sobre classes e instâncias

- *"Classes and object are separate yet intimately related concepts. Specifically,* ***every object is the instance of some class****, and every class has zero or more instances. For practically all applications,* ***classes are static****; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program. Similarly,* ***the class of most objects is static****, meaning that once an object is created, its class is fixed. In sharp contrast, however,* ***objects are typically created and destroyed at a furious rate*** *during the lifetime of an application."*

Definição do objecto classe

- Estado
  - identificação das variáveis de instância
- Comportamento
  - construtores/destrutores
  - getters e setters
  - outros métodos de instância, decorrentes do que representam

A referência this

- é usual precisarmos de referenciar o objecto que recebe a mensagem
- mas, no contexto da escrita do código da classe, ainda não sabemos como é que se vai chamar o objecto
- sempre que precisamos de ter acesso a uma variável do objecto podemos usar a referência *this*

- uma utilização muito normal é quando queremos desambiguar e identificar as variáveis de instância

- Por exemplo, em

```
// Métodos de Instância
public int    getX() { return this.x; }
public int    getY() { return this.y; }

public void setX( int    x) {this.x = x;}
public void setY( int    y) {this.y = y;}
```

- a utilização de *this* permite desambiguar a qual das variáveis nos estamos a referir

- a referência *this* pode ser utilizada para identificar um outro método da instância

```
// modificador - decrementa Coordenada X
void decCoordX(int deltaX) {
   this.decCoord(deltaX, 0);     // invoca decCoord() local
}
```

  - no caso de termos escrito apenas decCoord(deltaX,0)
    o compilador teria acrescentado automaticamente a referência *this*

  - também utilizada para invocar os contrutores (dentro de outros construtores)

Regras de acesso a variáveis e métodos

- a declaração das variáveis de instância pode ser precedida de informação sobre o nível de visibilidade

Modificador	Acessível a partir do código de
`public`	Qualquer classe
`protected`	Própria classe, qualquer classe do mesmo *package* e qualquer subclasse
`private`	Própria classe
nenhum	Própria classe e classes dentro do mesmo *package*

- para garantir o total encapsulamento do objecto as variáveis de instância devem ser declaradas como **private**

- ao ter encapsulamentomento total é necessário garantir que existem métodos que permitem o acesso e modificação das variáveis de instância.

- os métodos que se pretendem que sejam visíveis do exterior devem ser declarados como **public**

A classe Aluno

- declaração das variáveis de instância

```
/**
 * Classe Aluno.
 * Classe que modela de forma muito simples a
 * informação e comportamento relevante de um aluno.
 *
 * @author Antonio Nestor Ribeiro
 * @version 2006/03/20 (modificada Março 2009)
 */
public class Aluno {
    // instance variables
    private int numero;
    private int nota;
    private String nome;
```

- construtores: vazio, parametrizado e de cópia

```java
/**
 * Constructores para a classe Aluno
 */
public Aluno() {
  this.numero = 0;
  this.nota = 0;
  this.nome = "NA";
}

public Aluno(int numero, int nota, String nome) {
    this.numero = numero;
    this.nota = nota;
    this.nome = nome;
}


public Aluno(Aluno umAluno) {
    this.numero = umAluno.getNumero();
    this.nota = umAluno.getNota();
    this.nome = umAluno.getNome();
}
```

- métodos *getters* e *setters*

```
public int getNumero() {

    return this.numero;
}

public int getNota() {

    return this.nota;
}

public String getNome() {

    return this.nome;
}


public void setNota(int novaNota) {
    this.nota = novaNota;
}

private void setNumero(int numero) {
    this.numero = numero;
}

public void setNome(String nome) {
    this.nome = nome;
}
```

A classe Turma

- criação de um objecto que permita guardar instâncias de Aluno
- como estrutura de dados vamos utilizar um array de objectos do tipo Aluno
  - Aluno alunos[]
- A utilização de Aluno na definição de Turma corresponde à utilização de **composição** na definição de objectos mais complexos

- declação das v.i.

```
/**
 * Primeira implementao de uma turma de alunos.
 * Assume que a turma  mantida num array.
 *
 * @author Antonio Nestor Ribeiro
 * @version 2006/03/20
 * @version 2009/03/30
 * @version 2011/04/04
 */
public class Turma
{
    private String designacao;
    private Aluno[] lstalunos;
    private int capacidade;

    //variaveis internas para controlo do numero de alunos
    private int ocupacao;


    //se não for especificado o tamanho da turma usa-se esta constante
    private static final int capacidade_inicial = 20;
```

- construtores

```
/**
 * Constructor for objects of class Turma
 */
public Turma()
{
    this.designacao = new String();
    this.lstalunos = new Aluno[capacidade_inicial];

    this.ocupacao = 0;
}

public Turma(String designacao, int tamanho) {
     this.designacao = designacao;
     this.lstalunos = new Aluno[tamanho];

     this.ocupacao = 0;
}


public Turma(Turma outraTurma) {
    this.designacao = outraTurma.getDesignacao();

    this.ocupacao = outraTurma.getOcupacao();
    this.lstalunos = outraTurma.getLstAlunos();
}
```

- getters

```
public String getDesignacao() {
    return this.designacao;
}
```

```
public int getOcupacao() {
    return this.ocupacao;
}

/**
 * Método privado (auxiliar)
 *
 * Possível "buraco negro"!!! Como resolver?
 */
private Aluno[] getLstAlunos() {
    return this.lstalunos.clone(); //!!!
}
```

- o método getLstAlunos é auxiliar e privado

- inserir um novo Aluno

```
/**
 * Este método assume que se verifique previamente se
 * ainda existe espaço para mais um aluno na turma.
 *
 * Em futuras versões desta classe poderemos fazer internamente a
 * gestão das situações de erro. Neste momento assume-se que a
 * pré-condição é verdadeira.
 *
 * Este método deverá ser reescrito em futuras implementações
 * para evitar potenciais quebras de encapsulamento - já feito ao
 * agregar uma CÓPIA do aluno passado como parâmetro.
 *
 */
public void insereAluno(Aluno umAluno) {


    this.lstalunos[this.ocupacao] = new Aluno(umAluno); //encapsulamento garantido
    this.ocupacao++;
}
```

- utiliza-se o construtor de cópia de Aluno
  - porquê?!

O método clone

- este método tem como objectivo a criação de uma cópia do objecto a quem é enviado
  - a nocão de cópia depende muito da classe que faz a implementação
  - a noção geral é que x.clone() != x
  - sendo que,

    x.clone().getClass() == x.getClass()

O método clone

- regra geral, e de acordo com a visão em POO, a expressão seguinte deve prevalecer

  x.clone().equals(x),

  - embora isso dependa muito da forma como ambos os métodos estão implementados

- a implementação de clone é relativamente simples

O método clone

- na metodologia de POO já temos um método que faz cópia de objectos
  - o construtor de cópia de cada classe
- Dessa forma podemos dizer que apenas temos de invocar esse construtor e passar-lhe como referência o objecto que recebe a mensagem - neste caso o *this*

O método clone

- implementação do método clone da classe Aluno

```
/**
* Implementação do método de clonagem de um Aluno
*
* @param  umAluno   aluno que é comparado com o receptor
* @return  objecto do tipo Aluno
* ** * */

public Aluno clone() {
  return new Aluno(this);
}
```

- optamos por devolver um objecto do mesmo tipo de dados e não Object como é a definição padrão do Java.

Clone vs Encapsulamento

- a utilização de clone() permite que seja possível preservarmos o encapsulamento dos objectos, desde que:
  - seja feita uma cópia dos objectos à entrada dos métodos
  - seja devolvida uma cópia dos objectos e não o apontador para os mesmos

A clonagem de objectos

- Duas abordagens:
  - *shallow* clone: cópia parcial que deixa endereços partilhados
  - *deep* clone: cópia em que nenhum objecto partilha endereços com outro

- A sugestão é utilizar sempre *deep* clone, na medida em que podemos controlar todo o processo de acesso aos dados

- **REGRA**: clone do todo = "soma" do clone das partes

  - tipos simples e objectos imutáveis (String, Integer, Float, etc.) não precisam (não devem!) ser clonados.

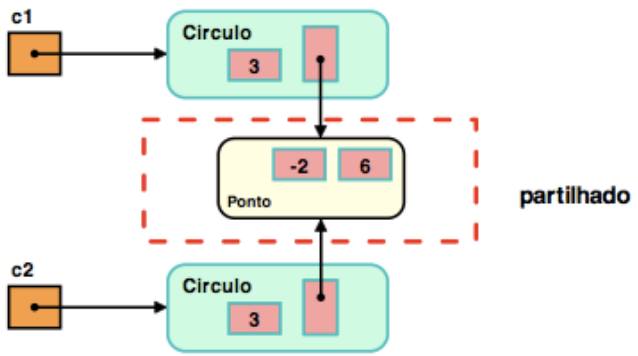
- A saber:
  - implementar o clone como sendo uma invocação do construtor de cópia

```
public Ponto2D getCentro() {
    return centro.clone(); // cria um novo Ponto2D, cópia do centro !!
}
```

  - o método clone() existente nas classes Java é sempre *shallow*, e devolve sempre um Object (se usado, é necessário fazer cast)
  - os clones que vamos fazer, nas nossas classes, devolvem sempre um tipo de dados da classe

- Exemplificação de um *shallow clone*

```
c2 = c1.clone1();
```



- existem conteúdos partilhados

- Como implementar os métodos
  - public boolean existeAluno(Aluno a)
  - public void removeAluno(Aluno a)
- como é que determinamos se o objecto está efectivamente dentro do array de alunos?

- A solução
  - lstalunos[i] == a, não é eficaz porque compara os apontadores
  - (lstalunos[i]).getNumero() == a.getNumero(), assume demasiado sobre a forma como se comparam alunos
- Quem é a melhor entidade para determinar como é que se comparam objectos do tipo Aluno?

- através da disponibilização de um método, na classe Aluno, que permita comparar instâncias de alunos
  - é importante que esse método seja universal, isto é, que tenha sempre a mesma assinatura
  - é importante que todos os objectos respondam a este método
- **public boolean equals(Object o)**

- dessa forma o método existeAluno(Aluno a) da classe Turma, assume a seguinte forma:

```
/**
 * De acordo com o funcionamento tipo destes métodos,
 * vai-se percorrer o array e enviar o método equals a cada objecto
 */

public boolean existeAluno(Aluno umAluno) {
    boolean resultado = false;

    for(int i=0; i< this.ocupacao && !resultado; i++)
      resultado = this.lstalunos[i].equals(umAluno);

    return resultado;
  }
```

- Em resumo:
  - método de igualdade é determinante para que seja possível ter colecções de objectos
  - o método de igualdade não pode codificado a não ser pela classe
  - existem um conjunto de regras básicas que todos os métodos de igualdade devem respeitar

O método equals

- a assinatura é:
  - **public boolean equals(Object o)**
- é importante referir, antes de explicar em detalhe o método, que:

O método equals

- a relação de equivalência que o método implementa é:
- é **reflexiva**, ou seja x.equals(x) == true, para qualquer valor de x que não seja nulo
- é **simétrica**, para valores não nulos de x e y se x.equals(y) == true, então y.equals(x) == true

- é **transitiva**, em que para x,y e z, não nulos, se x.equals(y) == true, y.equals(z) == true, então x.equals(z) == true

- é **consistente**, dado que para x e y não nulos, sucessivas invocações do método equals (x.equals(y) ou y.equals(x)) dá sempre o mesmo resultado

- para valores nulos, a comparação com x, não nulo, dá como resultado false.

- quando os objectos envolvidos sejam o mesmo, o resultado é true, ie, x.equals(y) == true, se x == y
  - dois objectos são iguais se forem o mesmo, ie, se tiverem o mesmo apontador
- caso não se implemente o método equals, temos uma implementação, por omissão, com o seguinte código:

```
public boolean equals(Object object) {
        return this == object;
}
```

- esqueleto típico de um método equals

```
public boolean equals(Object o) {

    if (this == o)
      return true;

    if((o == null) || (this.getClass() != o.getClass()))
      return false;

    <CLASSE> m = (<CLASSE>) o;
    return ( <condições de igualdade> );

}
```

- o método equals da classe Aluno

```
/**
 * 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());
     }

 }
```

- como é que será o método equals da classe Turma?

- quais as consequências de não ter o método equals implementado??
  - consideremos que Aluno não tem equals
  - o que acontece neste método de Turma?

```
/**
 * De acordo com o funcionamento tipo destes métodos,
 * vai-se percorrer o array e enviar o método equals a cada objecto
 */

public boolean existeAluno(Aluno umAluno) {
    boolean resultado = false;

    for(int i=0; i< this.ocupacao && !resultado; i++)
      resultado = this.lstalunos[i].equals(umAluno);

    return resultado;
  }
```