Resumen de Java

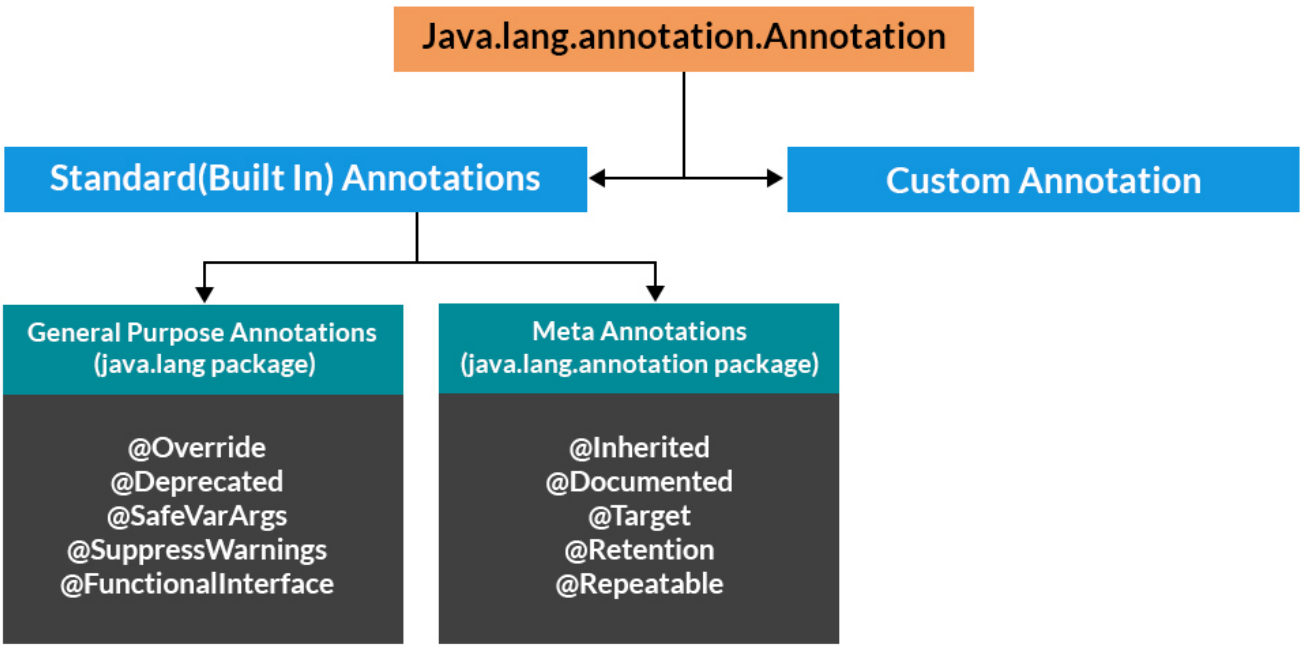
Anotaciones

Las anotaciones son metadatos que proporcionan información adicional sobre el código fuente al momento de compilar. Se utilizan para proporcionar información adicional al compilador, al depurador, frameworks y a otras herramientas que procesan el código fuente de Java.

```
@SuppressWarnings("deprecation")
public static void probarMetodosDeprecated() {
    MetodosDeprecated md = new MetodosDeprecated();

    md.metodoObsoleto();
}
```

```
void deprecated.MetodosDeprecated.metodoObsoleto()

@Deprecated

Deprecated. Se recomienda utilizar metodoNuevo en su lugar

Descripción del método

Press 'F2' for focus
```

Las Anotaciones provienen de la interfaz `Java.Lang.annotation.Annotation`:



Lista de Anotaciones Mas Comunes

- @Override: se utiliza para indicar que un método en una subclase está sobrescribiendo un método en la superclase o interfaz.
- @Deprecated: se utiliza para marcar un método o clase como obsoletos. Esto indica que el método o clase puede ser eliminado en versiones futuras del código y se recomienda no utilizarlo.
- @SuppressWarnings: se utiliza para suprimir advertencias específicas del compilador.

- @FunctionalInterface: se utiliza para indicar que una interfaz funcional tiene exactamente un método abstracto. Esto puede ser útil en marcos de trabajo (frameworks) que requieren interfaces funcionales.
- @SafeVarargs: se utiliza para suprimir advertencias relacionadas con el uso de varargs en métodos genéricos.
- @Target: se utiliza para indicar los elementos del código fuente a los que se aplica una anotación.
- @Retention: se utiliza para indicar cuánto tiempo se retiene una anotación en tiempo de ejecución.
- @Documented: se utiliza para indicar que una anotación debe incluirse en la documentación generada.
- @Inherited: se utiliza para indicar que una anotación se hereda por las subclases.

Programación Orientada a Objetos

La POO es un paradigma de la programación, es una forma de pensar, trabajar o estructurar un lenguaje de programación. Este poseé postulados sobre los que se basa su metodología. A continuación se presentaran los Terminos o Postulados de POO:

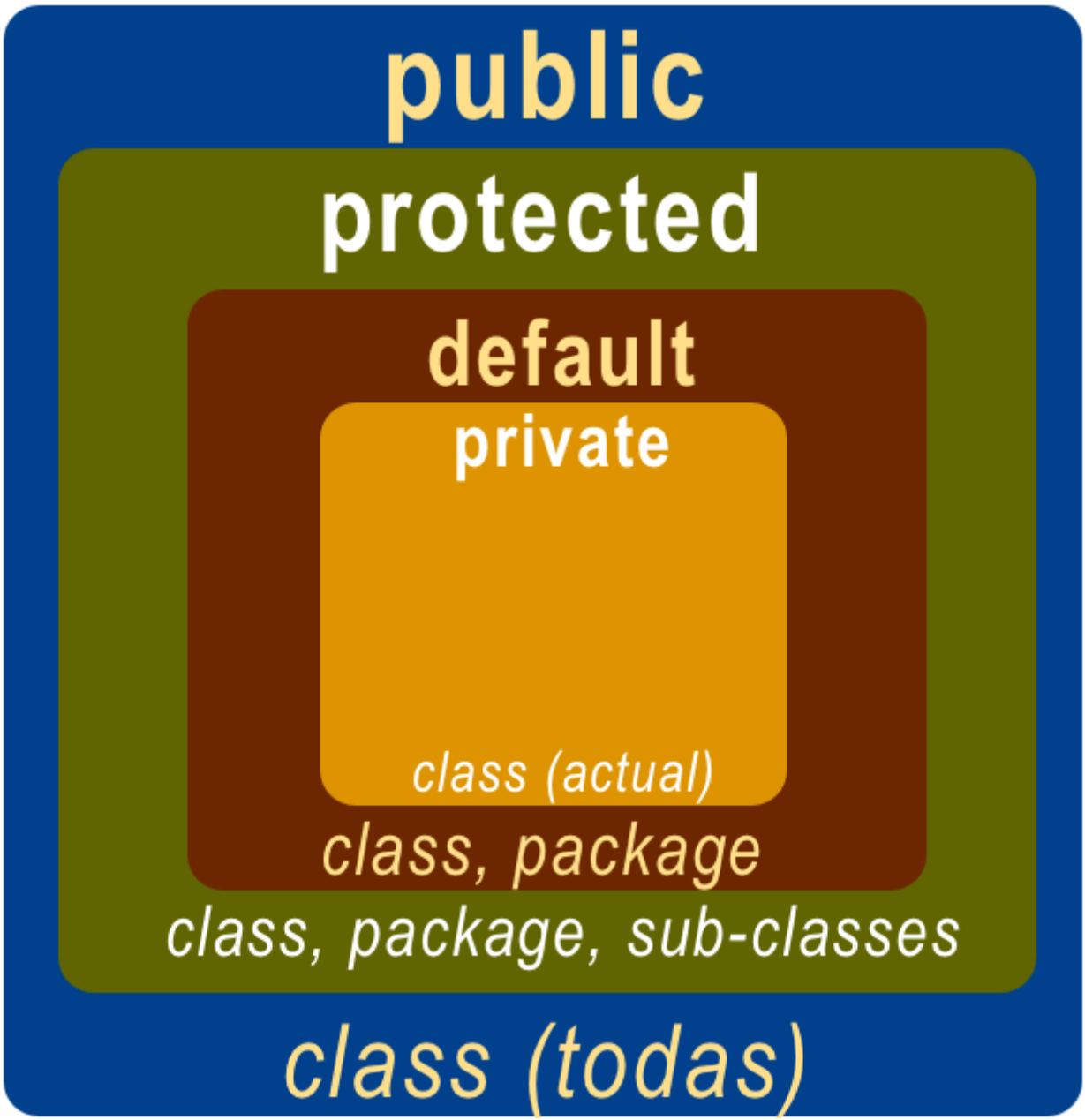
- **Encapsulamiento**: Es el mecanismo que permite ocultar los detalles internos de un objeto y exponer solo aquellos que sean relevantes para el uso de ese objeto. El encapsulamiento se logra mediante el uso de modificadores de acceso en los atributos y métodos de un objeto.
- **Abstracción**: Es la capacidad de representar los aspectos esenciales de un objeto de manera simplificada. La abstracción se logra mediante la definición de clases y objetos, que encapsulan datos y comportamientos relacionados.
- **Herencia**: Es el mecanismo que permite crear nuevas clases a partir de clases existentes, heredando sus atributos y métodos. La herencia permite la reutilización de código y la creación de jerarquías de clases.
- **Polimorfismo**: Es la capacidad de los objetos de una misma clase o de clases relacionadas mediante herencia de responder a un mismo mensaje o solicitud de manera diferente. El polimorfismo permite la creación de programas más flexibles y adaptables a diferentes situaciones.

Modificadores de Acceso

Los modificadores de acceso en Java son palabras clave que se utilizan para establecer los niveles de acceso y visibilidad para:

- Las variables
- Los Métodos
- Las Clases

isibilidad	Public	Protected	Default	Private
a misma Clase	SI	SI	SI	SI
ualquier Clase no Paquete	SI	SI	SI	NO
na SubClase del Paquete	SI	SI	SI	NO
na SubClase el mismo e	SI	SI, a través de la herencia	NO	NO
ualquier Clase el Paquete	SI	NO	NO	NO

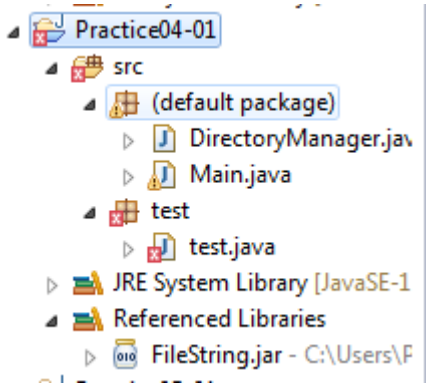


- **public**: Los miembros **(variables, métodos, clases)** con este modificador son accesibles desde cualquier parte del programa.
- **protected**: Los miembros con este modificador solo son accesibles dentro de la propia clase y sus subclases, así como dentro del mismo paquete.

- **default** (también llamado *"package-private"*): los miembros con este modificador solo son accesibles dentro del mismo paquete.
- **private**: Los miembros con este modificador solo son accesibles dentro de la propia clase donde se definen.

Java Packages

En Java, un paquete (o package en inglés) es un mecanismo que permite organizar las clases y otros elementos de un programa en grupos lógicos. Los paquetes ayudan a evitar conflictos de nombres entre clases y permiten una mejor gestión y mantenimiento del código.



Cada paquete en Java se identifica por su nombre único, que se compone de varios identificadores separados por puntos. Por ejemplo, el paquete java.util contiene clases para manipular colecciones de objetos, como listas, conjuntos y mapas.

Para utilizar una clase que se encuentra en un paquete, se debe indicar el nombre completo de la clase, incluyendo el nombre del paquete al que pertenece. Por ejemplo, para utilizar la clase ArrayList del paquete java.util, se debe escribir:

```
import java.util.ArrayList;

public class MiClase {
    // código de la clase
    ArrayList<String> miLista = new ArrayList<String>();
    // más código de la clase
}
```

Además un los paquetes también pueden contener subpaquetes, lo que permite organizar el código en una jerarquía más compleja. Por ejemplo, el paquete java.util contiene el subpaquete java.util.concurrent, que contiene clases para programación concurrente en Java.

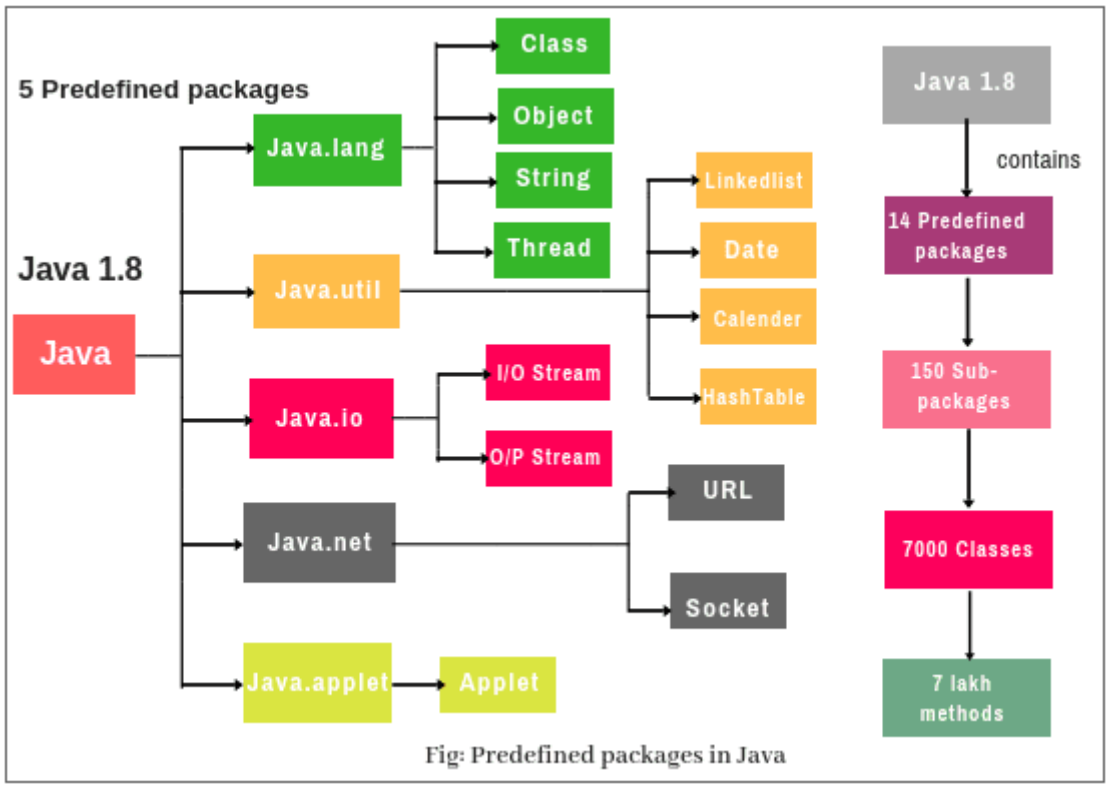


Fig: Predefined packages in Java

Como podemos ver, del paquete `java` se extienden otros como, `lang`, `util`, `io`, `net` y `applet`. De estos el más conocido es util, ya qué, como vimos en el ejemplo anterior, este contiene distintas clases y métodos que nos ayudan a trabajar con distintas estructuras de datos en Java.

Modificadores Java

Existen modificadores "normales", que se utilizan seguido de establecer un modificador de acceso a un elemento de Java. Estos no modifican la accesibilidad o visibilidad del elemento, sino que modifican el comportamiento del elemento en cuestión.

- **final**: Se utiliza para indicar que una variable, método o clase no puede ser modificado o extendido. Ejemplo:

```
public class Constantes {
    public static final double PI = 3.14159265359;
}
```

- **static**: Se utiliza para definir elementos que pertenecen a la **clase** en sí misma, y **NO** a la instancia de esa clase (objeto), por ejemplo: Podriamos llevar un registro del número de objetos creados a partir de la clase Contador.

```
  public class Contador {
      private static int contador = 0;

      public Contador() {
          contador++;
      }
```

```
    public static int getContador() {
        return contador;
    }
}
```

También se utiliza la palabra clave `static` para definir el método getContador(), que se puede acceder directamente a través del nombre de la clase (Contador.getContador()) sin necesidad de crear un objeto de la clase..

- **abstract**: Se utiliza para definir una clase o método que no tiene **implementación** y debe ser completado por una subclase. Recordemos que implementar una clase es cuando adjuntamos codigo adentro de `{}`

```
public abstract class Forma {
    public abstract double area();
}

public class Cuadrado extends Forma {
    private double lado;

    public Cuadrado(double lado) {
        this.lado = lado;
    }

    public double area() {
        return lado * lado;
    }
}
```

- **synchronized**: Se utiliza para definir un bloque de código que solo puede ser ejecutado por un hilo a la vez.

```
public class Contador {
    private int contador = 0;

    public synchronized void incrementar() {
        contador++;
    }
}
```

En este ejemplo, la palabra clave synchronized se utiliza para definir el método incrementar(), que sólo puede ser ejecutado por un hilo a la vez, lo que garantiza que el contador se incrementará correctamente aunque varias hebras traten de ejecutar este método simultáneamente.

- **volatile**: se utiliza para indicar que el valor de una variable puede ser modificado por varios hilos a la vez y se deben tomar medidas especiales para asegurar que los hilos acceden a la variable de manera

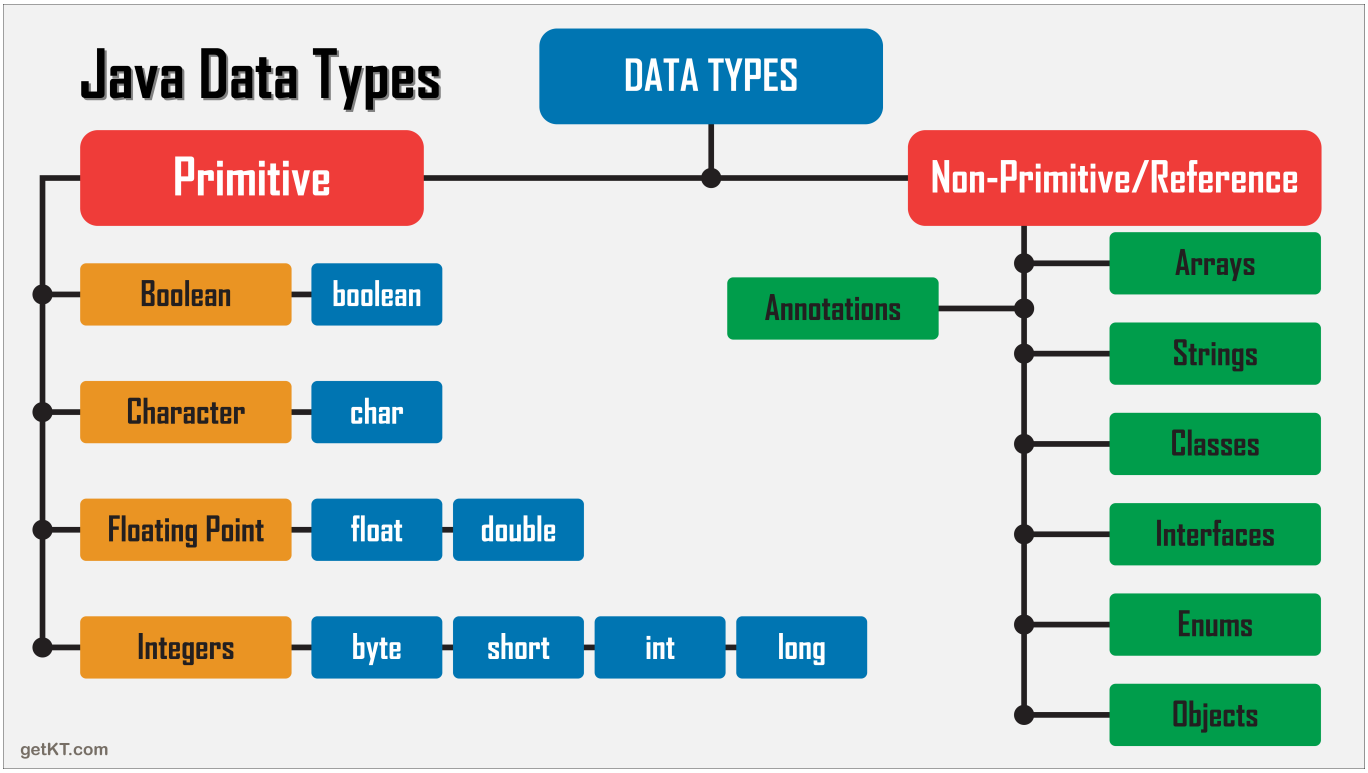
segura.

```
public class Contador {
    private volatile int contador = 0;

    public void incrementar() {
        contador++;
    }

    public int getContador() {
        return contador;
    }
}
```

- **transient**: Se utiliza para indicar que una variable no debe ser serializada cuando se guarda el estado de un objeto.

Tipos de Datos

Java tiene dos tipos de datos: tipos de datos primitivos y tipos de datos de referencia.



Datos Primitivos

Los tipos de datos primitivos son tipos de datos básicos y se almacenan directamente en la memoria de la computadora. Hay ocho tipos de datos primitivos en Java:

- `byte`: un número entero de 8 bits con signo.
- `short`: un número entero de 16 bits con signo.
- `int`: un número entero de 32 bits con signo.

- `long`: un número entero de 64 bits con signo.
- `float`: un número de punto flotante de 32 bits.
- `double`: un número de punto flotante de 64 bits.
- `boolean`: un valor booleano que puede ser true o false.
- `char`: un carácter Unicode de 16 bits.

Datos de Referencia

Los tipos de datos de referencia se utilizan para almacenar referencias a objetos y se almacenan en la memoria dinámica de la JVM (Java Virtual Machine). Algunos ejemplos de tipos de datos de referencia son String, Integer, Double, ArrayList, LinkedList, etc.

Por ejemplo, la clase String es un tipo de dato de referencia. Cuando se crea una instancia de la clase String, se reserva un bloque de memoria dinámica que contiene la cadena de caracteres. La variable que se utiliza para hacer referencia a la instancia de String almacenará la dirección de memoria donde se encuentra el objeto. Es decir, la variable no contendrá directamente la cadena de caracteres, sino que almacenará la referencia a la ubicación en la memoria donde se encuentra la cadena.

Otro ejemplo es la clase ArrayList. Cuando se crea una instancia de ArrayList, se reserva un bloque de memoria dinámica que contiene una lista de objetos. La variable que se utiliza para hacer referencia a la instancia de ArrayList almacenará la dirección de memoria donde se encuentra la lista. De nuevo, la variable no contendrá directamente los objetos de la lista, sino que almacenará la referencia a la ubicación en la memoria donde se encuentra la lista.

Wrapper Types

Un Wrapper Type, en Java, es un tipo de dato que envuelve a otro tipo de dato primitivo y lo convierte en un objeto. Por ejemplo, el tipo de dato primitivo `int` se puede envolver en el objeto `Integer` utilizando un Wrapper Type. De esta manera, un valor de tipo primitivo se puede tratar como un objeto, lo que facilita su uso en ciertas situaciones, como por ejemplo cuando se trabaja con colecciones de datos o se requiere su paso por referencia.

Los Wrapper Types se utilizan principalmente para proporcionar una funcionalidad adicional a los tipos de datos primitivos, ya que estos últimos no pueden tener métodos asociados. Por ejemplo, un objeto de tipo `Integer` tiene métodos como `parseInt()` o `toString()`, que no están disponibles para el tipo de dato primitivo int.

En Java existen los siguientes Wrapper Types:

- `Boolean`: envuelve al tipo de dato primitivo boolean.
- `Byte`: envuelve al tipo de dato primitivo byte.
- `Short`: envuelve al tipo de dato primitivo short.
- `Integer`: envuelve al tipo de dato primitivo int.
- `Long`: envuelve al tipo de dato primitivo long.
- `Float`: envuelve al tipo de dato primitivo float.
- `Double`: envuelve al tipo de dato primitivo double.
- `Character`: envuelve al tipo de dato primitivo char.

Array

Un arreglo es un tipo de datos por referencia que representa una colección de elementos del mismo tipo. Los elementos del arreglo se acceden por medio de un índice numérico que indica su posición en el arreglo. Por ejemplo, un arreglo de enteros (int[]) representa una colección de valores enteros que se pueden acceder y manipular individualmente.

```
// Definición y uso de un arreglo de enteros
int[] numeros = new int[5];
numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
numeros[3] = 4;
numeros[4] = 5;
```

Class

Una clase es un tipo de datos por referencia que define un conjunto de atributos y métodos. Las clases se utilizan para crear objetos, que son instancias de la clase, y que pueden tener un estado y un comportamiento definidos por los atributos y métodos de la clase. Por ejemplo, la clase String representa una cadena de caracteres y tiene métodos para manipularla.

Interface

Una interfaz es un tipo de datos por referencia que define un conjunto de métodos que deben ser implementados por cualquier clase que la implemente. Las interfaces se utilizan para definir un contrato que una clase debe cumplir para ser considerada de un tipo específico. Por ejemplo, la interfaz Comparable define un método compareTo() que debe ser implementado por cualquier clase que quiera ser comparada con otras instancias de la misma clase.

```
// Definición de la interfaz Ordenable
public interface Ordenable {
    public int compareTo(Object o);
}

// Clase que implementa la interfaz Ordenable
public class Persona implements Ordenable {
    private String nombre;
    private int edad;

    // Constructor de la clase Persona
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Método para obtener el nombre de la persona
    public String getNombre() {
        return nombre;
    }
```

```
    // Método para obtener la edad de la persona
    public int getEdad() {
        return edad;
    }

    // Implementación del método compareTo() de la interfaz Ordenable
    public int compareTo(Object o) {
        Persona otraPersona = (Persona) o;
        return this.edad - otraPersona.getEdad();
    }
}
```

Enums

Una enumeración es un tipo de datos por referencia que representa un conjunto fijo de valores constantes. Las enumeraciones se utilizan para definir un conjunto finito de valores que pueden ser utilizados en un programa. Por ejemplo, la enumeración Month define los doce meses del año como valores constantes.

```
// Definición de la enumeración Month
public enum Month {
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
}

// Uso de la enumeración Month
Month currentMonth = Month.APRIL;
System.out.println("El mes actual es: " + currentMonth); // Imprime "El mes
actual es: APRIL"
```

Casting y Conversion

La conversión se refiere a la transformación de un valor de un tipo de datos a otro tipo de datos. Las conversiones pueden ser implícitas o explícitas.

El casting se refiere a convertir un valor de un tipo de datos a otro de manera **explicita**. En Java, el casting se realiza utilizando el operador de paréntesis y la clase a la que se está convirtiendo.

Por ejemplo, si tienes una variable `double` y necesitas convertirla en una variable `int`, puedes hacerlo mediante el casting de la siguiente manera:

```
double miDouble = 3.14;
int miInt = (int) miDouble;
```

También se puede realizar en objetos. Por ejemplo, si tienes un objeto de la clase `Persona` y necesitas convertirlo en un objeto de la clase `Empleado`, puedes hacerlo mediante el casting de la siguiente manera:

```
Persona otraPersona = new Persona();
Empleado miEmpleado = (Empleado) otraPersona;
```

En cambio una conversión implicita ocurre automáticamente cuando el tipo de destino puede contener todos los valores posibles del tipo de origen. Por ejemplo, cuando asignas un valor de tipo int a una variable de tipo double, el compilador realiza la conversión implícita, ya que un double puede contener cualquier valor posible de un int.

Por ejemplo:

```
int miInt = 100;
long miLong = miInt;
```

Clases Abstractas Vs Interfaces

Las clases abstractas y las interfaces son ambos mecanismos de Java para establecer un contrato común que las clases concretas deben seguir. Sin embargo, hay algunas diferencias clave entre las dos:

- **Implementación de métodos**: Una ***clase abstracta*** **PUEDE** tener métodos abstractos y no abstractos, mientras que una ***interfaz*** **SOLO** puede tener métodos abstractos (sin implementación).
- **Herencia:** Una clase abstracta puede heredar de otra clase o implementar varias interfaces, mientras que una interfaz solo puede extender múltiples interfaces. En otras palabras, las clases abstractas permiten la herencia múltiple, mientras que las interfaces permiten la implementación múltiple.
- **Variables de instancia**: Las ***clases abstractas*** **PUEDEN** tener variables de instancia, mientras que las ***interfaces*** **NO**.
- **Constructor:** Las ***clases abstractas*** **PUEDEN** tener constructor, mientras que las ***interfaces*** **NO**.

En resumen, una clase abstracta es una clase que no se puede instanciar y que generalmente sirve como base para clases derivadas o extendidas. Puede contener métodos concretos, métodos abstractos y variables de instancia.

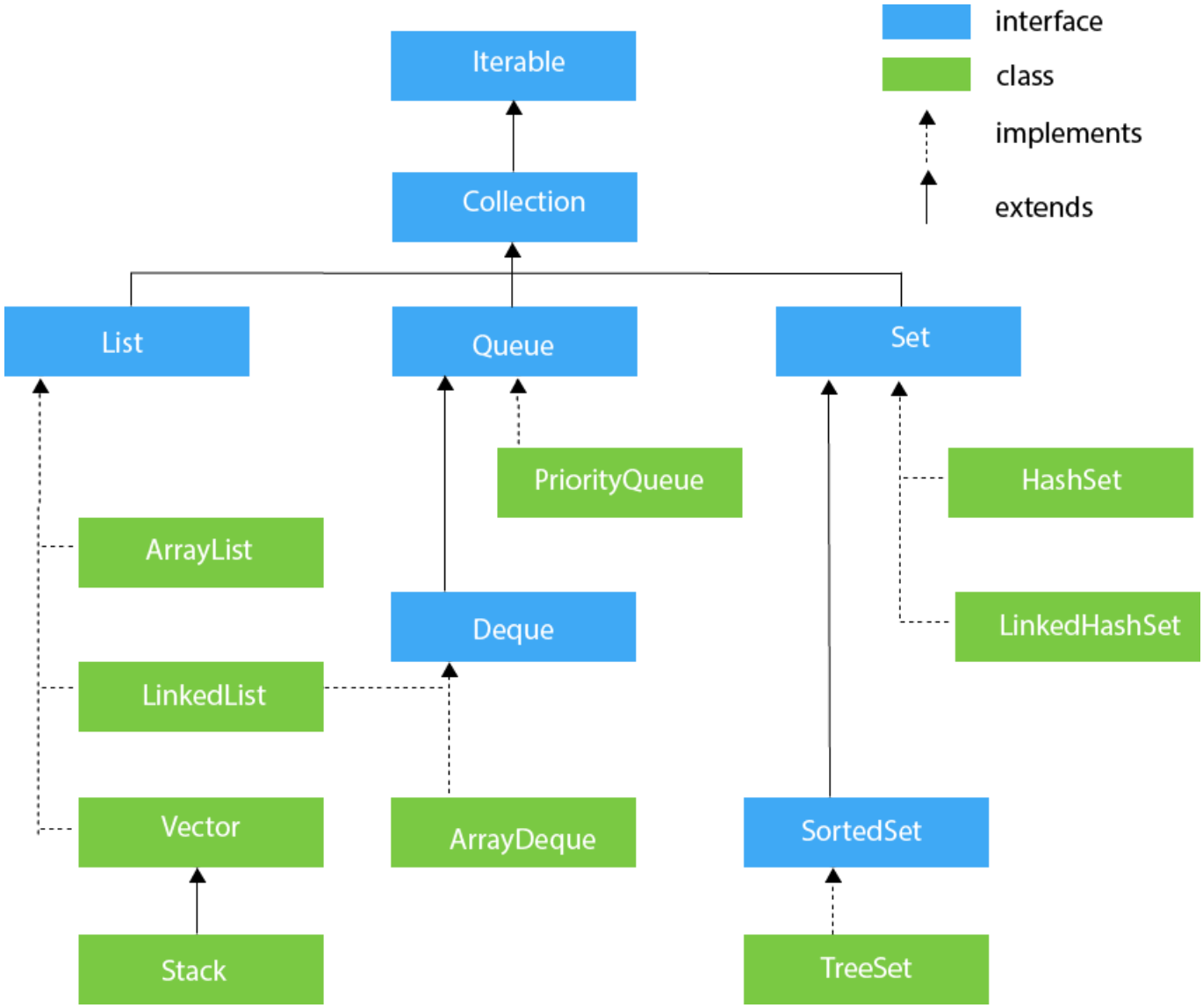
Una interfaz, por otro lado, define un conjunto de métodos que una clase DEBE implementar. No puede contener variables de instancia ni métodos concretos.

Una manera de pensar en la diferencia es que una clase abstracta proporciona cierta implementación predeterminada para que las clases derivadas la usen y se enfoque en la parte restante. En cambio, una interfaz simplemente establece un contrato que una clase debe cumplir sin proporcionar ninguna implementación predeterminada.

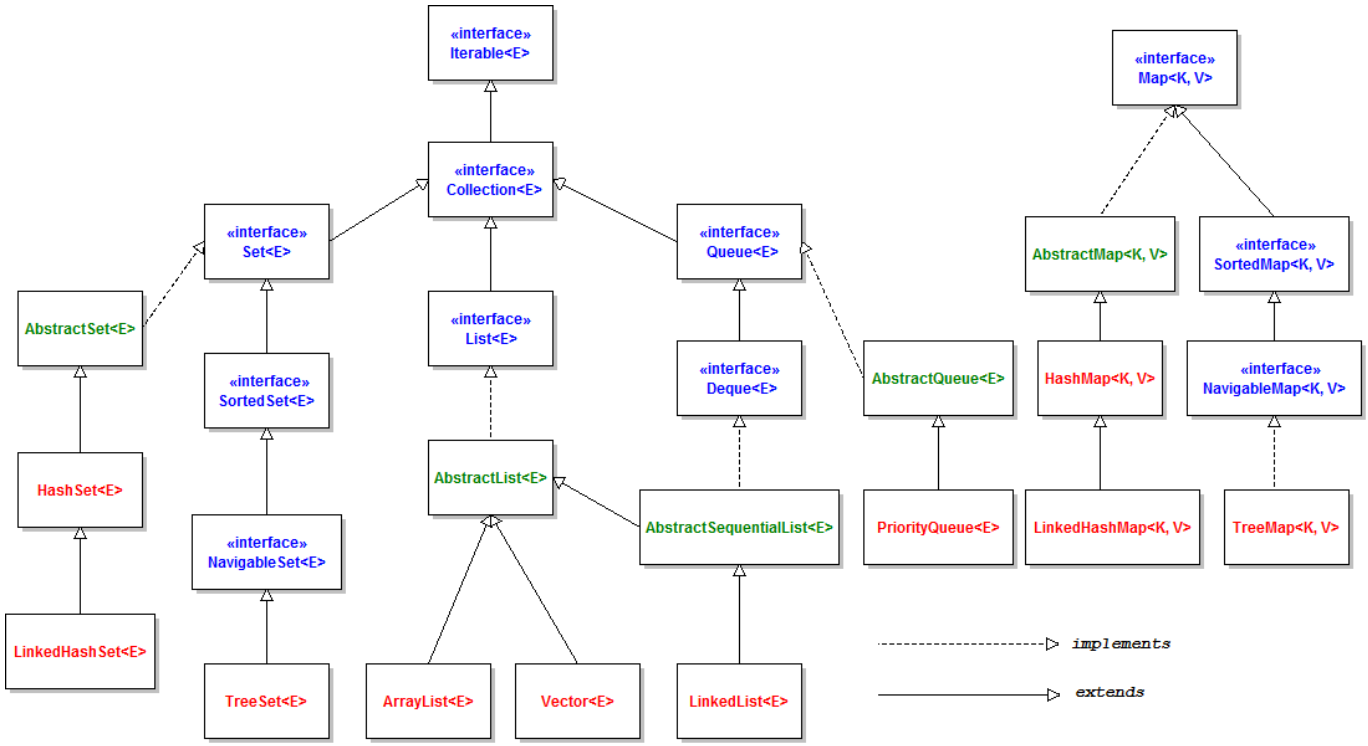
Colecciones

En Java, las colecciones son una estructura de datos que se utilizan para almacenar y manipular grupos de objetos. Estas colecciones proporcionan una forma conveniente y eficiente para almacenar, ordenar, buscar y manipular grandes cantidades de datos.

La API de colecciones de Java proporciona una variedad de interfaces y clases que implementan diferentes tipos de colecciones. Algunos de los tipos de colecciones más comunes incluyen List, Set, Map, Queue y Deque.



Como podemos ver en el gráfico anterior, estaría faltando la colección map. Esto ocurre porqué Map y sus distintas implementaciones no provienen de ***Iterable Interface***. En cambio este se encuentra en otro grupo, vease el siguiente gráfico:



Iterable

La interfaz Iterable es una interfaz genérica que se utiliza para proporcionar una forma de recorrer elementos de una colección en Java. Esta interfaz define el método iterator(), que devuelve un iterador que puede utilizarse para recorrer los elementos de la colección.

Collection

Collection es una interfaz en Java que representa un grupo de elementos que se pueden almacenar y manipular. Es la interfaz principal de la Jerarquía de Colecciones en Java y define un conjunto común de operaciones que se pueden realizar en cualquier tipo de colección. Estas operaciones son:

- `add()`: agrega un elemento a la colección.
- `remove()`: elimina un elemento de la colección.
- `size()`: devuelve la cantidad de elementos en la colección.
- `isEmpty()`: devuelve true si la colección está vacía, false de lo contrario.
- `contains()`: devuelve true si la colección contiene un elemento especificado, false de lo contrario.

List

Una lista es una colección de elementos ordenados por índice, lo que significa que los elementos se pueden acceder por su posición en la lista. Las implementaciones comunes de la interfaz List incluyen ArrayList y LinkedList.

```
List<String> nombres = new ArrayList<>();
nombres.add("Juan");
nombres.add("María");
nombres.add("Pedro");
System.out.println(nombres.get(1)); // Imprime "María"
```

Set

Un conjunto es una colección que no permite elementos duplicados. Los elementos en un conjunto no tienen un orden específico. Las implementaciones comunes de la interfaz Set incluyen HashSet y TreeSet.

```
Set<Integer> numeros = new HashSet<>();
numeros.add(1);
numeros.add(2);
numeros.add(1); // Se ignora porque 1 ya está en el conjunto
System.out.println(numeros.size()); // Imprime 2
```

sortedSet

SortedSet es una subinterfaz de Set en Java que mantiene sus elementos ordenados según su orden natural o utilizando un comparador definido por el usuario. Esto significa que los elementos de una SortedSet están ordenados en un orden específico, que puede ser ascendente o descendente.

```
// Crear un objeto TreeSet
        SortedSet<Integer> numeros = new TreeSet<>();

        // Agregar elementos al TreeSet
        numeros.add(5);
        numeros.add(3);
        numeros.add(9);
        numeros.add(1);
        numeros.add(7);

        // Imprimir los elementos del TreeSet (ordenados de menor a mayor)
        for (Integer numero : numeros) {
            System.out.println(numero);
        }

        // Obtener el primer elemento del TreeSet
        Integer primerNumero = numeros.first();
        System.out.println("El primer número es: " + primerNumero);

        // Obtener el último elemento del TreeSet
        Integer ultimoNumero = numeros.last();
        System.out.println("El último número es: " + ultimoNumero);
```

Queue

Una cola es una colección de elementos donde el primer elemento en entrar es el primero en salir (FIFO). Las implementaciones comunes de la interfaz Queue incluyen LinkedList y PriorityQueue.

```
Queue<String> cola = new LinkedList<>();
cola.add("Juan");
cola.add("María");
cola.add("Pedro");
System.out.println(cola.poll()); // Imprime "Juan"
```

Dequeue

Una doble cola (Deque) es una colección de elementos donde los elementos se pueden agregar y eliminar desde ambos extremos de la cola. Las implementaciones comunes de la interfaz Deque incluyen LinkedList y ArrayDeque.

```
Deque<String> deque = new LinkedList<>();
deque.addFirst("Juan");
deque.addLast("Pedro");
deque.addLast("María");
System.out.println(deque.pollFirst()); // Imprime "Juan"
```

Map

Un mapa es una colección de elementos que se almacenan como pares clave-valor, donde cada clave es única. Las implementaciones comunes de la interfaz Map incluyen HashMap y TreeMap.

```
Map<String, Integer> edades = new HashMap<>();
edades.put("Juan", 25);
edades.put("María", 30);
edades.put("Pedro", 35);
System.out.println(edades.get("María")); // Imprime 30
```