TIPOS PARAMETRIZADOS
CLASSES GENÉRICAS

PARTE I

JAVA5 – Tiger

JAVA6 - Mustang

EVOLUÇÃO

Oak (1991): Desenvolvida para dispositivos electrónicos

Java 1.0 (1995): Versão Original (com "applets")

Java 1.1 (1997): Com "inner classes" e um novo modelo de tratamento de eventos

} Java 1

Java 1.2 (1998): Inclui "Swing"

Java 1.3 (2000): Novos métodos e "packages"

Java 1.4 (2002): Mais "packages"

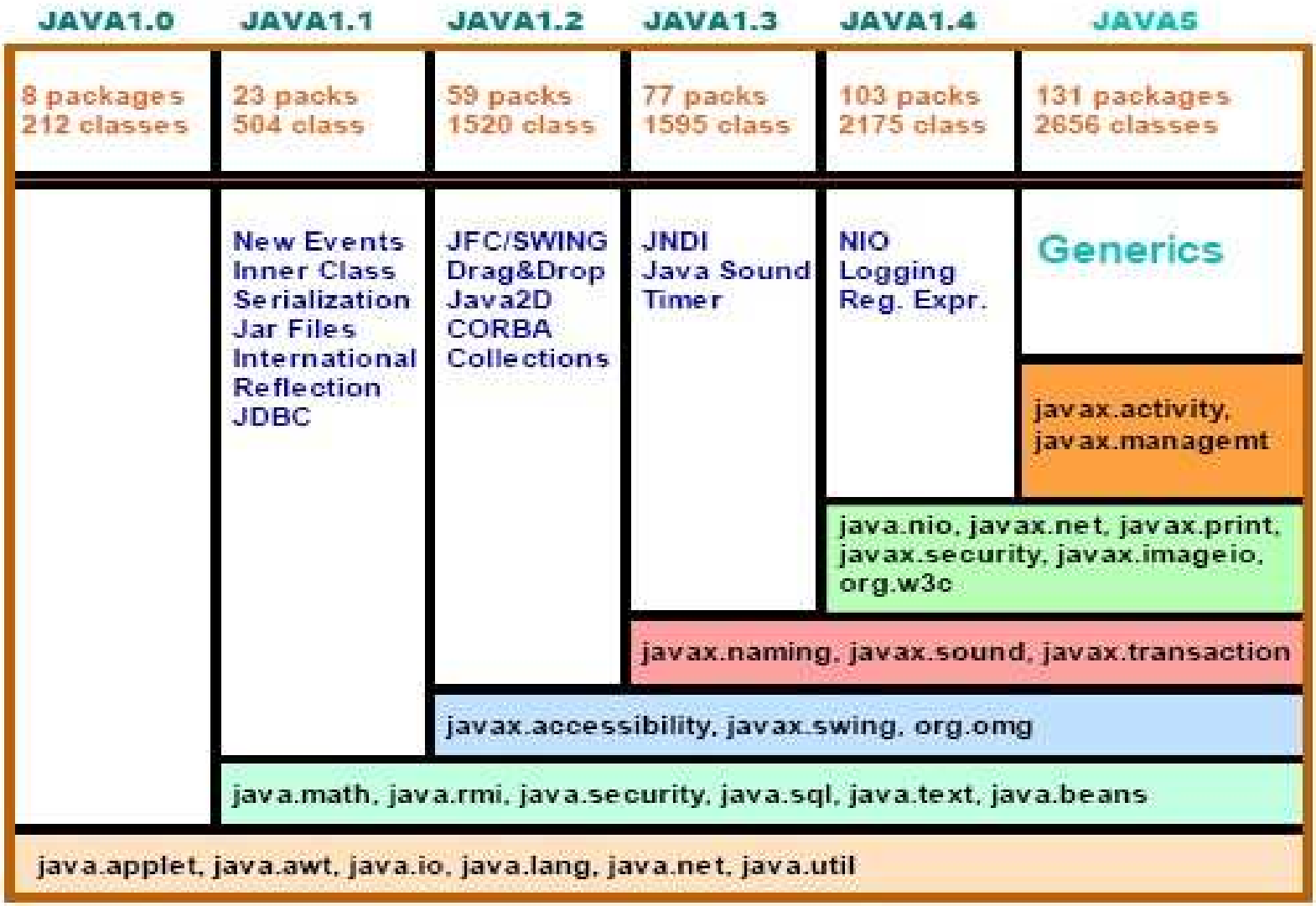
} Java 2

Java 1.5 (04/05): Nova sintaxe; Tipos enumerados, Colecções parametrizadas, novo iterador for, boxing e unboxing automáticos, genéricos (classes e métodos).

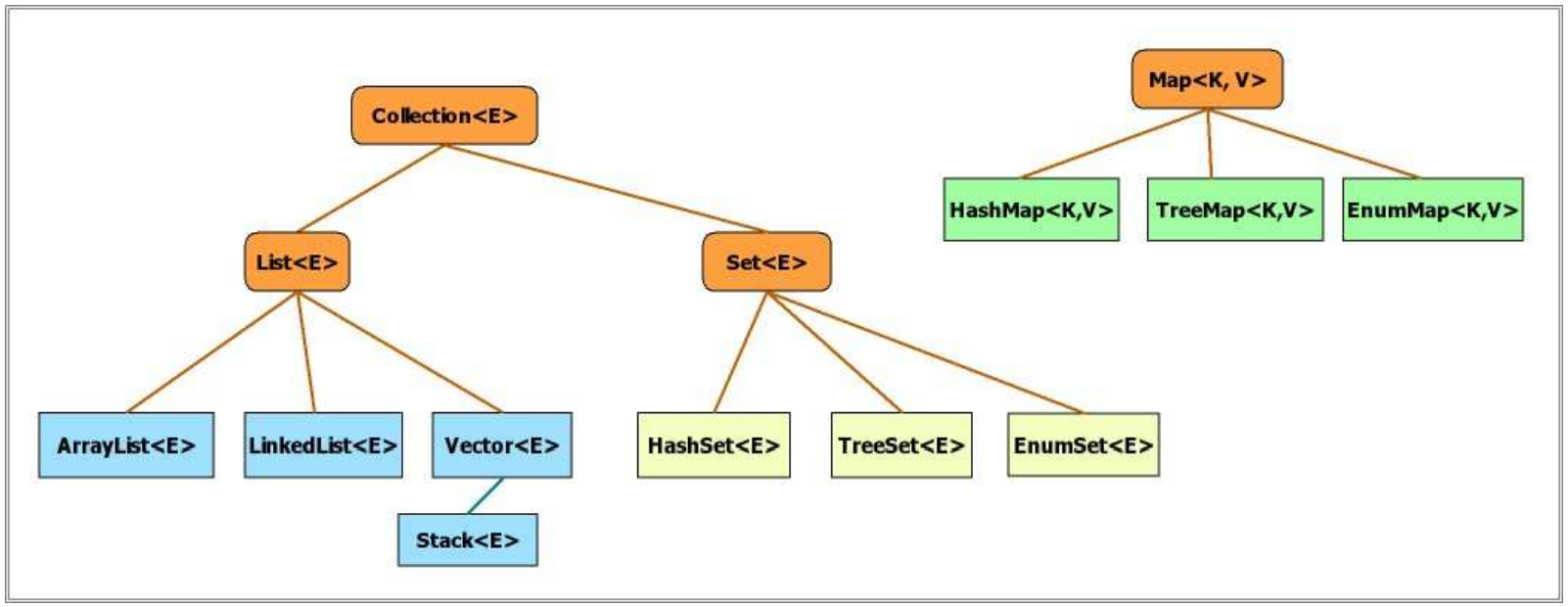
} Java 5

Java6

EVOLUÇÃO JSE



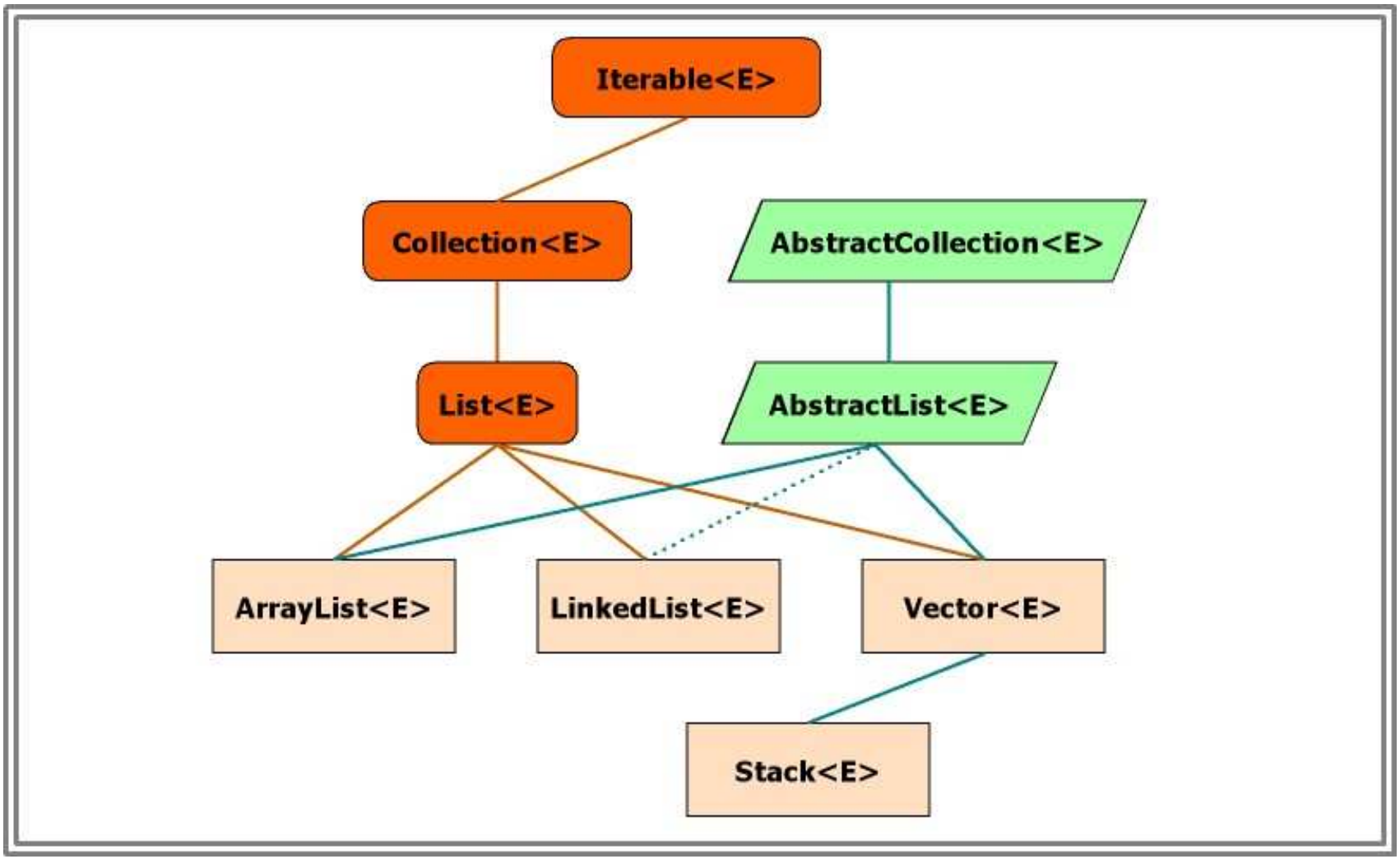
JCF – COLECÇÕES DE JAVA



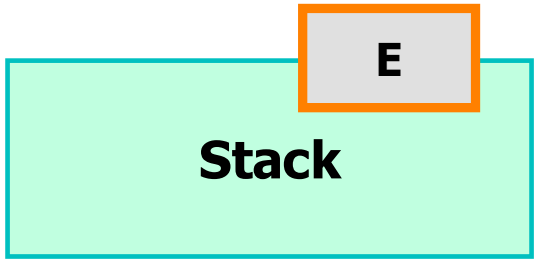
JAVA COLLECTIONS FRAMEWORK (JCF 5.0):

- Interfaces parametrizadas;
- Classes parametrizadas (concretas ou não).

JCF – LIST<E>


Iterable<E>
Collection<E>
AbstractCollection<E>
List<E>
AbstractList<E>
ArrayList<E>
LinkedList<E>
Vector<E>
Stack<E>

CLASSES GENÉRICAS => TIPOS PARAMETRIZADOS



Tipo genérico Stack<E>

⇓ para cada instanciação de E com uma classe, temos um tipo concreto

Stack<String>
Stack<Ponto2D>
Stack<Livro>
Stack<Integer>

todas satisfazem a API de Stack<E>

Stack<int> => restrição E ≠ tipo primitivo
(mas há Boxing e Unboxing, a ver) !!

JAVA2 – JAVA5

- **Problema:** Mudança de JAVA2 para JAVA5
- **Requisito:** Tornar a linguagem "type safe"

O impacto tecnológico será sempre brutal

Caminho 1

Alterar a sintaxe da linguagem mas manter a máquina virtual
(ou seja, manter byte-code)

JAVA5

Caminho 2

Alterar a sintaxe da linguagem e mudar a máquina virtual
ou o gerador de código intermédio

C# (CIL)

PROBLEMAS COM JAVA2

- Em JAVA2 tal como em JAVA5 a classe Object é a superclasse de todas as classes, simples ou colecções.

- Sendo classes TIPOS, o polimorfismo por inclusão garante que TODAS AS CLASSES (tipos) são compatíveis com Object (princípio da substituição de Liskov).

- Assim, todos os *arrays* e todas as colecções (JCF) de JAVA2 são colecções cujos elementos são instâncias de Object ou de classes compatíveis (ou seja, qualquer coisa).

```
ArrayList nomes = new ArrayList();
ArrayList idades = new ArrayList();
HashSet formas = new HashSet();
```

JAVA2 NÃO É "TYPE SAFE"

Antes

```
ArrayList amigos = new ArrayList();   // strings
nomes.add("ana"); nomes.add("rui");
nomes.add(new Integer(23));       // OK !!
nomes.add(new Motorista(2));      // OK !!
...
String nome = "";
for(Iterator it = amigos.iterator(); it.hasNext()){
    nome = (String) it.next(); // erro em runtime
    System.out.println(nome);
}
```

COMPILAÇÃO OK mas … ERROS DE RUNTIME

JAVA5 É "TYPE SAFE"

- 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 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 DE JAVA5

ERROS DE COMPILAÇÃO = ☹ !

OUTRAS COISAS BOAS DE JAVA5

- I/O: Classe Scanner;
- Importação estática;
- printf() “à la C”;
- Novo ciclo for();
- Métodos varargs;
- Métodos com tipos de retorno co-variantes;
- Novo JCF: Colecções parametrizadas e iterador for();
- Auto boxing e unboxing;
- Tipos enumerados “type safe”;
- Classes e métodos genéricos;

TIPO RESULTADO CO-VARIANTE

Permite redefinir de forma simples o método clone() de Object !!

Em JAVA5

```
public Ponto3D clone() {
 return new Ponto3D(this.getX(), this.getY(), z);
}

public Pixel clone() {
 return new Pixel(this.getX(), this.getY(), cor);
}
```

```
Ponto3D p = pt1.clone();

Pixel pixel2 = pixel1.clone();
```

Consequência (ex° classe Pixel):

a) Não usar o "shallow" clone() da classe Object !
b) Definir construtores de cópia, cf. Pixel(Pixel p);
c) Definir clone() como Pixel clone() { return new Pixel(this); }

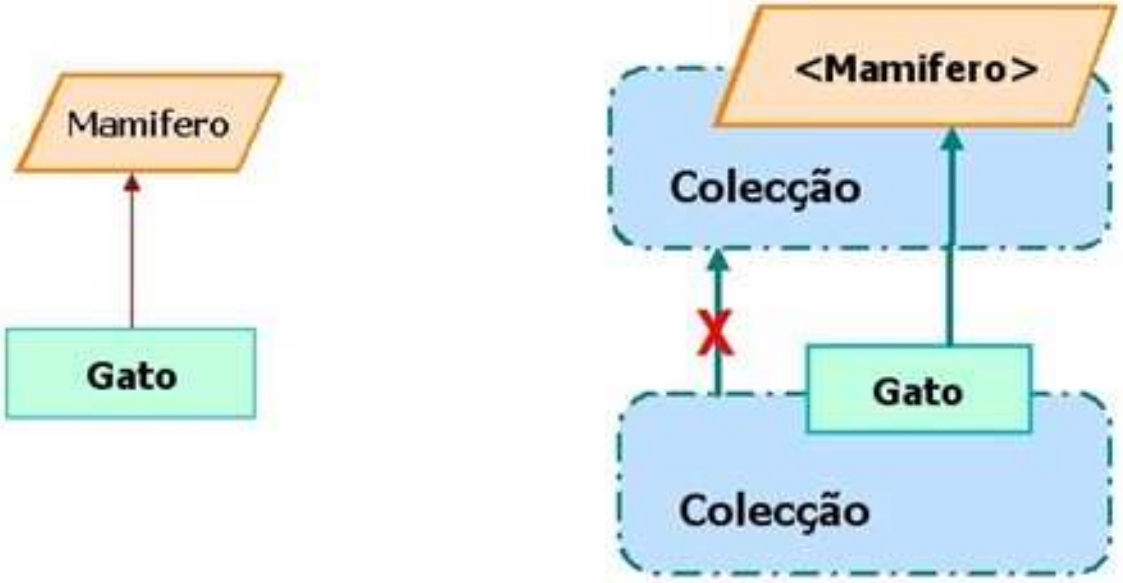
"The new language features (in Java5) all have one thing in common: they take some common idiom and provide linguistic support for it.

In other words, they shift the responsibility for writing the boilerplate code from the programmer to the compiler."

Joshua Bloch

Senior staff engineer, Sun Microsystems

COLECÇÕES DE JAVA NÃO SÃO CO-VARIANTES




```
ArrayList<Mamifero> mamif = new ArrayList<Mamifero>();
ArrayList<Gato> gatos = new ArrayList<Gato>();
gatos. add( new Gato("TIKO", "X", 3.5) ); .....
mamif = gatos;  // ERRO DE COMPILAÇÃO !
```

NÃO-COVARIÂNCIA

A NÃO-COVARIÂNCIA DAS COLECÇÕES É UMA PROPRIEDADE FUNDAMENTAL PARA A SUA SEGURANÇA DE TIPOS

```
1.ArrayList<Fruta> lstF= new ArrayList<Fruta>();
2.ArrayList<Laranja> lstL = new ArrayList<Laranja>();
3.lstF = lstL; // partilha de refs entre lstF e lstL
4.lstF.add(new Laranja(20));
5.lstF.add(new Banana(33));
6.for(Laranja larj : lstL) out.println(larj); // BUM !!
```

NÃO-COVARIÂNCIA

UTILIZAÇÃO DE WILDCARDS !!

Em vez de escrevermos:

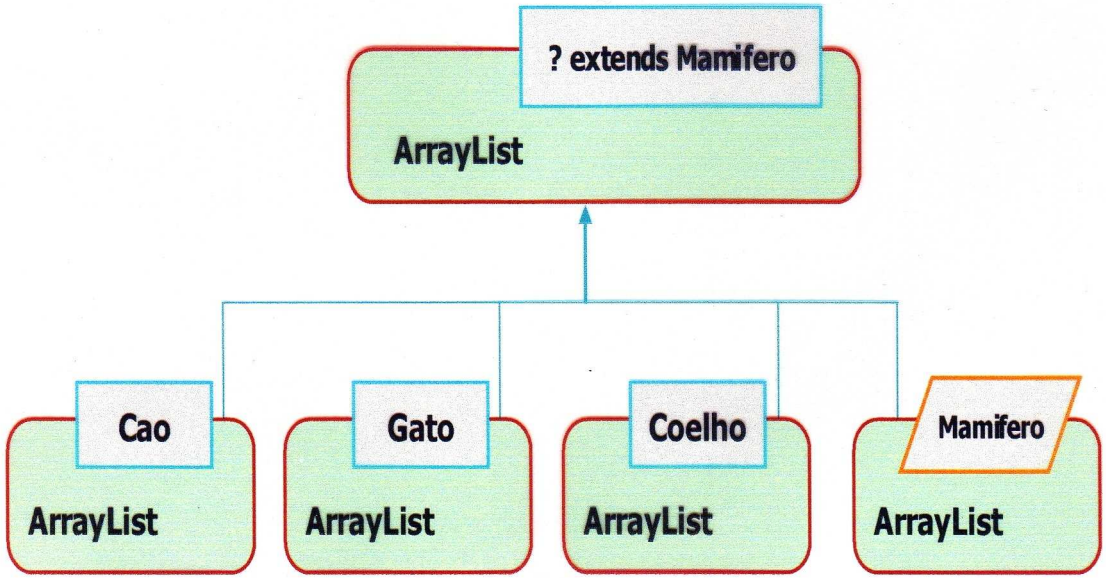
```
public void juntaMamif(Collection<Mamifero> cm) {
  ...
}
```

usamos o wildcard ? extends Mamifero que generaliza o tipo

```
Collection<? extends Mamifero> compatível com,
     Collection<Gato>  ou
     Collection<Cao>   ou
     Collection<Coelho>

public void juntaMamif(Collection<? extends Mamifero> cm) {
  ...
}
```

HIERARQIA COM WILDCARDS



WILDCARDS

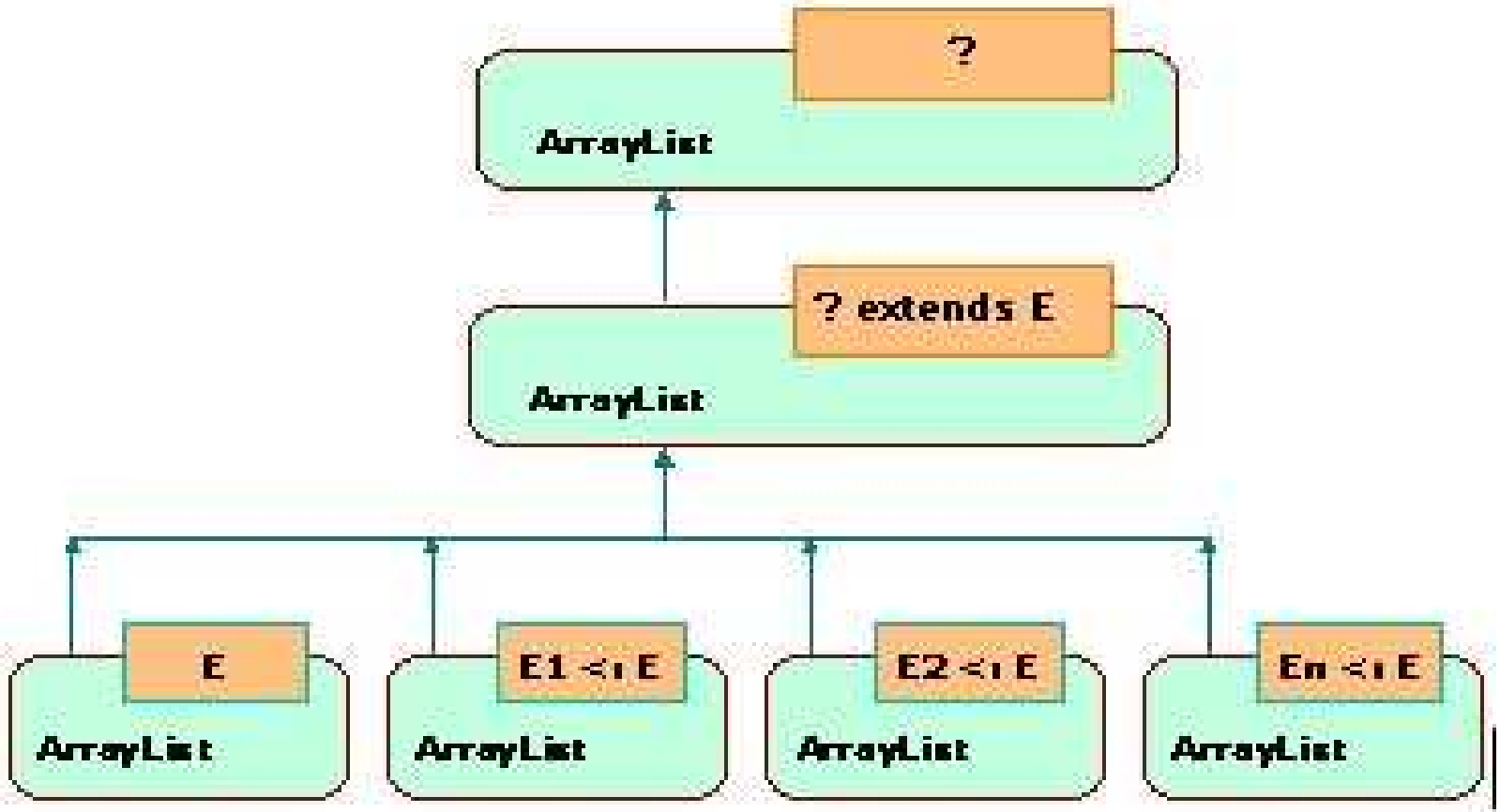
Wildcard	Tipos representados
`?`	Qualquer tipo
`? extends E`	`E` e qualquer subtipo de `E`
`? super E`	`E` e qualquer supertipo de `E`

Quadro 8.5 – *Wildcards* como especificadores de argumentos

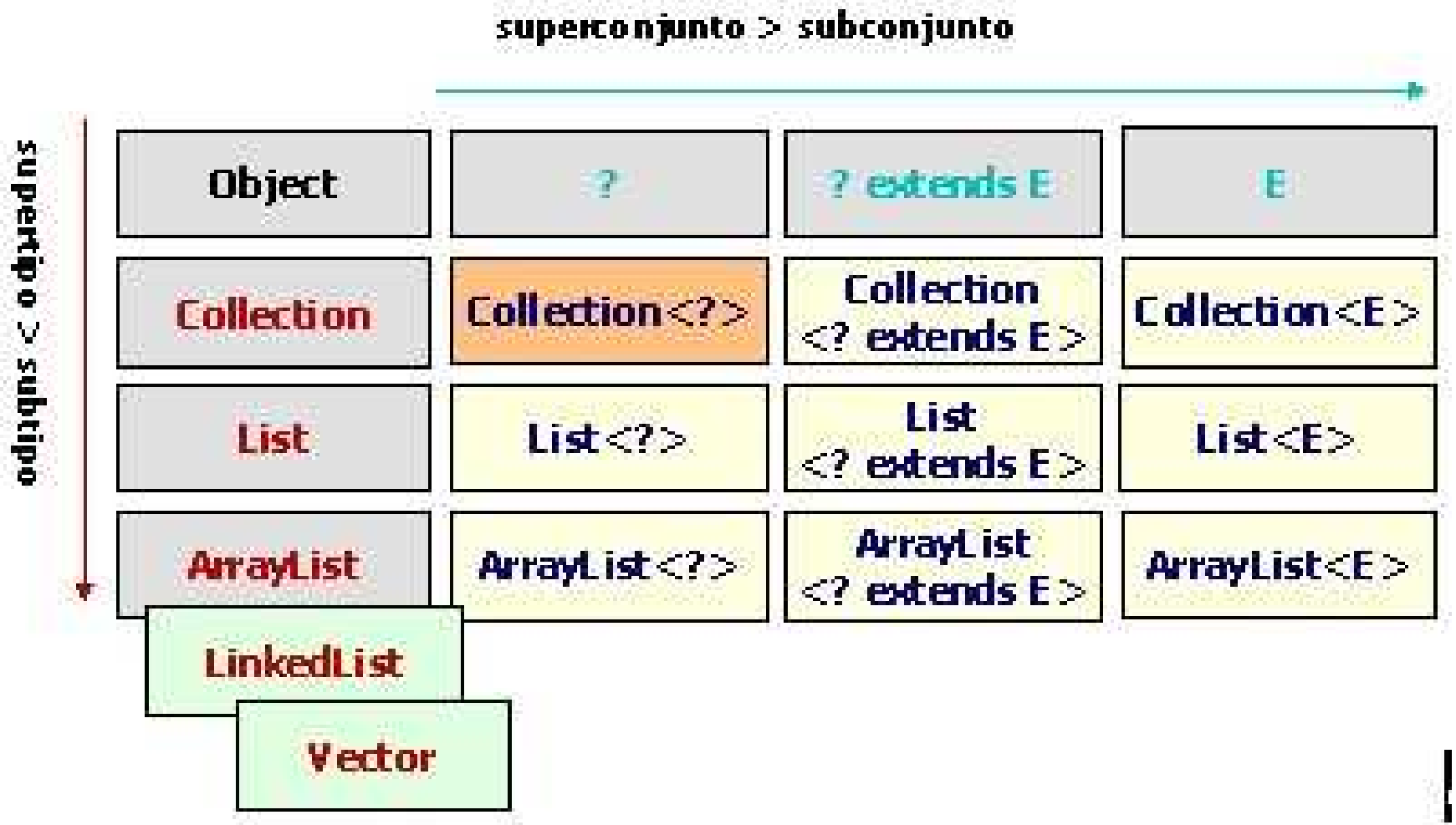
- Todas as colecções usam wildcards nas suas APIS

```
public boolean addAll(Collection<? extends E> c)
public boolean addAll(int index,
                      Collection<? extends E>  c)
public boolean contains(Object o)
public boolean containsAll(Collection<? extends E>  c)
```

HIERARQUIA


?
ArrayList
? extends E
ArrayList
E
ArrayList
E1 <: E
ArrayList
E2 <: E
ArrayList
En <: E
ArrayList

HIERARQUIA COM WILDCARDS



MODELO DE COMPILAÇÃO

NOTA: A MÁQUINA VIRTUAL DE JAVA NÃO FOI MODIFICADA.

O BYTE CODE É O MESMO !

- Cada instanciação de um tipo genérico ou método genérico tem uma representação própria;

- Porém, só existe uma representação para cada tipo genérico que é partilhada por todas as instanciações concretas;

- Mantêm-se as colecções heterogéneas de JAVA2, cf. ArrayList, etc., que passam a designar-se por **RAW TYPES**;

- **TYPE ERASURE (ELISÃO DE TIPOS)** => As classes genéricas têm por tipo os equivalentes raw types. Assim, por exemplo, `ArrayList<String>` e `ArrayList<Ponto>` são do tipo `ArrayList`, etc.
O resto baseia-se em **Object** e **casting**.

TYPE ERASURE - EXEMPLO

- Generic type

```
class LinkedList<A> implements Collection<A> {
  protected class Node {
    A elt; Node next = null;
    Node (A elt) { this.elt = elt; }
  }
  public void add (A elt) { ...  }
  ...
}
```

- After type erasure

```
class LinkedList implements Collection {
  protected class Node {
    Object elt; Node next = null;
    Node (Object elt) { this.elt = elt; }
  }
  public void add (Object elt) { ...  }
  ...
}
```

TYPE ERASURE - EXEMPLO

- Generic type

```
final class Test {
  public static void main (String[ ] args) {
    LinkedList<String> ys = new LinkedList<String>();
    ys.add("zero"); ys.add("one");
    String y = ys.iterator().next();
  }
}
```

- After type erasure

```
final class Test {
  public static void main (String[ ] args) {
    LinkedList ys = new LinkedList();
    ys.add("zero"); ys.add("one");
    String y = (String)ys.iterator().next();
  }
}
```

Additional Cast

TYPE ERASURE - PARÂMETROS

Tipo Parâmetro	Tipo Após Elisão
<T>	Object
<T extends Number>	Number
<T extends Cloneable & Comparable<T>>	Cloneable
<T extends Comparable<T> & Cloneable	Comparable
<T extends String & Comparable<T>>	String
<?>	Object
<? extends E>	E
<? super E>	Object

Quadro 11.1 – Elisão de tipos (*type erasure*)

Tipo Parametrizado - Exemplos	Tipo Após Elisão
List<String>	List
Par<String, String>	Par
Par<Integer, Integer>[]	Par[]
Set<?>	Set
Collection<?>[]	Object[]
List<? extends String>	List

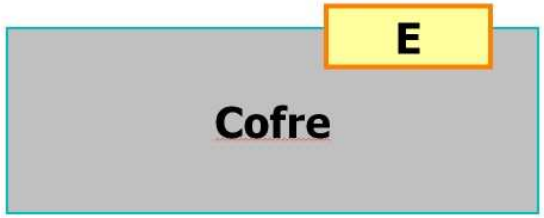
Quadro 11.2 – Resultados da elisão de tipos

CRIAÇÃO DE CLASSES GENÉRICAS

- Implica dominar o mecanismo de "type erasure" de JAVA5 que traz muitas complicações e "warnings" por vezes complexos;
- Implica conhecer o mecanismo de inferência de tipos;
- Implica conhecer o mecanismo de "wilcard capture";
- Implica conhecer os mecanismo de reflexão de JAVA5;
- Implica conhecer bem a classe Classe<T>.

VAMOS CRIAR DUAS CLASSES GENÈRICAS E VER, PASSO A PASSO, COMO TAL PODE SER FEITO

CRIAÇÃO DE CLASSES GENÉRICAS




```
public class Cofre<E> {
   private E elem = null;
   public Cofre(E objp) { elem = objp; }
   public void guarda(E objp) { elem = objp; }
   public E consulta() { return elem; }
   public void retira() { elem = null; }
   public boolean vazio() { elem == null; }
}
```

- Classe genérica Cofre<E> capaz de guardar qualquer coisa de um tipo E.

CRIAÇÃO DE CLASSES GENÉRICAS

```
Cofre<Integer> intgCofre = new Cofre<Integer>();
intgCofre.guarda(new Integer(12));
Cofre<Integer> intCofre = new Cofre<Integer>(12); // boxing
Cofre<String> strCofre = new Cofre<String>("abc");
```

O nosso Cofre<E> é *type safe*. Qualquer tentativa de inserir objectos de outro tipo ou ler objectos de tipo errado é de imediato detectada pelo compilador:

```
// Erros de Compilação por type checking
// strCofre.guarda(new StringBuilder("abc"));
// intgCofre.guarda("abc");
// Acesso aos cofre
// int val = (Integer) strCofre.consulta(); // Erro
int val1 = intCofree.consulta();
Integer intg = intgCofre.consulta();
```

CRIAÇÃO DE CLASSES GENÉRICAS

- A compilação da classe genérica Cofre<E> cria automaticamente o tipo raw que lhe corresponde, ou seja, Cofre.

```
public class Cofre {
   private Object obj = null;
   public Cofre(Object objp) { obj = objp; }
   public void guarda(Object objp) { obj = objp; }
   public Object consulta() { return obj; }
   public void retira() { obj = null; }
   public boolean vazio() { return obj == null; }
}
```

```
strCofre.getClass().getName() ======>  "Cofre"
intCofre.getClass().getName() ======>  "Cofre"
```

```
intCofre instanceOf Cofre<Integer>
```

ERRO !!

CRIAÇÃO DE CLASSES GENÉRICAS

Utilizações gerais

O código seguinte define uma classe genérica **CofreDuplo<E>** como subclasse da classe **Cofre<E>**, para criar um subtipo de **Cofre<E>** capaz de guardar dois objectos do tipo **E**:

```
public class CofreDuplo<E> extends Cofre<E> {
  private E elem;
  public CofreDuplo() { elem = null; }
  public CofreDuplo(E prim, E seg) {
      super(prim); elem = seg; }
  public void guarda(E prim, E seg) {
    super.guarda(prim); elem = seg; }
  public E primeiro() { return super.consulta(); }
  public E segundo() { return elem; }
  public void retira() { super.retira(); elem = null; }
  public bolean vazio() {
    return super.vazio() && (elem == null);
  }
}
```

Utilizações gerais

Na **instanciação do parâmetro de tipo** de uma classe genérica, isto é, na definição do tipo argumento, podemos também usar os *wildcards* que vimos antes: <?>, <? **extends E**> ou <? **super E**>. Vejamos alguns exemplos:

```
Cofre<?> cQq = new Cofre<String>("xxx");
out.println(cQq.consulta());
```

Note-se, no entanto, que se na especificação dos tipos argumento de uma classe genérica for usado mais de um *wildcard* ilimitado, cada ocorrência do ? **pode** representar um tipo concreto diferente. Vejamos exemplos:

```
Par<?, ?> par = new Par<String, String>("ABC", "DEF");

Par<?, ?> parDif = new Par<Ponto2D, Integer>();

Cofre<? extends Number> cNum = new Cofre<Integer>(123);
out.println(cNum.retira());

Cofre<? extends Appendable> cApp = new Cofre<StringBuilder>();
cApp.guarda(new StringBuilder("ABC"));
```

CRIAÇÃO DE CLASSES GENÉRICAS

Restrições e regras gerais

O **tipo parâmetro** `E` tem como âmbito de utilização (*scope*) os membros de instância da definição da classe genérica, podendo aí ser usado onde quer que um tipo possa ser usado. Porém, um tipo parâmetro de uma classe genérica não poderá ser usado na definição de um método `static` da classe. Há ainda outras limitações, como veremos.

Porém, **não é possível criar subclasses** não genéricas **de classes genéricas**, tal como em,

```
public class CofreGen extends Cofre<E> { // ERRO !!
public class MeuCofreDuplo extends CofreDuplo<E> { // ERRO !!
```

nem **de tipos parametrizados com *wildcards***, como em:

```
public class CofreDuplo<?> extends Cofre<?> { // ERRO !!
public class CofreW extends Cofre<? extends String> { // ERRO
```

CRIAÇÃO DE CLASSES GENÉRICAS

Por exemplo, admitamos que pretendemos que a classe `Cofre` apenas guarde objectos que sejam instâncias da classe `Number` ou de uma subclasse de `Number`, ou de `Ponto2D` ou de uma qualquer subclasse de `Ponto2D`. Nestes casos, é possível usar um parâmetro de tipo limitado, especificando exactamente tal restrição, usando a forma `E extends T`, em que `T` especifica o tipo que serve de limite (*bound*):

```
public class CofreNumeros<E extends Number> { ..... }
public class CofrePontos<E extends Ponto2D> { ..... }
```

A utilização da forma `<E extends T>` é mesmo obrigatória, sempre que na definição da classe genérica necessitarmos de ter acesso aos métodos definidos no tipo `E`, o que geralmente acontece. Por isso nos referimos antes a limtações no uso de `E` dentro da classe.

CRIAÇÃO DE CLASSES GENÉRICAS

EXERCÍCIO:

TENTAR CRIAR A CLASSE GENÉRICA

PAR<X, Y>