

Métodos comunes de todos los objetos

equals hashCode comparable

Método equals, cuando NO:

- Cada instancia de la clase es propiamente única.
- No hay necesidad para la clase de proveer una igualdad lógica.
- Una superclase ya implementó este método y el mismo es apropiado para la clase.
- La clase es privada y estamos seguros que el método equals nunca será invocado.

Método equals, cuando SI:

- Cuando una clase tiene una igualdad lógica que va más allá de la identidad del objeto.
Y una superclase no haya implementado este método.

Que establece su contrato?

El método equals implementa una relación de equivalencia. Con las siguientes propiedades:

- Reflexivo: Para cada valor de X no nulo.
X.equals(X) es **true**.
- Simétrico: Para cada valor de X e Y no nulo.
X.equals(Y) es **true** si y sólo si Y.equals(X) es **true**.
- Transitivo: Para cada valor de X, Y, Z no nulo.
Si X.equals(Y) es **true**
Y Y.equals(Z) es **true**
Entonces X.equals(Z) debe ser **true**.
- Consistente: Para cada valor de X e Y. Múltiples invocaciones de X.equals(Y) debe retornar consistentemente **true** o consistentemente **false**. Siempre que no se modifique la información utilizada en ambos.
Para cada valor de X no nulo. X.equals(null) debe retornar **false**.

Receta para un método equals de calidad:

- 1) Usar el operador `==` para chequear si el argumento es una referencia a este objeto. Si lo es retorna `true`.
- 2) Usar el operador `instanceof` para chequear si el argumento posee el tipo correcto. Si no, retorna `false`.
- 3) Castear el argumento al tipo correcto. Como el casteo se realiza después de un `instanceof` está garantizado que será satisfactorio.
- 4) Por cada atributo "significante" en la clase chequear si ese atributo es igual al atributo correspondiente de este objeto. Si todos estos son satisfactorios, retorna `true`. Caso contrario `false`.

Para los atributos primitivos cuyo tipo no es `float` ni `double` usar el operador `==` para comparar.

Para objetos, llamar al método `equals` recursivamente.

Para atributos `float` usar el método estático `compare` de la clase `Float`. `Float.compare(float1, float2)`

Para atributos `double` usar el método estático `compare` de la clase `Double`. `Double.compare(double1, double2)`

Ejemplo método equals

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        if (o == this) return true;  
        if (!(o instanceof Telephone)) return false;  
        Telephone tp = (Telephone)o;  
        return tp.lineNum == lineNum && tp.prefix == prefix && tp.areaCode ==  
areaCode;  
    }  
    ...  
}
```

Método hashCode:

**Siempre se debe hacer Override de hashCode cuando se hace
Override de equals.**

Que establece su contrato?

- Cuando el método hashCode es invocado en un objeto repetidas veces debe retornar el mismo valor consistentemente.
- Si dos objetos son iguales de acuerdo al método equals, entonces hashCode debe retornar el mismo valor para ambos.
- Si dos objetos son distintos de acuerdo al método equals, no es necesario que hashCode produzca un valor distinto. Sin embargo esto puede mejorar la performance de una hash table.

Cómo NO escribir un método hashCode

```
@Override
```

```
public int hashCode() {
```

```
    return 42;
```

```
}
```


Receta para un buen método hashCode:

1. Declarar una variable entera llamada resultado e inicializarla con el valor de hash code del primer atributo significativo en nuestro objeto.
2. Para cada atributo remanente en nuestro objeto hacer lo siguiente:
 - a. Si el atributo es de tipo primitivo computar `Type.hashCode(atributo)`. Donde Type es el Tipo correspondiente a la primitiva.
 - b. Si el atributo es un objeto y esa clase hace uso de equals de manera recursiva para sus atributos. Usar hashCode recursivamente en el atributo. Si el valor del atributo es null usar 0.
 - c. Si el atributo es un array, tratarlo como si cada elemento fuera un atributo separado. Esto significa calcular un hashCode para cada elemento en el array y combinar los valores. Si el array no tiene elementos significantes usar una constante diferente de 0. Si TODOS los elementos son importante usar `Arrays.hashCode`.
3. Combinar el resultado de esta forma: $\text{resultado} = 31 * \text{resultado} + \text{atributo}$;

Ejemplo método hashCode

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    public int hashCode() {  
        int result = Short.hashCode(areaCode);  
        result = 31 * result + Short.hashCode(prefix);  
        result = 31 * result + Short.hashCode(lineNum);  
        return result;  
    }  
}
```

Interfaz Comparable

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

Al implementar esta interfaz en una clase estamos indicando que las instancias de dicha clase poseen un **orden natural**.

El contrato de este método especifica que debe devolver lo siguiente:

- Entero negativo si el objeto es menor al especificado por parámetro.
- Cero si el objeto es igual al especificado por parámetro.
- Entero positivo si el objeto es mayor al especificado por parámetro.

Ejemplo método compareTo

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    public int compareTo(Telephone tp) {  
        int result = Short.compare(areaCode, tp.areaCode);  
        if (result == 0) {  
            result = Short.compare(prefix, tp.prefix);  
            if (result == 0) result = Short.compare(lineNum, tp.lineNum);  
        }  
        return result;  
    }  
    ...  
}
```

Clase 13: Map

— Programación y Laboratorio III —

Agenda

- **Interfaz Map**
- HashMap
- Principio de Hashing
- equals() & hashCode()
- HashMap vs. Hashtable
- LinkedHashMap
- TreeMap

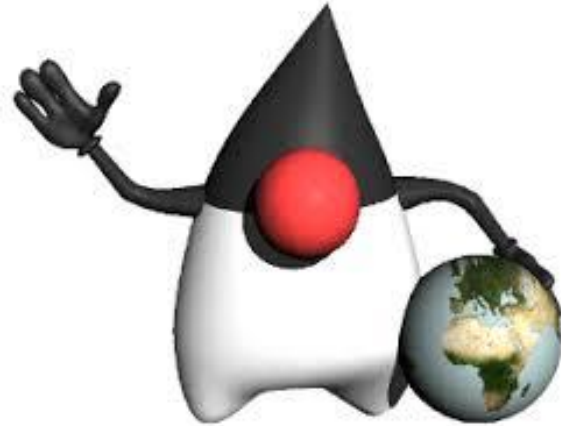


Interfaz Map

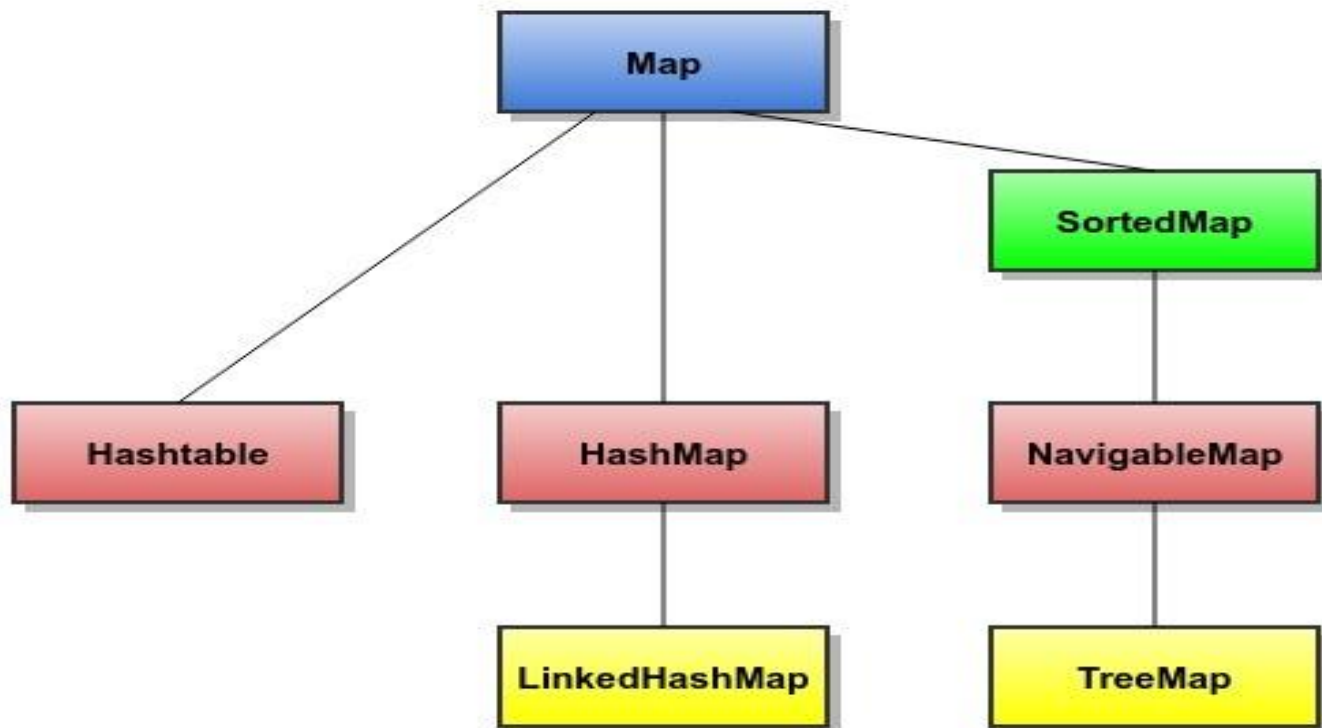
- Nos permite representar una estructura de datos para almacenar pares “clave-valor”.
- Para una clave sólo tenemos un valor.
- Map tienen implementada por debajo toda la teoría de las estructuras de datos de árboles, por lo tanto permiten añadir, eliminar y modificar elementos de forma transparente.
- La clave funciona como un identificador único y no se admiten claves duplicadas.

Map - Operaciones

- `boolean containsKey(Object key)`
- `boolean containsValue(Object var1);`
- `V get(Object var1);`
- `V put(K var1, V var2);`
- `V remove(Object var1);`
- `int size();`
- `boolean isEmpty();`



Interfaz Map



Agenda

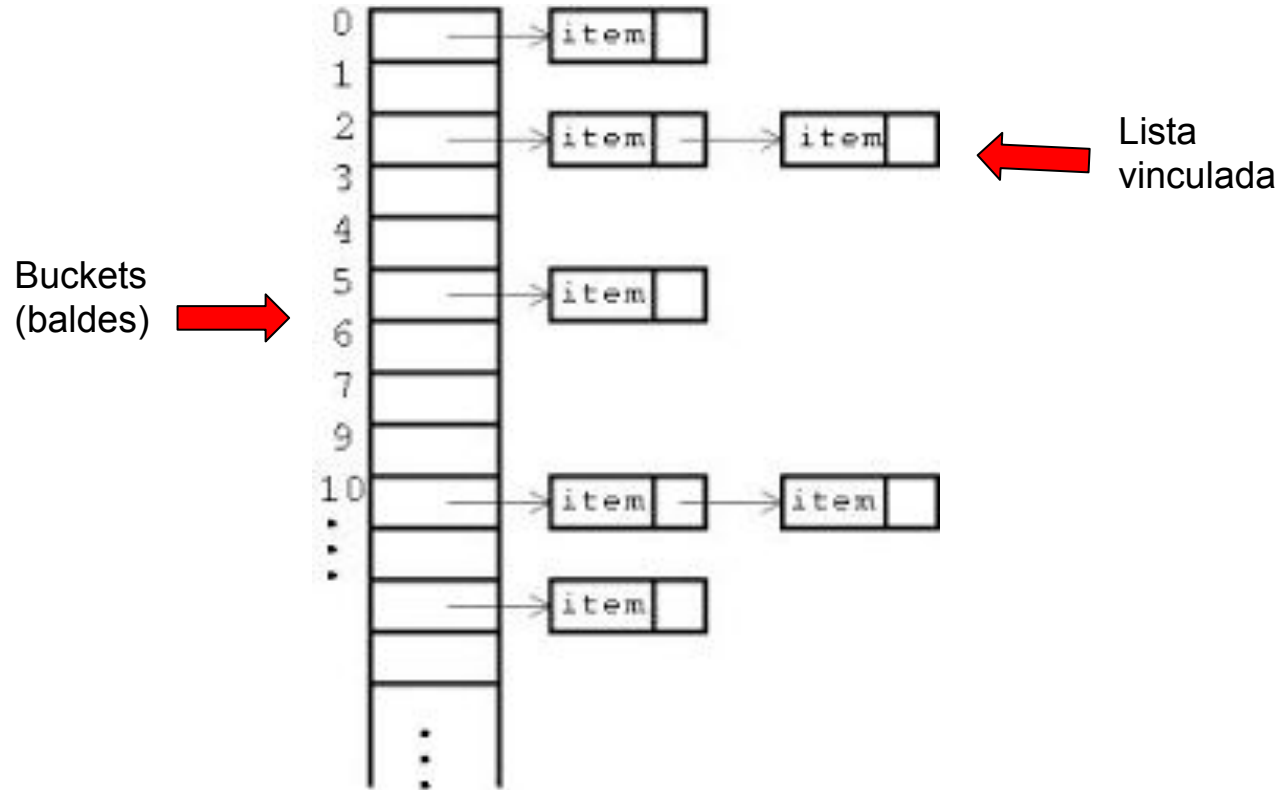
- ~~— Interfaz Map~~
- **HashMap**
- Principio de Hashing
- equals() & hashCode()
- HashMap vs. Hashtable
- LinkedHashMap
- TreeMap



HashMap

- Un HashMap es una colección de objetos, como los arrays, pero estos no tienen orden.
- Cada objeto se identifica mediante algún identificador y conviene que sea inmutable (`final`), de modo que no cambie en tiempo de ejecución.
- El nombre Hash hace referencia a una técnica de organización de archivos llamada hashing o “dispersión” en el cual se almacenan registros en una dirección del archivo que es generada por una función que se aplica a la clave del mismo.
- Los elementos que se insertan en un HashMap no tendrán un orden específico.
- Permite una clave null.

HashMap - Estructura

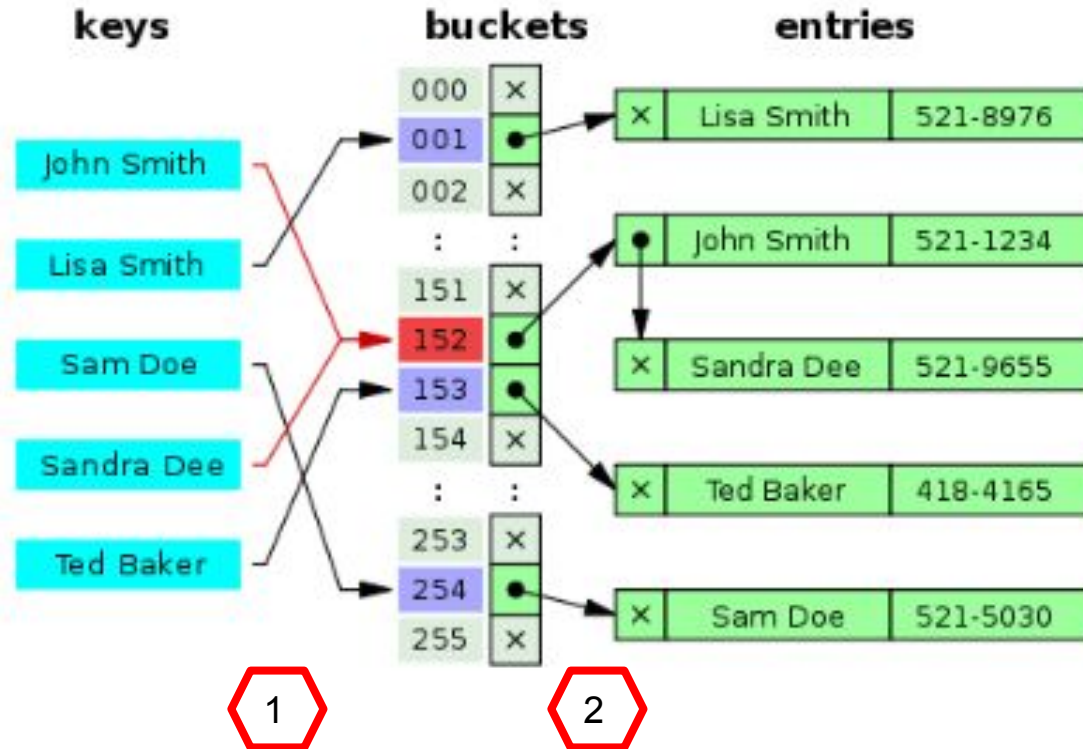


Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- **Principio de Hashing**
- equals() & hashCode()
- HashMap vs. Hashtable
- LinkedHashMap
- TreeMap



Principio de Hashing



Principio de Hashing (2)

- 1 Se utiliza el método `hashCode()` para encontrar el bucket correspondiente.
- 2 Se utiliza el método `equals()` para buscar el valor correspondiente a la clave dada.
 - Dos objetos idénticos tendrán el mismo identificador retornado por el método `hashCode()`.

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- **equals() & hashCode()**
- HashMap vs. Hashtable
- LinkedHashMap
- TreeMap



`equals()` & `hashCode()`

NO OLVIDAR:



“Si dos objetos son iguales usando `equals()`, entonces la invocación a `hashCode()` de ambos objetos debe retornar el mismo valor.”

equals() & hashCode() (2)

1) Objetos que son iguales pero retornan diferentes hashCodes.

Cada uno de estos baldes tiene asignado un número que lo identifica. Cuando agregamos un valor al HashMap, almacena el dato en uno de esos baldes. El balde que se usa depende del hashCode que devuelva el objeto a ser almacenado. Por ejemplo, si el método hashCode() del objeto retorna 49, entonces se almacena en el balde 49 dentro del HashMap.

Más tarde, cuando verifiquemos si la colección contiene al elemento invocando el método contains(elemento), el HashMap primero obtiene el hashCode de ese "elemento". Luego buscará el balde que corresponde a ese hashCode. Si el balde está vacío, significa que el HashMap no contiene al elemento y devuelve false.

equals() & hashCode() (3)

2) **Objetos que no son iguales pero retornan el mismo hashCode.**

Se utiliza este mecanismo de "baldes" por un tema de eficiencia. Si todos los objetos que se agregan a un HashMap se almacenaran en una única lista grande, entonces tendríamos que comparar la entrada con todos los objetos de la lista para determinar si un elemento en particular está contenido en el Map. Como se usan baldes, sólo se comparan los elementos del balde específico, y en general cada balde sólo almacena una pequeña cantidad de elementos en el HashMap.

Agenda

- ~~— Interfaz Map~~
- ~~— HashMap~~
- ~~— Principio de Hashing~~
- ~~— equals() & hashCode()~~
- **HashMap vs. Hashtable**
- LinkedHashMap
- TreeMap



HashMap vs. Hashtable

- La principal diferencia entre uno y otro es la sincronización interna del objeto. Para aplicaciones multihilos es preferible elegir Hashtable sobre HashMap, que no tiene sincronización.
- HashMap es mejor en cuanto a performance.
- Hashtable no admite valores nulos en ninguna de sus partes, mientras que HashMap sí.

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- ~~equals() & hashCode()~~
- ~~HashMap vs. Hashtable~~
- **LinkedHashMap**
- TreeMap



LinkedHashMap

- Similar a HashMap pero con la diferencia que mantiene una lista doblemente vinculada, además del array de baldes.
- La lista doblemente vinculada define el orden de iteración de los elementos

Agenda

- ~~Interfaz Map~~
- ~~HashMap~~
- ~~Principio de Hashing~~
- ~~equals() & hashCode()~~
- ~~HashMap vs. Hashtable~~
- ~~LinkedHashMap~~
- **TreeMap**



TreeMap

- Implementado mediante un árbol binario y permite tener un mapa ordenado.
- Cuando iteramos un objeto TreeMap los objetos son extraídos en forma ascendente según sus keys.
- TreeMap no sabe cómo ordenar la colección cuando se utiliza una key creada por el programador, sólo lo hace si trabajamos con objetos tipo String, Integer, etc.



Bibliografía oficial

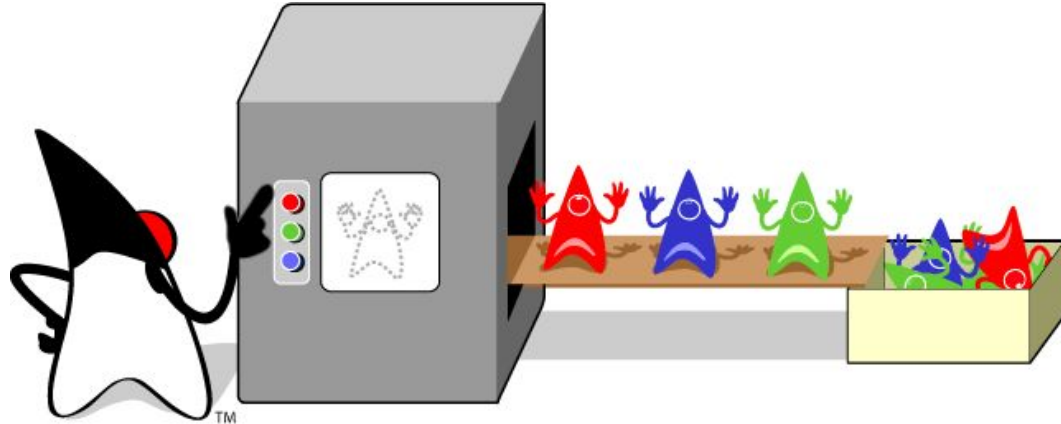
- <https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html>
-

Clase 14: Collection - Set

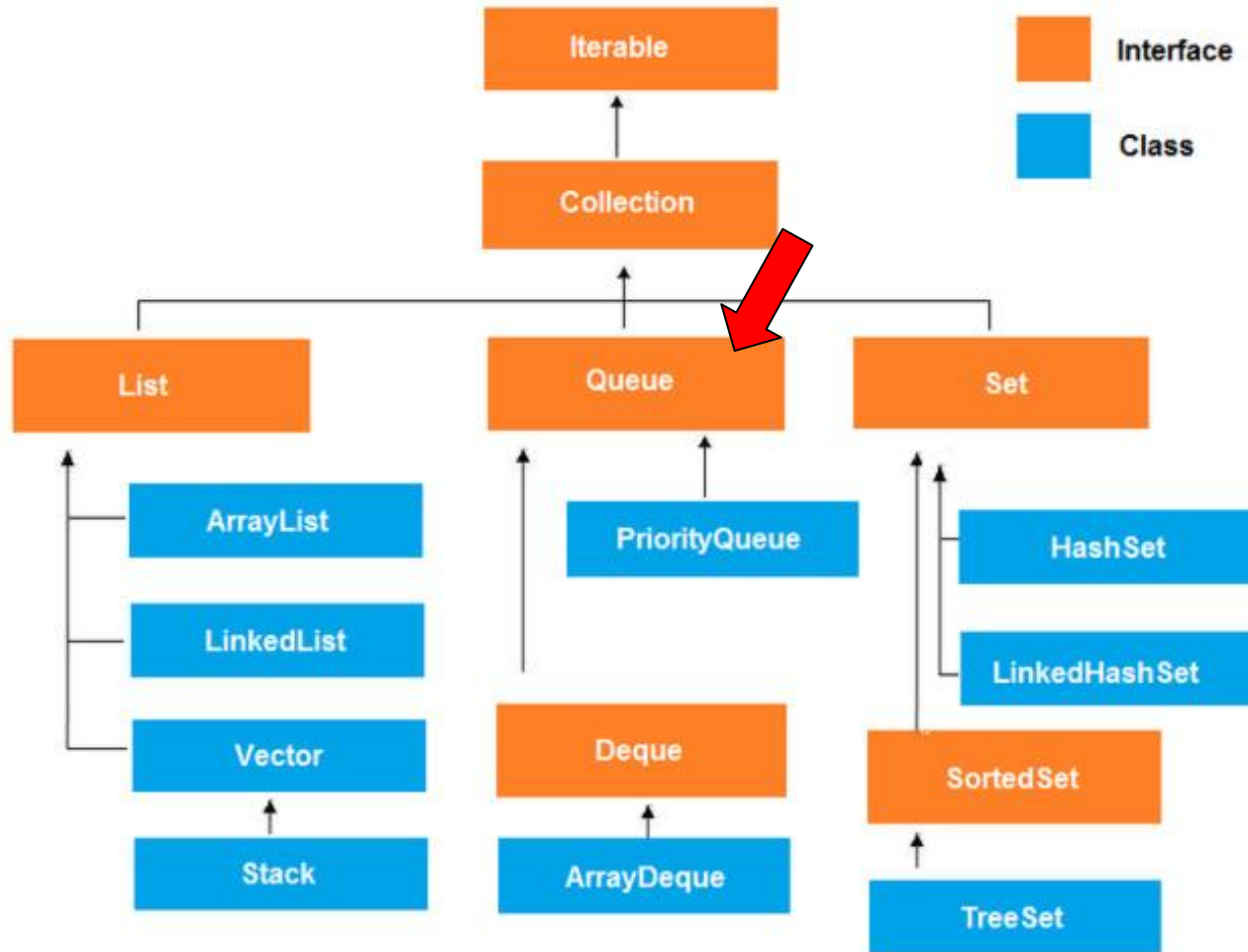
— Programación y Laboratorio III —

Agenda

- **Queue**
- Set
- HashSet
- LinkedHashSet
- TreeSet
- Diagrama de decisiones para uso de colecciones

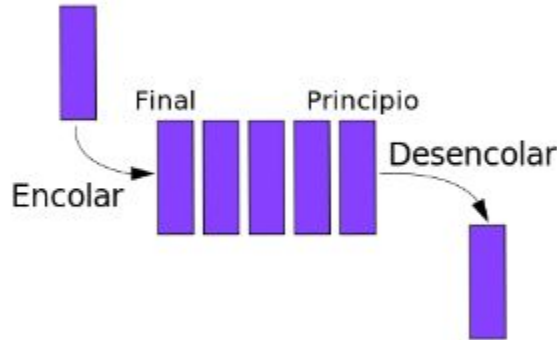


Collection Queue



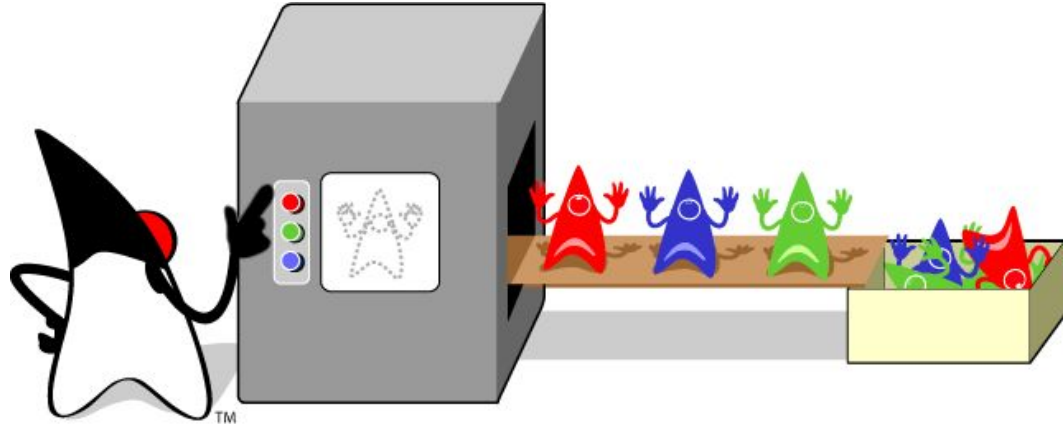
Queue

- Representa al tipo Cola, que es una lista en la que sus elementos se introducen únicamente por un extremo (fin de la cola) y se remueven por el extremo contrario (principio de la cola).

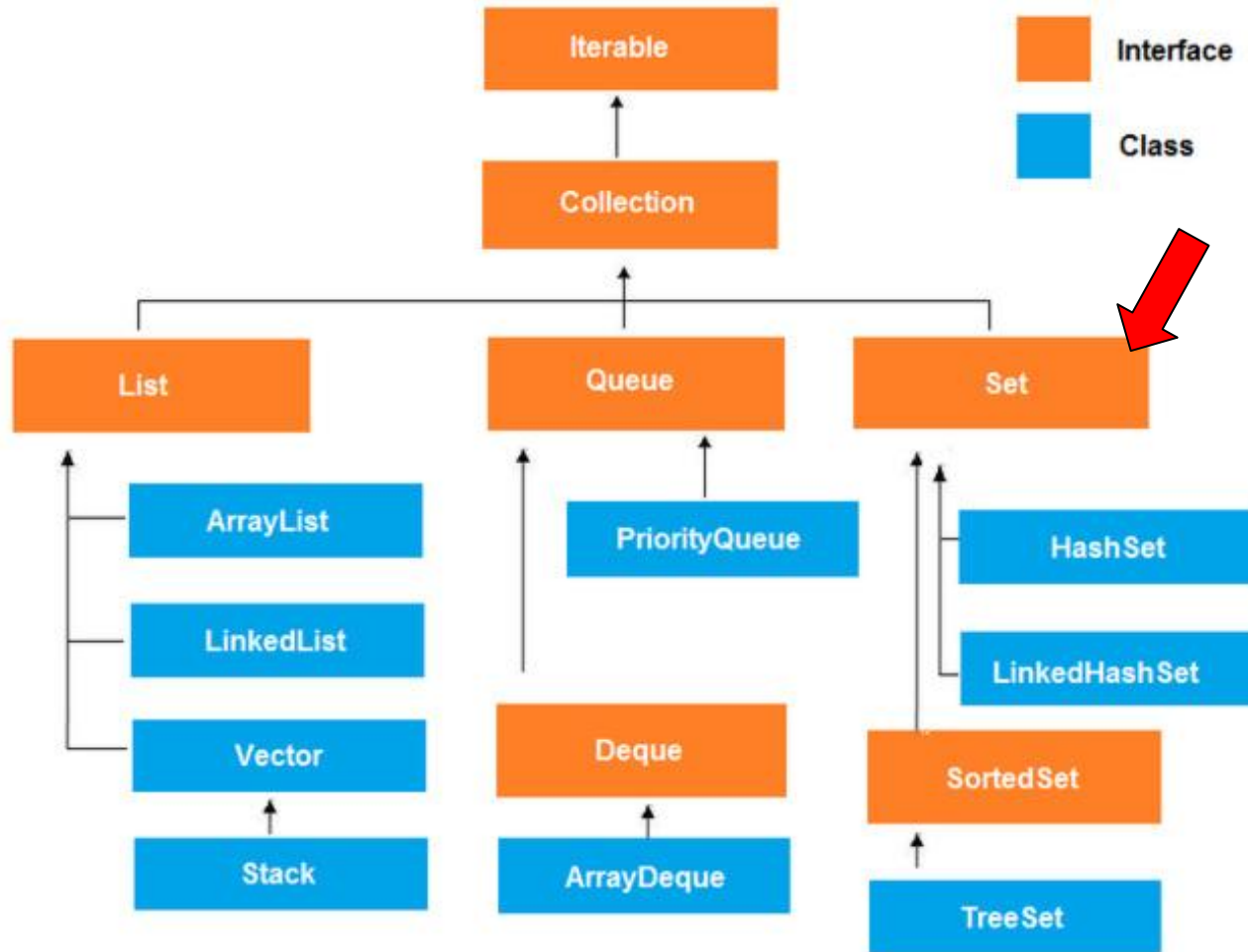


Agenda

- Queue
- **Set**
- HashSet
- LinkedHashSet
- TreeSet
- Diagrama de decisiones para uso de colecciones



Collection Set



Interfaz Set

- Modela los conjuntos de la matemática y sus propiedades.
- Set hereda los métodos de Collection y agrega sus propias restricciones para prohibir el duplicado de elementos.

Si tenemos un objeto que tienen las mismas características (equals) y el mismo hashCode que los objetos que ya se encuentran en el Set →
No se agrega a la colección

Interfaz Set - Algunas operaciones

- `int size();`
- `boolean isEmpty();`
- `boolean contains(Object var1);`
- `Iterator<E> iterator();`
- `boolean add(E var1);`
- `boolean remove(Object var1);`
- **`boolean addAll(Collection<? extends E> var1);`**
- **`boolean retainAll(Collection<?> var1);`**
- **`boolean removeAll(Collection<?> var1);`**

Interfaz Set - Operaciones

- `boolean addAll(Collection<? extends E> var1);`

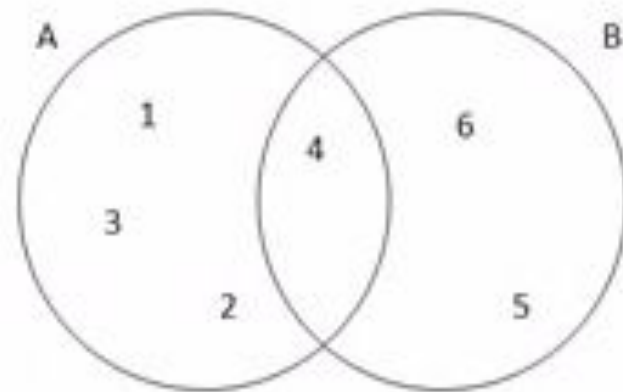
$$A \cup B = \{1,2,3,4,5,6\}$$

- `boolean retainAll(Collection<?> var1);`

$$A \cap B = \{4\}$$

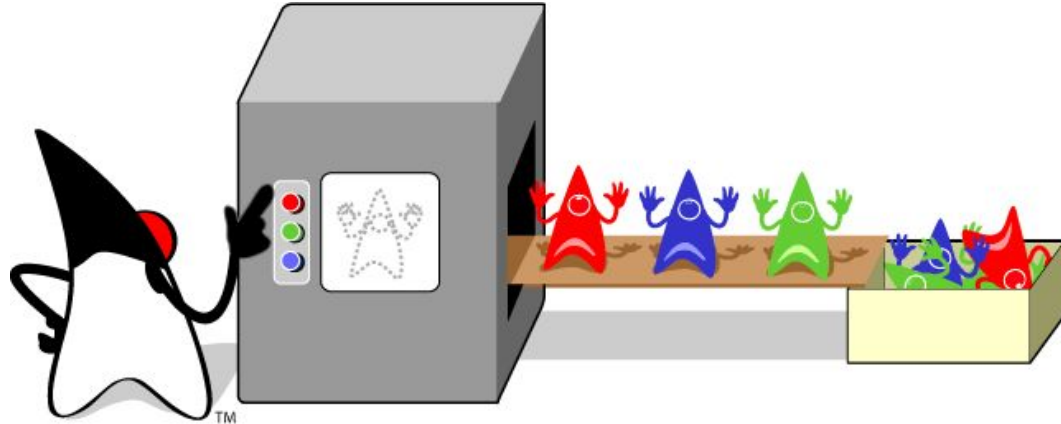
- `boolean removeAll(Collection<?> var1);`

$$A - B = \{1,2,3\}$$



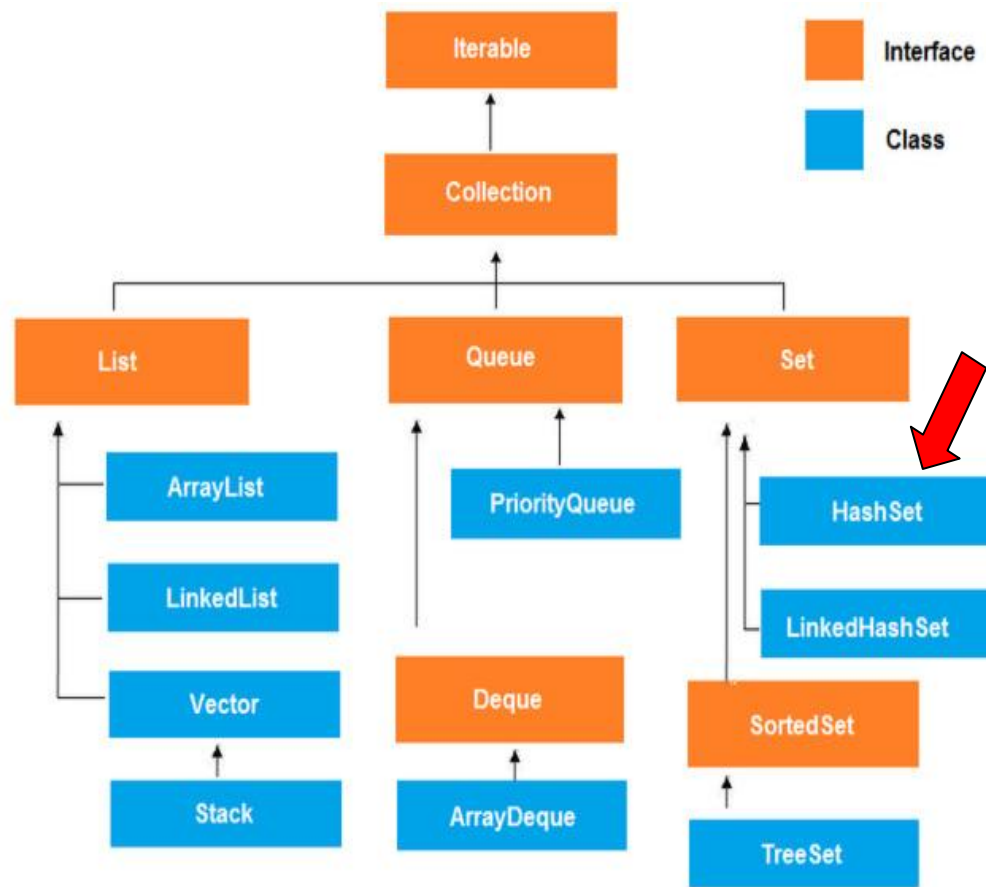
Agenda

- Queue
- Set
- **HashSet**
- LinkedHashSet
- TreeSet
- Diagrama de decisiones para uso de colecciones



HashSet

- Esta implementación de Set almacena los elementos en una **tabla hash**.
- Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash.



HashSet (2)

- Esta clase delega casi todas sus funcionalidades a un **HashMap<T, Object>**, donde el objeto que se inserta en el HashSet será la clave del mapa interno, y se registrará con el valor de un objeto por defecto dentro del mapa.
- Toda la lógica de verificación que el elemento no esté duplicado la maneja el mapa interno utilizado en la instancia del set → Las claves en un HashMap deben ser únicas.

HashSet (3)

1) Creación de HashSet:

```
HashSet<String> hashSet = new HashSet<String>();
```

2) Al llamar al constructor de HashSet, se ejecuta lo siguiente:

```
public HashSet() {  
    map = new HashMap<>();  
}
```

“map” es una variable de instancia de HashSet declarada como:
`HashMap<E, Object> map;`

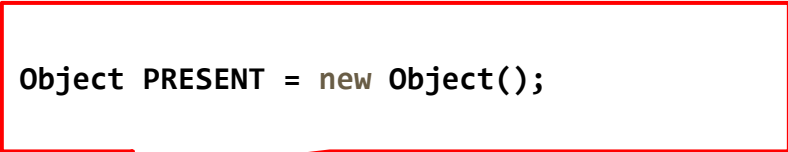
HashSet (4)

3) Una vez creado el HashSet, agregamos un conjunto de nombres:

```
hashSet.add("Pedro");  
hashSet.add("Pepe");  
hashSet.add("Luis");
```

4) Implementación del método add(E e):

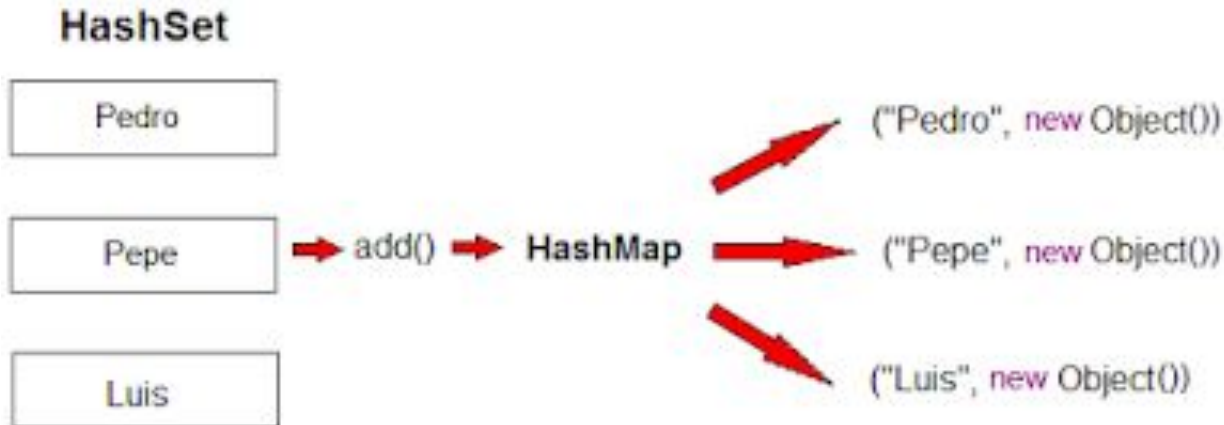
```
public boolean add(E e) {  
    return map.put(e, PRESENT) != null;  
}
```



```
Object PRESENT = new Object();
```

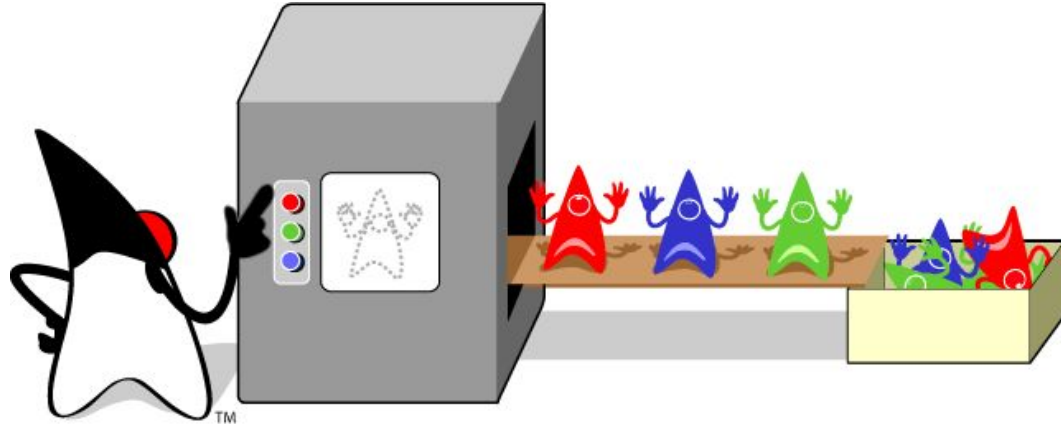

HashSet (4)

- Si ese objeto ya está agregado devuelve el valor anterior de esa key pero si el objeto no está y se agrega al HashMap devuelve null, entonces lo que hace el método `add()` es evaluar que devuelve y así se sabe si agrega o no el objeto.



Agenda

- Queue
- Set
- HashSet
- **LinkedHashSet**
- TreeSet
- Diagrama de decisiones para uso de colecciones

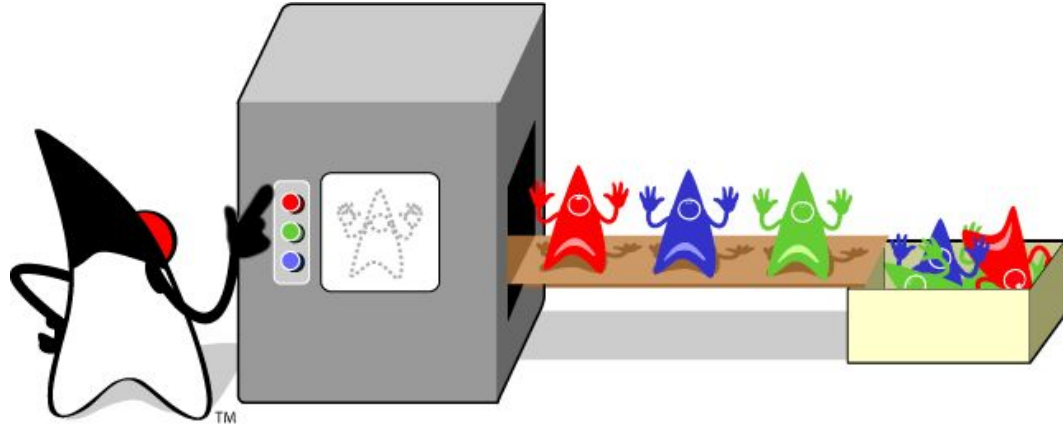


LinkedHashSet

- Es similar a HashSet pero la tabla de dispersión es doblemente enlazada.
- Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo coste constante, con la carga adicional que supone tener que gestionar los enlaces. Sin embargo habrá una mejora en la iteración, ya que al establecerse enlaces entre los elementos no tendremos que recorrer todas las entradas de la tabla, el coste sólo estará en función del número de elementos insertados.
- En este caso, al haber enlaces entre los elementos, estos enlaces definirán el **orden** en el que se insertaron en el conjunto, por lo que el orden de iteración será el mismo orden en el que se insertaron.

Agenda

- Queue
- Set
- HashSet
- ~~LinkedHashSet~~
- **TreeSet**
- Diagrama de decisiones para uso de colecciones



TreeSet

- Esta implementación almacena los elementos ordenándolos en función de sus valores.
- Es bastante más lento que HashSet.
- Los elementos almacenados deben implementar la **interfaz Comparable**.
- Garantiza, siempre, un rendimiento de $\log(N)$ en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

Agenda

- Queue
- Set
- HashSet
- LinkedHashSet
- TreeSet
- Diagrama de decisiones para uso de colecciones

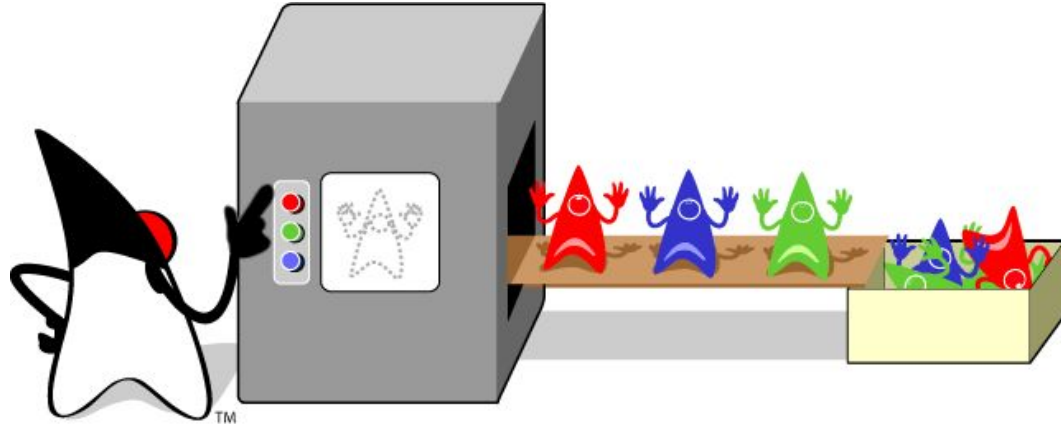
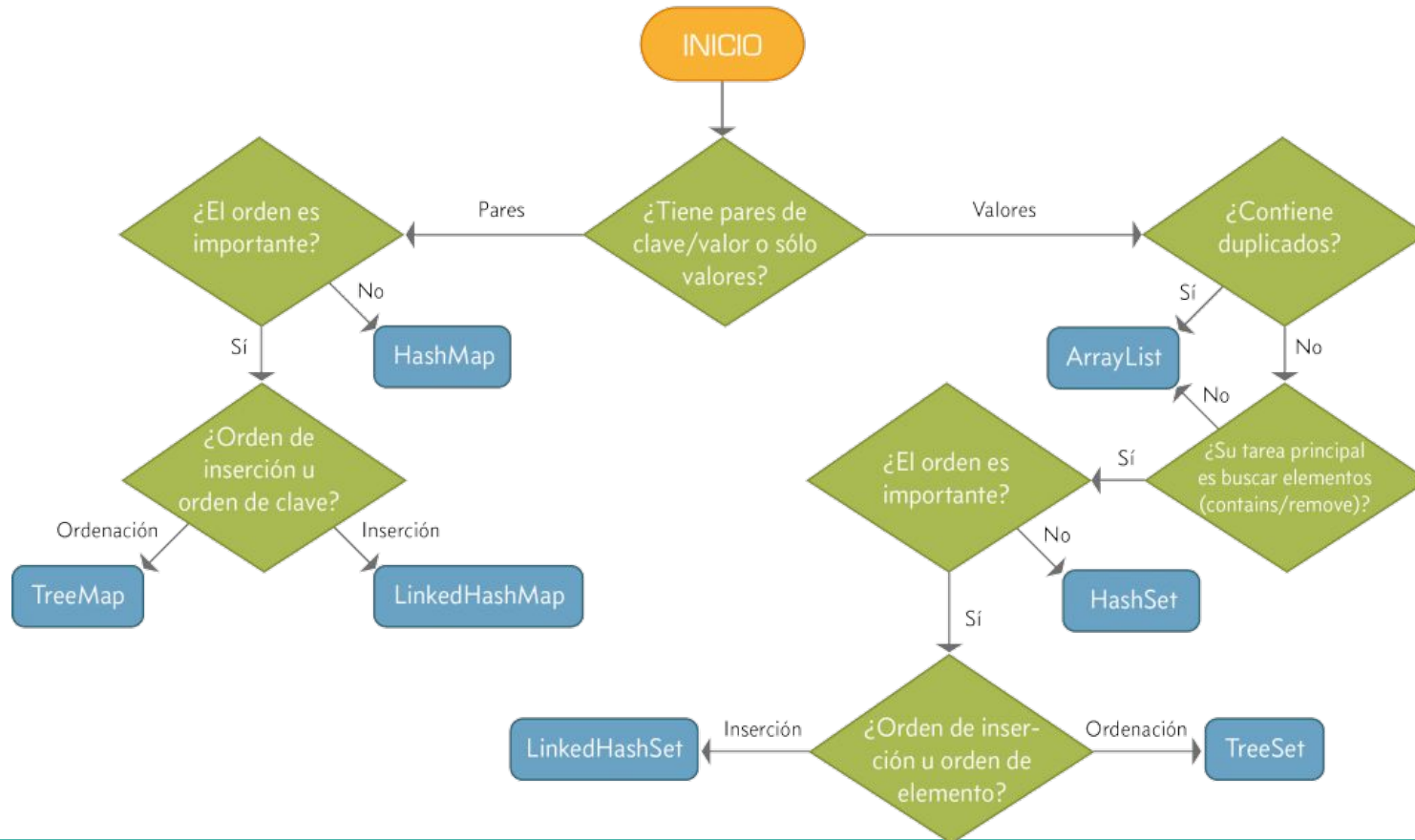


Diagrama de decisión para uso de colecciones Java



Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/collections/implementations/set.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>
- <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>
- <https://docs.oracle.com/javase/tutorial/collections/implementations/summary.html>

Clase 15: Genéricos

— Programación & Laboratorio III —

Agenda

- **Ejemplo**
- Genericidad
- Declaración de tipos genéricos
- Declaración de una clase genérica
- Tipos genéricos - Convención
- Ejemplos



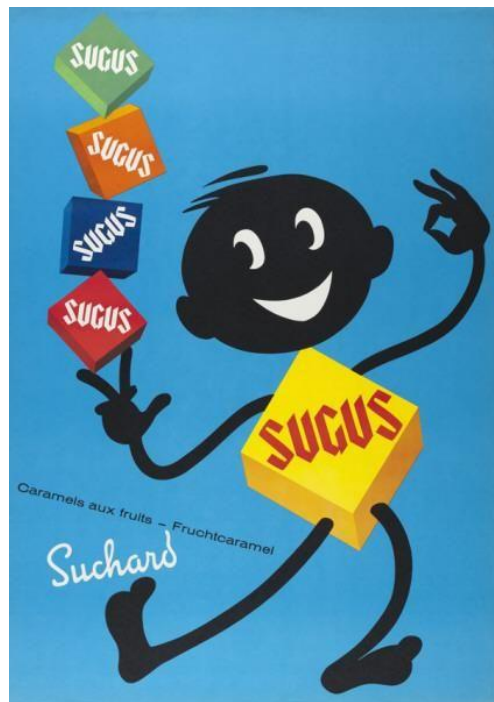
Ejemplo

```
public class Caja {  
  
    private List<Object> elementos = new ArrayList<>();  
    private int tope;  
  
    private Caja(int tope) {  
        this.tope = tope;  
    }  
  
    public boolean agregarElemento(Object o) {  
        if (tope < elementos.size()) {  
            elementos.add(o);  
            return true;  
        }  
        return false;  
    }  
}
```



Ejemplo (2)

```
public class Caramelo {  
  
    private String marca;  
    private String sabor;  
  
    private Caramelo(String marca, String sabor) {  
        this.marca = marca;  
        This.sabor = sabor;  
    }  
  
    @Override  
    public String toString() {  
        return "Caramelo= [marca= " + marca +  
            ", sabor= " + sabor + "]";  
    }  
}
```



Ejemplo (3)

```
public class Main {  
  
    public static void main(String[] args) {  
        Caja miCaja = new Caja(10);  
  
        Caramelo c1 = new Caramelo("Sugus", "menta");  
  
        miCaja.agregarElemento(c1);  
  
        Perro p1 = new Perro();  
        Gato g1 = new Gato();  
  
        miCaja.agregarElemento(p1);  
        miCaja.agregarElemento(g1);  
  
        ...  
    }  
}
```



Ejemplo (4)

...

```
//Imprimir los caramelos de la caja
for (Object o : miCaja.getElementos()) {
    if (o instanceof Caramelo) {
        System.out.println(((Caramelo) o).toString());
    }
}
}
```

Conclusión

- El uso de `Object` como una referencia genérica es potencialmente inseguro y no se puede hacer nada para que el programador no cometa un error equivocado.
- El error es descubierto en tiempo de ejecución, al momento de realizar el casteo cuando se lanza una excepción del tipo `ClassCastException`.

Solución

```
public class Caja<T> {  
  
    private List<T> elementos = new ArrayList<>();  
    private int tope;  
  
    private Caja(int tope) {  
        this.tope = tope;  
    }  
  
    public boolean agregarElemento(T t) {  
        if (tope < elementos.size()) {  
            elementos.add(t);  
            return true;  
        }  
        return false;  
    }  
}
```


Solución (2)

```
public class Main {  
  
    public static void main(String[] args) {  
        Caja<Caramelo> miCaja = new Caja(10);  
  
        Caramelo c1 = new Caramelo("Sugus", "menta");  
        miCaja.agregarElemento(c1);  
  
        Perro p1 = new Perro();  
        miCaja.agregarElemento(p1);  
  
        //Imprimir los caramelos de la caja  
        for (Caramelo c : miCaja.getElementos()) {  
            System.out.println(c.toString());  
        }  
    }  
}
```

Error en tiempo de compilación

Agenda

— ~~Ejemplo~~

- **Genericidad**
- Declaración de tipos genéricos
- Declaración de una clase genérica
- Tipos genéricos - Convención
- Ejemplos



Genericidad

- El término “genericidad” se refiere a una serie de técnicas que permiten escribir algoritmos o definir contenedores de forma que puedan aplicarse un amplio rango de tipos de datos.
- Es una construcción importante en los lenguajes de programación orientada a objetos, que si bien no es exclusiva de este tipo de lenguajes, ha adquirido verdadera importancia y uso.
- Permite definir una clase o un método sin especificar el tipo de dato o parámetros, de esta forma se puede cambiar la clase para adaptarla a diferentes usos sin tener que reescribirla.

Genericidad (2)

- La razón de la genericidad se base principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de dato que procesan. Por ejemplo: un algoritmo que implementa una pila de caracteres es igual al que se usa para implementar una pila de números enteros.
- La programación genérica significa escribir un código que puedan reutilizar muchos tipos diferentes de objetos

Agenda

- ~~— Ejemplo~~
- ~~— Genericidad~~
- **Declaración de tipos genéricos**
- Declaración de una clase genérica
- Tipos genéricos - Convención
- Ejemplos



Declaración de tipos genéricos

- Una declaración de tipos genéricos puede tener múltiples tipos parametrizados, separados por comas.
- Todas las invocaciones de clases genéricas son expresiones de una clase. Al instanciar una clase genérica no se crea una nueva clase.
- No se puede usar el tipo genérico <T> como tipo de un campo estático o en cualquier lugar dentro de un método estático o inicializador estático. Por ejemplo:

```
private static final T MI_CONSTANTE = new T();
```

Declaración de tipos genéricos (2)

- No se puede usar un tipo de datos genérico en la creación de objetos y arreglos. Por ejemplo:

```
public class Main {  
  
    public static void main(String[] args) {  
        T[] miArreglo = new T[10];  
  
        T t = new T();  
        miArreglo[0] = t;  
  
        //Imprimir los elementos del arreglo  
        for (int i=0; i<10; i++) {  
            System.out.println(t.toString());  
        }  
    }  
}
```

Declaración de tipos genéricos (3)

- Dentro de una definición de clases, el tipo genérico puede aparecer en cualquier declaración no estática donde se podría utilizar cualquier tipo de datos concreto.

```
public class Caja<T> {  
  
    private List<T> elementos = new ArrayList<>();  
    ...  
  
    public boolean agregarElemento(T t) {  
        ...  
    }  
}
```


Agenda

- ~~— Ejemplo~~
- ~~— Genericidad~~
- ~~— Declaración de tipos genéricos~~
- **Declaración de una clase genérica**
- Tipos genéricos - Convención
- Ejemplos



Declaración de una clase genérica

- Una clase genérica o parametrizable es una clase con una o más variables de tipo genérico.

```
public class MiClase<tipo_genérico> {  
  
    ...  
  
}
```

Donde “MiClase” es el nombre de la clase genérica y “tipo_genérico” es el tipo parametrizado genérico.

Agenda

- ~~— Ejemplo~~
- ~~— Genericidad~~
- ~~— Declaración de tipos genéricos~~
- ~~— Declaración de una clase genérica~~
- **Tipos genéricos - Convención**
- Ejemplos



Tipos genéricos - Convención

- $E \rightarrow$ elemento de una colección
- $K \rightarrow$ clave
- $N \rightarrow$ número
- $T \rightarrow$ tipo
- $V \rightarrow$ valor
- S, U, V , etc. \rightarrow para segundos, terceros y cuartos tipos.

Agenda

- ~~— Ejemplo~~
- ~~— Genericidad~~
- ~~— Declaración de tipos genéricos~~
- ~~— Declaración de una clase genérica~~
- ~~— Tipos genéricos Convención~~
- **Ejemplos**



Ejemplos

- Ejemplo 1:

```
private Box<Integer> numeros = new Box<>();
```

- Ejemplo 2:

```
public interface Par<K, V> {
```

```
    ...
```

```
}
```

```
public class ParOrdenado<K, V> implements Pair<K,V> {
```

```
}
```

Ejemplos (2)

- Ejemplo 3:

```
public class NumeroNatural<T extends Integer> {  
  
    ...  
  
}
```

- Ejemplo 4:

```
public class A {...}
```

```
public interface B {...}
```

```
public class C <T extends A & B> {...}
```

Bibliografía oficial

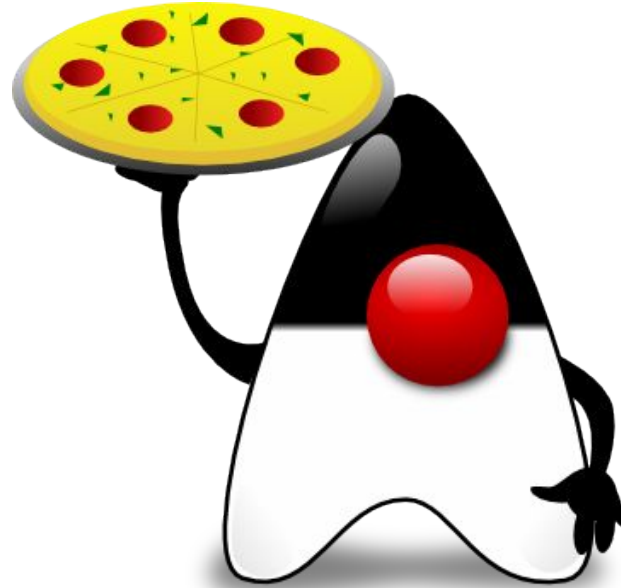
- <https://docs.oracle.com/javase/tutorial/java/generics/types.html>

Clase 16: Enum

— Programación & Laboratorio III —

Agenda

- **Enum - Características**
- Ejemplo 1
- Ejemplo 2
- Más características
- Ejemplo 3
- Operador "=="



Enum - Características

- La palabra reservada “enum” fue introducida en Java 5 para representar el tipo especial de clase que siempre extiende de java.lang.Enum.
- Constantes definidas de esta manera hacen el código más legible, permiten verificación en tiempo de compilación, documentan por adelantado la lista de valores aceptados y evitan el comportamiento inesperado si es que se reciben valores no válidos.
- Ejemplo:

```
public enum PizzaStatus {  
    ORDERED,  
    READY,  
    DELIVERED;  
}
```

Enum - Características (2)

- Un tipo enumerado restringe los posibles valores que puede tomar una variable. Esto ayuda a reducir los errores en el código y permite algunos usos especiales interesantes.
- Por convención, los nombres de los valores que puede tomar **se escriben en letras mayúsculas** para recordarnos que son valores fijos (que en cierto modo podemos ver como constantes).
- Una vez declarado el tipo enumerado, todavía no existen variables hasta que no las creamos explícitamente:

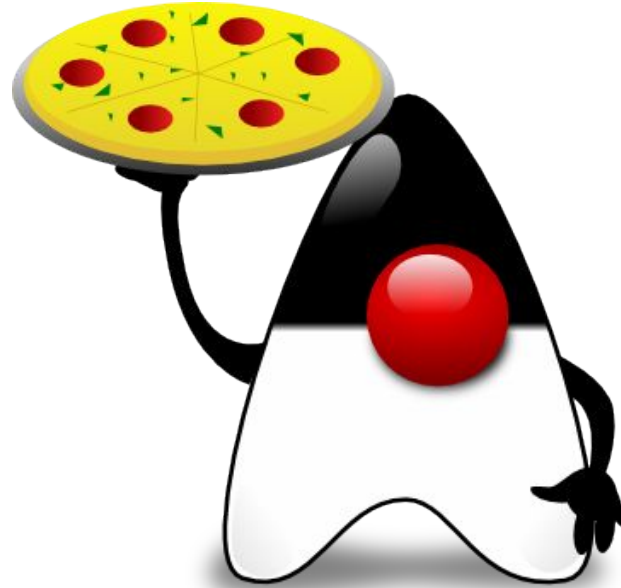
```
TipoEnumerado nombreVariable; → PizzaStatus status;
```

Enum - Características (3)

- Un tipo enumerado puede ser declarado dentro o fuera de una clase, pero no dentro de un método. Por tanto no podemos declarar un enum dentro de un método main; si lo hacemos nos saltará el error de compilación `"enum types must not be local"` (los tipos enumerados no deben ser locales a un método).
- Declararemos un tipo enumerado como una clase aparte o dentro de otra, pero fuera de cualquier método o constructor.

Agenda

- ~~Enum Características~~
- **Ejemplo 1**
- Ejemplo 2
- Más características
- Ejemplo 3
- Operador "=="

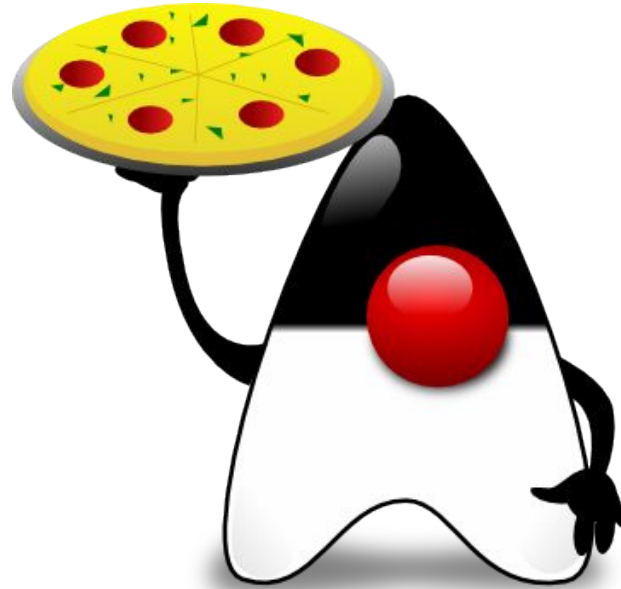


Ejemplo 1: Enum declarado dentro de clase

```
public class Pizza {  
  
    private PizzaStatus status;  
  
    public enum PizzaStatus {  
        ORDERED,  
        READY,  
        DELIVERED;  
    }  
  
    public boolean isDeliverable() {  
        if (status == PizzaStatus.READY) {  
            return true;  
        }  
        return false;  
    }  
}
```

Agenda

- ~~Enum Características~~
- ~~Ejemplo 1~~
- **Ejemplo 2**
- Más características
- Ejemplo 3
- Operador "=="

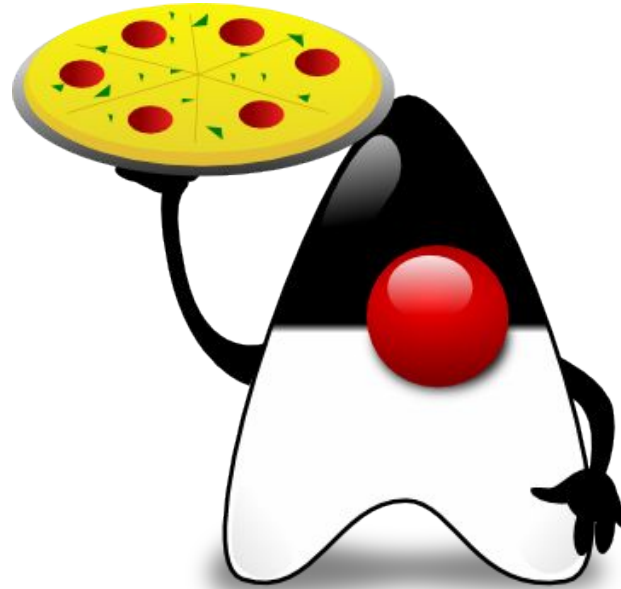


Ejemplo 2: Enum como clase

```
public enum PizzaStatus {  
    ORDERED,  
    READY,  
    DELIVERED;  
  
    public boolean isDeliverable() {  
        if (status == PizzaStatus.READY) {  
            return true;  
        }  
        return false;  
    }  
}
```

Agenda

- ~~Enum Características~~
- ~~Ejemplo 1~~
- ~~Ejemplo 2~~
- **Más características**
- Ejemplo 3
- Operador "=="

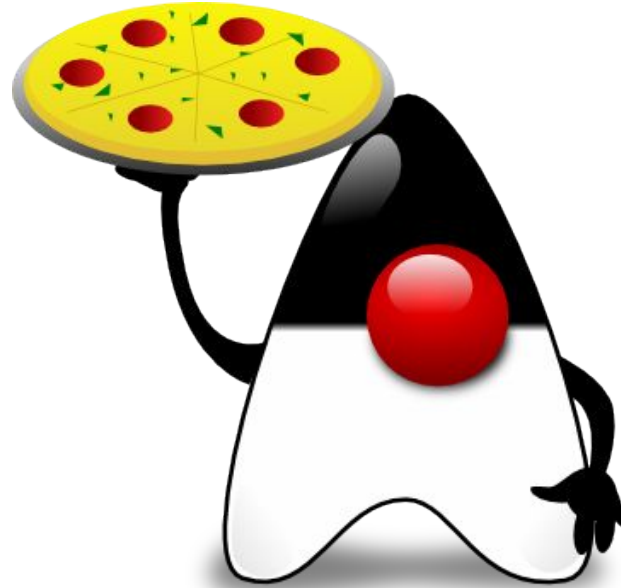


Más características

- Se dispone automáticamente de métodos especiales como por ejemplo el método `values()`, que el compilador agrega automáticamente cuando se crea un enum. Este método devuelve un array conteniendo todos los valores del enumerado en el orden en que son declarados
- Un tipo enumerado puede añadir campos constantes al objeto enumerado y recuperar esos campos. Para ello se usa un constructor especial para tipos enumerados.

Agenda

- Enum Características
- Ejemplo 1
- Ejemplo 2
- Más características
- **Ejemplo 3**
- Operador "=="



Ejemplo 3: Enum con constructor

```
public enum TipoDeMadera {  
    ROBLE ("Castaño verdoso"),  
    CAOBA ("Marrón oscuro"),  
    NOGAL("Marrón rojizo");
```

Por defecto el constructor es “private”, es redundante escribirlo.

```
    private String color;
```

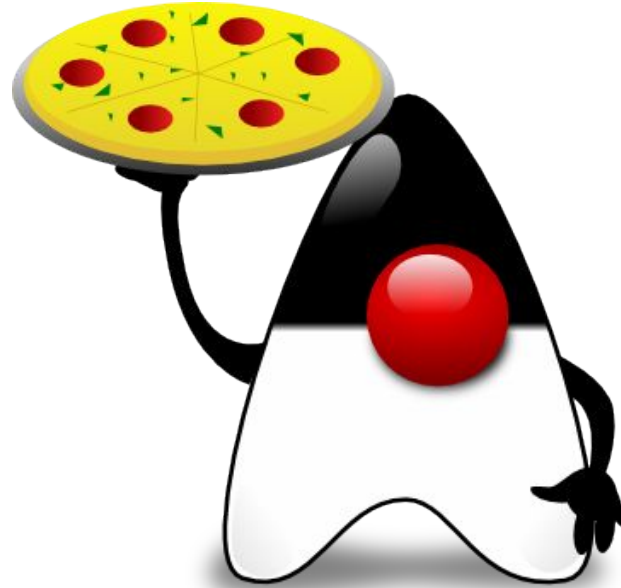
```
    TipoDeMadera (String color) {  
        this.color = color;  
    }
```

```
    public String getColor() {  
        return color;  
    }
```

```
}
```

Agenda

- ~~Enum~~ Características
- ~~Ejemplo 1~~
- ~~Ejemplo 2~~
- ~~Más características~~
- ~~Ejemplo 3~~
- **Operador "=="**



Operador “==”

- Usando el tipo enum nos aseguramos que sólo exista una instancia de la constante en la JVM. por lo tanto se puede usar el operador “==” para comparar dos variables del mismo tipo enum.
- Seguridad en tiempo de ejecución:

```
1)  if(pizza.getStatus().equals(Pizza.PizzaStatus.DELIVERED));  
2)  if(pizza.getStatus() == Pizza.PizzaStatus.DELIVERED);
```

En la opción 1) si el status de pizza es null, obtendremos una excepción del tipo `NullPointerException`. No así en la opción 2)

Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>
- <https://docs.oracle.com/javase/8/docs/technotes/guides/language/enum.html>

Clase 17: Excepciones

— Programación & Laboratorio III —

Agenda

- **Concepto de excepción**
- Tipos de excepciones
- Excepciones checked o comprobadas
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Concepto de excepción

- Una excepción en Java es un error o situación “excepcional” que se produce durante la ejecución de un programa.
- Ejemplos:
 - Leer un archivo que no existe
 - Acceder al valor N de una colección que tiene menos de N elementos.
 - Enviar o recibir información por red mientras se produce una pérdida de conectividad.
- Todas las excepciones en Java se representan a través de objetos que heredan de `java.lang.Throwable`.

Concepto de excepción (2)

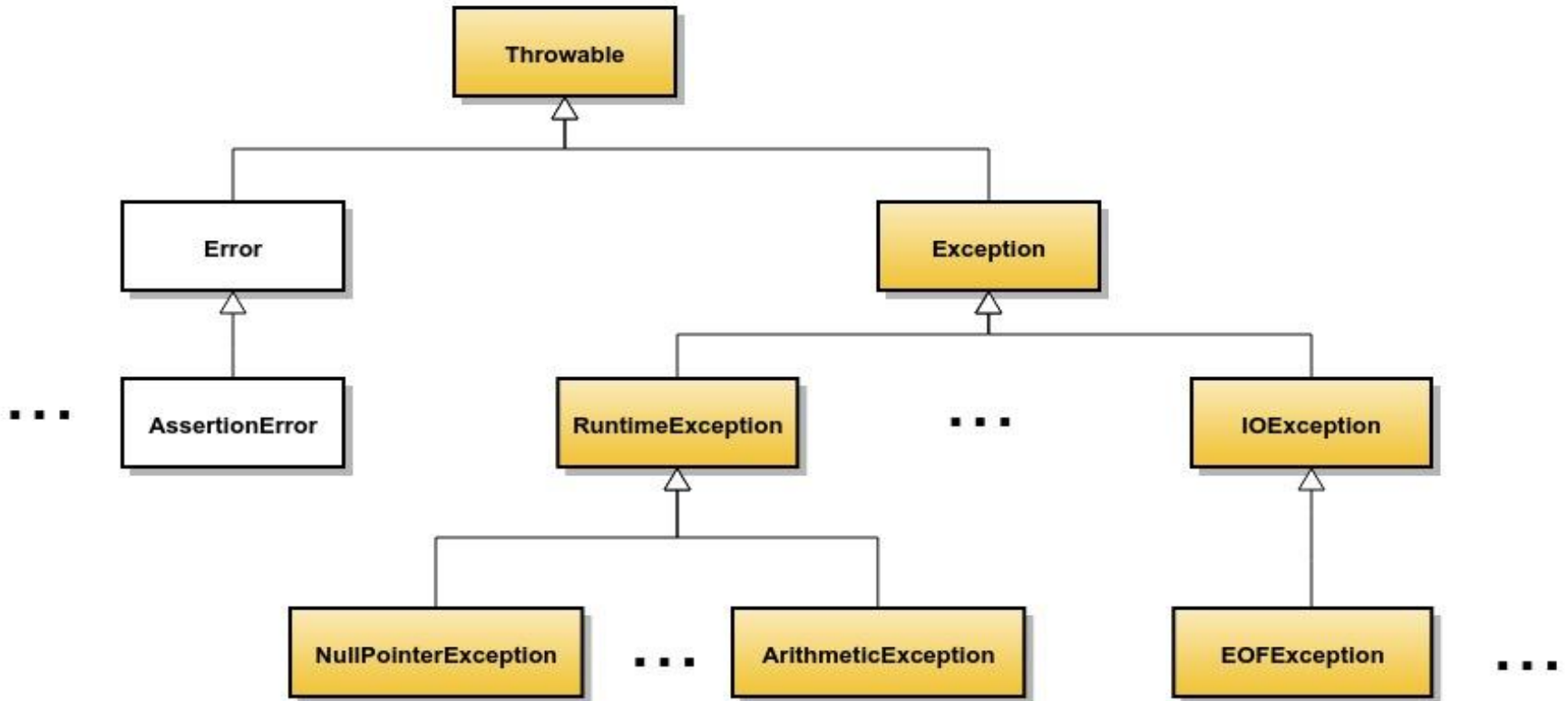
- Las excepciones proporcionan una forma clara de comprobar posibles errores sin oscurecer el código.
- Convierten las condiciones de error que un método puede señalar en una parte explícita del contrato del método.
- La lista de excepciones pueden ser vistas por el programador, comprobada por el compilador y preservada por las clases extendidas que redefinen el método.

Agenda

- ~~Concepto de excepción~~
- **Tipos de excepciones**
- Excepciones checked o comprobadas
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Tipos de excepciones



Tipos de excepciones (2)

- Las excepciones también son objetos.
- Todos los tipos de excepción deben extender de la clase `Throwable` o de alguna de sus subclases.
- De `Throwable` nacen dos ramas: `Error` y `Exception`.
 - 1) `Error` representa errores de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos. Por ejemplo: errores de la JVM, desbordamiento de buffer.
 - 2) `Exception` representa aquellos errores que normalmente sí solemos gestionar.

Tipos de excepciones (3)

- Por convenio, los nuevos tipos de excepción extienden a `Exception`.
- De `Exception` nacen múltiples ramas: `ClassNotFoundException`, `IOException`, `ParseException`, `SQLException` → **Excepciones checked.**
- `RuntimeException` es la única **excepción unchecked.**

Agenda

- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- **Excepciones checked o comprobadas**
- Excepciones unchecked o no comprobadas
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Excepciones checked (comprobadas)

- Una excepción de tipo checked representa un error del cual técnicamente podemos recuperarnos.
- Son todas las situaciones que son totalmente ajenas al propio código, por ejemplo: fallo de una operación de lectura/escritura.
- Este tipo de excepciones deben ser **capturadas** o **relanzadas**.

Agenda


- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- ~~— Excepciones checked o comprobadas~~
- **Excepciones unchecked o no comprobadas**
- Lanzamiento de excepciones
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Excepciones unchecked (no comprobadas)

- Representan errores de programación. Por ejemplo: intentar leer de un array de N elementos un elemento que se encuentra en una posición mayor que N.

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};  
System.out.println(numerosPrimos[10]);
```

 `ArrayIndexOutOfBoundsException`

Agenda

- ~~Concepto de excepción~~
- ~~Tipos de excepciones~~
- ~~Excepciones checked o comprobadas~~
- ~~Excepciones unchecked o no comprobadas~~
- **Lanzamiento de excepciones**
- Transferencia de control
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Lanzamiento de excepciones

- Las excepciones se lanzan utilizando la palabra reservada throw:

```
throw AException
```

- AException debe ser un objeto Throwable.
- Si se lanza la excepción no se regresa al flujo normal del programa.
- Las excepciones son objetos, por lo tanto se debe crear una instancia antes de lanzarse.

Lanzamiento de excepciones (2)

- Ejemplo:

```
public void buscarNumeroPrimo(int n) {  
    for (int i=0; i<10; i++) {  
        if (numerosPrimos[i] == n) {  
            System.out.println("El número " + n + " existe.");  
            break;  
        }  
    }  
    throw new NumeroNoExisteException(n);  
}
```

Agenda

- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- ~~— Excepciones checked o comprobadas~~
- ~~— Excepciones unchecked o no comprobadas~~
- ~~— Lanzamiento de excepciones~~
- **Transferencia de control**
- Cláusula throws
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Transferencia de control

- Una vez que se produce una excepción, las acciones que hubiera detrás del punto donde se produjo la excepción no tienen lugar.
- Las acciones que sí se ejecutarán serán las contenidas dentro de los bloques `catch` y `finally`.

Transferencia de control (2)

- Ejemplo:

```
public static void main(String[] args) {  
    FileWriter fichero = null;  
    try {  
        fichero = new FileWriter(PATH_ARCHIVO);  
        fichero.write("Escribiendo...")  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        if (fichero != null) {  
            fichero.close();  
        }  
    }  
}
```

Transferencia de control (3)

- Las sentencias dentro de la cláusula `try` se ejecutan hasta que se lance una excepción o hasta que finalice con éxito.
- Si se lanza una excepción, se examinan sucesivamente las sentencias de la cláusula `catch`.
- La cláusula `finally` de una sentencia proporciona un mecanismo para ejecutar una sección de código, se lance o no una excepción.
- Generalmente la cláusula `finally` se utiliza para limpiar el estado interno o para liberar recursos, como archivos abiertos o cerrar conexiones a bases de datos.

Agenda

- ~~— Concepto de excepción~~
- ~~— Tipos de excepciones~~
- ~~— Excepciones checked o comprobadas~~
- ~~— Excepciones unchecked o no comprobadas~~
- ~~— Lanzamiento de excepciones~~
- ~~— Transferencia de control~~
- **Cláusula throws**
- Creando nuestras propias excepciones
- Malas prácticas
- Buenas prácticas



Cláusula throws

- Se utiliza para declarar las excepciones checked que puede lanzar un método.
- Se pueden declarar varias, separadas por comas.
- RuntimeException y Error son las únicas excepciones que no hace falta incluir en las cláusulas throws.

Cláusula throws (2)

- Si se invoca a un método que tiene una excepción checked en su cláusula throws, existen tres opciones:
 - 1) Capturar la excepción y gestionarla.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}
```

```
public void leerArchivo(String path) {  
    try {  
        abrirArchivo(path);  
    } catch(IOException e) {  
        System.out.println("Se produjo un error al abrir el archivo").  
    }  
}
```

Cláusula throws (3)

- 2) Capturar la excepción y transformarla en una de nuestras excepciones.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) {  
    try {  
        abrirArchivo(path);  
    } catch(IOException e) {  
        throw new ArchivoNoExisteException(e);  
    }  
}
```

Cláusula throws (4)

3) Declarar la excepción en la cláusula throws y hacer que otro método la gestione.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}
```

```
public void leerArchivo(String path) throws IOException {  
    abrirArchivo(path);  
    ...  
}
```


Cláusula throws (5)

- No se permite que los métodos redefinidos declaren más excepciones checked en la cláusula throws que las que declara el método.
- Se pueden lanzar subtipos de las excepciones declaradas ya que podrán ser capturadas en el bloque catch correspondiente a su supertipo.
- Si una declaración de un método se hereda en forma múltiple, la cláusula throws de ese método

Agenda

- ~~Concepto de excepción~~
- ~~Tipos de excepciones~~
- ~~Excepciones checked o comprobadas~~
- ~~Excepciones unchecked o no comprobadas~~
- ~~Lanzamiento de excepciones~~
- ~~Transferencia de control~~
- ~~Cláusula throws~~
- **Creando nuestras propias excepciones**
- Malas prácticas
- Buenas prácticas



Creando nuestras excepciones

```
public class SaldoInsuficienteException extends Exception {  
    ...  
}  
  
public static void main(String[] args) {  
    Cliente cliente = new Cliente();  
    try {  
        cliente.pagar();  
    } catch(SaldoInsuficienteException e) {  
        e.printStackTrace();  
    }  
}
```

Agenda

- ~~Concepto de excepción~~
- ~~Tipos de excepciones~~
- ~~Excepciones checked o comprobadas~~
- ~~Excepciones unchecked o no comprobadas~~
- ~~Lanzamiento de excepciones~~
- ~~Transferencia de control~~
- ~~Cláusula throws~~
- ~~Creando nuestras propias excepciones~~
- **Malas prácticas**
- Buenas prácticas



Malas prácticas

1

```
FileWriter fichero = null;  
try {  
    fichero = new FileWriter("path");  
    fichero.write("Writing...");  
} catch (IOException e) {  
  
}
```

2

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};  
int undecimoPrimo = numerosPrimos[10];  
try {  
    int i = 0;  
    while(true) {  
        System.out.println(numerosPrimos[i++]);  
    }  
} catch (ArrayIndexOutOfBoundsException e) {}
```

Malas prácticas

3

```
public void pagar() throws Exception{  
    //Do something  
}
```

4

```
public void pagar() {  
    try {  
  
    } catch (Exception e) {  
        throw new RuntimeException();  
    }  
}
```

Agenda

- ~~Concepto de excepción~~
- ~~Tipos de excepciones~~
- ~~Excepciones checked o comprobadas~~
- ~~Excepciones unchecked o no comprobadas~~
- ~~Lanzamiento de excepciones~~
- ~~Transferencia de control~~
- ~~Cláusula throws~~
- ~~Creando nuestras propias excepciones~~
- ~~Malas prácticas~~
- **Buenas prácticas**



Buenas prácticas

- Utilizar excepciones que ya existen.
- Lanzar excepciones de acuerdo al nivel de abstracción en el que nos encontramos.
- Documentar con Javadoc todas las excepciones que se lanzan en los métodos .

Bibliografía oficial

- <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html>

Clase 18: Manejo de archivos

— Programación & Laboratorio III —

Agenda

- **Concepto de archivo**
- Archivos en Java
- Concepto de buffering
- Escribir un archivo
- Leer un archivo
- Cerrar un archivo



Concepto de archivo

- Desde el punto de vista de más bajo nivel, podemos definir un archivo (o fichero) como un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica

Por qué usar archivos?

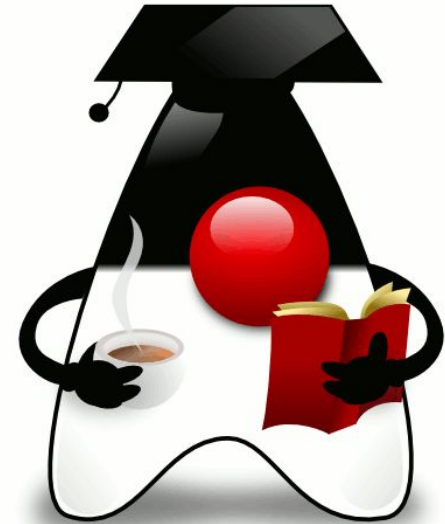
- Con frecuencia tendremos que guardar los datos de nuestro programa para poderlos recuperar más adelante. Los archivos se usan cuando el volumen de datos no es relativamente muy elevado.

Concepto de archivo (2)

- Los archivos suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos del disco.
- El nombre de un archivo está relacionado con su posición en el árbol de directorios que lo contiene, lo que permite no sólo identificar unívocamente cada fichero, sino encontrarlo en el disco a partir de su nombre.
- Los archivos suelen tener una serie de metadatos asociados, como pueden ser su fecha de creación, la fecha de última modificación, su propietario o los permisos que tienen diferentes usuarios sobre ellos

Agenda

- ~~Concepto de archivo~~
- **Archivos en Java**
- Concepto de buffering
- Escribir un archivo
- Leer un archivo
- Cerrar un archivo



Archivos en Java

- La clase `File` en java representa un fichero y nos permite obtener información sobre él, como por ejemplo atributos, directorios, etc.
- Tiene tres constructores:
 - `File(String path)`
 - `File(String path, String name)`
 - `File(File dir, String name)`

El parámetro “path” indica el camino hacia el directorio donde se encuentra el archivo, y “name” indica el nombre del archivo.

Archivos en Java (2)

- Algunos métodos importantes:
 - `String getName()`
 - `String getPath()`
 - `boolean exists()`
 - `boolean canWrite()`
 - `boolean canRead`
 - `boolean isFile()`
 - `boolean isDirectory()`
 - `long lastModified()`
 - `boolean renameTo(File dest);`
 - `boolean delete()`
 - `String[] list()`



Archivos en Java (3)

- Al instanciar la clase `File` con un archivo, no se abre el fichero ni es necesario que este exista y precisamente esa es la primera información útil que podemos obtener del fichero: saber si existe o no.

```
import java.io.File;
...
File f = new File("path/mi_fichero.txt");
if (f.exists())
    // El fichero existe.
else
    // el fichero no existe.
```

Archivos en Java (4)

- Si el archivo existe, es decir, si la función `exists()` devuelve `true`, entonces podemos obtener información acerca del archivo. Por ejemplo:

```
if(f.exists()){  
    System.out.println("Nombre del archivo " + f.getName());  
    System.out.println("Camino          " + f.getPath());  
    System.out.println("Camino absoluto  " + f.getAbsolutePath());  
    System.out.println("Se puede escribir " + f.canRead());  
    System.out.println("Se puede leer    " + f.canWrite());  
    System.out.println("Tamaño        " + f.length());  
}
```

Archivos en Java (5)

- Si el archivo existe, podemos no tener permisos para leerlo, así que otra comprobación útil es si se puede hacer en él la operación deseada.

```
if (f.canRead())  
    // El archivo existe y se puede leer.
```

```
if (f.canWrite())  
    // El archivo existe y se puede escribir en él
```

```
if (f.canExecute())  
    // El archivo existe y se puede ejecutar
```

Archivos en Java (6)

- Tenemos, además, los correspondientes métodos `setReadable()`, `setReadOnly()`, `setWritable()` y `setExecutable()` que nos permiten cambiar las propiedades del archivo siempre que seamos los propietarios del mismo o tengamos permisos para hacerlo.

Archivos en Java (7)

- La siguiente comprobación útil que se puede hacer con un `File` es determinar si es un archivo normal o es un directorio. Si es un directorio, podemos obtener un listado de los ficheros contenidos en él.

```
// Se crea el File del directorio
File directorio = new File("src/main/java/com/chuidiang/ejemplos/");

// Si es un directorio
if (directorio.isDirectory()) {
    // obtenemos su contenido
    File[] ficheros = directorio.listFiles();

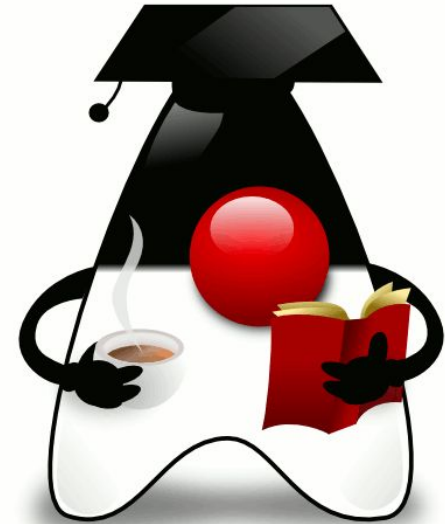
    for (File fichero : ficheros) {
        System.out.println(fichero.getName());
    }
}
```

Archivos en Java (8)

- Son archivos que podremos crear desde un programa Java y leer con cualquier editor de texto, o bien crear con un editor de textos y leer desde un programa Java.
- Para manipular archivos, siempre tendremos los mismos pasos:
 - 1) Abrir el archivo → Si no abrimos el archivo, obtendremos un mensaje de error al intentar acceder a su contenido.
 - 2) Escribir datos o leerlos.
 - 3) Cerrar el archivo → Si no cerramos el archivo, puede que realmente no se llegue a guardar ningún dato.

Agenda

- ~~— Concepto de archivo~~
- ~~— Archivos en Java~~
- **Concepto de buffering**
- Escribir un archivo
- Leer un archivo
- Cerrar un archivo



Concepto de Buffering

“Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que esté lleno. Por eso se realizarán menos viajes cuando usas el carrito”

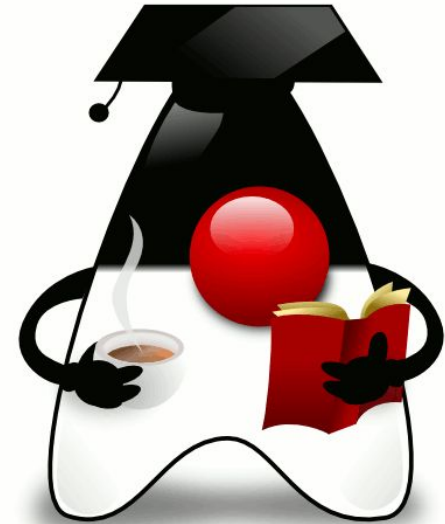
- Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante reducir las operaciones de lectura/escritura que realizamos sobre los archivos.

Concepto de Buffering (2)

- Un buffer es una estructura de datos que permite el acceso por trozos a una colección de datos.
- Los buffers son útiles para evitar almacenar en memoria grandes cantidades de datos, en lugar de ello, se va pidiendo al buffer porciones pequeñas de los datos que se van procesando por separado. También resultan muy útiles para que la aplicación pueda ignorar los detalles concretos de eficiencia de hardware subyacente, la aplicación puede escribir al buffer cuando quiera, que ya se encargará el buffer de escribir a disco siguiendo los ritmos más adecuados y eficientes.

Agenda

- ~~— Concepto de archivo~~
- ~~— Archivos en Java~~
- ~~— Concepto de buffering~~
- **Escribir un archivo**
- Leer un archivo
- Cerrar un archivo



Escribir archivo

- Para abrir un archivo usaremos un `BufferedWriter`, que se apoya en un `FileWriter`, que a su vez usa un "File" al que se le indica el nombre del archivo

```
BufferedWriter f = new BufferedWriter(  
    new FileWriter(new File("mi_archivo.txt")));
```

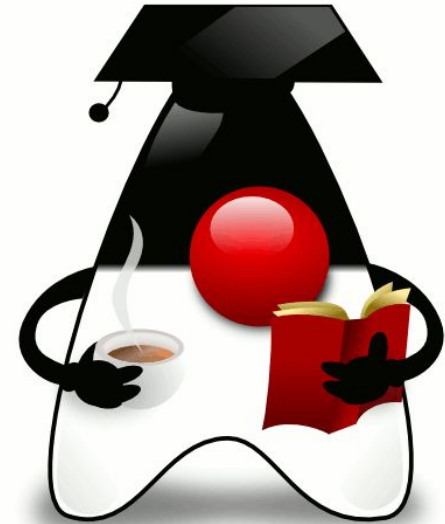
- En el caso de un archivo de texto, no escribiremos con `"println()"`, como hacíamos en pantalla, sino con `"write()"`. Cuando queramos avanzar a la línea siguiente, deberemos usar `"newline()"`:

Escribir archivo (2)

```
try {  
    BufferedWriter fSalida = new BufferedWriter(new FileWriter(new File("mi_archivo.txt")));  
    fSalida.write("Hola");  
    fSalida.newLine();  
    fSalida.write("Mundo!");  
    fSalida.newLine();  
    fSalida.close();  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
}
```

Agenda

- ~~Concepto de archivo~~
- ~~Archivos en Java~~
- ~~Concepto de buffering~~
- ~~Escribir un archivo~~
- **Leer un archivo**
- Cerrar un archivo



Leer archivo

- Para leer de un fichero de texto usaremos "readLine()", que nos devuelve un String. Si ese String es null, quiere decir que se ha acabado el archivo y no se ha podido leer nada. Por eso, lo habitual es usar un "while" para leer todo el contenido de un fichero.
- Existe otra diferencia con la escritura: no usaremos un BufferedWriter, sino un BufferedReader, que se apoyará en un FileReader:

```
BufferedReader fEntrada = new BufferedReader(  
    new FileReader(new File("mi_archivo.txt")));
```

Leer archivo (2)

- La clase `java.io.BufferedReader` resulta ideal para leer archivos de texto y procesarlos. Permite leer eficientemente caracteres aislados, arrays o líneas completas como Strings. Cada lectura a un `BufferedReader` provoca una lectura en el archivo correspondiente al que está asociado. Es el propio `BufferedReader` el que se va encargando de recordar la última posición del fichero leído, de forma que posteriores lecturas van accediendo a posiciones consecutivas del fichero.

Leer archivo (3)

```
if (!(new File("mi_archivo.txt")).exists()) {  
    return;  
}  
try {  
    BufferedReader fEntrada = new BufferedReader(new FileReader(new File("mi_archivo.txt")));  
    String linea=null;  
    while ((linea=fEntrada.readLine()) != null) {  
        System.out.println(linea);  
    }  
    fEntrada.close();  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
}
```


Agenda

- ~~Concepto de archivo~~
- ~~Archivos en Java~~
- ~~Concepto de buffering~~
- ~~Escribir un archivo~~
- ~~Leer un archivo~~
- **Cerrar un archivo**



Cerrar

```
if (!(new File("mi_archivo.txt")).exists()) {  
    return;  
}  
try {  
    BufferedReader fEntrada = new BufferedReader(new FileReader(new File("mi_archivo.txt")));  
    String linea=null;  
    while ((linea=fEntrada.readLine()) != null) {  
        System.out.println(linea);  
    }  
    fEntrada.close();  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage());  
}
```

Cerrar

```
if (!(new File("mi_archivo.txt")).exists()) {  
    return;  
}  
try {  
    BufferedReader fEntrada = new BufferedReader(new FileReader(new File("mi_archivo.txt")));  
    String linea=null;  
    while ((linea=fEntrada.readLine()) != null) {  
        System.out.println(linea);  
    }  
} catch (IOException e) {  
    System.out.println("Se produjo un error al escribir en el archivo: " + e.getMessage() );  
} finally {  
    fEntrada.close();  
}
```

Bibliografía oficial

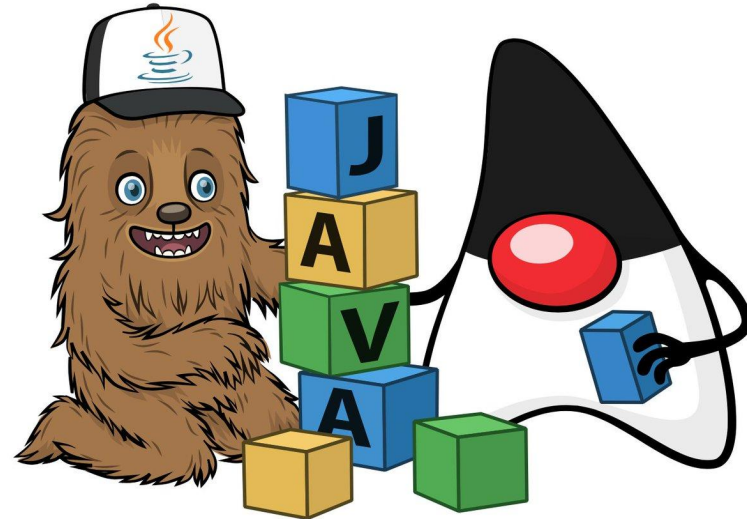
- <https://docs.oracle.com/javase/tutorial/essential/io/file.html>
- <https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

Clase 19: Serialización & JSON

— Programación & Laboratorio III —

Agenda

- **Serialización**
- Interfaz Serializable
- Serial Version UID
- Modificador transient
- Serialización y archivos
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Serialización

- La serialización permite convertir cualquier objeto (que implemente la interfaz Serializable) en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original.
- Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en una máquina que corre bajo el sistema operativo Windows, serializarlo y enviarlo a través de la red a una máquina que corre bajo UNIX donde será correctamente reconstruido.

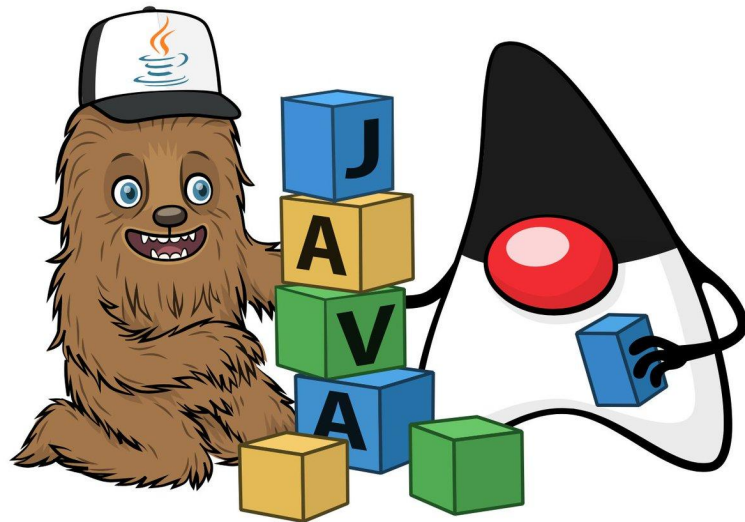
Serialización (2)

- La serialización es una característica añadida al lenguaje Java para dar soporte a:
 - La invocación remota de objetos
 - La persistencia.
- La invocación remota de objetos permite a los objetos que “viven” en otras máquinas comportarse como si vivieran en nuestra propia máquina.
- Para la persistencia, la serialización nos permite guardar el estado de un componente en disco, abandonar el IDE y restaurar el estado de dicho componente cuando se vuelve a correr el IDE.

Agenda

—Serialización

- **Interfaz Serializable**
- Serial Version UID
- Modificador transient
- Serialización y archivos
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Interfaz Serializable

- Un objeto se puede serializar si implementa la interfaz Serializable. Esta interfaz no declara ninguna función, se trata de una interfaz vacía.

```
public interface Serializable{  
}
```

- Para hacer una clase serializable simplemente debemos implementar esta interfaz:

```
public class Persona implements Serializable{  
    private String nombre;  
    ...  
}
```

Interfaz Serializable (2)

- Si dentro de la clase hay atributos que son otras clases, éstos a su vez también deben implementar Serializable. Con los tipos de java (String, Integer, etc.) no hay problema porque lo son. Por ejemplo:

```
public class Persona implements Serializable {  
    private String nombre;  
    private Direccion direccion;  
}
```

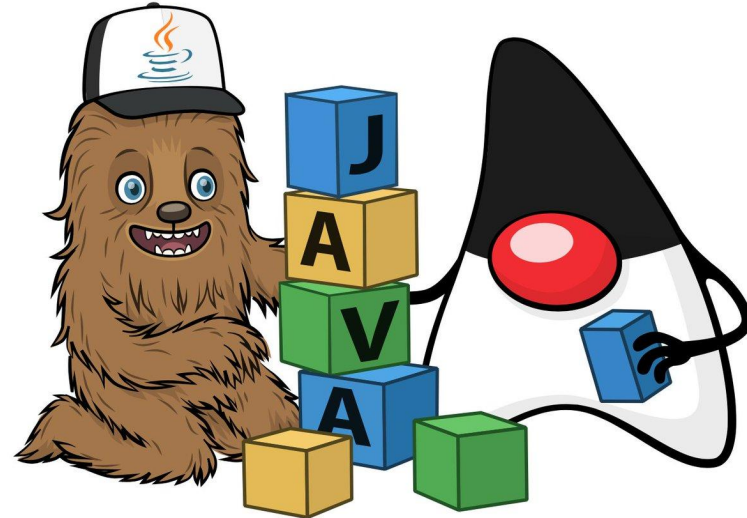
```
public class Direccion implements Serializable {  
    private String calle;  
    private String ciudad;  
    ...  
}
```

Interfaz Serializable (3)

- La serialización de un objeto como atributo de otro objeto se puede tomar como un árbol, donde las hojas son objetos que forman parte de otro objeto como un atributo, y todos ellos son marcados como serializables, para así entonces serializar primero las hojas del árbol y finalizar con la raíz.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- **Serial Version UID**
- Modificador transient
- Serialización y archivos
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Serial Version UID

- Cuando pasamos objetos serializables de un lado a otro, puede llegar a pasar que en las distintas versiones de nuestro programa la clase cambie. De esta manera es posible que un lado tenga una versión más antigua que en el otro lado. Si esto sucede, la reconstrucción de la clase en el lado que recibe es imposible.
- Para evitar este problema, se aconseja que la clase que queremos serializar tenga un atributo de la siguiente forma:

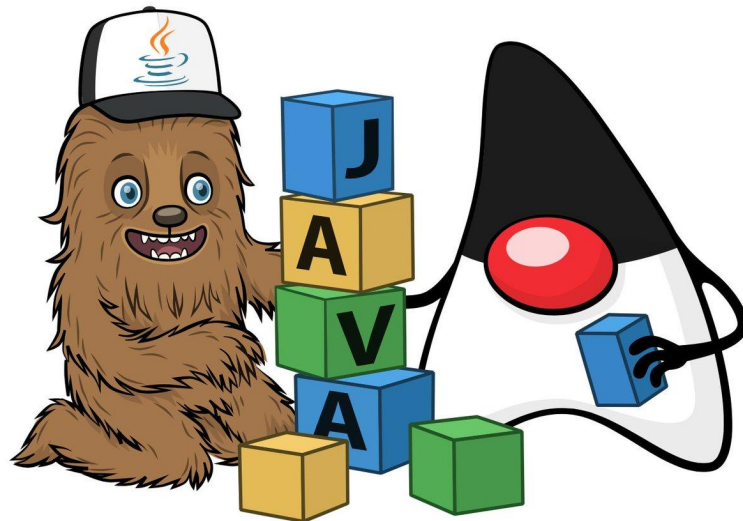
```
public class Persona implements Serializable {  
    private static final long serialVersionUID = 8799656478674716638L;  
  
    ...  
}
```

Serial Version UID (2)

- El número del final debe ser distinto para cada versión de compilado que tengamos.
- De esta forma, la JVM es capaz de detectar rápidamente que las versiones en ambos lados son diferentes.

Agenda

- ~~— Serialización~~
- ~~— Interfaz Serializable~~
- ~~— Serial Version UID~~
- **Modificador transient**
- Serialización y archivos
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Modificador transient

- Habrá ocasiones donde no será necesario incluir un atributo del objeto en la serialización, y para esto se encuentra el modificador transient.
- Este modificador le indica a la JVM que dicho atributo deberá ser exento de la serialización, en otras palabras, ignorará este atributo.
- Por otro lado, los atributos que lleven el modificador static nunca se tomarán en cuenta al serializar un objeto, ya que este atributo pertenece a la clase y no al objeto.

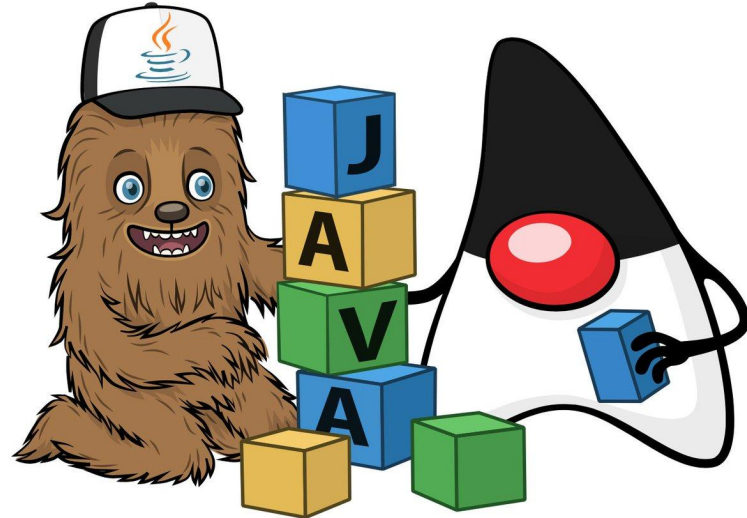
Modificador transient (2)

- Ejemplo:

```
public class Persona implements Serializable {  
    private String nombre;  
    private String apellidoCasada;  
    private transient String apellidoSoltera;  
  
    ...  
  
}
```

Agenda

- ~~— Serialización~~
- ~~— Interfaz Serializable~~
- ~~— SerialVersionUID~~
- ~~— Modificador transient~~
- **Serialización y archivos**
 - ObjectOutputStream
 - ObjectInputStream
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



Serialización y archivos

- Un uso de la serialización es la lectura y escritura de objetos en un archivo.
- Ejemplo:

```
public class Persona implements Serializable {  
    private String nombre;  
    private Direccion direccion;  
}
```

```
public class Direccion implements Serializable {  
    private String calle;  
    private String ciudad;  
    ...  
}
```

Serialización y archivos (2)

- Escribir en el archivo:

```
File file = new File("mi_archivo.txt");  
ObjectOutputStream objOutputStream = new ObjectOutputStream(new  
FileOutputStream(file));
```

```
Persona p = new Persona();  
objOutputStream.writeObject(p);
```

```
objOutputStream.close();
```

Serialización y archivos (3)

- Leer archivo:

```
File file = new File("mi_archivo.txt");
ObjectInputStream objInputStream = new ObjectInputStream(new
FileInputStream(file));
```

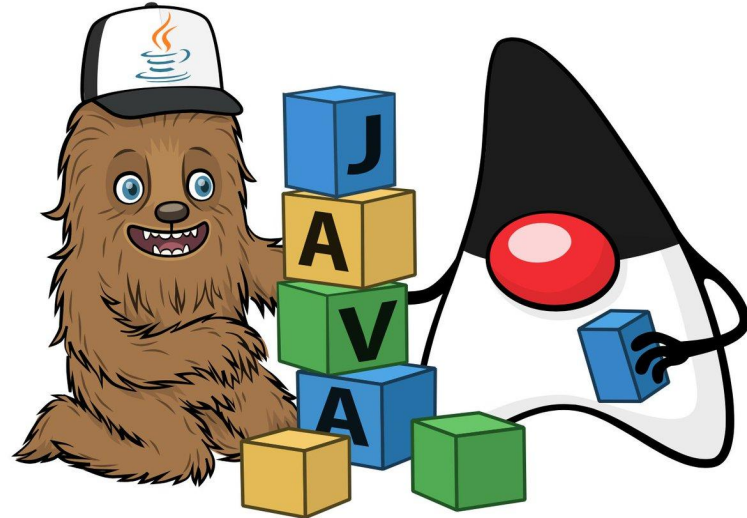
```
Object aux = objInputStream.readObject();
```

```
while (aux!=null) {
    if (aux instanceof Persona)
        System.out.println(aux);
    aux = objInputStream.readObject();
}
```

```
objInputStream.close();
```

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - **ObjectOutputStream**
 - **ObjectInputStream**
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson

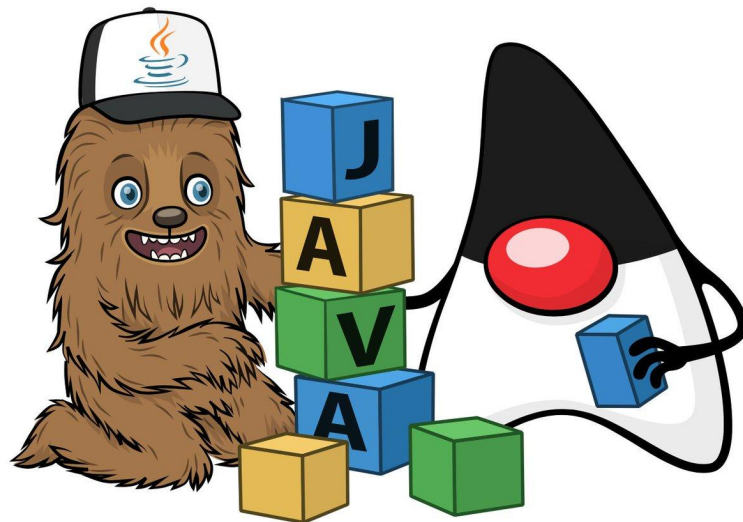


ObjectOutputStream

- Se utiliza para escribir objetos en un OutputStream en lugar de escribir el objeto convertido a bytes. De esta manera se encapsula el OutputStream en un ObjectOutputStream.
- Sólo los objetos que implementen la interfaz Serializable pueden escribirse en los streams.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ~~ObjectOutputStream~~
 - **ObjectInputStream**
- JSON
- Librerías para procesamiento de JSON
 - Jackson
 - Gson

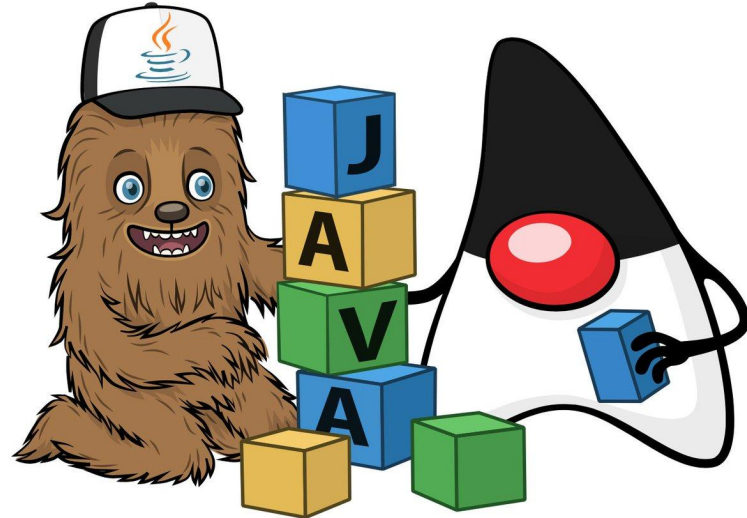


ObjectInputStream

- Se utiliza en conjunto con ObjectOutputStream para leer los objetos que fueron escritos.
- Se debe notar que al leer los objetos tal vez sea necesario realizar un casting sobre los mismos.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ~~ObjectOutputStream~~
 - ~~ObjectInputStream~~
- **JSON**
- Librerías para procesamiento de JSON
 - Jackson
 - Gson



JSON

- JSON (JavaScript Object Notation) es un formato para intercambiar datos liviano, basado en texto e independiente del lenguaje de programación fácil de leer tanto para seres humanos como para las máquinas.
- Puede representar dos tipos estructurados: objetos y matrices.
 - Un objeto es una colección no ordenada de cero o más pares de nombres/valores.
 - Una matriz es una secuencia ordenada de cero o más valores.
- Los valores pueden ser cadenas, números, booleanos, nulos y estos dos tipos estructurados.

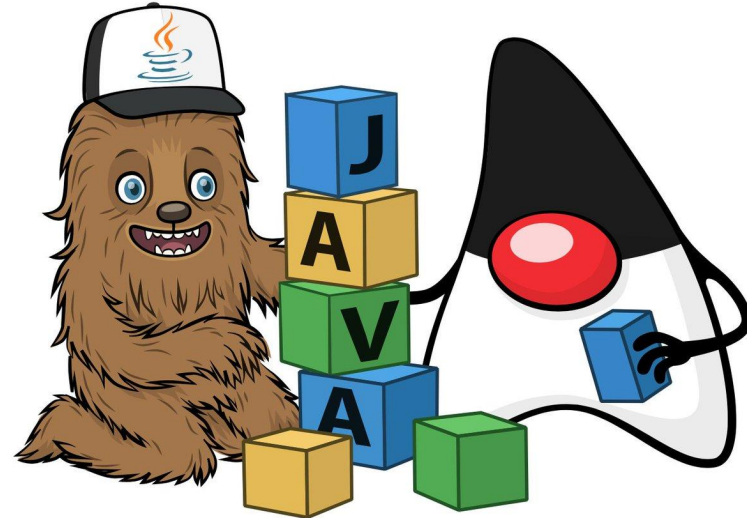
JSON (2)

```
{
  "nombre": "Homero",
  "apellido": "Simpson",
  "edad": 35,
  "direccion": {
    "calle": "Av. Colón 1145",
    "ciudad": "Mar del Plata",
    "codigoPostal": 7600
  },
  "telefonos": [
    {
      "tipo": "casa",
      "numero": "212 555-1234"
    },
    {
      "tipo": "celular",
      "numero": "646 555-4567"
    }
  ]
}
```

- Ejemplo: representación JSON de un objeto Persona, que tiene valores de cadena para nombre y apellido, un valor numérico para la edad, un valor de objeto que representa el domicilio de la persona y un valor de matriz de objetos de números telefónicos.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ~~ObjectOutputStream~~
 - ~~ObjectInputStream~~
- ~~JSON~~
- **Librerías para procesamiento de JSON**
 - Jackson
 - Gson



Librerías para procesamiento de JSON

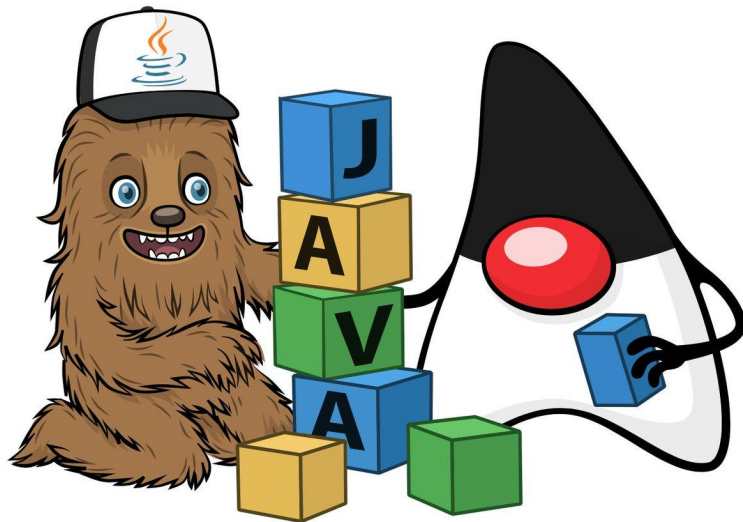
- En la actualidad existen varias librerías para transformar un objeto Java en una cadena JSON. Entre ellas:
 - 1) **Jackson:** es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y deserializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos JSON. Para ello usa básicamente la introspección de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la deserialización buscará un método “setName(String s)”.

Librerías para procesamiento de JSON (2)

2) **Gson**: el uso de esta librería se basa en el uso de una instancia de la clase Gson. Dicha instancia se puede crear de manera directa (`new Gson()`) para transformaciones sencillas o de forma más compleja con `GsonBuilder` para añadir distintos comportamientos.

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ~~ObjectOutputStream~~
 - ~~ObjectInputStream~~
- ~~JSON~~
- ~~Librerías para procesamiento de JSON~~
 - **Jackson**
 - Gson



Jackson

- Escribir en archivo:

```
File file = new File("mi_archivo.json");
```

```
ObjectMapper mapper = new ObjectMapper();
```

```
Persona persona = new Persona();
```

```
//Object to JSON in file
```

```
mapper.writeValue(file, persona);
```

Jackson (2)

- Leer desde archivo:

```
File file = new File("mi_archivo.json");
```

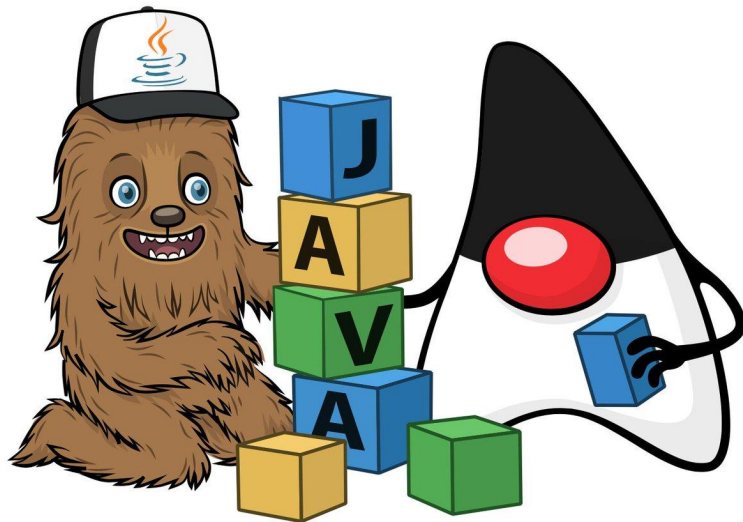
```
ObjectMapper mapper = new ObjectMapper();
```

```
Persona p = mapper.readValue(file, Persona.class);
```

```
System.out.println(p);
```

Agenda

- ~~Serialización~~
- ~~Interfaz Serializable~~
- ~~Serial Version UID~~
- ~~Modificador transient~~
- ~~Serialización y archivos~~
 - ~~ObjectOutputStream~~
 - ~~ObjectInputStream~~
- ~~JSON~~
- ~~Librerías para procesamiento de JSON~~
 - ~~Jackson~~
 - **Gson**



Gson

- Escribir en archivo:

```
File file = new File("mi_archivo.json");
```

```
BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file));
```

```
Persona persona = new Persona("Juan", "Gson");
```

```
Gson gson = new Gson();
```

```
gson.toJson(persona, Persona.class, bufferedWriter);
```

Gson (2)

- Leer desde archivo:

```
File file = new File("mi_archivo.json");
```

```
BufferedReader bufferedReader = new BufferedReader(new FileReader(file));
```

```
Gson gson = new Gson();
```

```
Persona persona = gson.fromJson(bufferedReader, Persona.class);
```

```
System.out.println(persona);
```

Bibliografía oficial

- https://www.tutorialspoint.com/java/java_serialization.htm
- <https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>
- <https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html>
- <https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html>
- <http://jackson.codehaus.org/>
- <https://sites.google.com/site/gson/gson-user-guide>