

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
    }

    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 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)
        listaTelem.add(amigo.getNome() + " - " + amigo.getTelem());
    }
    return listaTelem;
}

```

```
// dá um conjunto de String Nome-Número dos amigos de uma dada cidade
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 **ArrayList<Circulo>** E PROGRAMAR ALGUNS MÉTODOS

```
public class MapaCirc {
    // Variáveis de Instância
    private ArrayList<Circulo> circulos;

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

    // Métodos de Instância
    public ArrayList<Circulo> getCirculos() {
        ArrayList<Circulo> aux = new ArrayList<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;
    for(Circulo circ : circulos)
        if(circ.getRaio() > maiorRaio ) {
            circMaiorRaio = circ; maiorRaio = circ.getRaio();
        }
    return circ.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

public MapaCirc clone1() { return new MapaCirc(circulos); }     // construtor faz clone

```

AUTO-BOXING E AUTO-UNBOXING EM JAVA5

- AINDA QUE FORMALMENTE **NÃO POSSAMOS TER COLECÇÕES DE TIPOS SIMPLES**, COMO POR EXEMPLO `ArrayList<int>`, EM JAVA , MESMO ANTES DE JAVA5, É POSSÍVEL INTRODUIR 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, **JAVA5** 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 `tipoValue()`, CF. `intValue()`, `doubleValue()`, ETC. (UNBOXING).

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



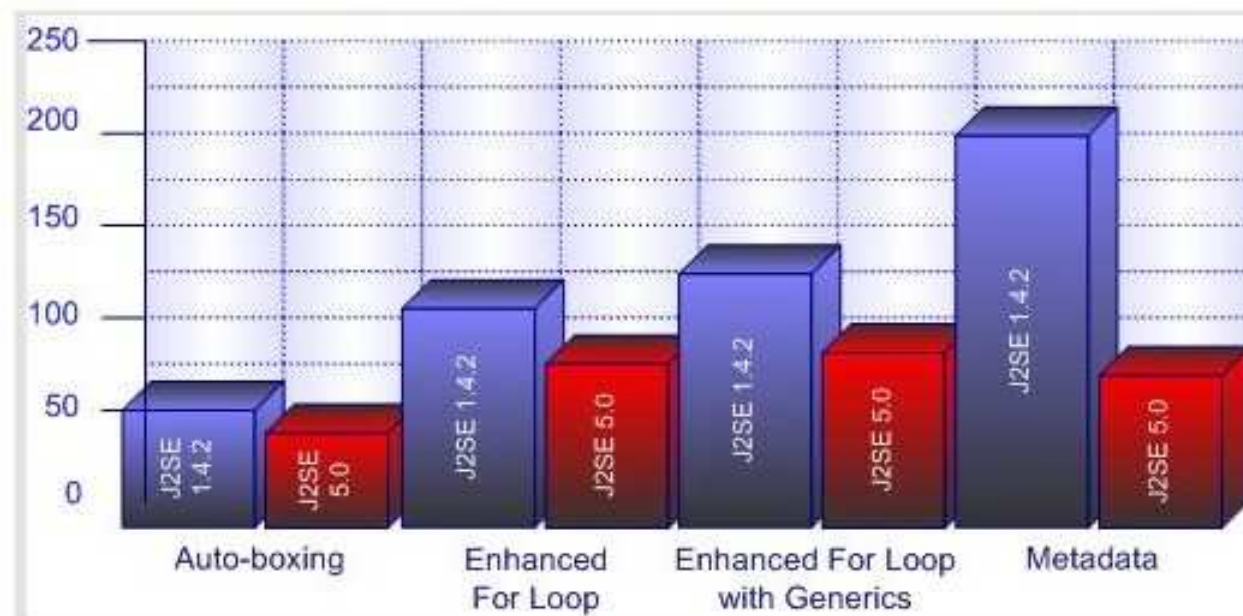
- PORÉM, JAVA5 OFERECE UM MECANISMO AUTOMÁTICO PARA TAIS CONVERSÕES, DESIGNADO AUTO-UNBOXING: CONVERSÃO AUTOMÁTICA DE OBJECTOS DE CLASSES WRAPPER PARA OS RESPECTIVOS TIPOS SIMPLES

```
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.



- EM JAVA2-4, AS COLECÇÕES NÃO ERAM PARAMETRIZADAS. TODAS AS COLECÇÕES ERAM IMPLEMENTADAS USANDO O SUPERTIPO **Object**. A UMA VARIÁVEL DA CLASSE **Object** PODIA 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 JAVA5) OS MESMOS NOMES DAS ACTUAIS COLECÇÕES:

```
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
```

- EM 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 NOTACÃO 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 DE **JAVA5**

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 JAVA5, 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
```

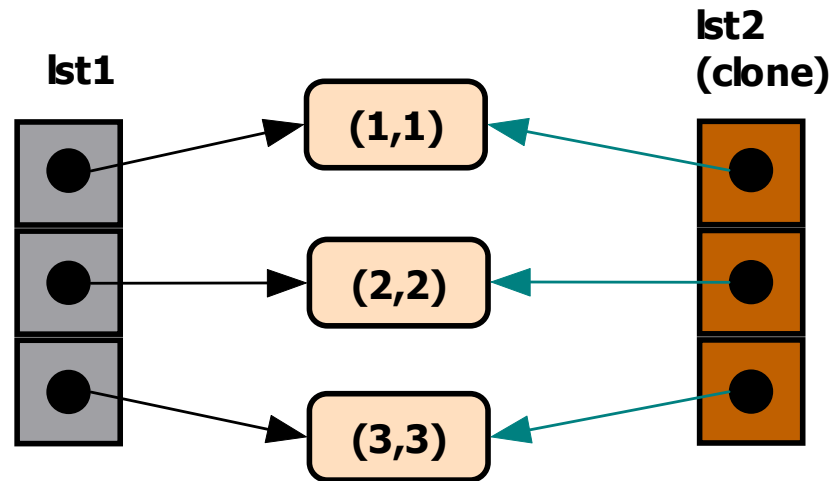
```
Ponto2D pt = pontos.get(0); // de pontos só podem sair Ponto2D
```

NOTA IMPORTANTE: ERROS DE RUNTIME POR ERROS DE TIPO ≈ 0 EM JAVA5 !!

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 SIMPLEMENTE 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 !!**

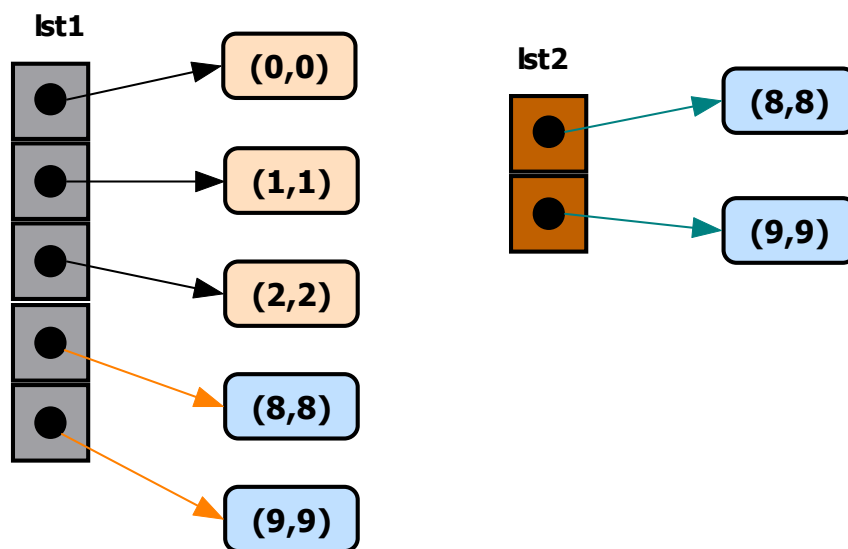
ENTÃO, COMO GARANTIR QUE, QUANDO SE ADICIONA UMA COLECÇÃO A OUTRA `addAll()` OU SE USA `clone()` NÃO FICAM ELEMENTOS PARTILHADOS ENTRE ELAS? A RESPOSTA É CLARA AGORA: COPIANDO DE FORMA **DEEP** CADA UM DOS ELEMENTOS DA COLECÇÃO PARÂMETRO, **REALIZANDO O `clone()` DE CADA ELEMENTO**.

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.

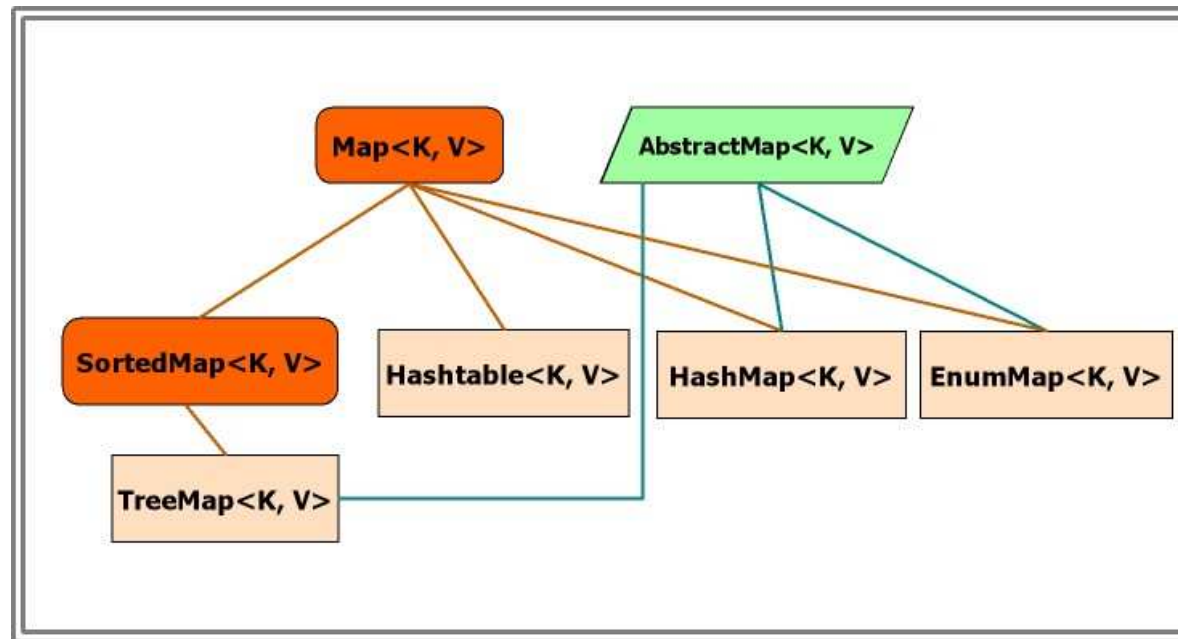


```
for(Ponto2D p : lst2) lst1.add(p.clone());
```

Colecções independentes

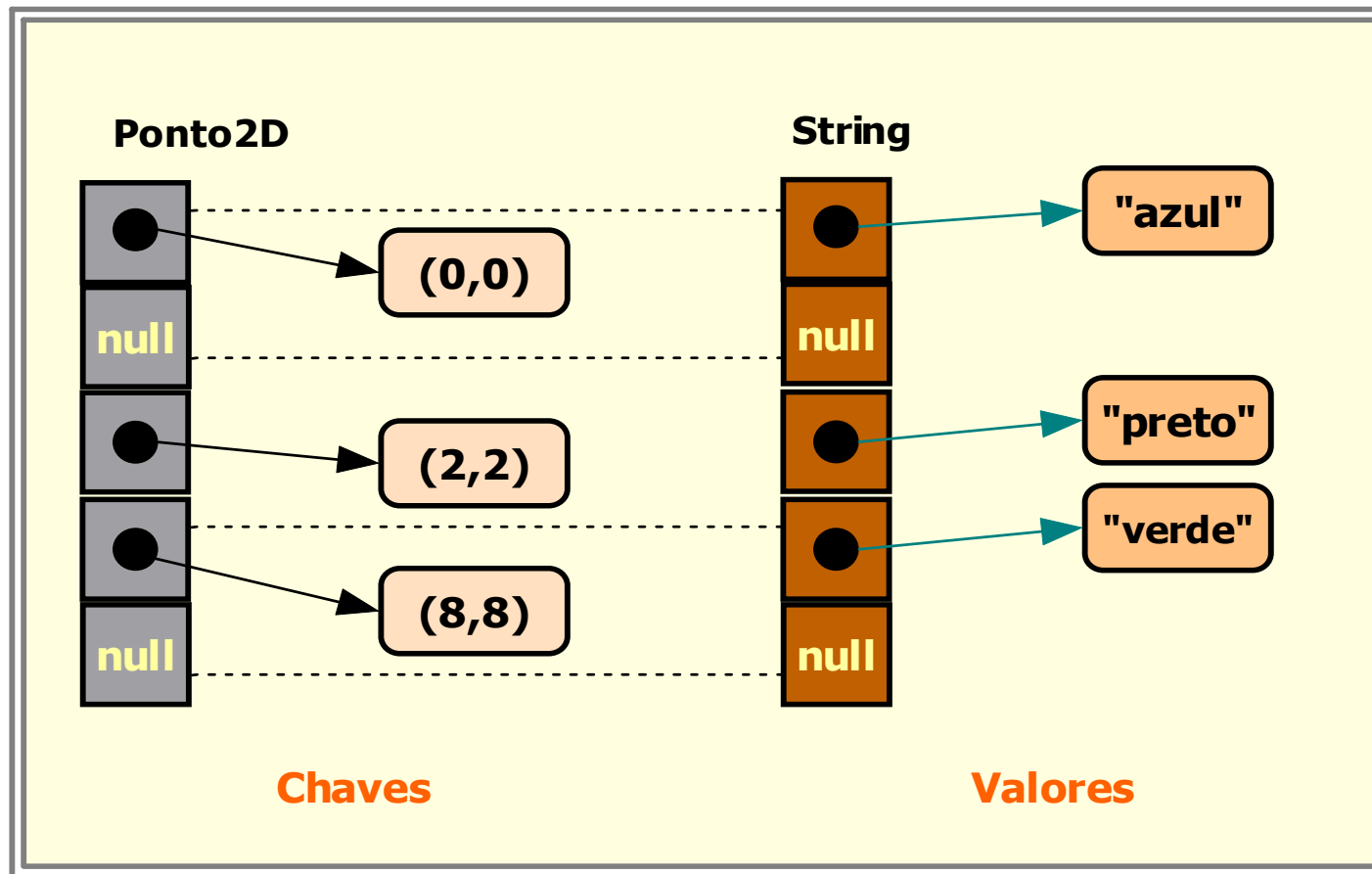
ESTUDO DA INTERFACE/API `Map<K,V>`

- 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

OPERAÇÕES 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();
```