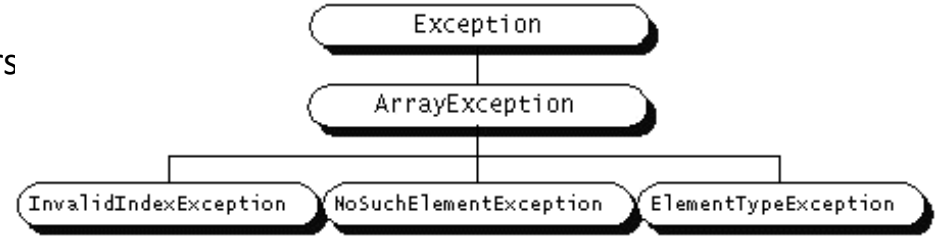
Excepciones - Genericidad – Colecciones

Agrupación Errores

- Las excepciones pueden agrupars



- Si se desea atrapar excepciones de tipo InvalidIndexException:

```
catch (InvalidIndexException e) {
  ...
}
```

- Si se desea atrapar todas las excepciones de arreglos, independiente de su tipo específico:

```
catch (ArrayException e) {
  ...
}
```

Manejo de Excepciones

```
Connection conn = null;
try {
   // conexión a base de datos
   conn = DriverManager.getConnection(…);
   // uso de conn
   // ...
} catch(SQLException e) {
   // manejo de error
   System.out.println(…);
} finally {
   // liberación de recursos
   if (conn != null) {
      conn.close();
   }
}
```

Propagación de Excepciones

- Si un método no atrapa (catch) una excepción, el método aborta, propagando la excepción

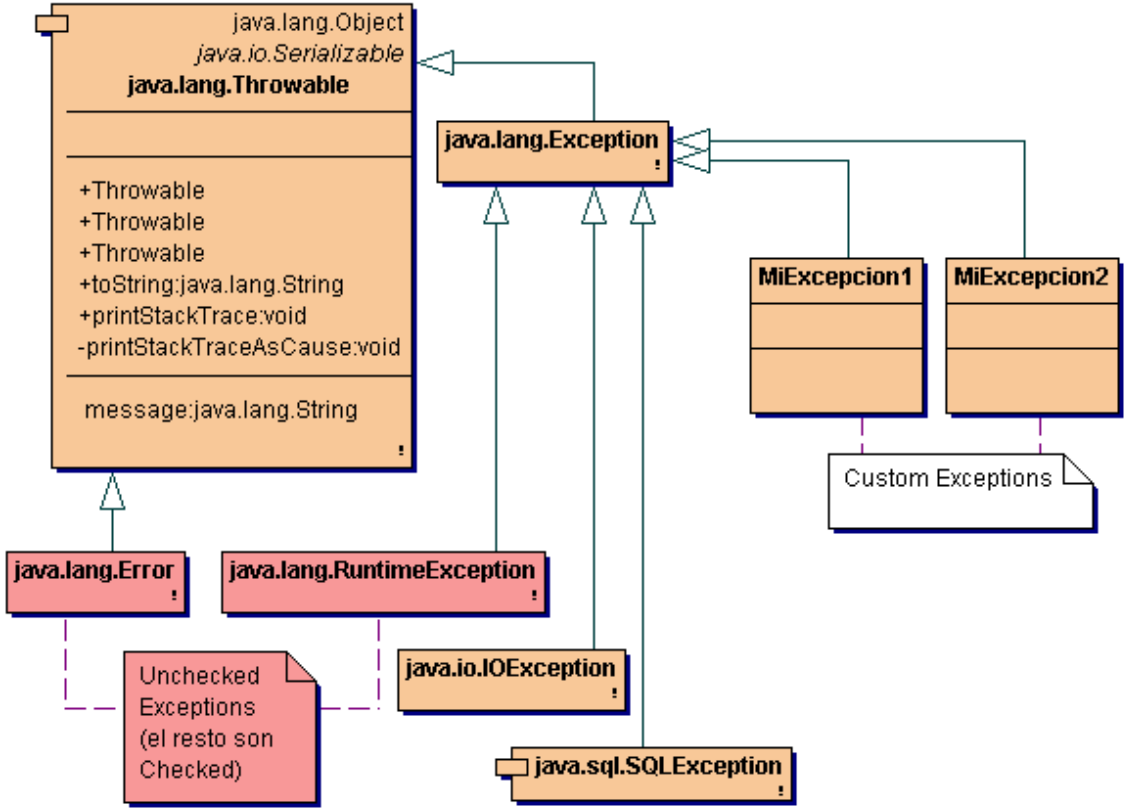
- Un método debe declarar el conjunto de excepciones "checked" que lanza o propaga, con la sentencia throws

```
void miMetodo() throws ExceptionType {
  // código que puede generar excepciones
}
```

Propagando Excepciones

```
void executeQuery() throws SQLException {
    Connection conn = null;
    try {
        // conexión a base de datos
        conn = DriverManager.getConnection(…);
        // uso de conn
        // …
    } finally {
        // liberación de recursos
        if (conn != null) {
            conn.close();
        }
    }
}
```

Jerarquía de Excepciones



Dos Tipos de Excepciones

- Excepciones "checked"
  - Si un método genera (throw) o propaga una excepción checked, debe declararlo (throws) en su firma

- Excepciones "unchecked"
  - No es necesario que un método declare (throws) las excepciones unchecked que genera (throw) o propaga (aunque puede hacerlo)

Excepciones "Checked"

- Clases derivadas de Throwable, exceptuando aquellas derivadas de Error y RuntimeException
- El compilador exige que un método declare el conjunto de excepciones "checked" que lanza o propaga

```
void f() throws IOException, SQLException {
  ...
}
```

- Ejemplos
  - FileNotFoundException
  - SQLException

Excepciones "Unchecked"

- Clases Error, RuntimeException, y derivadas
- El compilador no exige que un método declare las excepciones unchecked que genera o propaga, de modo de no complicar la programación
- Ejemplos
  - OutOfMemoryException
  - NullPointerException
  - ArrayIndexOutOfBoundsException

Creación de Excepciones

- Parte del diseño de un paquete es la definición de las excepciones que su uso puede generar
- Para crear un nuevo tipo de excepciones, debe crearse una clase derivada de Throwable
- Para definir excepciones checked, lo aconsejable es derivarlas de la clase Exception

```
public class UsuarioRequeridoException extends Exception {
    public UsuarioRequeridoException() {
        super("Debe establecerse el usuario!");
    }
}
```

- Para definir excepciones unchecked, lo aconsejable es derivarlas de la clase RuntimeException

Lanzamiento de Excepciones

- Para lanzar una excepción se utiliza la sentencia throw

```
void generaReporte() throws UsuarioRequeridoException
{
   ...
   if (usuario == null) {
      throw new UsuarioRequeridoException();
   }
   ...
}
```

Resumen

- Java permite manejar los errores de una manera cómoda y segura, utilizando excepciones
- Las excepciones son clases derivadas de la clase Throwable
- El bloque try-catch-finally permite programar separadamente el código normal y el manejo de errores
- Las excepciones no atrapadas en un bloque catch son automáticamente propagadas al método "anterior" en el stack de llamadas

Resumen

- Si una excepción no es atrapada en un programa, éste aborta
- Un método debe declarar en la cláusula throws de su firma el conjunto de excepciones "checked" que lanza o propaga, lo que no es necesario para las excepciones "unchecked" (derivadas de las clases Error y RuntimeException)
- Se recomienda que las excepciones propias se deriven de las clases Exception (checked) o RuntimeException (unchecked)
- Para lanzar una excepción se utiliza la sentencia throw

Genericidad

- **Genericidad**:
  - Definición de clases genéricas.
  - Declaración y construcción de tipos genéricos.
  - Genericidad y sistema de tipos.
  - Genericidad y tipos dinámicos.
  - Métodos genéricos.
  - Características avanzadas:
    - Genericidad restringida.
    - Declaración de tipos *puros*.
    - Genericidad y herencia.

Genericidad

- Facilidad de un lenguaje de programación para definir **clases, interfaces y métodos parametrizados con tipos** de datos.
- Resultan de utilidad para la implementación de **tipos de datos contenedores** como las **colecciones** (`List<T>`, `HashSet<T>`)
- La genericidad sólo tiene sentido en **lenguajes con comprobación estática de tipos**, como Java.
- **La genericidad permite escribir código reutilizable.**

Definición de clase genérica

- Una **clase genérica** es una clase que en su declaración utiliza un tipo variable (**parámetro**), que será establecido cuando sea utilizada.

- Al **parámetro** de la clase genérica se le proporciona un nombre (`T`, `K`, `J`, etc.) que permite utilizarlo como **tipo de datos en el código de la clase**.

- Sobre las variables cuyo tipo sea el parámetro (`T`, `K`, `J`, etc.) **sólo es posible aplicar métodos de la clase `Object`**:
  - dado que representan “cualquier dato” sólo podemos aplicar operaciones disponibles en todos los tipos de datos del lenguaje Java.

Clase genérica Contenedor

```
public class Contenedor<T> {

        private T contenido;

        public void setContenido(T contenido) {

                this.contenido = contenido;
        }

        public T getContenido() {

                return contenido;
        }
}
```

Operaciones disponibles

- Sobre una variable de tipo T, sólo podemos aplicar **métodos públicos de la clase `Object`**
  - **Nota**: el método `clone()` no es público en la clase Object.

- También podemos utilizar la **asignación** (=) y la comparación de **identidad** (== o !=).

- Dentro de la clase genérica, **NO es posible construir objetos de los tipos parametrizados**:

  - `T contenido = new T(); // No compila`

Ejemplo: hashCode y equals en Contenedor

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((contenido == null) ? 0 :
                               contenido.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
      if (this == obj) return true;
      if (obj == null) return false;
      if (getClass() != obj.getClass()) return false;

      Contenedor<T> other = (Contenedor<T>) obj;

      return contenido.equals(other.contenido);
}
```

Uso de una clase genérica

- La **parametrización** de una clase genérica se realiza en la **declaración** de una variable y en la **construcción de objetos**.

```
Contenedor<String> contenedor =
            new Contenedor<String>();

contenedor.setContenido("hola");
```

Genericidad y tipos primitivos

- Las clases genéricas **no pueden ser parametrizadas a tipos primitivos**.
- Para resolver este problema el lenguaje define **clases envoltorio** de los tipos primitivos:
  - **`Integer`**, **`Float`**, **`Double`**, **`Character`**, **`Boolean`**, etc.
- El compilador transforma automáticamente tipos primitivos en clases envoltorio y viceversa: **autoboxing**.

```
Contenedor<Integer> contenedor =
                    new Contenedor<Integer>();
contenedor.setContenido(10);
int valor = contenedor.getContenido();
```

Genericidad y sistema de tipos

```
List<Burbuja> burbujas = new
              LinkedList<Burbuja>();

              //OK
Collections.addAll(burbujas, basica, limitada, debil,
                                                creciente,
sensible);


LinkedList<BurbujaDebil> debiles =
                                                    new
LinkedList<BurbujaDebil>();

Collections.addAll(debiles, debil, creciente);


burbujas = debiles;          //error en tiempo de
                             compilación
```

Genericidad y sistema de tipos

- Las reglas del **polimorfismo** se mantienen entre clases genéricas.

- Sin embargo, en una asignación polimórfica **no está permitido que tengan distintos parámetros**.
- En el ejemplo,
  - `LinkedList` es compatible con `List` y han sido parametrizadas al mismo tipo (`Burbuja`).
  - En la última asignación, aunque `LinkedList` es compatible con `List`, están parametrizadas a tipos distintos (`Burbuja` y `BurbujaDebil`).
    - No importa que `Burbuja` y `BurbujaDebil` sean compatibles.

- Es una **limitación en el paso de parámetros**.

Genericidad y sistema de tipos

- **Problema**: el método sólo permite variables cuyo tipo estático sea compatible con `List<Burbuja>`.

```
public class PruebaSimulador {

  private static void simular(List<Burbuja> burbujas){
    Simulador simulador = new Simulador(710, 710);

    for (Burbuja burbuja : burbujas)
         simulador.simular(burbuja);
  }
}
```

- ¿Cómo podemos pasar una variable de tipo `List<BurbujaDebil>`?

Genericidad y sistema de tipos

- **Solución**: **tipo comodín**.

```
private static void simular(
                          List<? extends Burbuja> burbujas){
```

- En el ejemplo significa: permite cualquier lista genérica parametrizada a la clase `Burbuja` o a un tipo compatible (subclase).
- El tipo comodín se puede usar **también para declarar variables locales o atributos**.
- **No se puede utilizar para construir objetos**.
- Si se indica simplemente `<?>`, significa “cualquier tipo”.

Genericidad – Tipos dinámicos

- En **tiempo de ejecución** se pierde la información sobre el tipo utilizado para parametrizar la clase genérica.
  - Toda clase genérica parametrizada (`Contendor<String>, LinkedList<Burbuja>`) se transforma a un ***tipo puro*** (`Contenedor`, `LinkedList`).
  - El código compilado (*bytecodes*) sólo contiene clases, interfaces y métodos ordinarios, sin parámetros.

- En **tiempo de ejecución** no se puede consultar el parámetro al que fue instanciada una clase genérica.

Genericidad – Tipos dinámicos

- Sólo podemos comprobar si el tipo dinámico de una variables es del tipo puro o compatible con él.

```
// No compila
if (contenedor instanceof Contenedor<Burbuja>) { … }

// Sí compila
if (contenedor instanceof Contenedor) { … }

//Sí compila
if (contenedor.getClass() == Contenedor.class)
```

Genericidad – Conversión de tipos

- Una conversión de tipos a un tipo genérico lo marca el compilador como un *warning*.
  - No se puede comprobar el tipo utilizado para la parametrización.

```
public static void main(String[] args) {
     LinkedList<Punto> puntos = new LinkedList<Punto>();

     // ... se crean y añaden objetos punto a la lista


     @SuppressWarnings("unchecked")
      LinkedList<Punto> copia =
                          (LinkedList<Punto>)puntos.clone();
}
```

Métodos genéricos

- Un método que declara una variable de tipo (por ejemplo, `<T>`) se denomina **método genérico**.
- Antes de la declaración del tipo de retorno del método se indica una variable que representa el tipo (`<T>`).
- El alcance de la variable de tipo (`<T>`) es local al método, esto es, puede aparecer en la signatura del método y en el cuerpo del método.
- Es posible definir métodos genéricos incluso en clases que no son genéricas. Por ejemplo, la clase **`Collections`** tiene métodos genéricos y no es genérica.

Métodos genéricos - Ejemplo 1

- Método que acepta una secuencia de valores de cualquier tipo y lo convierte en una lista:

```
public static <T> List<T> asList(T... datos) {
    List<T> lista = new ArrayList<T>(datos.length);

    for (T elemento : datos)
        lista.add(elemento);

    return lista;
}
```

Métodos genéricos - Ejemplo 1

- El método `asList` se podría invocar como sigue:

```
public static void main(String[] args) {

    List<Integer> listaEnteros = asList(1,2,3);

    String[] arrayPalabras = {"hola", "ciao", "hello"};

    List<String> listaPalabras = asList(arrayPalabras);
}
```

- **El tipo de `T` se infiere** a partir del tipo de los argumentos o la variable a la que se asigna el resultado.

Métodos genéricos - Ejemplo 2

- Añade una secuencia variable de elementos a una lista:

```
public static <T> void addAll (List<T> lista,
                               T... elementos) {
     for (T elemento : elementos)
             lista.add(elemento);
}
```

```
List<Integer> enteros = new ArrayList<Integer>();
addAll(enteros, 1, 2, 3);
```

Métodos genéricos - Ejemplo 3

- El siguiente ejemplo declara un método genérico que retorna un elemento aleatorio de cualquier lista:

```
public static <T> T getElementoAleatorio(List<T> lista){
    Random random = new Random();
    int index = random.nextInt(lista.size());
    return lista.get(index);
}

// Programa

List<Integer> enteros = new ArrayList<Integer>();
addAll(enteros, 1, 2, 3);

int entero = getElementoAleatorio(enteros);
```

Genericidad – Características avanzadas

- Dentro de una clase genérica se pueden utilizar otras clases genéricas.
- Una clase genérica puede tener **varios parámetros**.

```
public class ContenedorDoble <T,K> {
    private String nombre;
    private Contenedor<T> clave;
    private K valor; … }
```

```
ContenedorDoble<String, Cuenta> contenedor = …
```

- Una **interfaz** también puede declarar parámetros:
  - Un ejemplo son las interfaces que definen las colecciones (`List<T>`, `Set<T>`, etc.).

Genericidad – Características avanzadas

- Se puede restringir el conjunto de clases que se pueden utilizar para parametrizar un tipo genérico (**genericidad restringida**).
- Es posible utilizar una clase genérica y no establecer sus parámetros (**tipo puro**).
- Se puede aplicar **herencia** con clases genéricas.

Genericidad restringida

- **Objetivo**: limitar los tipos a los que puede ser parametrizada una clase genérica.

- Al restringir los tipos obtenemos el **beneficio** de poder **aplicar métodos** (además de los de `Object`) **sobre los objetos del tipo parametrizado**.

- Una clase con genericidad restringida sólo permite ser parametrizada con tipos **compatibles** **con el de la restricción** (clase o interfaz).

Genericidad restringida

- **Ejemplo**: la clase `Escenario` sólo puede ser parametrizada con tipos compatibles con `Animable`.
- `Animable` es una interfaz que declara los métodos necesarios para animar un elemento:

```
public class Escenario<T extends Animable> {

        private LinkedList<T> elementos;
        …
        public void accion() {
                for (T elemento : elementos)
                        elemento.animar();
        …}
}
```

Genericidad restringida

- **Ejemplo**: Si queremos utilizar burbujas en el escenario, debemos hacer que la clase `Burbuja` implemente la interfaz `Animable`:
  - La implementación de `animar` podría llamar a `ascender`.
- Una clase genérica puede estar **restringida por varios tipos**:

```
public class Escenario<T extends Animable & Atrapable>
```

- Las operaciones disponibles para objetos de tipo `T` es la unión de todos los tipos de la restricción.
  - En el ejemplo, todas las operaciones de la interfaz `Animable` y la interfaz `Atrapable`.

Genericidad – Tipo puro

- Cuando se declara una variable cuyo tipo se corresponde con una clase genérica y no se especifica el parámetro se asigna el **tipo puro** (*raw*) que corresponde a:
  - Sin genericidad restringida, la clase `Object`.
  - Con genericidad restringida, la clase a la que se restringe.

```
Contenedor contenedor = new Contenedor();      // Object
Escenario escenario = new Escenario();         // Animable
```

  - Siendo:
    - Clase `Contenedor<T>`
    - Clase `Escenario<T extends Animable>`

Genericidad y herencia

- Una clase puede heredar de una clase genérica.
- Una clase puede implementar una interfaz genérica.
- En cualquiera de los dos casos, si no se establece el tipo del parámetro, la clase descendiente sigue siendo genérica.
- Al heredar se puede reducir el número de parámetros.

Ejemplo herencia

```
public class Par<T, P> {
     private T valor1;
     private P valor2;

     public Par(T valor1, P valor2) {
          this.valor1 = valor1;
          this.valor2 = valor2;
     }

     public T getValor1() { return valor1; }

     public P getValor2() {  return valor2; }

     public void setValor1(T valor1) { this.valor1 = valor1; }

     public void setValor2(P valor2) { this.valor2 = valor2; }
     //continúa ...
}
```

Ejemplo herencia

```
public class Par<T, P> {

     //...

     @Override
     public String toString() {
            return getClass().getName()
                        + "[valor1=" + valor1
                        + ", valor2=" + valor2
                        + "]";
     }
}
```

Ejemplo herencia

```
public class ParUniforme<T> extends Par<T, T> {

    public ParUniforme(T valor1, T valor2) {
        super(valor1, valor2);
    }

    public boolean contiene(T valor) {
        return getValor1().equals(valor)
            ||
            getValor2().equals(valor);
    }
}
```

Ejemplo herencia

```
public class ParEntero extends ParUniforme<Integer>{

        public ParEntero(Integer valor1, Integer valor2)
                { super(valor1, valor2);
        }

        public ParEntero suma(ParEntero
                otroPar){ return new
                ParEntero(
                        getValor1() +
                        otroPar.getValor1(),
                        getValor2() +
                        otroPar.getValor2());
        }
}
```

Ejemplo herencia

```
public class PruebaPar {
   public static void main(String[] args) {

      ParEntero par1 = new ParEntero(3,5);

      ParUniforme<Integer> par2 = par1;

      System.out.println(par2.contiene(5));

      ParEntero par3 = new ParEntero(2,6);

      System.out.println(par1.suma(par3));

   }
}
```

Colecciones

Introducción

- Este tema introduce la biblioteca de **colecciones** de Java y el concepto de **genericidad**.
- La genericidad es el mecanismo que permite programar una clase como `LinkedList` de modo que pueda ser utilizada con cualquier tipo de datos.
- El tipo al que se *parametriza* la colección se especifica entre <>:

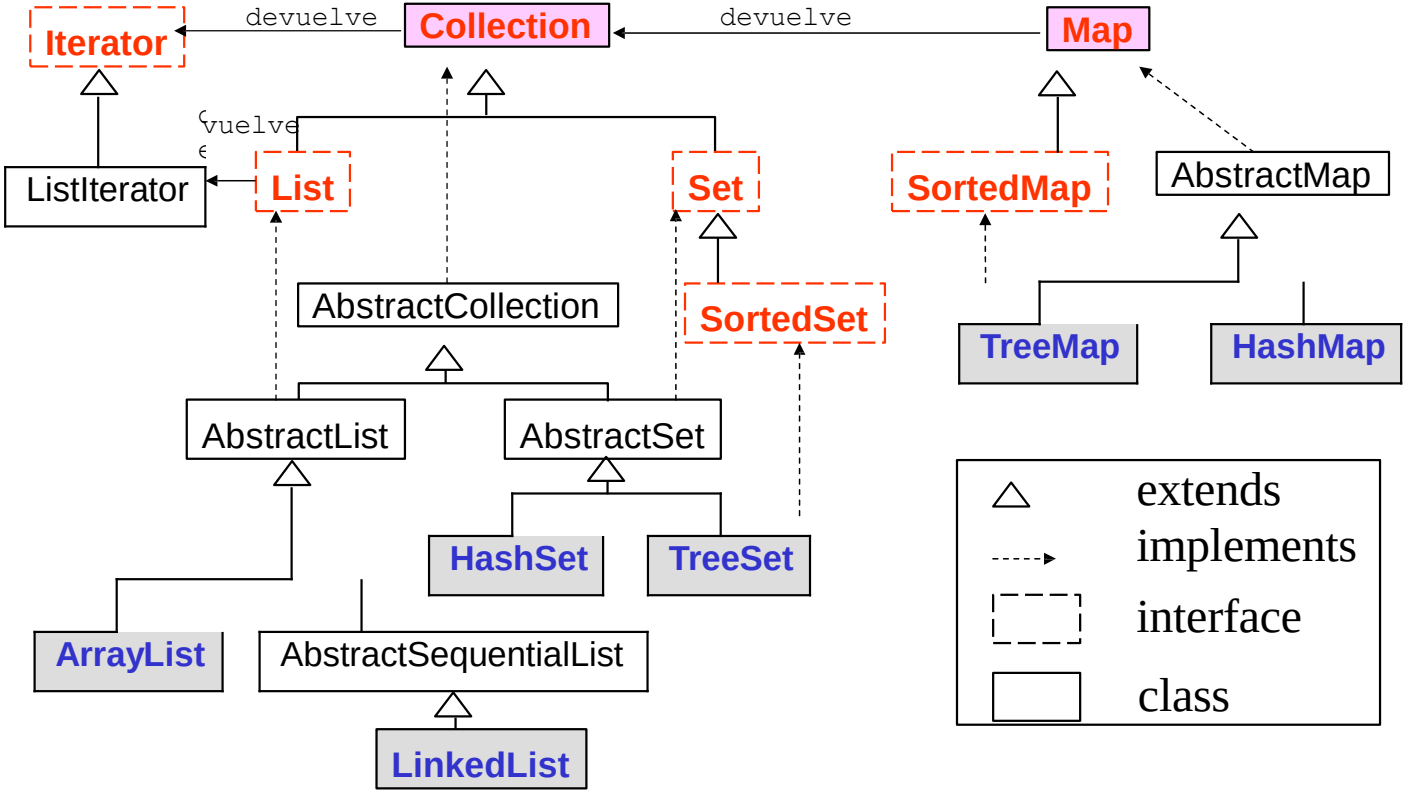
```
LinkedList<Punto> puntos = new LinkedList<Punto>();
```

- En caso de utilizar **tipos primitivos**, es necesario hacer uso de su tipo envoltorio: `Integer` para `int`, etc.
- En el tema se presentan primero las colecciones y después el mecanismo de genericidad.

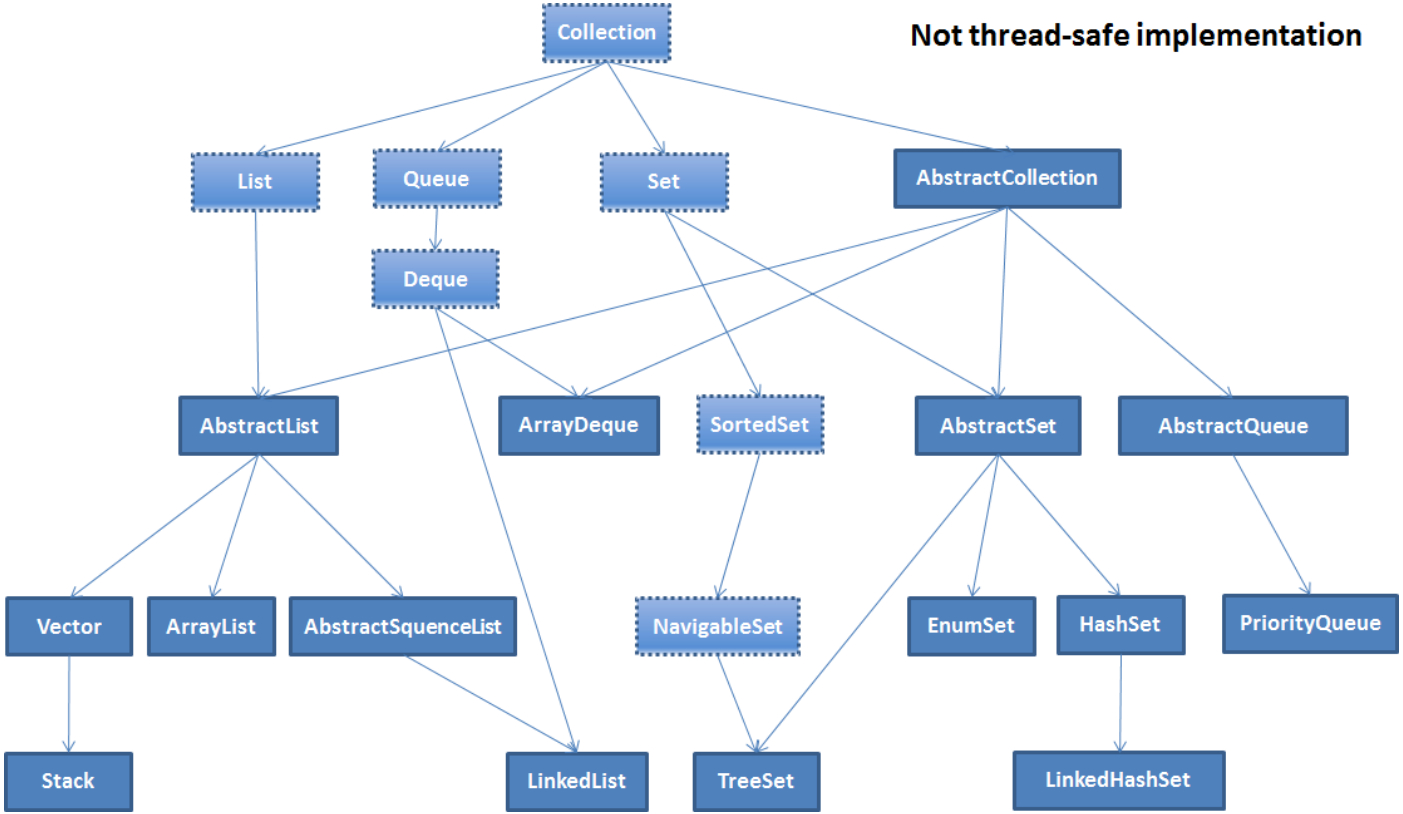
Parte 1 - Colecciones en Java

- Las colecciones en Java son un **ejemplo** destacado de implementación de **código reutilizable** utilizando un lenguaje orientado a objetos.
- Todas las colecciones son **genéricas**. Están disponibles en el paquete `java.util`.
- Los tipos abstractos de datos se definen como **interfaces**.
- Se implementan **clases abstractas** que permiten factorizar el comportamiento común a varias implementaciones.
- Un mismo **TAD** puede ser implementado por varias clases
  `List`: **`LinkedList`**, **`ArrayList`**

Colecciones en Java



Colecciones



Interfaz `Collection<T>`

- Define las operaciones comunes a todas las colecciones de Java.
- Permite usar colecciones basándonos en su interfaz en lugar de en la implementación.
- Los tipos básicos de colecciones son (subtipos de `Collection<T>`) :
  - Listas, definidas en la interfaz `List<T>`
  - Conjuntos, definidos en la interfaz `Set<T>`

Interfaz `Collection<T>`

- **Operaciones básicas de consulta**:
  - `size()`: devuelve el número de elementos.
  - `isEmpty()`: indica si tiene elementos.
  - `contains(Object e)`: indica si contiene el elemento pasado como parámetro.

Interfaz `Collection<T>`

- **Operaciones básicas de consulta:**
  - Observa que el método **`contains`** recibe como parámetro un objeto compatible con `Object`, es decir, de cualquier tipo de datos.

  - Es un error frecuente consultar en una colección si existe un elemento que no es del tipo de la colección.

```
LinkedList<String> lista = new LinkedList<>();
// … añadir cadenas

lista.contains(10); // ¿compila?
```

Interfaz `Collection<T>`

- **Operaciones básicas de consulta:**
  - Los tipos de colecciones (listas, conjuntos) determinan de forma distinta cuándo un elemento está en la colección.
  - Por ejemplo, las listas buscan los elementos utilizando el método `equals`.
  - En cambio, los conjuntos implementados con tablas de dispersión utilizan el código de dispersión (`hashcode`) y `equals`.

Interfaz `Collection<T>`

- **Operaciones básicas de modificación**:
  - `add(T e)`: añade un elemento a la colección.
    - Retorna un **booleano** indicando si acepta la inserción.
    - Las listas siempre aceptan los elementos, por tanto, siempre retornan verdadero.
    - Sin embargo, los conjuntos retornan un valor falso si el elemento está repetido.
  - `remove(Object e)`: intenta eliminar el elemento.
    - Retorna un booleano indicando si ha sido eliminado.

Nota: al igual que sucede con `contains`, para ambas operaciones, listas y conjuntos determinan si un elemento está en la colección de forma diferente.

Interfaz `Collection<T>`

- **Operaciones básicas de modificación**:
  - **`clear`**`()`: elimina todos los elementos.
  - **`addAll`**`(otra)`: añade todos los elementos de la colección `otra`.
  - **`removeAll`**`(otra)`: elimina los elementos de la colección (objeto receptor) que estén contenidos en la colección establecida como parámetro (`otra`).

Interfaz `List<T>`

- La interfaz **`List<T>`** define **secuencias** de elementos a los que se puede acceder atendiendo a su posición.
- En la librería se ofrecen varias implementaciones: `LinkedList<T>` y `ArrayList<T>`.

- Las posiciones válidas van de `0` a `size()-1`.
  - En caso de acceso fuera de rango, se produce un error de ejecución.
- El método **`add`**`(T e)`:
  - Añade al elemento al final de la lista.
  - Siempre acepta la inserción y retorna verdadero.

Interfaz `List<T>`

- Añade a las operaciones de `Collection` métodos de acceso por posición como:
  - `T get (int indice)`
  - `T set (int indice, T nuevo)`
    - Reemplaza el elemento situado en la posición `indice` por el elemento `nuevo`.
    - Retorna el elemento que ha sido sustituido.
  - `void add (int indice, T nuevo)`
    - Sitúa en la posición indiceel elemento nuevo. Los
    - elementos que estuvieran situados en indicey posteriores, son desplazados a la derecha.
  - T remove (int indice)
    - Elimina el elemento en la posición indicey lo retorna.
    - Los elementos situados en las posiciones indice + 1y siguientes son desplazados a la izquierda.

Lista `LinkedList<T>`

- La clase `LinkedList<T>` ofrece una implementación basada en **listas de nodos doblemente enlazados**

- Añade a la interfaz `List` operaciones para gestionar los extremos de la lista:
  - `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst` y `getLast`
  - Con ellas podemos simular **pilas** y **colas**.

- Interesa utilizar `LinkedList` cuando la lista irá creciendo dinámicamente por los extremos.

Lista `LinkedList<T>`

- **Ventajas**:
  - Las operaciones en los extremos de la lista (principio y final) son eficientes (orden constante).

- **Desventajas**:
  - Las consultas por posición son lentas.
  - Para obtener el elemento en la posición `i` es necesario recorrer los `i-1` nodos precedentes.
  - Inserciones y modificaciones en posiciones intermedias tampoco son eficientes.

Lista `ArrayList<T>`

- La clase `ArrayList<T>` implementa la interfaz `List<T>` utilizando **arrays redimiensionables:**
  - Inicialmente tiene una *capacidad* (tamaño del array).
  - Cuando se agota la capacidad, construye un nuevo array de mayor tamaño y se copian los elementos del antiguo.

```
Construye un lista con capacidad inicial 10
ArrayList<String> lista = new ArrayList<String>(10);

// Añadimos 3 elementos, tamaño (size) es 3
Collections.addAll(lista, "hola", "hello", "hallo");
```

- **Nota**: `java.utils.Collections` ofrece rutinas de utilidad sobre las colecciones. El método `addAll` permite añadir una secuencia de elementos (argumento de tamaño variable).

Lista `ArrayList<T>`

- **Ventajas**:
  - Las consultas por posición son rápidas (orden constante).
  - Las **inserciones por el final** (`add`) también son eficientes si no se desborda la capacidad.

- **Desventajas**:
  - Las operaciones de inserción y modificación en posiciones intermedias son ineficientes:
    - Por ejemplo, eliminar el primer elemento supone tener que desplazar el resto de elementos del array.
  - Las operaciones de inserción pueden desbordar la capacidad y tener que reconstruir el array.

Lista `ArrayList<T>`

- Interesa utilizar `ArrayList` cuando conocemos a priori el tamaño que tendrá la lista y la colección solo será consultada.
- Ejemplo:

```
class Procesador {
   private ArrayList<String> lista;

   public Procesador(String... palabras) {
      // Conocemos la capacidad de la lista
      this.lista = new ArrayList<String>(palabras.length);

      for (String palabra : palabras) {
         // Inserción eficiente, no supera capacidad
          this.lista.add(palabra);
      }
   }
}
```

Búsqueda y borrado en listas

- En las listas, la operación de **búsqueda** (`contains`) y **borrado** (`remove`) de objetos utiliza el método **`equals`** para localizarlos.
- En general ambas operaciones no son eficientes:
  - Se recorre la colección desde el principio hasta localizar el objeto.
  - Las implementaciones `ArrayList<T>` y `LinkedList<T>` tienen similar rendimiento para localizar el objeto.
  - Sin embargo, en la operación de borrado, `LinkedList<T>` es más eficiente porque sólo requiere eliminar un nodo.
  - En cambio, `ArrayList<T>` puede necesitar desplazar objetos en el array.

Interfaces `Set<T>` y `SortedSet<T>`

- La interfaz **`Set<T>`** define **conjuntos**, esto es, colecciones con elementos no repetidos.
  - Esta interfaz extiende la interfaz `Collection<T>`. Sin embargo, no añade nuevas operaciones.
  - Precisa la semántica del método **`add()`** para indicar que no se admiten elementos repetidos.

- La interfaz `Set<T>` es especializada por la interfaz **`SortedSet<T>`**, que define **conjuntos ordenados**.

Interfaces `Set<T>` y `SortedSet<T>`

- La librería de Java ofrece dos implementaciones de conjuntos:
  - **`HashSet<T>:`** conjunto implementado con tablas de dispersión.
  - **`TreeSet<T>:`** **conjunto ordenado** implementado con árboles binarios de búsqueda balanceados
    - Para su funcionamiento es necesario definir un **orden** (se estudia más adelante).

Conjunto `HashSet<T>`

- La clase **`HashSet<T>`** implementa un conjunto utilizando una **tabla de dispersión**.

- Para su correcto funcionamiento exige que la clase de los objetos que se almacenan en la colección ofrezca una **implementación consistente de los métodos `equals()` y `hashCode()`**:

  - Si `o1.equals(o2) == true` entonces `o1.hashCode() == o2.hashCode()`

Conjunto `HashSet<T>`

- La clase **`HashSet<T>`** determina si un **objeto está repetido** en el conjunto utilizando los métodos `equals/hashCode`.
- Al insertar un nuevo objeto solicita su código de dispersión (método `hashCode`):
  - 1. Si el código de dispersión no está en la tabla, entonces se inserta el objeto (no está repetido).
  - 2. Si el código está en la tabla (colisión), consulta si el objeto con el mismo código de dispersión que ya está en la tabla es igual (`equals`) que el objeto que se quiere insertar.
  - 3. En caso de ser iguales, se descarta por ser repetido.
- En definitiva, la operación **`add()`** es **eficiente**.

Conjunto HashSet<T>

- A diferencia de las listas, **los elementos de un conjunto no se pueden recuperar por posición**.
- Para recuperar los elementos de un conjunto es necesario utilizar un *iterador* (se estudia más adelante) o su versión alternativa mediante un **recorrido *for each***:

```
HashSet<String> conjunto = new HashSet<String>();
Collections.addAll(conjunto, "Juan", "Luis", "Pedro", "Juan");
System.out.println(conjunto.size()); // 3, ha rechazado uno

for (String nombre : conjunto) {
    System.out.println(nombre); // Luis, Pedro y Juan
}
```

Conjunto `HashSet<T>`

- El orden en el que se recuperan los objetos de un conjunto durante el **recorrido** no necesariamente coincide con el orden en el que fueron insertados.
  - Se obtienen en el orden en el que aparecen en la tabla de dispersión.
  - En el ejemplo anterior el orden de recorrido no coincide con el orden en el que se han insertado.

- Los **conjuntos ordenados** (`TreeSet<T>`) mantienen los objetos ordenados (se estudia más adelante).

Conjunto `HashSet<T>`

- Las operaciones de **consulta** (`contains`) y de **borrado** (`remove`) son **eficientes**.

- Se utiliza el código de dispersión para localizar el objeto en la tabla:
  - Un elemento está en el conjunto (`contains`) si el código de dispersión está registrado en la tabla y el elemento asociado al código es `equals` al elemento que se busca.
  - Borrar un elemento implica buscarlo (similar a `contains`) y si se localiza eliminarlo de la tabla.

Ordenación de una lista

- La librería de Java ofrece la clase `java.util.Collections` con rutinas de utilidad para gestionar colecciones.
- El método **`sort`** ordena una lista.
- El siguiente ejemplo muestra la ordenación de una lista de cadenas:

```
LinkedList<String> saludos = new LinkedList<>();
Collections.addAll(saludos, "hola", "hello", "hallo");

Collections.sort(saludos);

for (String saludo : saludos) {
    System.out.println(saludo); // hallo, hello, hola
}
```

Ordenación de una lista

- El método `sort` es capaz de ordenar una lista si el tipo de datos de los objetos que contiene es *comparable*.
- Una clase es comparable si implementa la interfaz **`Comparable<T>`**:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

  - El método `compareTo` compara el objeto receptor y el parámetro. Devuelve un entero positivo si el objeto receptor es mayor, negativo si es menor y cero si es igual al parámetro.
- En el ejemplo anterior la clase `String` es *comparable*: implementa el orden alfabético de las cadenas.

Ordenación de una lista

- **Ejemplo**: clase que representa un pedido con dos propiedades (producto y cantidad).
- El criterio de ordenación está basado en el orden de la propiedad *producto*. Así pues, un pedido es menor que otro si el nombre del producto es menor en orden alfabético.

```
public class Pedido implements Comparable<Pedido> {
    private final String producto;
    private final int cantidad;

    // Se omite constructor y métodos get

    @Override
    public int compareTo(Pedido otro) {
        return this.producto.compareTo(otro.producto);
    }
}
```

Ordenación de una lista

- Las clases que implementan la interfaz `Comparable<T>` se dice que tienen **orden natural**.

- No todas las clases ofrecen orden natural. Incluso si lo ofrecen, podría interesar ordenar los objetos de acuerdo a otro criterio.

- La clase `Collections` ofrece una versión sobrecargada del método `sort` que recibe como argumento un criterio de ordenación (interfaz **`java.util.Comparator<T>`**).

Criterios de ordenación

- Interfaz **`Comparator<T>:`**

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- El método `compare` devuelve un entero positivo si `o1` es mayor que `o2`, negativo si es menor y cero si son iguales.

Criterios de ordenación

- **Ejemplo**: implementación de un criterio de ordenación para la clase `Pedido`.

- La ordenación que realiza es la siguiente:
  - Primero ordena por orden alfabético (orden natural) de la propiedad *producto*.
  - En caso de empate, ordena por orden ascendente de la cantidad (orden natural de la clase `Integer`).

- Observa que para utilizar el orden natural de un tipo primitivo se utiliza el tipo envoltorio, por ejemplo, `Integer` para `int`.

Criterios de ordenación

```
public class ComparadorPedidos implements Comparator<Pedido> {

    @Override
    public int compare(Pedido arg0, Pedido arg1) {

        int criterio1 =
          arg0.getProducto().compareTo(arg1.getProducto());
        int criterio2 =
        ((Integer)arg0.getCantidad()).compareTo(arg1.getCantidad());

        if (criterio1 == 0) { // empate criterio 1
            return criterio2;
        }
        else return criterio1;
    }
}
```

Criterios de ordenación

- Utilizamos el comparador implementado para ordenar una lista de pedidos:

```
LinkedList<Pedido> pedidos = new LinkedList<Pedido>();
// Se añaden pedidos ...

Collections.sort(pedidos, new ComparadorPedidos());
```

- En este ejemplo no importa que la clase `Pedido` tenga orden natural (`Comparable`).

- El método `sort` ordena la colección utilizando el criterio que se establece como parámetro.

Conjuntos Ordenados `TreeSet<T>`

- La clase **`TreeSet<T>`** implementa la interfaz `SortedSet<T>` que define **conjuntos ordenados**.

- Un conjunto ordenado **evita elementos repetidos** y además permite recorrer los **elementos en orden**.

```
TreeSet<String> ordenado = new TreeSet<String>();
Collections.addAll(ordenado, "hello", "hola", "hallo");

for (String saludo : ordenado) {
    System.out.println(saludo); // hallo, hello, hola
}
```

Conjuntos Ordenados `TreeSet<T>`

- La clase **`TreeSet<T>`** requiere que las clases de los objetos de la colección implementen un orden natural (`Comparable`):
  - En el ejemplo anterior, la clase `String` es *comparable*.

- Si la clase no tiene orden natural o queremos que se ordene de acuerdo a otro criterio, en el constructor se estable un objeto `Comparator<T>`.

```
TreeSet<String> ordenado =
          new TreeSet<String>(new OrdenInverso());
Collections.addAll(ordenado, "hello", "hola", "hallo");

for (String saludo : ordenado) {
    System.out.println(saludo); // hola, hello, halo
}
```

Conjuntos Ordenados `TreeSet<T>`

- La interfaz **`SortedSet<T>`** que implementa `TreeSet<T>` añade operaciones a la interfaz `Set<T>`.
- Las más destacables son:
  - `first()` y `last()`: retorna el menor o mayor elemento del conjunto, respectivamente.
  - `headSet(T elem)` y `tailSet(T elem)`: retornan un **conjunto ordenado** con los elementos estrictamente menores (`headSet`) o mayores (`tailSet`) que el parámetro.
- No obstante, la característica más destacada de un conjunto ordenado es que durante un recorrido los elementos se obtienen en el orden establecido.

Conjuntos Ordenados `TreeSet<T>`

- La clase `TreeSet<T>` utiliza el criterio de ordenación (`Comparable` o `Comparator`) para:
  - Para insertar un elemento evitando repetidos (`add`).
  - Localizar un elemento en la colección (`contains`).
  - Borrar un elemento de la colección (`remove`).

- Por ejemplo, al insertar un elemento se considera que está repetido si al compararlo con algún elemento de la colección el criterio de ordenación da como resultado 0.
- Por tanto, no se utiliza `equals` en ninguna de esas operaciones.

Colecciones ordenadas

- Para trabajar con colecciones ordenadas tenemos dos opciones:
  - Lista que ordenamos con `sort` cuando sea necesario.
  - Conjunto ordenado.

- La elección de una u otra depende de la gestión de los elementos repetidos:
  - Si queremos mantener **elementos repetidos**, la opción es utilizar una **lista**.
  - Si la semántica **conjunto** es importante, debemos optar por un conjunto ordenado.

Recorridos

- El **problema** de los recorridos:
  - Solo las listas permiten recorrido por posición.
  - El rendimiento de un recorrido por posición varía según la implementación (`ArrayList` vs `LinkedList`)
  - No es posible recuperar individualmente los elementos de un conjunto.
  - Los conjuntos ordenados solo ofrecen operaciones para recuperar los elementos por los extremos (primero y último) y para extraer subconjuntos.
- Es necesario utilizar un **esquema de recorrido que sea eficiente y común a todas las colecciones**.
- La solución al problema son los **iteradores**.

Iteradores

- Un **iterador** es un objeto que permite recorrer los elementos de una colección.

- El **esquema de recorrido** que ofrece un iterador es el siguiente:

  - `Mientras` *`quedan elementos:`*
    - *`Recupera elemento`*
    - `Procesa el elemento …`

- Este esquema es definido por la interfaz **`Iterator<T>`**

Iteradores

- Interfaz **`Iterator<T>`**:
  - **`hasNext`**`()`: indica si quedan elementos en la iteración.
  - **`next`**`()`: devuelve el siguiente elemento de la iteración.
  - **`remove`**`()`: elimina el último elemento devuelto por el iterador.

```
public interface Iterator<T> {

    boolean hasNext();

    T next();

    void remove();
}
```

Iteradores

- Las colecciones de Java ofrecen un iterador para su recorrido utilizando el método `iterator()`.
- **Ejemplo**:

```
public static int contarBurbujasExplotadas(List<Burbuja> burbujas) {

        Iterator<Burbuja> iterador =
        burbujas.iterator(); int contador = 0;
        while (iterador.hasNext()) {


            Burbuja burbuja = iterador.next();
            if (burbuja.isExplotada())
                contador++;
        }
        return contador;
}
```

Iteradores

- Las colecciones implementan la interfaz `Iterable<T>` lo que les obliga a ofrecer un iterador:

```
public interface Iterable<T> {

    Iterator<T> iterator();
}
```

- Cualquier otro tipo de datos que quiera ser *iterable* debe implementar esta interfaz.

- Los **arrays** también son iterables.

Recorrido *for each*

- El recorrido **for each** permite recorrer objetos *iterables* sin manejar un objeto iterador.
- Es la opción más común de recorrido.

```
public static int contarBurbujasExplotadas(List<Burbuja> burbujas) {

        int contador = 0;
        for (Burbuja burbuja : burbujas) {
            if (burbuja.isExplotada())
                contador++;
        }
        return contador;
}
```

- El código anterior es equivalente a utilizar un iterador.

Eliminar elementos en un recorrido

- Durante el **recorrido** de una colección con iterador explícito o con *for each* (implícito) **no está permitido modificar la colección** (añadir o quitar elementos) .
- A través de un recorrido con **iterador explítico** sí podemos eliminar el último elemento recuperado:

```
public static void eliminarExplotadas(List<Burbuja> burbujas) {

        Iterator<Burbuja> iterador = burbujas.iterator();
        while (iterador.hasNext()) {
            Burbuja burbuja = iterador.next();
            if (burbuja.isExplotada())
                iterador.remove();

        }
}
```

Mapas

- La interfaz **`Map<K,V>`** representa una estructura de datos (mapa) que asocia pares <*clave, valor*>.

- En general un mapa es gestionado como un **conjunto** en el que los elementos son las **claves** y estas claves tienen asociado un valor.

- Por tanto, la implementación de un mapa y un conjunto es similar. Encontramos dos implementaciones:
  - **`HashMap<K,V>`**: implementación basada en una tabla de dispersión.
  - **`TreeMap<K,V>`**: implementación basada en árboles binarios de búsqueda balanceados. Representa **mapas ordenados**.

Mapas

- Un mapa no es una colección de elementos, sino una *tabla* que asocia <clave, valor>.
- Un mapa no es *iterable*. Sin embargo, su contenido se puede recorrer a través de sus claves.
- Aunque no es una colección, ofrece operaciones similares a las colecciones:
  - **Consulta**: `size()`, `isEmpty()`, **`containsKey(clave)`**, **`containsValue(valor)`**, **`get(clave) -> valor`**
  - **Modificación**: **`put(clave, valor)`**, **`remove(clave)`**, `clear()`, `putAll(otroMapa)`

Mapas

- **Ejemplo**: construye un mapa que asocia palabras con el número de veces que se repiten en una secuencia

```
public static Map<String, Integer> contarPalabras(String... palabras) {

        HashMap<String, Integer> mapa = new HashMap<>();
        for (String palabra: palabras) {
            if (! mapa.containsKey(palabra)) { //  primera aparición
                mapa.put(palabra, 1);
            }
            else { // suma 1 al contador
                int contador = mapa.get(palabra);
                mapa.put(palabra, contador + 1);
            }
        }
        return mapa;
}
```

Mapas

- Aunque un mapa no es iterable, podemos recorrerlo a través de sus claves.
- El método **`keySet()`** retorna el **conjunto de claves** del mapa.
- Utilizando la clave, podemos recuperar el valor asociado con el método **`get(clave)`**.

```
Map<String, Integer> mapa =
          contarPalabras("hola", "Juan", "hola", "adiós");

Set<String> claves = mapa.keySet();

for (String clave : claves) {
    System.out.printf("Clave: %s -> Valor: %s \n",
                      clave, mapa.get(clave));
}
```

Mapas

- También podemos obtener una colección con todos los valores que están asociados a claves del mapa.
- El método **`values()`** retorna una colección (interfaz `Collection`) con los valores.
- **Ejemplo**: calcular el mayor número de repeticiones del mapa.

```
Map<String, Integer> mapa =
   contarPalabras("hola", "Juan", "hola", "adiós");
Collection<Integer> valores = mapa.values();

int mayor = 0;
for (int valor : valores) {
   if (valor > mayor)
      mayor = valor;
}
```

Mapas

- De forma análoga a como sucede con las colecciones, no podemos modificar un mapa mientras recorremos sus claves o valores.
- **Ejemplo**: eliminar asociaciones cuyo valor sea 1.

```
Map<String, Integer> mapa =
   contarPalabras("hola", "Juan", "hola", "adiós");

for (String clave : mapa.keySet()) {

   if (mapa.get(clave) == 1)
      mapa.remove(clave); // Se produce un error
}
```

- El fragmento de código anterior NO FUNCIONA.

Mapas

- En un mapa, la modificación de sus colecciones de claves (`keySet`) o valores (`values`) tiene efecto sobre el mapa.
- Por tanto, si quitamos una clave de la colección de claves estaríamos quitando una entrada del mapa.
- De nuevo, utilizando un **iterador explícito** podríamos borrar durante el recorrido:

```
Map<String, Integer> mapa =
    contarPalabras("hola", "Juan", "hola", "adiós");

Iterator<String> iterador = mapa.keySet().iterator();

while (iterador.hasNext()) {
    String clave = iterador.next();
    if (mapa.get(clave) == 1)
        iterador.remove();
}
```

Mapas

- **Ejemplo**: añadir nuevas entradas al mapa cuya clave sea igual a la original con el prefijo “>” y que mantenga el mismo valor.

```
Map<String, Integer> mapa =
      contarPalabras("hola", "Juan", "hola", "adiós");

HashMap<String, Integer> nuevasEntradas = new HashMap<>();

for (String clave : mapa.keySet()) {
   String nuevaClave = ">" + clave;
   nuevasEntradas.put(nuevaClave, mapa.get(clave));
}
```

```
mapa.putAll(nuevasEntradas);
```

Mapas ordenados

- Al igual que con conjuntos, podemos optar por dos implementaciones de un mapa: `HashMap<K,V>` y `TreeMap<K,V>`, esta última implementa la interfaz **`SortedMap<K,V>`**.
- La diferencia fundamental es que un **`TreeMap<K,V>`** ordena las entradas según la clave.
- De este modo, al recorrer las claves del mapa (`keySet`) obtenemos las claves ordenadas.

```
public static SortedMap<String, Integer> contarPalabras(

    tring... palabras) { TreeMap<String, Integer> mapa = new
    TreeMap<>();

    /

    /

    E
```

Mapas ordenados

- La clase **`TreeMap<K,V>`** ofrece un constructor en el que podemos establecer un criterio de ordenación (`Comparator`).
- Resulta útil cuando queremos aplicar un orden a las claves distinto al orden natural de la clase que implementa las claves o bien si las claves no tienen orden.
- En el ejemplo las claves son cadenas (`String`) y le aplicamos un orden alfabético descendente.

```
public static SortedMap<String, Integer> contarPalabras(
                                                String... palabras) {

   TreeMap<String, Integer> mapa = new TreeMap<>(new OrdenInverso());

   // El resto igual ...
   return mapa;
}
```

Mapas - Rendimiento

- El funcionamiento de los mapas es análogo al de los **conjuntos**.
- La clase `HashMap<K,V>` utiliza una implementación similar a `HashSet<T>`:
  - Por tanto, se utilizan del mismo modo los métodos `hashCode/equals` para determinar si una clave está en el mapa.
- La clase `TreeMap<K,V>` comparte implementación con `TreeSet<T>`:
  - Así pues, las operaciones para añadir entradas (`put`), buscar claves (`containsKey`) y eliminar entradas (`remove`) solo utilizan el criterio de ordenación (`Comparable` o `Comparator`). No se utiliza el método `equals`.

Métodos de la clase Object

- Las clases que implementan las colecciones (`LinkedList<T>`, `HashSet<T>`, etc.) redefinen los métodos `equals`, `hashCode`, `toString` y `clone`.

- Las implementaciones de los métodos **`equals/hashCode`** es consistente.
- **Igualdad de listas**:
  - Dos listas son iguales si tienen el mismo tamaño y los elementos en cada posición son iguales (`equals`).
- **Igualdad de conjuntos**:
  - Dos conjuntos son iguales si tienen el mismo tamaño y todos los elementos de un conjunto están incluidos (`contains`) en el otro conjunto.

Métodos de la clase Object

- La implementación del método **`toString`** muestra el contenido de la colección utilizando el método `toString` de sus elementos.
- El método **`clone`** solo puede ser utilizado cuando el tipo de la variable es una clase (por ejemplo, `LinkedList`) y no se puede utilizar si fuera una interfaz (por ejemplo, `List`):
  - Ninguno de los métodos de la clase `Object` están declarados en las interfaces.
  - Dado que `equals`, `hashCode` y `toString` son públicos, siempre están disponibles en cualquier objeto
  - Sin embargo, el método `clone` se hace público en la implementación de las clases.

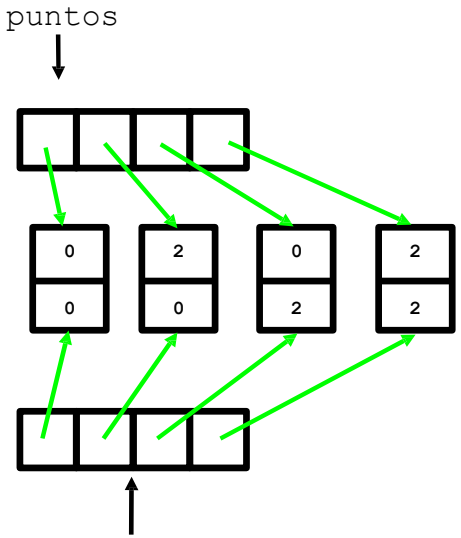
Copia de colecciones

- Todas las clases que implementan colecciones ofrecen un constructor de copia y el método `clone`.
- En ambos casos construye una **copia superficial** del objeto receptor.

```
LinkedList<Punto> puntos;
…
LinkedList<Punto> copia;

// Opción 1: copia con clone
copia = (LinkedList<Punto>)puntos.clone();

// Opción 2: constructor de copia
copia = new LinkedList<Punto>(puntos);
```



Colecciones y objetos “mutantes”

- Los **conjuntos y mapas** son sensibles a la modificación de los objetos una vez que han sido insertados en la colección.
- **Ejemplo**:

```
HashSet<Punto> puntos = new
HashSet<>(); Punto punto1 = new
Punto(0, 1); puntos.add(punto1);
punto1.setY(2); // (0, 2)
Punto punto2 = new
Punto(0, 2);
puntos.add(punto2);


System.out.println(puntos.size()); // ¿tiene 2 puntos?
```

Colecciones y objetos “mutantes”

- El método `hashCode` de la clase `Punto` es el siguiente:

```
@Override
public int hashCode() {
   int primo = 31;
   int resultado = 1;
   resultado = primo * resultado + x;
   resultado = primo * resultado + y;
   return resultado;
}
```

- Al insertar el punto (0, 1) se calcula su código de dispersión: 962. Al no estar en la tabla, lo inserta.
- La modificación del objeto a través de la variable `punto1` (*aliasing*) no es tenida en cuenta por el conjunto.

Colecciones y objetos “mutantes”

- Al insertar el segundo punto (0, 2) su código de dispersión es 963.
- El conjunto determina que no está repetido porque en la tabla de dispersión no tiene una entrada asociada al código 963.
- Por tanto, el segundo punto se introduce en el conjunto.
- El resultado es que el conjunto tiene dos puntos iguales (0, 2).
- **Resumen**: en conjuntos y mapas no deben alterarse los objetos una vez insertados. Si así fuera, primero hay que borrarlos (`remove`) y después volver a introducirlos.
- Las listas no tienen el problema de los objetos “mutantes”.

Recomendaciones

- **Programar hacia el TAD**
  - En **constructores y métodos públicos**, el tipo de retorno y el tipo de los parámetros se especifica utilizando la interfaz (por ejemplo `List` en lugar de `LinkedList`)
  - Observa como el método `contarPalabras` presentado en las diapositivas anteriores declara retornar un mapa (`Map`) y no un `HashMap`.

```
public static Map<String, Integer> contarPalabras(String... palabras) {
        HashMap<String, Integer> mapa = new HashMap<>();
        /
        /
        .
        .
        .
        r
        e
```

Recomendaciones

- **Evitar el uso de arrays**
  - Los arrays tienen una funcionalidad limitada.
  - Además, las operaciones fundamentales de la clase `Object` no están redefinidas en los arrays.
  - Es necesario utilizar los métodos static de la clase `java.util.Arrays` para comparar por igualdad dos arrays (`equals`), obtener el código de dispesión (`hashCode`) y la representación textual (`toString`).
  - Podemos obtener una lista a partir de un array:

```
String[] array = {"a", "b", "c", "d"};
List<String> lista = Arrays.asList(array);
```

  - **Nota**: la lista obtenida es de solo consulta.

Recomendaciones

- **Evitar los problemas de aliasing**:
  - En general, el aliasing de las colecciones suele ser incorrecto.
  - Por tanto, debemos evitar compartirlas tanto cuando recibimos una como parámetro de un método o constructor y cuando la ofrecemos en un método de consulta.
- **Solución 1**:
  - **Copiar la colección** (`clone` o constructor de copia).

```
// atributos
private LinkedList<Punto> vertices;
// ...

public List<Punto> getVertices() {
    return (List<Punto>) vertices.clone();
}
```

Recomendaciones

- **Evitar los problemas de aliasing …**
- **Solución 2**:
  - Devolver una ***vista* no modificable** de la colección:

```
// atributos
private LinkedList<Punto> vertices;
// ...
public List<Punto> getVertices() {
    return Collections.unmodifiableList(vertices);
}
```

  - Existe una operación análoga para cada interfaz de las colecciones: `unmodifiableSet`, `unmodifiableMap`, etc.
  - Es recomendable documentar que se devuelve una *vista* no modificable.
  - Es más eficiente que construir una copia.

Guía Rápida

- Junto a los ejercicios de programación está disponible una anexo con una **guía rápida** de uso de las colecciones.

- Incluye los siguientes tópicos:
  - Uso de `LinkedList` (**listas**).
  - Uso de `HashSet` (**conjuntos**).
  - Uso de `HashMap` (**mapas**).
  - **Ordenación de listas**. Interfaces `Comparable` y `Comparator`.