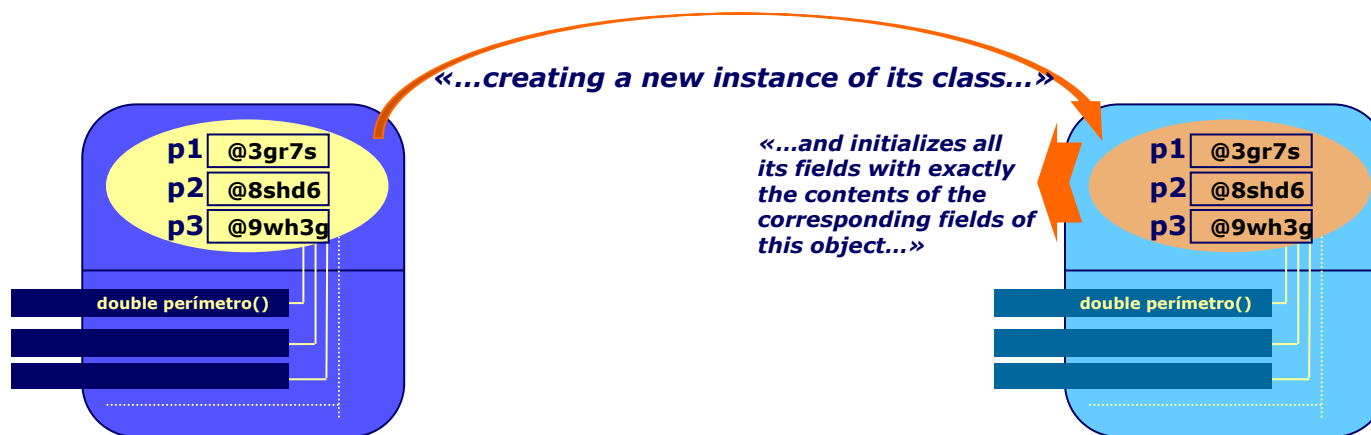
```
public Object clone() throws CloneNotSupportedException{
    return super.clone();
}
```

... CloneNotSupportedException

- Será que neste momento o método `clone` da classe `Ponto`, está disponível para ser usado por qualquer outro objecto?
- Afinal em que situação é lançada a excepção `CloneNotSupportedException`?

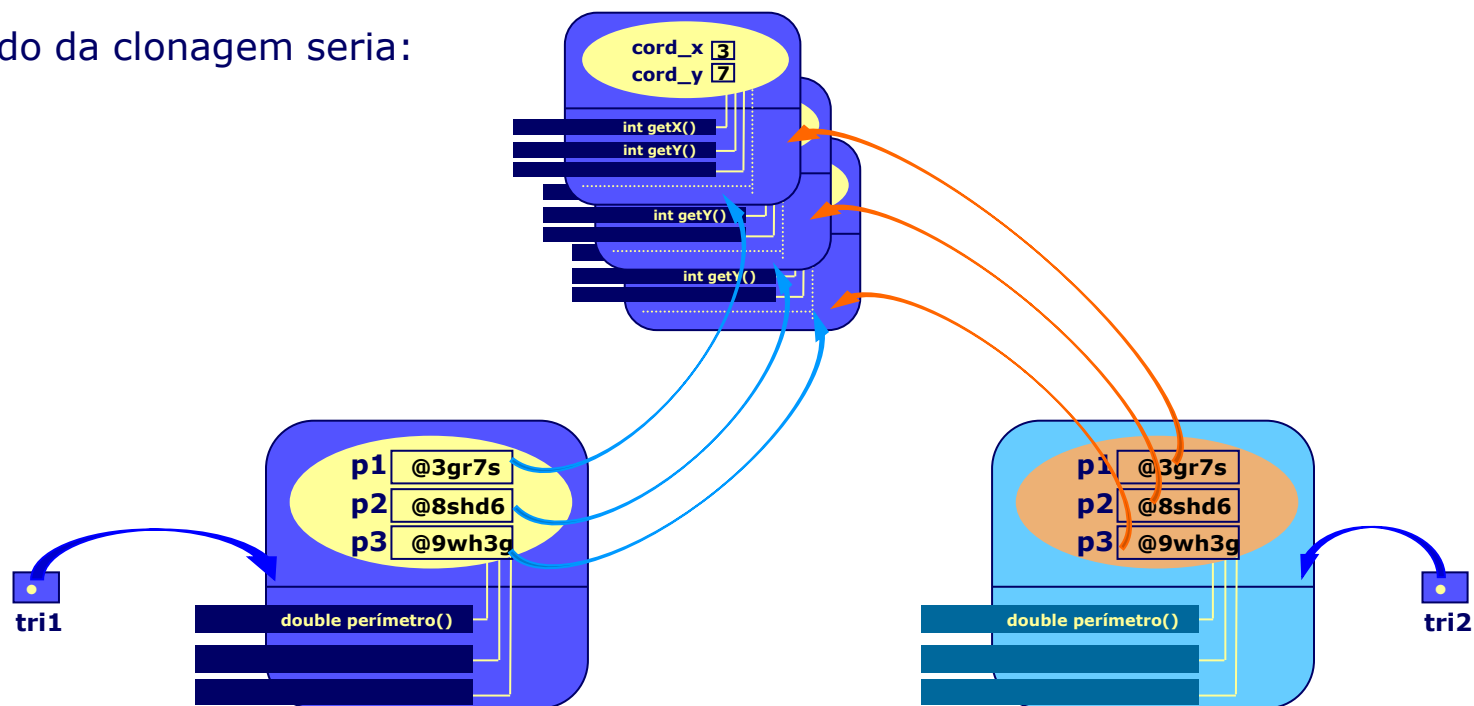
Antes de analisarmos a especificação desta excepção, vamos ver se a implementação nativa do método `clone` serve os objectivos de clonagem de **todas** as classe em Java, tal como serve para a classe **Ponto**

Façamos a mesma análise do processo de clonagem nativo, mas agora para a classe `Triangulo`.



... CloneNotSupportedException

Ou seja, o resultado da clonagem seria:



Ou seja, as alterações sobre "tri1" actuam directamente sobre "tri2":

```
Triangulo tri2, tri1 = new Triangulo (2,3,2,5,4,5);
tri2 = (Triangulo) tri1.clone();
tri2.pl.setX(0); tri2.pl.setY(0);
tri2.p2.setX(1); tri2.p2.setY(1);
tri2.p3.setX(0); tri2.p3.setY(2);
IO.cout.writeln("\nTriangulo " + tri1 + " e o Triangulo " + tri2 +
    " sao " + (tri1.equals(tri2)? "iguais":"diferentes"));
```

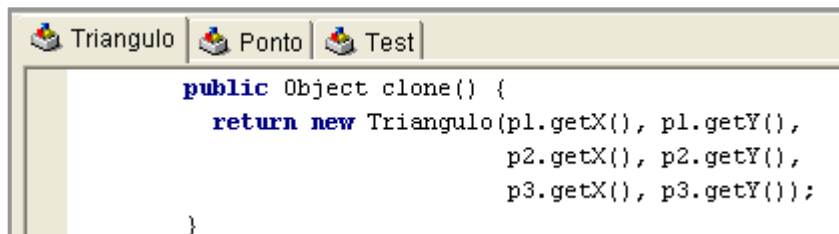
```
Command Prompt
D:\MyFolder\ISEL\Pg II - I02\Projects>java pg2.aula02.Triangulo
Triangulo [(0,0)<(1,1)<(0,2)] e o Triangulo [(0,0)<(1,1)<(0,2)] sao iguais
D:\MyFolder\ISEL\Pg II - I02\Projects>
```

... CloneNotSupportedException

Este comportamento é descrito na especificação do método `clone` nativo, como tratar-se de uma cópia superficial ou "**shallow copy**".

«...the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.»

Então, para que o método `clone` da classe `Triangulo` implemente uma "**deep copy**" não basta invocar na sua redefinição, `super.clone()`, é necessário fazer algo mais:

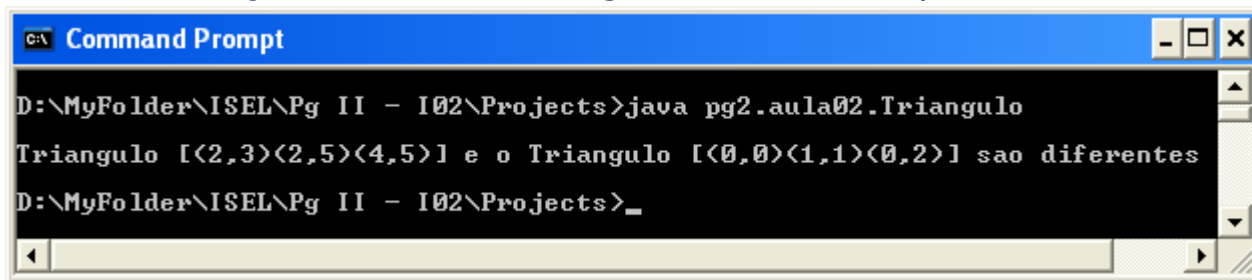


```

public Object clone() {
    return new Triangulo(p1.getX(), p1.getY(),
        p2.getX(), p2.getY(),
        p3.getX(), p3.getY());
}
    
```

(veremos mais à frente que esta implementação não é a mais correcta)

deste modo já obteríamos o seguinte resultado para o teste anterior:



```

C:\> Command Prompt

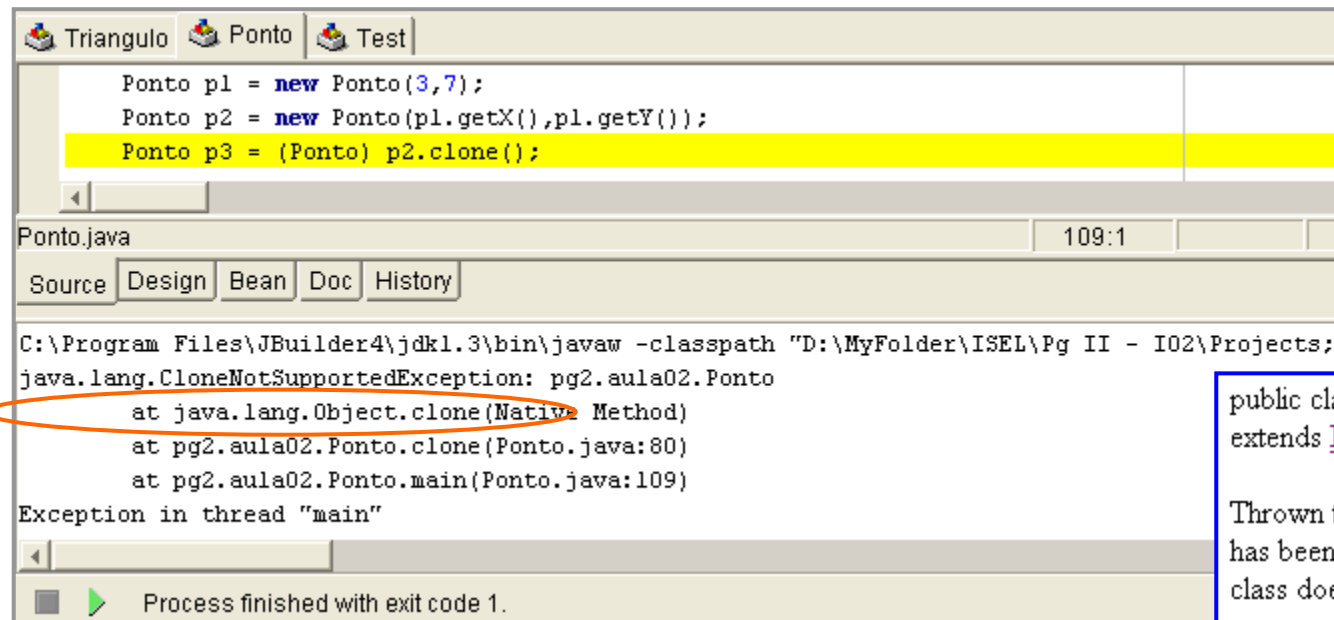
D:\MyFolder\ISEL\Pg II - I02\Projects>java pg2.aula02.Triangulo
Triangulo [(2,3)<(2,5)<(4,5)] e o Triangulo [(0,0)<(1,1)<(0,2)] sao diferentes
D:\MyFolder\ISEL\Pg II - I02\Projects>_
    
```

Marker Interface Cloneable

Para o utilizador, que acede à **Classe Ponto** e à **Classe Triângulo** como é que se sabe à priori que o comportamento do método `clone` da Classe Ponto é nativo e o da classe Triângulo não o é?

Porque a classe Ponto **tem** que implementar a **interface Cloneable** e a classe Triângulo não?

Porquê? O que acontece se a classe Ponto não implementar esta interface?



```

Triangulo Ponto Test
Ponto p1 = new Ponto(3,7);
Ponto p2 = new Ponto(p1.getX(),p1.getY());
Ponto p3 = (Ponto) p2.clone();

Ponto.java 109:1
Source Design Bean Doc History
C:\Program Files\JBuilder4\jdk1.3\bin\javaw -classpath "D:\MyFolder\ISEL\Pg II - IO2\Projects;
java.lang.CloneNotSupportedException: pg2.aula02.Ponto
    at java.lang.Object.clone(Native Method)
    at pg2.aula02.Ponto.clone(Ponto.java:80)
    at pg2.aula02.Ponto.main(Ponto.java:109)
Exception in thread "main"
Process finished with exit code 1.
  
```

```

public class CloneNotSupportedException
extends Exception
  
```

Thrown to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface.

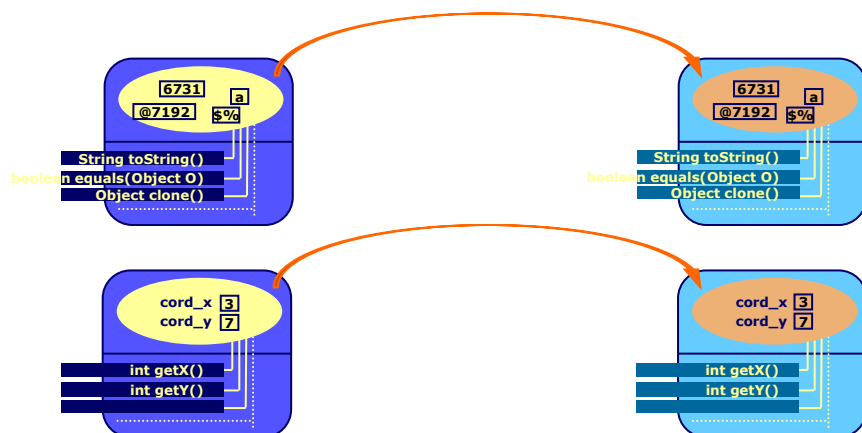
Applications that override the `clone` method can also throw this exception to indicate that an object could not or should not be cloned.

Conclusões

Para que as instâncias de uma determinada classe, tenham na sua interface a implementação nativa do método `Object.clone()` disponível a qualquer outro objecto, é necessário que:

- Redefina o método `clone` dando-lhe acesso `public`;
- Reportem e "tratem" a excepção `CloneNotSupportedException`;
- Implementem a interface `Cloneable`.

É ainda interessante notar que o método nativo `Object.clone()`, retorna sempre uma instancia da Classe a quem foi invocado o método. Ou seja:



Object => Instância de Object

Ponto => Instância de Ponto

... Conclusões

Este comportamento é transitivo...

Ou seja, se derivarmos a **Classe Ponto** numa nova classe **Ponto3D**, o método `clone()` herdado da Classe Ponto, quando invocado a uma instância da classe Ponto3D irá retornar uma instância desta Classe.

Como se processa então o mecanismo do `Object.clone()`?

- Este método é responsável por dimensionar com exactidão o tamanho do objecto que será copiado, e duplicá-lo. Uma vez reservado o respectivo espaço de memória, necessário para alojar o novo objecto, é então efectuada uma cópia "**bitwise**" dos bits do objecto original para o espaço do novo objecto.

Desta forma é garantido que independentemente do ponto onde seja invocado o método `clone()` o retorno será sempre um objecto da mesma classe que o original.

A **classe Triangulo** no seu processo de clonagem não invoca o método `super.clone()` (não implementa `Cloneable`), pelo que a condição anterior não é garantida.

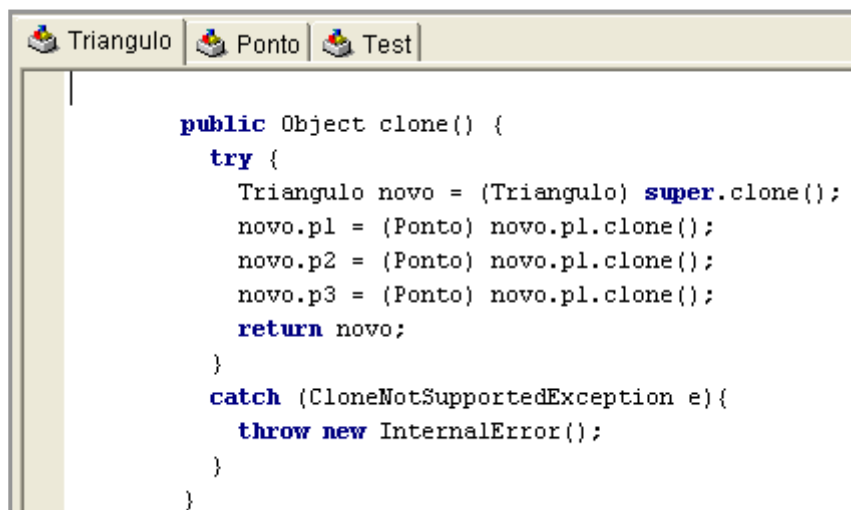
Ou seja, derivando a **classe Triangulo** numa nova **classe TrianguloColorido**, sem ser redefinido novamente o método `clone()`, não será retornado uma instancia desta classe quando lhe for invocado este método, mas sim de Triangulo.

... Conclusões



Efectivamente, já tinha sido referido aquando da sua implementação na classe `Triangulo`, que aquela não era a definição mais correcta para o método `clone()`.

Assim a correcta redefinição do método `clone()` na classe `Triangulo` seria:



```
public Object clone() {  
    try {  
        Triangulo novo = (Triangulo) super.clone();  
        novo.p1 = (Ponto) novo.p1.clone();  
        novo.p2 = (Ponto) novo.p1.clone();  
        novo.p3 = (Ponto) novo.p1.clone();  
        return novo;  
    }  
    catch (CloneNotSupportedException e) {  
        throw new InternalError();  
    }  
}
```

Mesmo quando o comportamento pretendido no método `clone()` não é semelhante ao nativo no `Object.clone()`, ou seja uma "**shallow copy**", o novo objecto deve ser o resultado do método `super.clone()` e não do construtor dessa classe. Assim é garantido que qualquer que seja o ponto da hierarquia a quem é invocado o método `clone` que o resultado verifica a condição:

- `x.clone.getClass() == x.getClass`

... Conclusões

E se quisermos voltar a retirar a propriedade Cloneable de uma classe derivada?

Se tiver uma **Classe A** que implementa a interface **Cloneable**,
Então a classe B que deriva de A, por herança também implementa **Cloneable**,

Mas se eu quiser que B já não seja Cloneable não consigo tirar-lhe esta interface.

No entanto, ainda me resta a opção de redefinir o método, lançando a excepção `CloneNotSupportedException` e assim impedir a clonagem.