- List: é uma interface (mãe de todas as listas). Quando falamos de List nem sempre estamos lidando com arrays. Uma lista significa, simplesmente, que estamos armazenando elementos em sequência, ou seja, o primeiro elemento adicionado também é o primeiro que será retornado. Além disso, temos um índice, e métodos que trabalham com ele.
- ARRAYs: Um array é uma estrutura de dados e serve para guardar elementos (valores primitivos ou referências). Possui tamanho fixo, são zero-based (no caso de referência é null) e sempre iniciam com os valores padrões.

  Sintaxe:

  - Primitivo: tipo[] nome= new tipo[tamanho];
  - Literal: tipo [] nome = {1,2,3,4,5};
  - Referência: Classe [] nome = new Classe [tamanho]; (Aqui pode ser usada a classe Object que é a mais genérica possível e pode guardar qualquer tipo de referência. Como essa classe não possuirá os métodos que nós criamos é preciso fazer uma conversão ou um type cast - transformamos uma referência de um tipo mais genérico, para uma de um tipo mais específico).
    - Cast implícito:

      ```
      int numero = 3;
      double valor = numero;  //cast implícito.
      ```

    Repare que colocamos um valor da variável numero (tipo int) na variável valor (tipo double), sem usar um cast explícito. Isso funciona? A resposta é sim, pois qualquer inteiro cabe dentro de um double. Por isso o compilador fica quieto e não exige um cast explícito, mas nada impede de escrever.

Agora, o contrário não funciona sem cast, uma vez que um double não cabe em um int:

- Cast explícito:

  double valor = 3.56;

  int numero = (int) valor; //cast explícito é exigido pelo compilador.

Nesse caso o compilador joga todo valor fracional fora e guarda apenas o valor inteiro.

Nas referências, o mesmo princípio se aplica. Se o cast sempre funciona, não é necessário deixá-lo explícito.

O Type cast explícito sempre funciona? A resposta é não. O cast explícito só funciona se ele for possível, mas há casos em que o compilador sabe que um cast é impossível e aí nem compila, nem com type cast.

❖ ArrayList: implementação de uma List. É um array dinâmico, ou seja, por baixo dos panos é usado um array, mas sem se preocupar com os detalhes e limitações. Serve para guardar referências. É do pacote java.util (que encapsula o uso do array). E o único limite do ArrayList é a memória da JVM.). Para limitarmos o ArrayList podemos utilizar os Generics que parametrizam classes e assim podemos definir o tipo dos elementos. A sintaxe são os símbolos de menor e maior e, dentro, indicar o tipo de classes e objetos que aquela lista poderá armazenar. Benefícios:

- ➢ Código mais legível, já que fica explícito o tipo dos elementos;
- ➢ Evitar casts excessivos;

➢ Antecipar problemas de casts no momento de compilação.

- Para inicializar um ArrayList com limite definido usar o construtor da classe ArrayList que é sobrecarregado e possui um parâmetro que recebe a capacidade:

  ```
  ArrayList lista = new ArrayList(26); //capacidade inicial
  ```

- Para inicializar um ArrayList a partir de outro:

  ```
  ArrayList lista = new ArrayList(26); //capacidade inicial
  lista.add("RJ");
  lista.add("SP");
  //outros estados
  ArrayList nova = new ArrayList(lista); //criando baseado na primeira lista
  ```

- LinkedList: implementação de uma List. Não utiliza um array internamente, mas possui algumas características semelhantes, como, sequência, ordem de inserção e índice. Seu funcionamento ocorre da seguinte forma: ao adicionarmos, por exemplo, cc1 e, em seguida, cc2, ela se lembrará do elemento que foi adicionado anteriormente, ou seja, cc2 se lembra de cc1, cc3 de cc2, e assim por diante. Da mesma forma, o primeiro elemento se lembra daquele que o segue, ou seja, cc1 lembra de cc2, cc2 de cc3, e assim sucessivamente. A isso, damos o nome de lista duplamente encadeada.

  Diferentemente do array, não temos como acessar uma determinada posição diretamente. Se quisermos, por exemplo, acessar a posição 3, temos que iniciar na primeira e seguir, até

atingirmos a desejada. Isso faz com que a iteração seja algo negativo na LinkedList.

- Vector: implementação de uma List. Ele, na verdade, é igual a um ArrayList. Internamente, ele também utiliza um array. O Vector tem uma diferença importante em relação ao ArrayList, ele é o que chamamos de thread safe.

  Qualquer programa em Java inicia com um método main, que forma uma "pilha" e, a partir dele, podemos ter uma nova "pilha". Dessa forma, elas podem ser executadas em paralelo. O Java permite a criação de inúmeros métodos main. Quando temos esse tipo de situação, e desejamos que as execuções sejam feitas em paralelo, em cima de uma mesma lista, utilizamos o java.util.Vector. Este tipo de operação só funciona dessa forma, o ArrayList e o LinkedList não servem. A utilização dele, em si, tem um custo em desempenho. Assim, se não for estritamente necessário, é melhor utilizar outros tipos de lista, como o ArrayList, que é mais eficiente.

  O Vector é utilizado como exceção. As ocasiões em que ele é necessário são raras.