COLECÇÕES DE OBJECTOS EM JAVA6

ESTUDO DO
JAVA COLLECTIONS FRAMEWORK 6.0

PARTE II

F. MÁRIO MARTINS
DI/UNIVERSIDADE DO MINHO

2007/2008

Exemplo 1 com ArrayList<E> e HashSet<E>

Definida uma classe Amigo com as características,

```
public class Amigo {
   // Variáveis de Instância
      private String nome;
      private String telemovel;
      private String cidade;
      private int idade;
   // Construtores
      …
   // Métodos de Instância
   public String getNome() { return nome; }
   public String getTelem() { return telemovel; }
   public String getCidade() { return cidade; }
   public int getIdade() { return idade; }
   public Amigo clone() { return new Amigo(this); }
   …
}
```

Vamos definir a classe AgendaAmigos usando um ArrayList<Amigo> e programar alguns métodos

```
public class AgendaAmigos {
   // Variáveis de Instância
   private ArrayList<Amigo> listaAmigos;   // não é verdadeiramente uma lista e
                                           // deveria ser um HashSet<Amigo>.
                                           // Ver as diferenças no fim …
   // Construtores
   public AgendaAmigos() { listaAmigos = new ArrayList<Amigo>(); }

   public AgendaAmigos(ArrayList<Amigo> listaInic) { // Não partilha. Faz cópia !!
      listaAmigos = new ArrayList<Amigo>();
      for(Amigo amigo : listaInic)
         listaAmigos.add(amigo.clone());   // faz a cópia
   }

   // Métodos de Instância
   public int numAmigos() { return listaAmigos.size(); }

   public ArrayList<Amigo> getListaAmigos() {  // devolve uma cópia ☺
      ArrayList<Amigo> listaAux = new ArrayList<Amigo>();
      for(Amigo amigo : listaAmigos)
         listaAux.add(amigo.clone());   // faz a cópia
      return listaAux;
   }

   public ArrayList<Amigo> getListaAmigosErrado() {  // partilha a lista ☹
      return listaAmigos;
   }
```

```
// determina o número de amigos de dada cidade
public int numAmigosCidade(String cidade) {
  int conta = 0;
  for(Amigo amigo : listaAmigos)
    if(amigo.getCidade().equals(cidade)) conta++;
  return conta;
}

// determina o número de amigos com mais de X anos
public int numAmigosComMaisDe(int idade) {
  int conta = 0;
  for(Amigo amigo : listaAmigos)
    if(amigo.getIdade() > idade) conta++;
  return conta;
}

// devolve o conjunto dos nomes dos amigos de dada cidade
public HashSet<String> amigosDe(String cidade) {
  HashSet<String> amigosDe = new HashSet<String>();
  for(Amigo amigo : listaAmigos)
    if(amigo.getCidade().equals(cidade))
         amigosDe.add(amigo.getNome());
  return amigosDe;
}
```

```
// dá o número de telemóvel do amigo de nome dado
public String daTelemDe(String nome) {
  Iterator<Amigo> itAmigos = listaAmigos.iterator();
  boolean encontrado = false;
  Amigo amigo = null;
  while(itAmigos.hasNext() && !encontrado) {
    amigo = itAmigos.next();
    if(amigo.getNome().equals(nome)) encontrado = true;
  }
  return encontrado ? amigo.getTelem(): "";
}

// dá a média das idades dos amigos
public double mediaIdades() {
  int somaIdades = 0;
  for(Amigo amigo : listaAmigos) somaIdades += amigo.getIdade();
  return (double) (somaIdades/this.numAmigos());
}

// dá um conjunto de String Nome-Número dos amigos de uma dada cidade
public HashSet<String> listaTelemCidade(String cidade) {
   HashSet<String> listaTelem = new HashSet<String>();
   for(Amigo amigo : listaAmigos)
      if(amigo.getCidade().equals(cidade))
         listaTelem.add(amigo.getNome() + " – " + amigo.getTelem());
   return listaTelem;
}
```

```
// dá um conjunto de String Nome-Número dos amigos
public String toString() {
   StringBuilder sb = new StringBuilder("---- Lista de Amigos ----\n");
   for(Amigo amigo : listaAmigos)
      sb.append(amigo.getNome() + " – " + amigo.getTelem() + "\n");
   return sb.toString();
}
```

NOTA: SE EM VEZ DE USARMOS `ArrayList<Amigo>` TIVÉSSEMOS USADO `HashSet<Amigo>`, NEM UMA LINHA DESTE CÓDIGO NECESSITARIA DE SER MUDADA., EXCEPTUANDO AS DECLARAÇÕES.

Exemplo 2 com `ArrayList<E>` e `HashSet<E>`

```
import static java.lang.Math.PI;
public class Circulo {
   // Variáveis de Instância
   private double raio;
   private Ponto2D centro;
   // Construtores
   public Circulo() { raio = 0.0; centro = new Ponto2D(0.0, 0.0); }
   public Circulo(double r, Ponto2D p) { raio = r; centro = p.clone(); }
   public Circulo(Circulo c) {  raio = c.getRaio(); centro = c.getCentro(); // faz clone() }
   // Métodos de Instância
   public double getRaio() { return raio; }
   public Ponto2D getCentro() { return centro.clone(); }
   public double perimetro() { return 2*PI*raio; }
   public double area() { PI*raio*raio; }
   public boolean maior(Circulo c) { return raio > c.getRaio(); }
   public void aumentaRaio(double rx) { raio += rx; }
   public void desloca(double dx, double dy) { centro.incCoord(dx, dy); }
   public boolean equals(Circulo c) { return raio == c.getRaio() && centro.equals(c.getCentro()); }
   public boolean equals(Object obj) {
     if (this == obj) return true;
     if ( (obj == null) || (this.getClass() != obj.getClass()) ) return false;
     Circulo c = (Circulo) obj; return this.equals(c);
   }
   public Circulo clone() { return new Circulo(this); }
   public String toString() {
    StringBuilder s = new StringBuilder("\n --- Circulo ---\n");
    s.append("raio = " + raio + "\t centro = " + centro.toString() + "\n");
    return s.toString();
   }
}
```

VAMOS DEFINIR A CLASSE `MapaCirc` USANDO UM `HashSet<Circulo>` E PROGRAMAR ALGUNS MÉTODOS

```
public class MapaCirc {  // um conjunto de círculos
   // Variáveis de Instância
   private HashSet<Circulo> circulos;

   // Construtores
   public MapaCirc() { circulos = new HashSet<Circulo>(); }
   public MapaCirc(HashSet<Circulo> auxCircs) {
      circulos = new HashSet<Circulo>();
      for(Circulo circ : auxCircs)
        circulos.add(circ.clone());
   }
   public MapaCirc(MapaCirc planAux) {
      HashSet<Circulo> circAux = planAux.getCirculos();
      for(Circulo c : circAux) circulos.add(c);
   }

   // Métodos de Instância
   public HashSet<Circulo> getCirculos() {
      HashSet<Circulo> aux = new HashSet<Circulo>();
      for(Circulo circ : circulos) aux.add(circ.clone());
      return aux;
   }

   public int numCirculos { circulos.size(); }
```

```
// número de círculos de raio superior ao dado
public int numCircRaioMaiorQue(double raio) {
   int conta = 0;
   for(Circulo circ : circulos)
     if(circ.getRaio() > raio) conta++;
   return conta;
}

// conjunto dos círculos com centro à direita do ponto dado
public HashSet<Circulo> circDeCentroADireitaDe(Ponto2D ponto) {
   HashSet<Circulo> circs = new HashSet<Circulo>();
   for(Circulo circ : circulos)
     if(circ.getCentro().getX() > ponto.getX())
       circs.add(circ.clone());
   return circs;
}

// Círculo com maior raio
public Circulo circMaiorRaio() {
   Circulo circMaiorRaio = null;
   double maiorRaio = Double.MIN_VALUE;
   for(Circulo circ : circulos)
     if(circ.getRaio() > maiorRaio ) {
       circMaiorRaio = circ; maiorRaio = circ.getRaio();
     }
   return circ.clone();  // só no final se faz clone()
}
```

```
   // Incrementar todos os raios de dado valor
   public void incrementaRaios(double increm) {
      for(Circulo circ : circulos) circ.aumentoRaio(increm);
   }

   public String toString() {
      StringBuilder sb = new StringBuilder("---- Mapa de Circulos ----\n");
      for(Circulo c : circulos) sb.append(c.toString());
      return sb.toString();
   }

   public MapaCirc clone() { return new MapaCirc(this); }       // construtor faz clone

}
```

Auto-boxing e auto-unboxing em Java (desde Java5)

- Ainda que formalmente não possamos ter colecções de tipos simples, como por exemplo `ArrayList<int>`, em Java , mesmo antes de java5, é possível introduzir valores de tipos simples em colecções que sejam do tipo da respectiva classe wrapper (`Integer`, `Float`, `Double`, etc.).
- Para tal, usa-se os métodos `valueOf()` de conversão para objectos definidos em tais classes (cf. **BOXING**).

```
ArrayList<Integer> lstInt = new ArrayList<Integer>();
int i = Input.lerInt();
Integer objI = Integer.valueOf(i);  // converte int em Integer
lstInt.add(objI);
```

- Porém, **Java** oferece um mecanismo automático para tais conversões, designado **AUTOBOXING**: Conversão automática de tipos simples para instâncias das Wrapper Classes. Assim, podemos adicionar directamente valores de tipos simples a colecções das respectivas wrapper.

```
ArrayList<Integer> lstInt = new ...
int i = Input.lerInt();
lstInt.add(i);
```

```
ArrayList<Double> lstDb = new ....
double val = Input.lerDouble();
lstDb.add(val);
```

- Para converter os objectos extraídos das colecções em valores, em Java2 teríamos que usar o método associado ao seu tipo *tipo*`Value()`, cf. `intValue()`, `doubleValue()`, etc. (UNBOXING).

```
ArrayList<Integer> lstI = new ArrayList<Integer>();
  .....
Integer objI = lstI.get(1);
int i = objI.intValue();
```

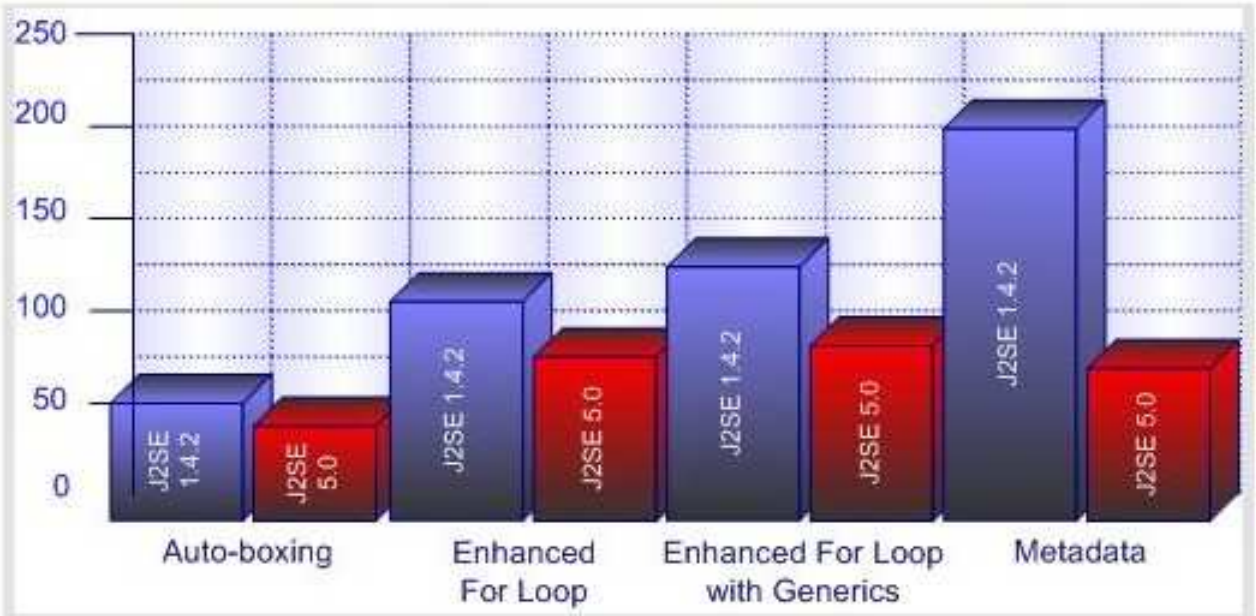
- Porém, **Java** oferece um mecanismo automático para tais conversões, designado **auto-unboxing**: Conversão automática de objectos de classes wrapper para os respectivos tipos simples

```
ArrayList<Integer> lstI = new ...
…
int i = lstI.get(x);
```

```
ArrayList<Double> lstD = new ...
…
double val = lstD.get(x);
```

```
for(Integer intObj : lstI) soma += intObj;  // somatório de int
```

A introdução das classes genéricas e os tipos parametrizados, os novos métodos `clone()` que podem dar como resultado um objecto do tipo da classe onde são implementados, deixando de ser necessário casting, os ciclos `for()` (foreach) sobre colecções e arrays, e os mecanismos de auto-boxing e auto-unboxing, vieram reduzir significativamente o esforço de codificação em JAVA5 comparativamente com as versões anteriores.



Porquê?

É importante saber !!

- Em Java2-4, as colecções não eram parametrizadas. Todas as colecções eram implementadas usando o supertipo `Object`. A uma variável da classe `Object` pode ser atribuída uma qualquer instância de qualquer classe.

```
Object obj = new Ponto2D();
             new Integer(12);
             “abcde”;
             new ArrayList();
```

- As colecções de Java2-4 eram, portanto, colecções de objectos de qualquer tipo, porque eram colecções de variáveis de tipo `Object`. No entanto, tinham (e têm ainda, porque continuam a existir em JAVA) os mesmos nomes das actuais colecções, não sendo porém parametrizadas.

```
ArrayList nomes = new ArrayList();
ArrayList pontos = new ArrayList();
HashSet circulos = new HashSet();
```

- Porém, o Programador tinha que ser muito disciplinado, porque não havendo indicação do tipo dos elementos o compilador não poderia verificar as ideias do programador, só ele mesmo !! Em geral, os tipos ficavam em comentário ...

```
ArrayList nomes = new ArrayList();    // ArrayList de String
ArrayList pontos = new ArrayList();   // ArrayList de Ponto2D
HashSet cidades = new HashSet();      // HashSet de InfoCidade
```

- Claro que adicionar elementos a estas colecções nunca poderia dar erro (mesmo que conceptualmente o programador estivesse a cometer erros), cf.

```
nomes.add("Rui"); nomes.add("Ana"); nomes.add("Lia");
nomes.add( new Ponto2D() ); 😐

pontos.add( new Ponto2D(2.0, 5.5) );
pontos.add( new Ponto2D(6.0, 1.5) );
pontos.add("abcd"); 😐
```

COMPILADOR DE JAVA2-4

Tudo Ok.
Sem erros de compilação !!

- Claro que qualquer operação de consulta a estas colecções estava programada para devolver uma variável de tipo `Object`. Tudo se baseava no supertipo. Por isso, o nosso programador Java2-4, "suava" para que tudo estivesse correcto, tendo que obrigatoriamente fazer **casting** de `Object` para o tipo que ele pretendia que a colecção tivesse (cf. comentários !!).

```
String nm = (String) nomes.get(0); // casting obrigatório excepto p/ Object
Ponto2D pt = (Ponto2D) pontos.get(0);
```

- E tudo estaria bem se tivesse "começado" bem. Porém, qualquer erro não detectado pelo compilador na inserção geraria um irrecuperável erro durante a execução do programa (o pior momento possível ... tipo, 5 da manhã na actualização de preços de um hipermercado...).

```
String nm = (String) nomes.get(3);    RunTimeError(91): String expected
Ponto2D pt = (Ponto2D) pontos.get(2); RunTimeError(91): Ponto2D expected
```

- A Partir de Java5, no JCF5.0, as colecções são implementadas em classes genéricas que o programador usa para criar colecções dos mais variados tipos, mas em que a correcção do tipo dos elementos destas colecções é verificada pelo compilador em **tempo de compilação** do programa. Assim, declarado na nova notação um `ArrayList` que apenas deve conter `String`, ou um `ArrayList` que apenas deve conter `Ponto2D`, cf.

```
ArrayList<String> nomes = new ArrayList<String>(); // no comments needed
ArrayList<Ponto2D> pontos = new ArrayList<Ponto2D>(); // no comments needed

nomes.add("Rui"); nomes.add("Ana"); nomes.add("Lia");
nomes.add( new Ponto2D() ); ☹

pontos.add( new Ponto2D(2.0, 5.5) );
pontos.add( new Ponto2D(6.0, 1.5) );
pontos.add("abcd"); ☹
```

COMPILADOR ACTUAL DE JAVA

ERROS DE COMPILAÇÃO = ☹ !

- Claro que qualquer operação de consulta a estas colecções está agora programada para devolver uma variável de tipo **E**. O programador de Java, não precisa de fazer **CASTING** porque sabe que cada elemento que pode extrair da colecção é, garantidamente, de tipo **E**, cf.

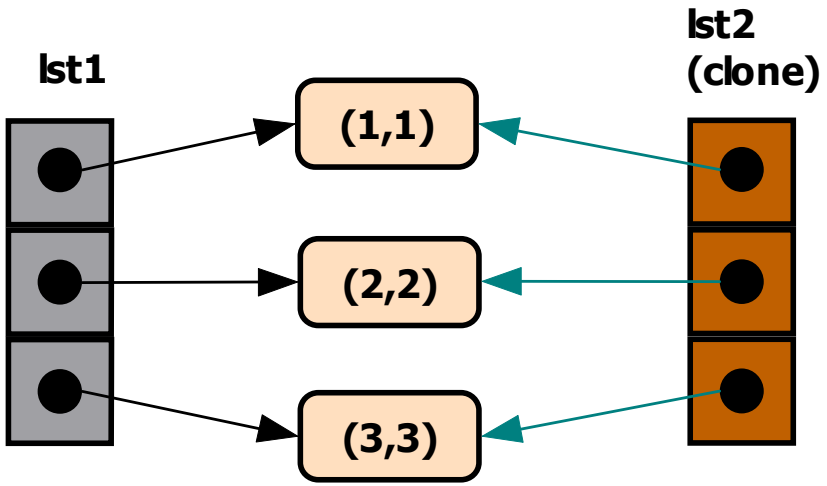
```
String nm = nomes.get(0);      // de nomes só podem "sair" String
Ponto2D pt = pontos.get(0);    // de pontos só podem sair Ponto2D
```

NOTA: ERROS DE RUNTIME DEVIDOS A ERROS DE TIPO ≈ 0 EM JAVA ACTUAL!

CLONE DE COLECÇÕES

CLONE PREDEFINIDO DE JAVA

Todas as colecções oferecem um método `clone()`. Este método é, como já vimos, um método de cópia *shallow*, pelo que o que o método faz é simplesmente copiar endereços. O resultado de `lst2 = lst1.clone();` será portanto (cf. Figura):



Resultado de `lst2 = lst1.clone();`

Ou seja, tudo é "apenas" partilhado !!

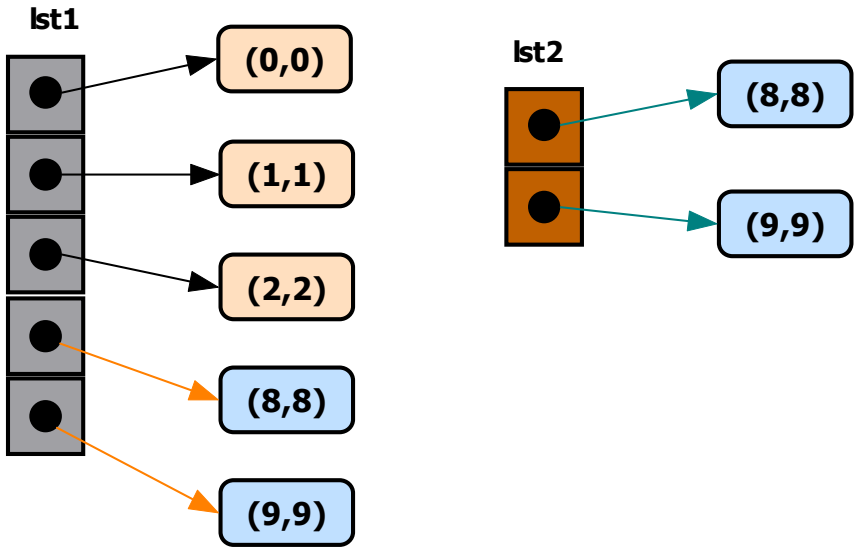
Partilha interna-externa implica não existência de encapsulamento e protecção de dados.
Assim, o método `clone()` definido para cada colecção não nos serve porque não preserva o encapsulamento !!

Então, como garantir que, quando se adiciona uma colecção a outra `addAll()` ou se usa `clone()` não ficam elementos partilhados entre elas? A resposta é clara agora: copiando de forma **DEEP** cada um dos elementos da colecção parâmetro, realizando o `clone()` de cada elemento.

Para tal, basta escrever o código seguinte:

```
for(Ponto2D p : lst2) lst1.add(p.clone());  // clone de lst2
```

Como a Figura mostra, usando esta operação as colecções são completamente independentes uma da outra, pelo que todos os problemas relacionados com alterações indevidas aos seus elementos deixam de existir.
É esta independência e protecção que pretendemos ter nas nossas variáveis de instância, garantindo que os seus valores apenas são modificados pelos métodos próprios.

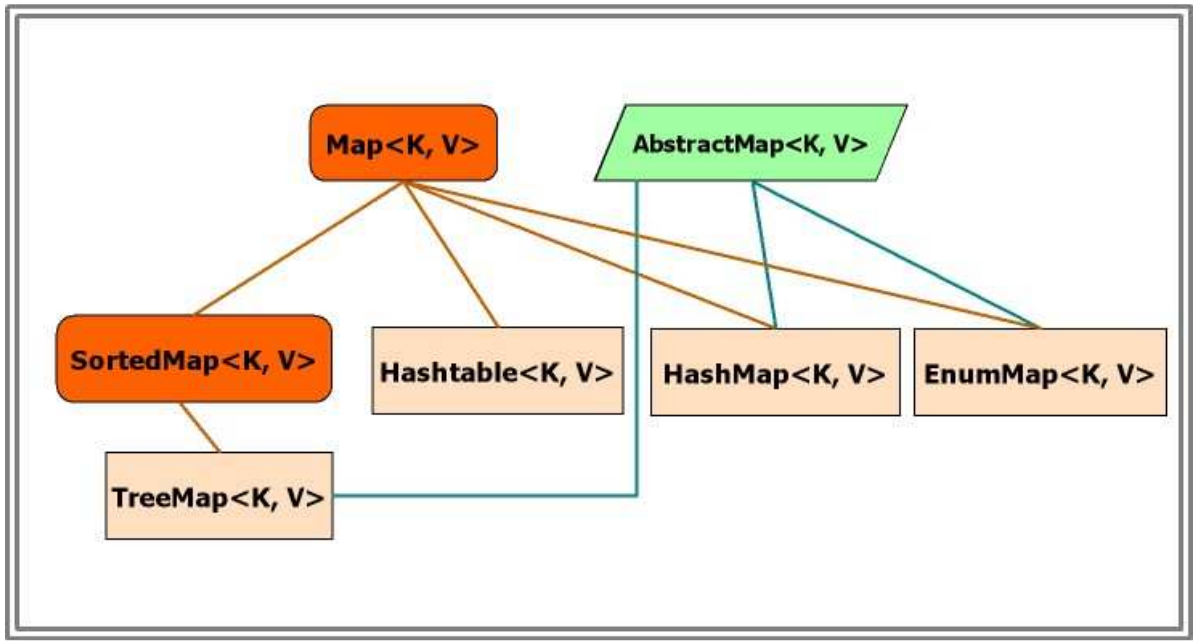


for(Ponto2D p : lst2) lst1.add(p.clone())

Colecções independentes

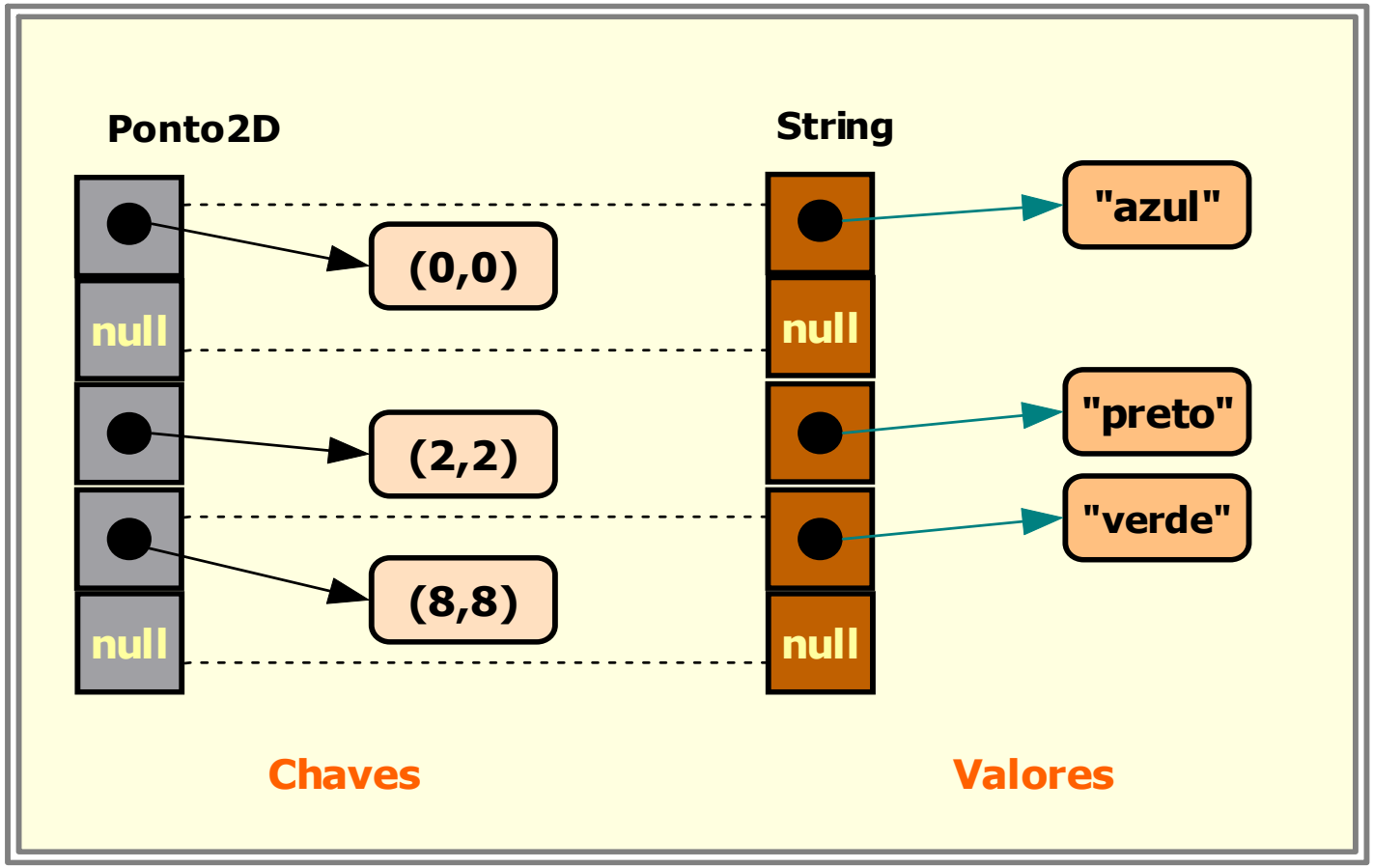
Estudo da Interface/API Map<K,V>

- São correspondências 1 para 1 entre objectos; associações típicas únicas de um objecto-chave a um objecto-valor;
- Possuem diversas classes de implementação, das quais estudaremos de momento em particular `HashMap<K,V>` e `TreeMap<K,V>`.



- `TreeMap<K,V>` irá oferecer-nos implementações de correspondências que permitem manter as chaves ordenadas, de forma automática se forem String, Integer, Double, etc.). Se não forem destas classes (exº `Ponto2D`) indicaremos ao construtor o algoritmo de ordenação (veremos como !!).

HashMap<Ponto2D, String> pixels = new HashMap<Ponto2D, String>();



Representação Interna

public interface Map<K, V>

```
public abstract boolean
      containsKey(Object chave)              // chave existe ?
      containsValue(Object valor)            // valor existe ?
      equals(Object o)
      isEmpty()                              // vazio ?

public abstract V put(K chave, V valor)      // inserir par Chave - Valor
public abstract V get(Object chave)          // dada a chave obtém o valor
public abstract V remove(K chave)            // remove a associação de chave K

public abstract void putAll(Map<? extends K,
                                ? extends V> m)   // insere um Map

public abstract Set<K> keySet()              // devolve o conjunto das chaves
public abstract Collection<V> values()       // devolve a colecção dos valores

public abstract void clear()                 // apaga associações
public abstract int size()                   // nº de associações
```

OPRAÇÕES TÍPICAS EM MAPS

```
HashMap<Ponto2D, String> pixels = new HashMap<Ponto2D, String>();

// insere um par Ponto2D – Cor
pixels.put(new Ponto2D(1.2,4.5), “azul”);

// dada uma chave Ponto2D devolve a respectiva Cor
String cor = pixels.get(p1);

// já existe uma dada chave ?
boolean teste = pixels.containsKey(pt1) ;

// existe algum pixel da cor dada ?
boolean teste = pixels.containsValue(“amarelo”) ;

// varrimento dos valores para determinar o número de pixels de cor c
int conta = 0;
for(String cor : pixels.values())
  if(cor.equals(c)) conta ++;

// varrimento das chaves para criar um Set< Ponto2D> dos ponto com X > vx
HashSet<Ponto2D> pts = new HashSet<Ponto2D>();
for(Ponto2D pt : pixels.keySet())
  if(pt.getX() > vx) pts.add(pt);
```

```
// mudar cor de um dado ponto p1; fácil porque há sempre “partilha” após get()
pixels.get(p1).mudaCor(“cinza”);

// remover uma associação Ponto2D - Cor
pixels.remove(new Ponto2D(1.0, -5.0));

// toString de um Map
String stringPixels = pixels.toString();

// iteração sobre as chaves para determinar o primeiro ponto com x == y
Iterator<Ponto2D> it = pixels.keySet();
boolean enc = false; Ponto2D pt = null;
while(it.hasNext() && !enc) {
  pt = it.next();
  if(pt.getX() == pt.getY()) enc = true;
}
if(enc) pt = pt.clone();
```