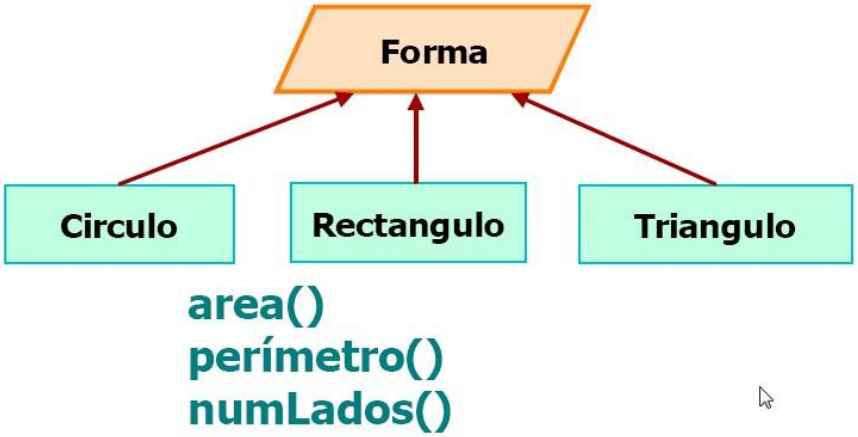
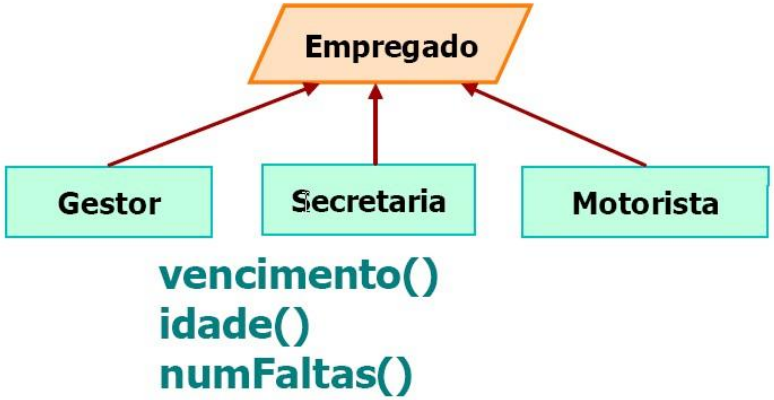
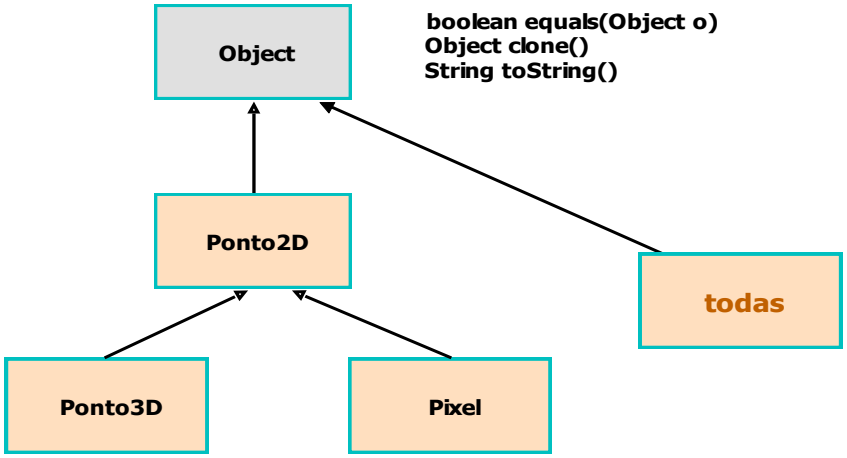
HIERARQUIAS TÍPICAS








```
public class Ponto2D { // Exemplo de Herança

   // Construtores
   public Ponto2D(double x, double y) { this.x = x; this.y = y; }
   public Ponto2D() { this(0, 0) ; }
   public Ponto2D(Ponto2D p) { x = p.cx(); y = p.cy(); }

   // Variáveis de Instância
   private double x, y;

   // Métodos de Instância
   public double cx() { return x; }
   public double cy() { return y; }
   public void somaponto(double cx, double cy) {x += cx; y += cy; }
   public void somaponto(Ponto2D p) { x += p.cx() ;  y += p.cy() ; }
```

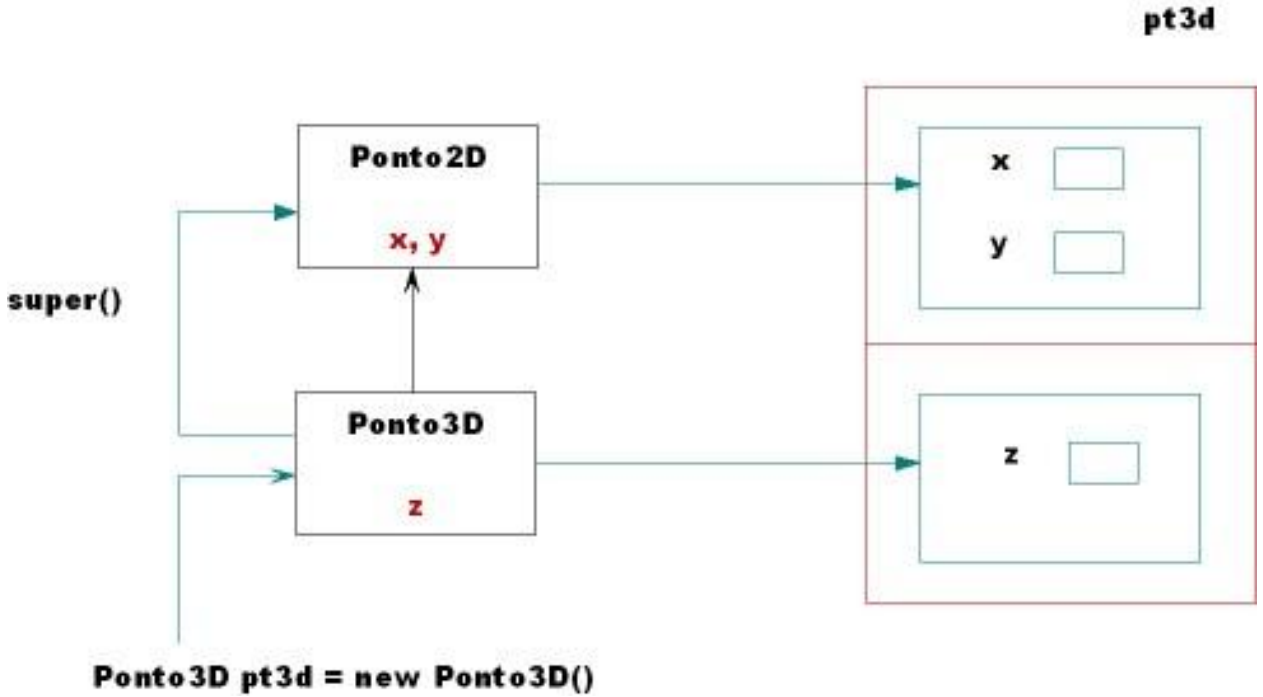
```
  public Ponto2D pontoSoma(Ponto2D p) {
    return new Ponto2D(x + p.cx(), y + p.cy());
  }

  public String toString() { return new String("Ponto2D(" + x + "," + y +")")) }

  public boolean equals(Object obj) {
    if (this == obj) return true;
    if( (obj == null) || (this.getClass() != obj.getClass()) )
      return false;
    Ponto2D p = (Ponto2D) obj;
    return  x ==p.cx() && y == p.cy();
 }

 public Ponto2D clone() { return new Ponto2D(this); }

}
```

A Classe Ponto3D deve ser criada usando as coordenadas x, y de ponto2D e os métodos de Ponto2D herdaddos, acrescentadndo a variável z e os métodos que tratam da variável z.

```
public class Ponto3D extends Ponto2D {

  // Variáveis de instância
  private double z ;

  // Construtores
  public Ponto3D() { super(); z = 0.0; }

  public Ponto3D(double x, double y, double z) {
    super(x, y); this.z = z ;
  }

  public Ponto3D(Ponto3D pt3) {
    super(pt3.cx(), pt3.cy()); this.z = pt3.cz() ;
  }

  // Métodos de Instância
  public int cz() { return z; }

  public void somaponto(double x, double y,
                                  double z) {
    this.somaponto(x, y); this.z = this.z + z;
   }
```

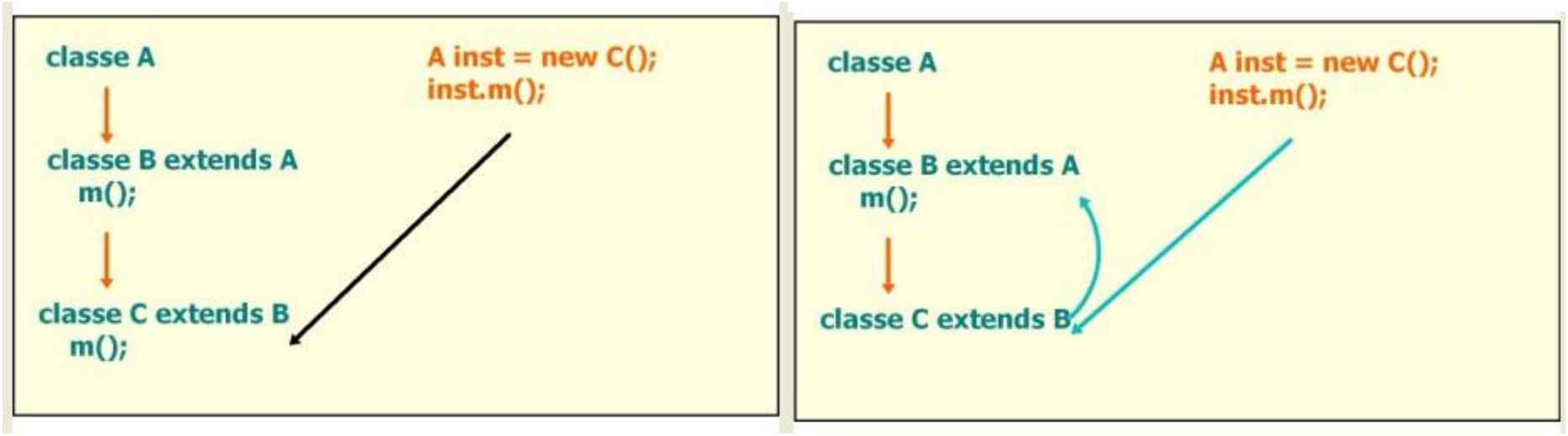
```
  public Ponto3D pontoSoma(Ponto3D p) {
    return new Ponto3D(this.cx() + p.cx(), this.cy() + p.cy(), z + p.cz());
  }

  public String toString() {
    StringBuilder s = new StringBuilder();
    s.append( "Ponto3D(" ); s.append(this.cx());
    s.append(", "); s.append(this.cy());
    s.append(", "); s.append(z); s.append("\n");
    return s.toString();
  }
}
```

Ponto3D = Ponto2D + Δ



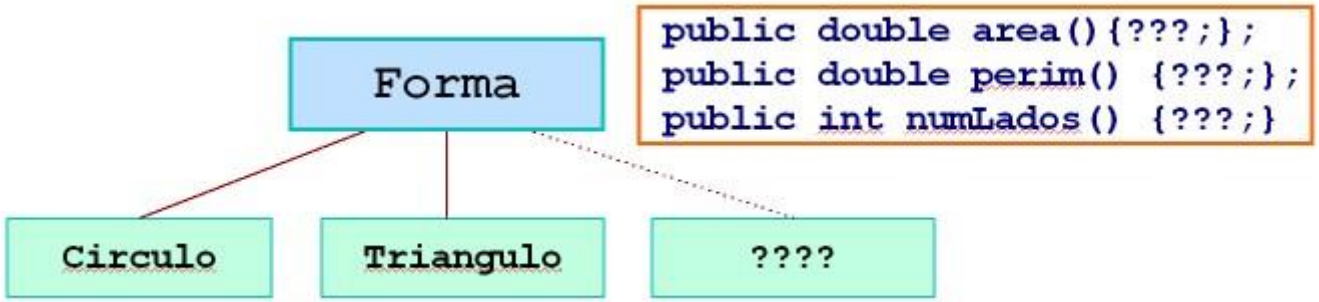
HERANÇA NA PRÁTICA

- OS CONSTRUTORES DA SUBCLASSE DEVEM, ANTES DE OUTRA COISA QUALQUER, INVOCAR OS CONSTRUTORES ADEQUADOS DA SUPERCLASSE, USANDO `super()` OU `super(…,…,…);`
- ESTA INVOCAÇÃO DOS CONSTRUTORES DA SUPERCLASSE TEM POR OBJECTIVO INICIALIZAR AS VARIÁVEIS QUE PERTENCEM À SUPERCLASSE;
- DADO QUE AS VARIÁVEIS DAS NOSSAS CLASSES SÃO PRIVADAS, NÃO SÃO HERDADAS NUNCA, MAS TEMOS ACESSO AOS SEUS VALORES USANDO OS MÉTODOS GET;
- O INTERPRETADOR INVOCA SEMPRE O MÉTODO MAIS LOCAL:.



CLASSES ABSTRACTAS

- COMO VIMOS ANTES COM A CLASSE FORMA E SUA HIERARQUIA, POR VEZES TEMOS DIFICULDADES EM CODIFICAR MÉTODOS QUE, DEVENDO SER DEFINIDOS PARA QUE TODAS AS SUBCLASSES OS DEVAM IMPLEMENTAR, POSSUEM UM CÓDIGO QUE NÃO TEM SENTIDO, E QUE CODIFICAMOS ASSIM APENAS PORQUE SOMOS OBRIGADOS;



QUESTÃO:

Que código reutilizável pelas suas classes pode ou deve ser colocado na superclasse, ou seja, que código é comum a todas as actuais e futuras subclasses de Forma ??

RESPOSTAS POSSÍVEIS:

1) Código "dummy" só para obrigar à sua reescrita nas várias subclasses;

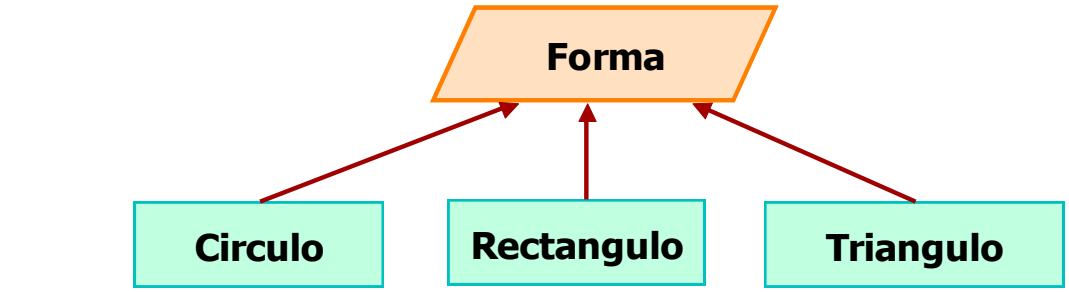
```
public double area() { return 9999.99; }
public double perim() { return -9999.99; }
```

2) Não definir NENHUM e, assim, não "comprometer" !!

apenas irá conter as assinaturas dos métodos.

NESTE CASO, A CLASSE É DECLARADA COMO **ABSTRACTA** PORQUE ALGUNS (OU TODOS) DOS SEUS MÉTODOS NÃO VÃO TER CÓDIGO MAS APENAS A SUA DEFINIÇÃO SINTÁCTICA.
ESTES MÉTODOS ABSTRACTOS SÃO HERDADOS PELAS SUBCLASSES QUE SÃO OBRIGADAS A DAR-LHES UMA IMPLEMENTAÇÃO.

```
/**
 * Abstract class Forma
 */
public abstract class Forma {
  public abstract double area();
  public abstract double perimetro();
  public abstract String toString();
  public abstract Forma clone();
 }
```



area()
perimetro()
toString()
clone()()

Esta classe **Forma** é, neste caso 100% abstracta pois não possui qualquer implementação. As subclasses devem implementar os métodos que formam a sua linguagem comum.

- A utilização da classe abstracta numa colecção não sofre qualquer alteração ao que se disse atrás e ao código atrás apresentado;

```
Forma[] formas = new Forma[50];
formas[0] = new Forma(); ……..
formas[i] = new ???();
```

ou até

```
ArrayList<Forma> formas = new ArrayList<Forma>()
formas.add(new ??);
formas.add( new ??);
```

- Uma classe abstracta pode conter variáveis próprias que serão herdadas pelas suas subclasses;
- Uma classe abstracta pode conter métodos completamente definidos e pode até conter métodos que contêm código útil comum às subclasses e que estas depois complementam;
- Tendo variáveis de instância, uma classe abstracta deverá ter construtores que inicializam estas variáveis. Estes construtores são chamados de forma natural pelos construtores das suas subclasses;
- Porém, uma classe abstracta nunca poderá criar instâncias porque, ao conter 1 ou mais métodos abstractos, estes métodos gerariam erros.
- Assim, uma classe abstracta pode ser desde 0% a 99,9% abstracta !

EXEMPLO DE CLASSE ABSTRACTA COM CÓDIGO COMUM ÚTIL:

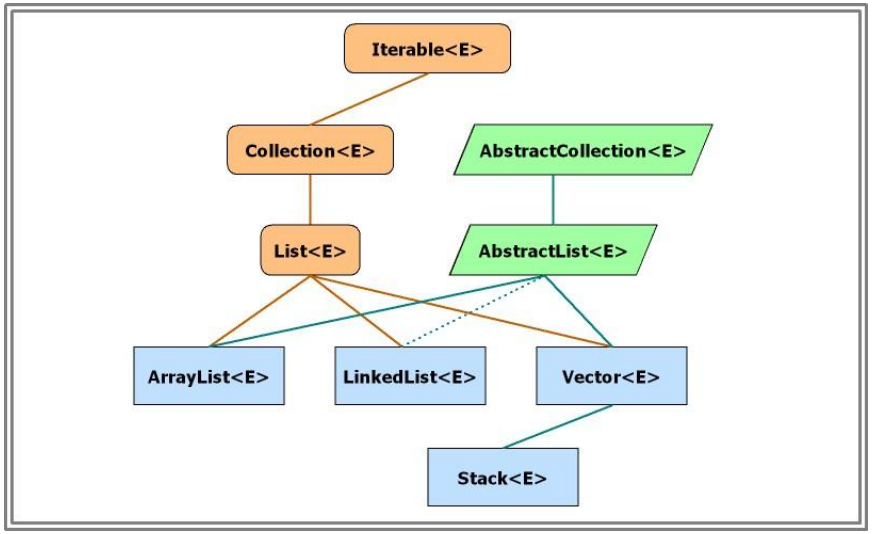
CLASSE ABSTRACTA:

Contém variáveis e métodos comuns às suas subclasses;
Como possui atributos tem construtores para os inicializar.
Também define nas suas variáveis de classe coisas comuns.

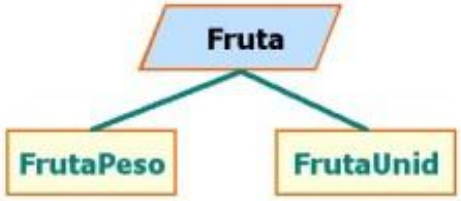
```
import java.io.Serializable;
public abstract class Empregado implements Serializable {
  // de classe
  private static double salDia = 50.00;
  public static double getSalDia() { return salDia; }
  public static void setSalDia(double nvSalDia) { salDia = nvSalDia; }
  // construtores
  public Empregado(String cod, String nom, int dias) {
     codigo = cod; nome = nom; this.dias = dias;
  }
  public Empregado(Empregado emp) {
     codigo = emp.getCodigo(); nome = emp.getNome();
     dias = emp.getDias();
  }
  // de instância
  private String codigo;
  private String nome;
  private int dias;
  // métodos concretos
  public String getNome() { return nome; }
  public String getCodigo() { return codigo; }
  public int getDias() { return dias; }
```

```
  public boolean equals(Object obj) {
    if(this == obj) return true;
    if((obj == null) || (this.getClass() != obj.getClass())) return false;
    Empregado e = (Empregado) obj;
      return nome.equals(e.getNome()) &&
             codigo.equals(e.getCodigo()) && dias == e.getDias();
  }
  // abstractos
  public abstract double salario();
  public abstract String toString();    // algum código poderia ser colocado aqui
                                        // para passer a String as variáveis de instância
  public abstract Empregado clone();
}
```

- Esta classe **Empregado** é uma classe abstracta mas possui bastante estrutura e comportamento já implementado. Tal é bastante comum nas classes abstractas de JCF, ainda que em JCF as **interfaces** sejam os verdadeiros supertipos das colecções.



OUTRO EXEMPLOS DE POLIMORFISMO E CLASSE ABSTRACTA




```
public abstract class Fruta {
  private double preco;
  private String nome;
  //
  public Fruta(String nm, double p) {
    nome = nm; preco = p);
  }
  public Fruta(Fruta f) {
    nome = f. getNome(); preco = f.getPreco();
  }
  public String getNome() { return nome; }
  public double getPreco() { return preco; }
  //
  public abstract String toString();
  public abstract Fruta clone();
  public abstract double aPagar();
}
```

```
public class FrutaPeso extends Fruta {
   private double peso;
  //
  public FrutaPeso(String nm,
                   double pr, double ps) {
    super(nm, pr); peso = ps;
  }
  public FrutaPeso(FrutaPeso f) {
    super(f. getNome(), f.getPreco());
    peso = f. getPeso();
  }
  //
  public double getPeso() { return peso; }
  public double aPagar() {
     return peso*super.getPreco();
  }
  public String toString() { ..... };
  public FrutaPeso clone() {
      return new FrutaPeso(this);
  }
}
```

```
public class FrutaUnid extends Fruta {
   private int quant;
  //
  public FrutaUnid(String nm,
                   double pr, int qtd) {
    super(nm, pr); quant = qtd;
  }
  public FrutaUnid(FrutaUnid f) {
    super(f. getNome(), f.getPreco());
    quant = f. getQuant();
  }
  //
  public int getQuant() { return quant; }
  public double aPagar() {
     return quant*super.getPreco();
  }
  public String toString() { ..... };
  public FrutaUnid clone() {
      return new FrutaUnid(this);
  }
}
```

- **VAMOS AGORA USAR ESTAS CLASSES PARA CRIAR UMA COLECÇÃO POLIMÓRFICA DE FRUTA E REALIZAR ALGUMAS OPERAÇÕES.**

```
public class Cabaz {

   private ArrayList<Fruta> cabaz = new ArrayList<Fruta>();  // cabaz polimórfico
....
   // Juntar uma Fruta ao cabaz
   public void junta(Fruta f) { cabaz.add(f.clone()); }
   // Valor total a pagar pelo cabaz
   public double aPagar() {
     double total = 0.0;
     for(Fruta f : cabaz) total += f.aPagar();
    return total;
   }
   // Total de frutos por peso
   public int numFrutosPorPeso() {
     int total = 0;
     for(Fruta f : cabaz)
         if(f instanceof FrutaPeso) total ++;  // usar apenas se não existirem subclasses !!
    return total;
   }
   // Conjunto dos nomes dos frutos do cabaz
   public TreeSet<String> nomesFrutos() {
     TreeSet<String> nomes = new TreeSet<String>();
     for(Fruta f : cabaz) nomes.add(f.getNome());
     return nomes;
   }
```

```
// Total de frutos à unidade comprados
public int totalFrutosUnidade() {
  int total = 0;
  for(Fruta f : cabaz)
      if(f instanceof FrutaUnid)
              total += ((FrutaUnid) f).getQuant();  // porque getQuant() é específico de FrutaUnid
 return total;
}
// Junta frutos ao cabaz
public void juntaAoCabaz(HashSet<Fruta> cab) {
  for(Fruta f : cab) cabaz.add(f.clone());
}
```

Questão a ver: Podemos atribuir directamente a um ArrayList<Fruta> um ArrayList<FrutaPeso> ou seja de um subtipo de Fruta??

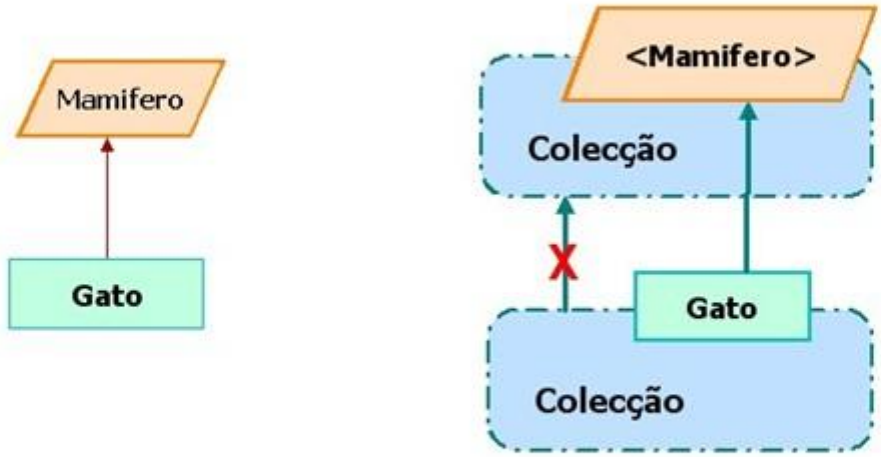
Nota: Não se trata de inserir 1 a 1 os elementos de ArrayList<FrutaPeso> usando o método junta(), mas sim realizar uma atribuição directa entre os ArrayList, por exemplo, no código,

```
ArrayList<Fruta> cabFruta = new ArrayList<Fruta>();
ArrayList<FrutaPeso> cabPeso = ArrayList<FrutaPeso>();
cabFruta.add(..); cabPeso.add(..); ……..
// e agora, depois de ter os dois
cabFruta = cabPeso; ????
```

ou tendo o método juntarCabaz(ArrayList<Fruta> cf) invocá-lo usando:

```
cabaz.juntarCabaz(cabPeso);
```

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 !
```

AS COLECÇÕES DE JAVA NÃO SÃO CO-VARIANTES, OU SEJA, MESMO QUE OS TIPOS DOS SEUS CONTEÚDOS SEJAM COMPATÍVEIS, SENDO UM SUBTIPO DO OUTRO, A COLECÇÃO DO SUPERTIPO NÃO É COMPATÍVEL COM A COLECÇÃO DO SUBTIPO.

SOLUÇÃO: USAR "WILDCARDS" O MECANISMO SINTÁCTICO DE EXTENSÃO DE TIPOS.

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

Nos exemplos anteriores, em vez de escrevermos métodos que aceitam como parâmetro uma Collection<E>, compatível com ArrayList<E>, HashSet<E> mas não com ArrayList<subtipo de E>, vamos expandir o tipo parâmetro tornando-o compatível com E e com qualquer dos subtipos de E usando o "wildcard limitado" ? extends E.

Se os métodos juntarCabaz(ArrayList<Fruta> cf) e juntarMamifero(HashSet<Mamifero> mam) forem agora definidos como:

a) public void juntarCabaz(ArrayList<? extends Fruta> cf) { ... }
b) public void juntarMamifero(HashSet<? extends Mamifero> mam) { ... }

então poderão receber como parâmetros, respectivamente,

a) ArrayList<Fruta>, ArrayList<FrutaPeso>, ArrayList<FrutaUnid>

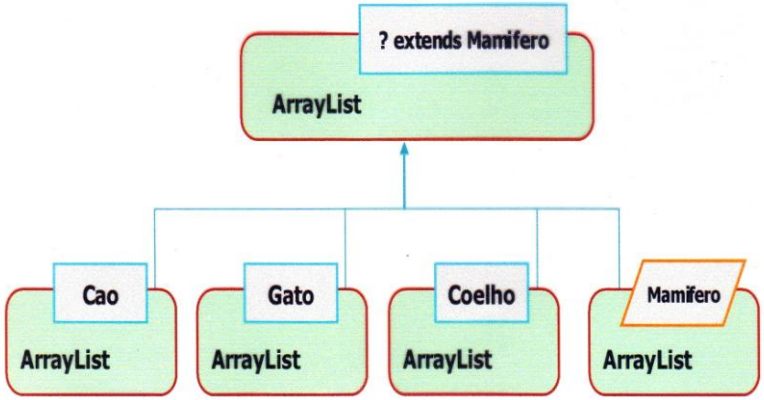
b) HashSet<Mamífero>, HashSet<Gato>, HashSet<Cao>

NOTA: wildcards não se usam em variáveis de instância !

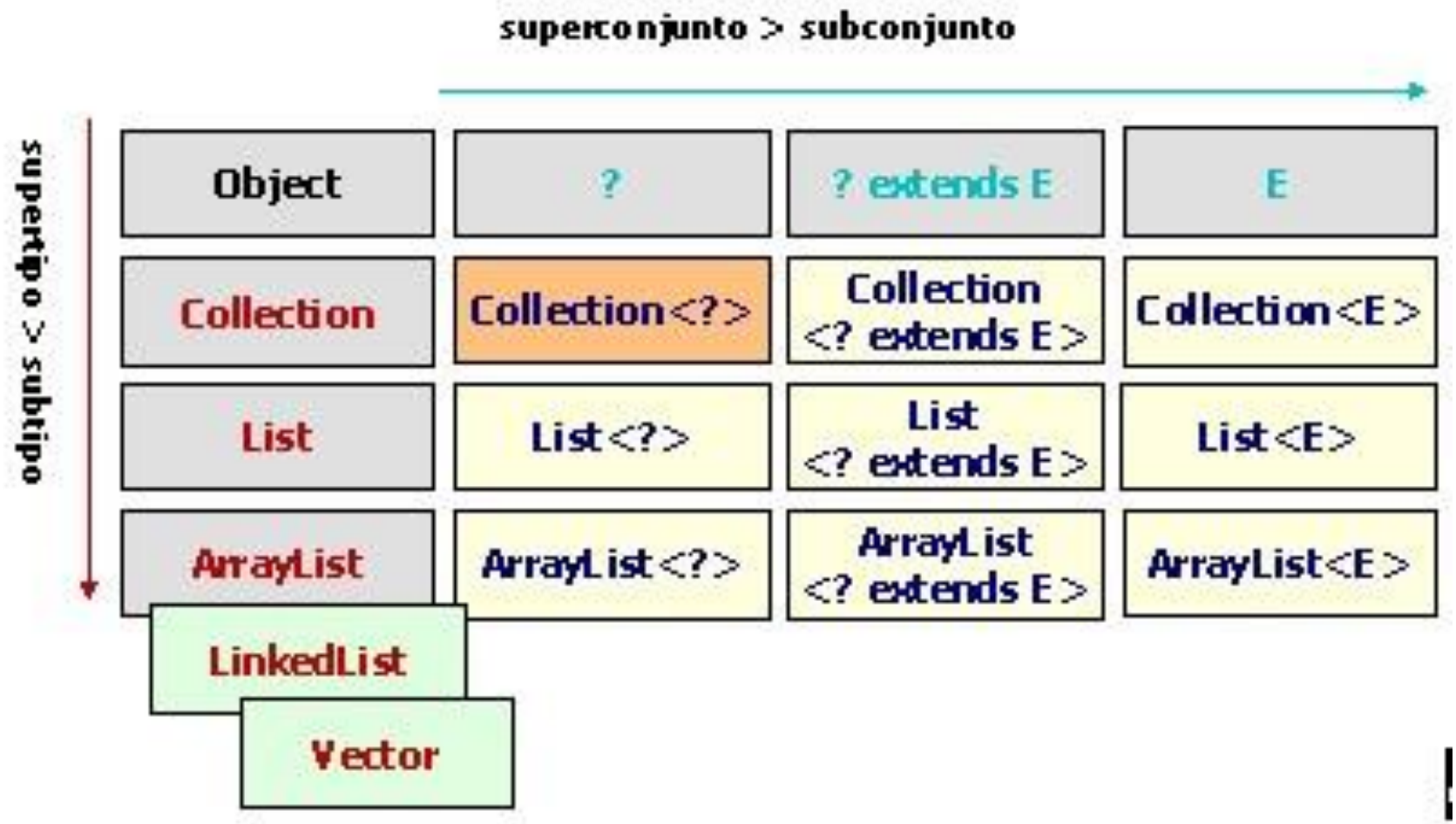
As colecções de Java Possuem APIs o mais genéricas e compatíveis possível, pelo que, para além de usarem wildcards, usam o supertipo de colecção.

```
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)
```

Collection<? extends E> é compatível com:
ArrayList<E>, HashSet<E>, TreeSet<E>, Stack<E>, LinkedList<E> e ainda de todas as outras colecções de subtipos de E.



AS COLECÇÕES DE JAVA ACABAM POR TER DUAS RELAÇÕES HIERÁRQUICAS QUE RESULTAM DA HIERARQUIA NORMAL DE CLASSES E INTERFACES E DA UTILIZAÇÃO DE WILDCARDS.



UTILIZAREMOS SEMPRE QUE NECESSÁRIO `? extends E` E EM ALGUMAS SITUAÇÕES SEREMOS OBRIGADOS A USAR `? super E`.

O WILDCARD ? (TIPO DESCONHECIDO) SERÁ DE POUCA UTILIDADE, EXCEPTO NA LEITURA E COMPREENSÃO DE ALGUM CÓDIGO EXISTENTE EM JAVA.