1. Tipus de dades

La majoria de llenguatges de programació classifiquen les dades de la següent manera:

Tipus de dades simples	Numèriques	*Enter, Real*
	Alfanumèriques	*Caràcter*
	Lògiques	*Booleà*
Tipus de dades compostes o estructures de dades	Estàtiques	*Vectors, matrius, registres*
	Dinàmiques	*Piles, cues, llistes, arbres, conjunts*
	Homogènies	*Vectors, matrius, piles, cues, llistes, arbres, conjunts*
	Heterogènies	*Registres*
	Accés per posició	*Vectors, matrius*
	Accés per nom	*Registres*
	Accés per clau	*Piles, cues, llistes, arbres, conjunts*

Les estructures de dades són agrupacions d'elements que alhora poden ser dades simples o altres estructures.

Classificació segons l'emmagatzemament de les dades

- Estructures de dades **estàtiques**: Es componen per un nombre fix d'elements.
- Estructures de dades **dinàmiques**: Es componen per un nombre variable d'elements. Per tant es poden afegir i treure elements en temps d'execució.

Classificació segons el tipus d'elements que les componen

- Estructures de dades **homogènies**: Els elements que les componen són tots del mateix tipus
- Estructures de dades **heterogènies**: Es componen per elements que poden ser de diferent tipus.

Classificació segons el mètode d'accés

- Per posició: Cal indicar la posició per accedir a un element.
- Per nom: Cal indicar el nom per accedir als diferents elements.
- Per clau: Cada element té una clau (hash) que permet accedir-hi.

2. Estructures dinàmiques

Una estructura dinàmica és un conjunt variable d'elements, aquests elements s'anomenen **nodes**, i estan **enllaçats** (lligats) uns amb els altres, així un node està format per,

- Un o més enllaços
- Dades, en general un objecte per node

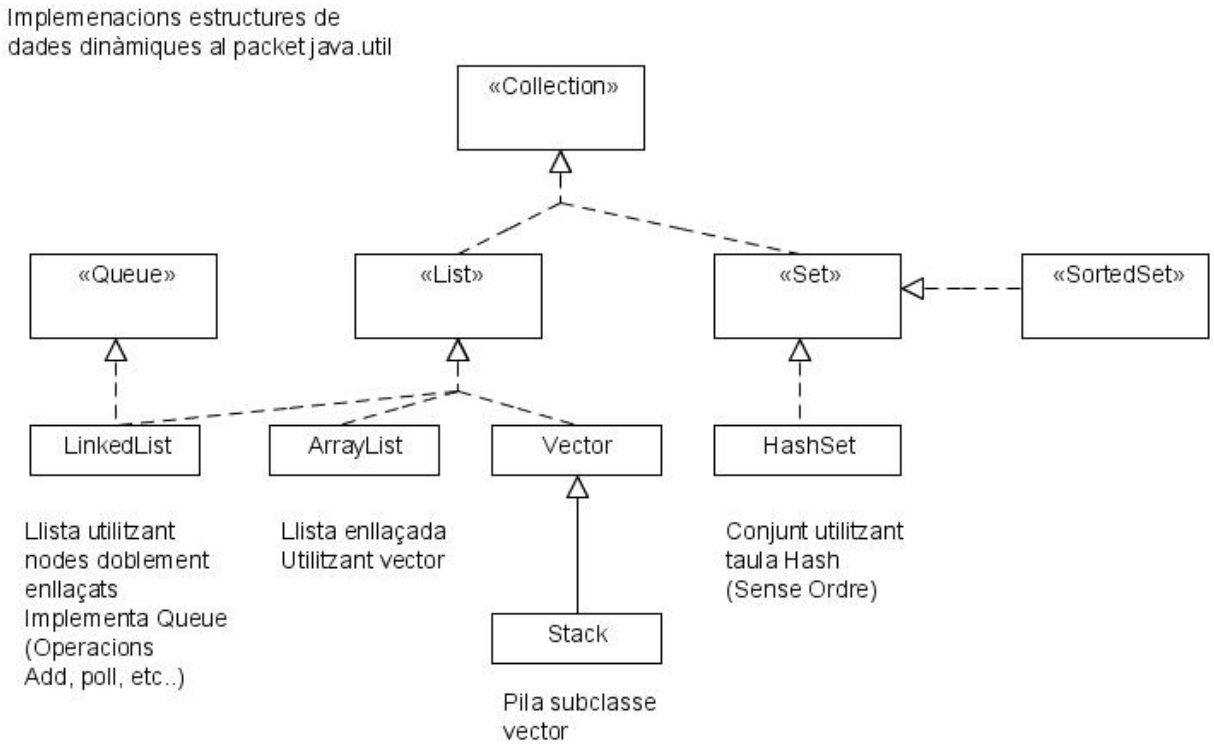
Aquestes estructures les podem recórrer, afegir i treure elements i d'altres operacions específiques.

Cal destacar que les estructures dinàmiques s'emmagatzemen en un espai diferent de memòria (heap, en comptes de a l'espai de dades), i que en principi no tenen limitacions de mida, al contrari que les estructures estàtiques.

Aquestes estructures poden ser de molts tipus diferents segons el seu comportament, el nombre d'índex que defineixen, el nombre d'elements que relacionen, però es poden agrupar segons si són:

- Estructures Lineals: cada node s'enllaça només amb el següent i opcionalment amb l'anterior
  - **Cua**. Una cua és una estructura de dades de tipus FIFO. Només es pot accedir a la cua pel final (No està permès colar-se). Els elements entren per la cua i surten pel principi.
  - **Pila**. Una pila és una estructura de dades de tipus LIFO. Només es pot accedir a la pila pel seu cim (top). Els elements entren pel cim i surten per aquest, de manera que no es pot desempilar un element mentre tingui altre elements per sobre.
  - **Llista**. Una llista és una seqüència de nodes ordenada on cada elements manté referències al següent i/o anterior. És una estructura més general que les piles o les cues ja que permet afegir i treure elements de qualsevol punt de la llista. De fet piles i cues es poden implementar a partir de les llistes.
  - **Conjunts**. Un conjunt és una col·lecció d'objectes. Aquests objectes han de ser diferenciables. En un conjunt no importa l'ordre i els elements no estan mai repetits. Es defineix el conjunt buit com el que no té cap element. C = {}, el conjunt universal aquell que els té tots. Es diu que dos conjunts són iguals si tenen els mateixos elements.
- Estructures no lineals: cada node s'enllaça amb múltiples nodes
  - **Arbres**. Els arbres per contra són estructures no lineals. En general un node té molts successors i un únic predecessor. Donada la seva estructura són de gran utilitat per a realitzar cerques amb un cost molt baix.
  - **Grafs**. Aquesta és la estructura més complexe doncs permet enllaçar un node amb qualsevol altre.

3. API. Classes d'utilitat



Aquest esquema representa de manera sintetitzada una relació de les interfases (<< ... >>) i classes que ofereix Java per treballar amb les estructures dinàmiques.

Cal notar que aquestes classes tenen moltes operacions i per tant abans de fer servir una estructura dinàmica i posar-se a implementar-la és aconsellable donar una ullada a l'API i comprovar si tot allò que necessitem ja està fet.

L'únic inconvenient és que Java no ofereix encara estructures específiques per a arbres.

Totes aquestes classes, com implementen interfaces comunes, tenen molts mètodes similars.

Collection: Interface genèrica que representa un conjunt d'elements sense ordre

- **List**: *Collection* amb ordre, admet duplicats
- **Set**: *Collection* sense ordre, sense duplicats
    - **SortedSet**: Afegeix ordre entre els elements
- **Queue**: *Collection* amb operacions per al funcionament de Cua, afegir pel final (offer), treure pel principi (poll) i consultar el cap de la cua (peek)

Com a comentari, tenir en compte que sempre que una col·lecció gestioni l'ordre dels seus elements, cal que aquests (la classe dels objectes que la componen), implementin la interface Comparable i per tant defineixin quin és l'ordre entre objectes.

Exemple:

public class Person {...} passaria a ser

public class Person implements Comparable<Person>{

...

```
	@Override
	public int compareTo(Person o) {
		// Ordena per edat i si tenen la mateixa edat a llavors per nom.
		int result = this.getAge() - o.getAge();
		if (result == 0){
			result = this.getName().compareTo(o.getName());
		}
		return result;
	}
}
```

A part de la necessitat d'establir o no un ordre entre els elements, o bé el requeriment de comportaments específics (Cua o Pila per exemple), les diferències entre unes estructures o altres resideix en temes de rendiment, per exemple unes són millors per inserció i d'altres per l'accés a un element.

Per exemple, ús de la classe *LinkedList:* afegir, treure, conté? i mida

```
	System.out.println ("Llista Java Demo");
	List l = new LinkedList();
	for (int i=0; i<=100; i++ ) {
	  l.add(new Integer(i));
	}

	System.out.println("Està 35 " + l.contains(new Integer(35)));
	System.out.println("longitud " + l.size());
	for (int i=100; i>=0; i-- ) {
	  l.remove(new Integer(i));
	}
	System.out.println("longitud " + l.size());
	System.out.println ();
```

4. Iteradors

Un Iterador és una eina que permet recòrrer una col·lecció d'elements.
Totes les estructures de l'API de java implementen la interface <<*Iterable*>>, aquesta interface obliga a crear un mètode *iterator()* que retorna un iterador sobre els elements.

Un iterador té les operacions següents:

- hasNext(): boolean per detectar si encara hi ha elements
- next(): element retorna el següent objecte de la col·lecció
- remove() esborra el darrer element retornat

El problema de iterator() és que només pot anar capo en davant i no pot visitar l'element anterior. **LisIterator** és l'iterator que si que ho pemet.

Exemple:

```
        List l = new LinkedList();
        Iterator iter = l.iterator();
        int element = 0;
        while (iter.hasNext()) {
          System.out.println("Element " + element + " " + iter.next());
          element++;
        }
```

5. Tipus generic

Les col·leccions admeten qualsevol tipus de dada.

Per evitar la problemàtica del casting, Java permet definir classes i mètodes genèrics, sense usar aquest sistema caldria treballar amb elements de la classe Object.

Per usar els tipus genèrics s'utilitza els símbols "< *tipus* >".

Les interfaces, classes i mètodes de l'arbre de les col·leccions estan adaptades per permetre l'ús dels genèrics.

A l'API es pot detectar quines classes i mètodes admeten genèrics si tenen la cadena **"<E>"**.

Exemple:

```
 public class Persona implements Comparable<Persona> {
        ...
 }

 TreeSet<Persona> persones = new TreeSet<Persona>();

 Iterator<Persona> iter = persones.iterator();

 // S'ha d'implementar "public int compareTo(Person o) {...}" per a indicar quin serà l'ordre en que
 // s'ordenaran els objectes de la classe Persona.
```

6. Control de repetits

Les implementacions de **Set** han de tenir dues propietats, sense ordre i sense duplicats ==> (HashSet)
Aquesta responsabilitat queda per al responsable de la implementació, a través dels mètodes:

- ***hashCode():int***, aquest mètode indica com calcular la clau hash per accedir a l'element (en quin calaix es guardarà). Dos elements iguals han de generar la mateixa clau hash.
- ***equals(o:Object):boolean***, aquest mètode defineix quan dos elements són iguals. 2 o més elements es poden guardar en el mateix calaix (perquè a la hora de calcular la seva clau hash ens dona el mateix valor hash) però es diferenciaran amb equals().

Exemple:

```
public class Persona {
    private String nom;
    private String dni;
    private int edat;

    public Persona(String nom, String dni, int edat) {
        super();
        this.nom = nom;
        this.dni = dni;
        this.edat = edat;
    }

    public String getNom() {
        return nom;
    }

    public String getDni() {
        return dni;
    }

    public int getEdat() {
        return edat;
    }

    // hashCode calcula la clau hash que permet emmagatzemar / recuperar un element.
    // Si dos elements tenen la mateixa clau hash van al mateix calaix.
    // 2's persones amb nom diferent tenen un hasCode diferent i per tant són diferents i no fa falta fer servir
    // el equals() per a veure si són iguals.
    public int hashCode () {
        return this.nom.hashCode();
    }
```

```
    // Sense definir "equals", no es pot determinar quan dos elements són iguals si tenen la mateixa clau
    // hash i no controlaria els repetits. Diferenciarà les persones pel dni. 2 persones amb diferent nom ja
    // són diferents perquè tenen un hash diferent i ja no fa falta fer servir el equals().
    public boolean equals(Object a) {
        return this.dni.equals(((Persona) a).getDni());
    }
}


// Ús de la classe
        Persona p;

        // Sense duplicats
        HashSet persones = new HashSet();
        persones.add(new Persona("P1", "1111111Z", 23));
        persones.add(new Persona("P2", "2222222Z", 24));
        persones.add(new Persona("P3", "3333333Z", 25));
        persones.add(new Persona("P3", "3333333Z", 25));   // Aquest està repe per culpa del hash i l'equals()
        persones.add(new Persona("P3", "6666666Z", 25));   // Aquest no està repe per culpa del l'equals()
        persones.add(new Persona("P4", "3333333Z", 25));

        // Sense ordre i sense definir operació hashCode aquesta iteració retorna ordres aleatoris
        // Cada vegada es calcula hashCode diferent
        Iterator<Persona> iter = persones.iterator();
        while (iter.hasNext()) {
            p = iter.next();
            System.out.println(p.getNom() + " " +p.getDni() + " " + p.getEdat());
        }
```

7. Ordenació

Les implementacions de **SortedSet** gestionen sense duplicats i amb ordre ==> (TreeSet)

Aquesta responsabilitat recau en el responsable de la implementació

- Interface Comparable
- mètode "compareTo(o:Object):int"

Si no s'implementa aquest mètode es genera una excepció quan es treballa amb aquesta estructura

Exemple:

```
public class Persona implements Comparable<Persona> {
    private String nom;
    private String dni;
    private int edat;

    public Persona(String nom, String dni, int edat) {
        super();
        this.nom = nom;
        this.dni = dni;
        this.edat = edat;
    }

    public String getNom() {
        return nom;
    }

    public String getDni() {
        return dni;
    }

    public int getEdat() {
        return edat;
    }

    public int compareTo(Persona p) {
        return this.nom.compareTo(p.getNom());
    }
}
```

8. Mapes

Accés indexat. Sense ordre (Per garantir l'ordre dels elements LinkedHashMap).

Exemple:

```
Map<Integer, String> exempleBuit = new HashMap<Integer, String>();
// El 1r valor és la clau que ens permetrà accedir contingut que s'emmagatzemarà en el 2n valor.

exempleBuit.put(1, "text qualsevol");
System.out.println(exempleBuit.get(new Integer(100))); // Clau no existeix

Map<Integer, String> exempleErrors = new HashMap<Integer, String>() {{
    put(100, "Aquest projecte ja existeix");
    put(101, "Cal indicar la empresa");
    put(102, "Cal indicar les tasques");
    put(103, "El salari no pot ser negatiu");
    put(104, "Cal incloure alguna tecnologia");
    put(200, "No existeix aquest projecte");
    put(400, "El valors permesos per al sexe són \"H\" i \"D\"");
    put(500, "Error desant el Curriculum");
    put(501, "Error carregant el Curriculum");
}};

exempleErrors.put(601, "Altre missatge");

System.out.println(exempleErrors.get(new Integer(100)));


Map<String, String> exempleErrorsCodi = new HashMap<String, String>() {{
    put("ERROREXI", "Aquest projecte ja existeix");
    put("ERROREMP", "Cal indicar la empresa");
    put("ERRORTAS", "Cal indicar les tasques");
}};

exempleErrorsCodi.put("ERRALT", "Altre missatge");

System.out.println(exempleErrorsCodi.get("ERRORALT"));
```

Altres mètodes per HashMap<K,V>:

- clear()
- size():int
- remove(Object key): V;
- values(): Collection<V>
- keySet(): Set<K>