Forma Canónica de Classes

$$\begin{pmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \lambda & 1 & \dots & 0 \\ 0 & 0 & \lambda & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 \\ 0 & 0 & 0 & \dots & \lambda \end{pmatrix}$$

Forma canónica de classes

A **forma canónica de classes públicas** garante que as suas instâncias têm um comportamento correcto quando manipuladas pelo *runtime* Java e por outras classes (como a *framework* de colecções).

Forma canónica de classes públicas:

- Construtor sem parâmetros:
  - Disponibiliza construtor sem parâmetros público (instanciar dinamicamente sem conhecer a existência classe);
- **Igualdade entre objectos:**
  - Redefine o método **`equals()`** e **`hashCode()`**
- Representação em `string` :
  - Redefine o método `toString()` (incluir a representação em *string* de todos os campos do objecto)
- **Clonagem**:
  - Implementar a interface **`Cloneable`**
- Seriação:
  - Implementar a interface `java.io.Serializable` e implementar o métodos `readObject()` e `writeObject()`

Igualdade entre objectos

Compara a igualdade do estado de objectos da mesma classe.
A implementação por omissão testa a identidade.

Todas as implementações de `equals()` devem satisfazer as seguintes condições:

- Reflexividade: `x.equals(x)` é sempre `true`
- Simetria: Se `x.equals(y)` é `true` então `y.equals(x)` é `true`.
- Transitividade: Se `x.equals(y)` é `true` e `y.equals(z)` é `true` então `x.equals(z)` é também `true`
- Consistência: `x.equals(y)` deve retornar consistente mente `true` ou `false` enquanto o estado do objecto não mudar
- Não nullidade: `x.equals(null)` retorna sempre `false`

Implementação de `equals()`

Exemplo do padrão `equals`:

```
public class C extends BaseClass {
   private int p;
   private Reftype r;
   public boolean equals(Object other) {
      if(this == other)
         return true;
      if(!(other instaceof C)
         return false;
      C otherC = (C)other;
      // Comparar cada um dos campos e retornar false caso não sejam iguais
         if(p != otherC.p)
            return false;
         if(r == null ? otherC.r != null : !r.equals(otherC.r) )
            return false;
         return true;
      }
}
```

Campos cujo valor é temporário, derivado de outros campos ou não essencial é designado por campo não significante e deve ser excluído das comparações (Ex: area de uma figura).

`Hash Code` de objectos

O método `hashCode()` é utilizado por classes que implementam HashTables (Exº: `HashMap` e `HashSet`).

A redefinição do `equals()` requer a redefinição de `hashCode()`.

Contrato a cumprir em `hashCode()`:

- Objectos iguais segundo o método `equals()` retornam o mesmo `hashCode()`, ou seja, se `x.equals(y) == true` então, `x.hashCode() == y.hashCode()`

Objectos diferentes podem ter *hash codes* iguais ou diferentes:

- `Hash Codes` diferentes para objectos diferentes melhoram o desempenho das *HashTables.*

Cálculo do *hash code*:

- Calcular o *hash code* de cada campo significativo. Para tipos primitivos converter o valor para inteiro. Para tipos referência chamar o método `hashCode()` se o campo não for null.
- Combinar os *hash codes* de todos os campos significativos.

Implementação de `hashCode()`

```
public int hashCode() {
   int hash = 0; // O hash code acumulado
   int c; // O hash code para um campo

   ... // Para cada campo calcular e combinar o hash code
   return hash;
}
```

```
// Or bit a bit
hash = hash << n | c;
// n é uma constante arbitrária
```


```
// Adição
hash = hash * p + c;
// p é um numero primo (ex° 37)
```

Clonagem de objectos

O método `clone()` retorna uma cópia do próprio objecto (análogo ao construtor por cópia em C++)

Contrato a cumprir no método `clone()` :

- O objecto retornado não deve ser o objecto clonado, i.e., `o.clone() != o`
- O objecto clonado e o original são instâncias da mesma classe
- O objecto clonado é igual ao original, i.e., `o.clone().equals(o) == true`

A classe Object implementa a versão por omissão do método clone():

```
Class Object {
   protected Object clone() throws CloneNotSupportedException {
      if(!(this insanceof Cloneable)) {
         throw CloneNotSupportedException();
      Object clone;
      // Cria e retorna uma cópia "superficial" do objecto
      return clone;
   }
}
```

Shallow Copy (Cópia superficial)

Uma classe que suporta clonagem tem que:

- implementar a "marker" interface Cloneable;
- redefinir o método clone() e declará-lo público.

```
public class Point implements Cloneable {
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}
```