

UNIVERSIDAD NACIONAL DE INGENIERÍA

Programación Orientada a Objetos

Genéricos

Clases y métodos genéricos, wildcards y type erasure

Gerson Garrido

Semana 10

Esta semana

Sesión A: clases y métodos genéricos

1. El problema de guardar `Object`
2. Clases genéricas: `Box<T>`, `Pair<A, B>`
3. Métodos genéricos e inferencia
4. Primitivos, boxing y raw types

Sesión B: wildcards y type erasure

5. Invarianza: `List<Integer>` no es `List<Number>`
6. Bounded types y wildcards: PECS
7. Type erasure y sus restricciones
8. Laboratorio: `BoundedStack<E>`

> **Lectura**
>
> Eck (v. 9): 10.1.1–10.1.3, 10.1.7 y 10.5.1–10.5.4. Complemento: serie *Generics* de dev.java.

SECCIÓN 01

Genéricos: clases y métodos

01



De dónde venimos

- Semana 4: `ArrayList<Song>` solo acepta canciones, y `get` devuelve una `Song` sin casting. ¿Qué significa ese `<Song>`?

De dónde venimos

- Semana 4: `ArrayList<Song>` solo acepta canciones, y `get` devuelve una `Song` sin casting. ¿Qué significa ese `<Song>`?
- Semana 6: un casting equivocado es una `ClassCastException`, en ejecución.

De dónde venimos

- Semana 4: `ArrayList<Song>` solo acepta canciones, y `get` devuelve una `Song` sin casting. ¿Qué significa ese `<Song>`?
- Semana 6: un casting equivocado es una `ClassCastException`, en ejecución.
- Semana 9: un error que detecta el compilador es mucho más barato que una excepción frente al usuario.

De dónde venimos

- Semana 4: `ArrayList<Song>` solo acepta canciones, y `get` devuelve una `Song` sin casting. ¿Qué significa ese `<Song>`?
- Semana 6: un casting equivocado es una `ClassCastException`, en ejecución.
- Semana 9: un error que detecta el compilador es mucho más barato que una excepción frente al usuario.

Genéricos ***(generics)***

Escribir una clase o un método una sola vez para muchos tipos, sin perder la verificación del compilador.

Una caja para cualquier cosa

ObjectBox.java

```java
public class ObjectBox {
    private Object content;

    public void put(Object content) {
        this.content = content;
    }

    public Object get() {
        return content;
    }
}
```

- Antes de Java 5 (2004), la única forma de escribir una caja una vez para todos los tipos.

Una caja para cualquier cosa

ObjectBox.java

```java
public class ObjectBox {
    private Object content;

    public void put(Object content) {
        this.content = content;
    }

    public Object get() {
        return content;
    }
}
```

- Antes de Java 5 (2004), la única forma de escribir una caja una vez para todos los tipos.
- `Object` es el supertipo de todas las clases: entra cualquier cosa.

Una caja para cualquier cosa

ObjectBox.java

```java
public class ObjectBox {
    private Object content;

    public void put(Object content) {
        this.content = content;
    }

    public Object get() {
        return content;
    }
}
```

- Antes de Java 5 (2004), la única forma de escribir una caja una vez para todos los tipos.
- `Object` es el supertipo de todas las clases: entra cualquier cosa.
- …y ese es justamente el problema.

El error aparece tarde

RawProblem.java

```java
ObjectBox box = new ObjectBox();
box.put("Lima");                    // compila: un String es un Object
String city = (String) box.get();   // hay que hacer casting al sacar
box.put(42);                        // también compila: nadie lo impide
String other = (String) box.get();  // el error aparece aquí, en ejecución
```

bash · salida recortada

```
$ java -cp build RawProblem
ciudad: Lima
ClassCastException: class java...
```

El error aparece tarde

RawProblem.java

```
1  ObjectBox box = new ObjectBox();
2  box.put("Lima");                    // compila: un String es un Object
3  String city = (String) box.get();   // hay que hacer casting al sacar
4  box.put(42);                        // también compila: nadie lo impide
5  String other = (String) box.get();  // el error aparece aquí, en ejecución
```

bash · salida recortada

```
$ java -cp build RawProblem
ciudad: Lima
ClassCastException: class java...
```

- Cada `get` necesita casting: el compilador solo sabe que sale un `Object`.

El error aparece tarde

RawProblem.java

```java
ObjectBox box = new ObjectBox();
box.put("Lima");                    // compila: un String es un Object
String city = (String) box.get();   // hay que hacer casting al sacar
box.put(42);                        // también compila: nadie lo impide
String other = (String) box.get();  // el error aparece aquí, en ejecución
```

bash · salida recortada

```
$ java -cp build RawProblem
ciudad: Lima
ClassCastException: class java...
```

- Cada `get` necesita casting: el compilador solo sabe que sale un `Object`.
- El error está en `put(42)`, pero aparece después, en el `get`, y al ejecutar.

Una clase genérica: Box<T>

Box.java

```java
public class Box<T> {
    private T content;   // null si está vacía

    public void put(T content) {
        this.content = content;
    }
    public T get() {
        return content;
    }
    public boolean isEmpty() {
        return content == null;
    }
}
```

- <T>: un type parameter, un tipo que todavía no se conoce.

Una clase genérica: Box<T>

Box.java

```java
public class Box<T> {
    private T content;   // null si está vacía

    public void put(T content) {
        this.content = content;
    }
    public T get() {
        return content;
    }
    public boolean isEmpty() {
        return content == null;
    }
}
```

- `<T>`: un type parameter, un tipo que todavía no se conoce.
- Dentro de la clase, `T` se usa como cualquier tipo.

Una clase genérica: Box<T>

Box.java

```java
public class Box<T> {
    private T content;   // null si está vacía

    public void put(T content) {
        this.content = content;
    }
    public T get() {
        return content;
    }
    public boolean isEmpty() {
        return content == null;
    }
}
```

- `<T>`: un type parameter, un tipo que todavía no se conoce.
- Dentro de la clase, `T` se usa como cualquier tipo.
- `T` no es una clase: se reemplaza por un tipo concreto en cada uso.

Una clase genérica: Box<T>

Box.java

```java
public class Box<T> {
    private T content;   // null si está vacía

    public void put(T content) {
        this.content = content;
    }
    public T get() {
        return content;
    }
    public boolean isEmpty() {
        return content == null;
    }
}
```

- `<T>`: un type parameter, un tipo que todavía no se conoce.
- Dentro de la clase, `T` se usa como cualquier tipo.
- `T` no es una clase: se reemplaza por un tipo concreto en cada uso.
- `Box<String>`: `put` recibe `String` y `get` devuelve `String`. Lo que va entre `<>` es el **type argument**.

Usar `Box<String>`

BoxDemo.java

```java
Box<String> city = new Box<>();      // diamante: T = String se infiere
city.put("Lima");
String s = city.get();               // sin casting
```

Usar `Box<String>`

BoxDemo.java

```java
Box<String> city = new Box<>();     // diamante: T = String se infiere
city.put("Lima");
String s = city.get();              // sin casting
```

Y la línea que antes compilaba (`errores/WrongType.java`):

bash

```
$ javac -Xlint:all -cp build errores/WrongType.java
errores/WrongType.java:5: error: incompatible types: int cannot be converted to
     String
        city.put(42);
```

Usar `Box<String>`

BoxDemo.java

```java
Box<String> city = new Box<>();     // diamante: T = String se infiere
city.put("Lima");
String s = city.get();              // sin casting
```

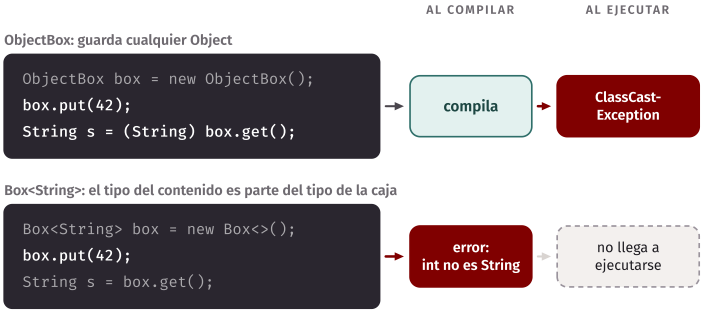
Y la línea que antes compilaba (`errores/WrongType.java`):

bash

```
$ javac -Xlint:all -cp build errores/WrongType.java
errores/WrongType.java:5: error: incompatible types: int cannot be converted to
     String
        city.put(42);
```

El error se detecta en la línea donde se comete y antes de ejecutar.

El error se mueve a la compilación



con genéricos, el mismo error se descubre antes: al compilar, no frente al usuario

Varios parámetros: `Pair<A, B>`

Pair.java

```java
public record Pair<A, B>(A first, B second) {
    public Pair<B, A> swap() {
        return new Pair<>(second, first);
    }
}
```

BoxDemo.java

```java
Pair<String, Integer> stock =
        new Pair<>("arroz", 40);
System.out.println(stock + " -> " + stock.swap
    ());
```

```
Pair[first=arroz, second=40] -> Pair[first=40, second=arroz]
```

- Varios type parameters, separados por comas.

Varios parámetros: `Pair<A, B>`

Pair.java

```java
public record Pair<A, B>(A first, B second) {
    public Pair<B, A> swap() {
        return new Pair<>(second, first);
    }
}
```

BoxDemo.java

```java
Pair<String, Integer> stock =
        new Pair<>("arroz", 40);
System.out.println(stock + " -> " + stock.swap
    ());
```

```
Pair[first=arroz, second=40] -> Pair[first=40, second=arroz]
```

- Varios type parameters, separados por comas.
- Los `record` también pueden ser genéricos.

Varios parámetros: `Pair<A, B>`

Pair.java

```java
public record Pair<A, B>(A first, B second) {
    public Pair<B, A> swap() {
        return new Pair<>(second, first);
    }
}
```

BoxDemo.java

```java
Pair<String, Integer> stock =
        new Pair<>("arroz", 40);
System.out.println(stock + " -> " + stock.swap
    ());
```

```
Pair[first=arroz, second=40] -> Pair[first=40, second=arroz]
```

- Varios type parameters, separados por comas.
- Los `record` también pueden ser genéricos.
- `swap` devuelve `Pair<B, A>`: el compilador sabe que `stock.swap().first()` es un `Integer`.

`ArrayList<E>` por dentro

- La API declara `ArrayList<E>`: `add(E e)`, `E get(int i)`, `E remove(int i)`.

`ArrayList<E>` por dentro

- La API declara `ArrayList<E>`: `add(E e)`, `E get(int i)`, `E remove(int i)`.
- En `ArrayList<Song>`, cada `E` pasa a ser `Song`.

`ArrayList<E>` por dentro

- La API declara `ArrayList<E>`: `add(E e)`, `E get(int i)`, `E remove(int i)`.
- En `ArrayList<Song>`, cada `E` pasa a ser `Song`.
- Nada mágico: es una clase genérica como `Box`.

`ArrayList<E>` por dentro

- La API declara `ArrayList<E>`: `add(E e)`, `E get(int i)`, `E remove(int i)`.
- En `ArrayList<Song>`, cada `E` pasa a ser `Song`.
- Nada mágico: es una clase genérica como `Box`.

Convención de nombres

Una letra mayúscula, para distinguirlos de las clases:

`T` *(type)* · `E` *(element)*
`K`, `V` *(key, value)*
`A`, `B` o `S`, `U` si hay varios

Métodos genéricos

ListUtils.java

```java
/** El primer elemento de la lista, o fallback si está vacía. */
public static <T> T firstOrDefault(List<T> list, T fallback) {
    return list.isEmpty() ? fallback : list.get(0);
}
```

Métodos genéricos

ListUtils.java

```java
/** El primer elemento de la lista, o fallback si está vacía. */
public static <T> T firstOrDefault(List<T> list, T fallback) {
    return list.isEmpty() ? fallback : list.get(0);
}
```

- El método declara su propio `<T>`, antes del tipo de retorno.

Métodos genéricos

ListUtils.java

```java
/** El primer elemento de la lista, o fallback si está vacía. */
public static <T> T firstOrDefault(List<T> list, T fallback) {
    return list.isEmpty() ? fallback : list.get(0);
}
```

- El método declara su propio `<T>`, antes del tipo de retorno.
- `List`: la interfaz que implementa `ArrayList` (las interfaces de colecciones, en la semana 11).

Métodos genéricos

ListUtils.java

```java
/** El primer elemento de la lista, o fallback si está vacía. */
public static <T> T firstOrDefault(List<T> list, T fallback) {
    return list.isEmpty() ? fallback : list.get(0);
}
```

- El método declara su propio `<T>`, antes del tipo de retorno.
- `List`: la interfaz que implementa `ArrayList` (las interfaces de colecciones, en la semana 11).
- La firma dice: lo que devuelve es del mismo tipo que los elementos y que el valor por defecto.

Inferencia de tipos

BoxDemo.java · salida: alice, nadie

```
1  List<String> empty = new ArrayList<>();
2  List<String> names = List.of("alice", "bob");
3  System.out.println(ListUtils.firstOrDefault(names, "nadie"));
4  System.out.println(ListUtils.firstOrDefault(empty, "nadie"));
```

- Nadie escribe cuánto vale `T`: el compilador lo infiere de los argumentos.

Inferencia de tipos

BoxDemo.java · salida: alice, nadie

```java
List<String> empty = new ArrayList<>();
List<String> names = List.of("alice", "bob");
System.out.println(ListUtils.firstOrDefault(names, "nadie"));
System.out.println(ListUtils.firstOrDefault(empty, "nadie"));
```

- Nadie escribe cuánto vale `T`: el compilador lo infiere de los argumentos.
- El diamante `<>` usa la misma inferencia: completa `new ArrayList<String>()`.

Inferencia de tipos

BoxDemo.java · salida: alice, nadie

```
1  List<String> empty = new ArrayList<>();
2  List<String> names = List.of("alice", "bob");
3  System.out.println(ListUtils.firstOrDefault(names, "nadie"));
4  System.out.println(ListUtils.firstOrDefault(empty, "nadie"));
```

- Nadie escribe cuánto vale `T`: el compilador lo infiere de los argumentos.
- El diamante `<>` usa la misma inferencia: completa `new ArrayList<String>()`.

Error común: el `<T>` en otro lugar

Sin `<T>`, `public static T firstOrDefault(...)` busca una *clase* `T`: `cannot find symbol`.

Genéricos y primitivos

BoxDemo.java

```java
Box<Integer> count = new Box<>();
count.put(7);            // boxing
int n = count.get() + 1;  // unboxing
```

```
n = 8
```

- `Box<int>` no existe: el type argument es una referencia.

Genéricos y primitivos

BoxDemo.java

```java
Box<Integer> count = new Box<>();
count.put(7);            // boxing
int n = count.get() + 1;  // unboxing
```

```
n = 8
```

- `Box<int>` no existe: el type argument es una referencia.
- Se usa la clase wrapper: `Integer`, `Long`, `Double`...

Genéricos y primitivos

BoxDemo.java

```java
Box<Integer> count = new Box<>();
count.put(7);            // boxing
int n = count.get() + 1;  // unboxing
```

```
n = 8
```

- `Box<int>` no existe: el type argument es una referencia.
- Se usa la clase wrapper: `Integer`, `Long`, `Double`...
- **Boxing**: `int` → `Integer`. **Unboxing**: al revés. Automáticos.

Genéricos y primitivos

BoxDemo.java

```
1  Box<Integer> count = new Box<>();
2  count.put(7);            // boxing
3  int n = count.get() + 1;  // unboxing
```

```
n = 8
```

- `Box<int>` no existe: el type argument es una referencia.
- Se usa la clase wrapper: `Integer`, `Long`, `Double`...
- **Boxing**: `int` → `Integer`. **Unboxing**: al revés. Automáticos.
- No es gratis: cada `Integer` es un objeto en el heap. Para millones de números, un `int[]`.

Error común: == entre `Integer`

BoxDemo.java

```java
Integer a = 127, b = 127, c = 1000, d = 1000;
System.out.println("127 == 127: " + (a == b) + ", 1000 == 1000: " + (c == d)
        + ", equals: " + c.equals(d));
```

bash

```
127 == 127: true, 1000 == 1000: false, equals: true
```

Error común: == entre `Integer`

BoxDemo.java

```java
Integer a = 127, b = 127, c = 1000, d = 1000;
System.out.println("127 == 127: " + (a == b) + ", 1000 == 1000: " + (c == d)
        + ", equals: " + c.equals(d));
```

bash

```
127 == 127: true, 1000 == 1000: false, equals: true
```

- Son objetos: `==` compara referencias (semana 2).

Error común: == entre `Integer`

BoxDemo.java

```java
Integer a = 127, b = 127, c = 1000, d = 1000;
System.out.println("127 == 127: " + (a == b) + ", 1000 == 1000: " + (c == d)
        + ", equals: " + c.equals(d));
```

bash

```
127 == 127: true, 1000 == 1000: false, equals: true
```

- Son objetos: `==` compara referencias (semana 2).
- Java guarda en caché al menos los `Integer` de -128 a 127 (la JVM puede ampliar el rango): con números chicos *parece* funcionar.

Error común: == entre `Integer`

BoxDemo.java

```java
Integer a = 127, b = 127, c = 1000, d = 1000;
System.out.println("127 == 127: " + (a == b) + ", 1000 == 1000: " + (c == d)
        + ", equals: " + c.equals(d));
```

bash

```
127 == 127: true, 1000 == 1000: false, equals: true
```

- Son objetos: `==` compara referencias (semana 2).
- Java guarda en caché al menos los `Integer` de -128 a 127 (la JVM puede ampliar el rango): con números chicos *parece* funcionar.
- Con wrappers, como con cualquier objeto: `equals`.

Raw types: el error que todavía compila

errores/RawList.java

```java
ArrayList names = new ArrayList
    ();
names.add("alice");
names.add(42);
String first =
    (String) names.get(1);
```

`ArrayList` sin type argument: un raw type. Existe por compatibilidad con el código anterior a Java 5.

bash · sin opciones

```
$ javac -d build errores/RawList.java
Note: errores/RawList.java uses
  unchecked or unsafe operations.
$ java -cp build RawList
Exception in thread "main"
  java.lang.ClassCastException: ...
```

Raw types: el error que todavía compila

errores/RawList.java

```java
ArrayList names = new ArrayList
    ();
names.add("alice");
names.add(42);
String first =
    (String) names.get(1);
```

`ArrayList` sin type argument: un raw type. Existe por compatibilidad con el código anterior a Java 5.

bash · sin opciones

```
$ javac -d build errores/RawList.java
Note: errores/RawList.java uses
  unchecked or unsafe operations.
$ java -cp build RawList
Exception in thread "main"
  java.lang.ClassCastException: ...
```

Solo una nota, fácil de pasar por alto. Y el problema de `ObjectBox`, de vuelta.

Con `-Xlint:all`, el compilador lo dice

bash · salida recortada

```
$ javac -Xlint:all -Werror -d build errores/RawList.java
errores/RawList.java:6: warning: [rawtypes] found raw type: ArrayList
  missing type arguments for generic class ArrayList<E>
errores/RawList.java:7: warning: [unchecked] unchecked call to add(E) as a
    member
  of the raw type ArrayList
error: warnings found and -Werror specified
```

Con `-Xlint:all`, el compilador lo dice

bash · salida recortada

```
$ javac -Xlint:all -Werror -d build errores/RawList.java
errores/RawList.java:6: warning: [rawtypes] found raw type: ArrayList
  missing type arguments for generic class ArrayList<E>
errores/RawList.java:7: warning: [unchecked] unchecked call to add(E) as a
    member
  of the raw type ArrayList
error: warnings found and -Werror specified
```

Error común: raw types “porque compila”

`unchecked` significa que el compilador *ya no garantiza* los tipos. Siempre con type argument: `ArrayList<String>`, o `new ArrayList<>()`.

SECCIÓN 02

Wildcards y type erasure

02



¿Una `List<Integer>` es una `List<Number>`?

errores/Invariance.java

```java
static double total(List<Number> numbers) { ... }
List<Integer> ints = new ArrayList<>(List.of(1, 2, 3));
List<Number> nums = ints;
System.out.println(total(ints));
```

¿Una `List<Integer>` es una `List<Number>`?

errores/Invariance.java

```java
static double total(List<Number> numbers) { ... }
List<Integer> ints = new ArrayList<>(List.of(1, 2, 3));
List<Number> nums = ints;
System.out.println(total(ints));
```

bash · salida recortada

```
Invariance.java:16: error: incompatible types: List<Integer> cannot be
    converted to List<Number>
Invariance.java:17: error: incompatible types: List<Integer> cannot be
    converted to List<Number>
```

`Integer` es subtipo de `Number`… pero la lista no.

¿Por qué? Imaginemos que sí se pudiera

1. `List<Number> nums = ints;` → `nums` e `ints` son la misma lista.

¿Por qué? Imaginemos que sí se pudiera

1. `List<Number> nums = ints;` → `nums` e `ints` son la misma lista.
2. `nums.add(2.5);` → válido: un `Double` es un `Number`.

¿Por qué? Imaginemos que sí se pudiera

1. `List<Number> nums = ints;` → `nums` e `ints` son la misma lista.
2. `nums.add(2.5);` → válido: un `Double` es un `Number`.
3. `int x = ints.get(3);` → ¡un `Double` en una lista de enteros!

¿Por qué? Imaginemos que sí se pudiera

1. `List<Number> nums = ints;` → `nums` e `ints` son la misma lista.
2. `nums.add(2.5);` → válido: un `Double` es un `Number`.
3. `int x = ints.get(3);` → ¡un `Double` en una lista de enteros!

Los genéricos son invariantes

`List<A>` y `List<B>` no tienen ninguna relación de subtipo, aunque `A` y `B` sí la tengan. Así el paso 3 no puede pasar.

Los arreglos lo hicieron al revés

ErasureDemo.java

```java
Integer[] ints = {1, 2, 3};
Number[] nums = ints;    // compila
try {
    nums[0] = 2.5;       // un Double
} catch (ArrayStoreException e) {
    System.out.println(
        "ArrayStoreException: "
        + e.getMessage());
}
```

bash

```
ArrayStoreException: java.lang.Double
```

- `Integer[]` **sí** es subtipo de `Number[]`: los arreglos son **covariantes**.

Los arreglos lo hicieron al revés

ErasureDemo.java

```java
Integer[] ints = {1, 2, 3};
Number[] nums = ints;    // compila
try {
    nums[0] = 2.5;       // un Double
} catch (ArrayStoreException e) {
    System.out.println(
        "ArrayStoreException: "
        + e.getMessage());
}
```

bash

```
ArrayStoreException: java.lang.Double
```

- `Integer[]` sí es subtipo de `Number[]`: los arreglos son **covariantes**.
- El error compila… y se detecta al ejecutar.

Los arreglos lo hicieron al revés

ErasureDemo.java

```java
Integer[] ints = {1, 2, 3};
Number[] nums = ints;    // compila
try {
    nums[0] = 2.5;       // un Double
} catch (ArrayStoreException e) {
    System.out.println(
        "ArrayStoreException: "
        + e.getMessage());
}
```

bash

```
ArrayStoreException: java.lang.Double
```

- `Integer[]` sí es subtipo de `Number[]`: los arreglos son **covariantes**.
- El error compila… y se detecta al ejecutar.
- Cada arreglo recuerda el tipo de sus elementos, y la JVM revisa cada asignación.

Los arreglos lo hicieron al revés

ErasureDemo.java

```java
Integer[] ints = {1, 2, 3};
Number[] nums = ints;    // compila
try {
    nums[0] = 2.5;       // un Double
} catch (ArrayStoreException e) {
    System.out.println(
        "ArrayStoreException: "
        + e.getMessage());
}
```

bash

```
ArrayStoreException: java.lang.Double
```

- `Integer[]` sí es subtipo de `Number[]`: los arreglos son **covariantes**.
- El error compila… y se detecta al ejecutar.
- Cada arreglo recuerda el tipo de sus elementos, y la JVM revisa cada asignación.
- Con listas, el compilador no deja llegar hasta ahí.

Bounded types: `<T extends Comparable<T>>`

ListUtils.java

```java
public static <T extends Comparable<T>>
        T max(List<T> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException(
            "lista vacía");
    }
    T best = list.get(0);
    for (T x : list) {
        if (x.compareTo(best) > 0) {
            best = x;
        }
    }
    return best;
}
```

- No sirve para *cualquier* `T`: hay que poder comparar.

Bounded types: `<T extends Comparable<T>>`

ListUtils.java

```java
public static <T extends Comparable<T>>
        T max(List<T> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException(
            "lista vacía");
    }
    T best = list.get(0);
    for (T x : list) {
        if (x.compareTo(best) > 0) {
            best = x;
        }
    }
    return best;
}
```

- No sirve para *cualquier* `T`: hay que poder comparar.
- El bound: `T` implementa `Comparable<T>` (semana 7). `extends` vale también para interfaces.

Bounded types: `<T extends Comparable<T>>`

ListUtils.java

```java
public static <T extends Comparable<T>>
        T max(List<T> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException(
            "lista vacía");
    }
    T best = list.get(0);
    for (T x : list) {
        if (x.compareTo(best) > 0) {
            best = x;
        }
    }
    return best;
}
```

- No sirve para *cualquier* `T`: hay que poder comparar.
- El bound: `T` implementa `Comparable<T>` (semana 7). `extends` vale también para interfaces.
- Adentro: `x.compareTo(best)` compila.

Bounded types: `<T extends Comparable<T>>`

ListUtils.java

```java
public static <T extends Comparable<T>>
        T max(List<T> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException(
            "lista vacía");
    }
    T best = list.get(0);
    for (T x : list) {
        if (x.compareTo(best) > 0) {
            best = x;
        }
    }
    return best;
}
```

- No sirve para *cualquier* `T`: hay que poder comparar.
- El bound: `T` implementa `Comparable<T>` (semana 7). `extends` vale también para interfaces.
- Adentro: `x.compareTo(best)` compila.
- Afuera: acepta `List<Integer>` y `List<String>`; no `List<Box<String>`.

`max` en acción

WildcardDemo.java · salida: max: 18, uva

```java
List<Integer> marks = List.of(15, 18, 11);
List<String> words = List.of("pera", "uva", "mango");
System.out.println("max: " + ListUtils.max(marks) + ", " + ListUtils.max(words));
```

`max` en acción

WildcardDemo.java · salida: max: 18, uva

```
1  List<Integer> marks = List.of(15, 18, 11);
2  List<String> words = List.of("pera", "uva", "mango");
3  System.out.println("max: " + ListUtils.max(marks) + ", " + ListUtils.max(words));
```

- Las palabras se comparan en orden alfabético.

`max` en acción

WildcardDemo.java · salida: max: 18, uva

```
1  List<Integer> marks = List.of(15, 18, 11);
2  List<String> words = List.of("pera", "uva", "mango");
3  System.out.println("max: " + ListUtils.max(marks) + ", " + ListUtils.max(words));
```

- Las palabras se comparan en orden alfabético.
- Devuelve un `T`: `max(marks)` es un `Integer`, no un `Comparable` al que haya que hacer casting.

`max` en acción

WildcardDemo.java · salida: max: 18, uva

```
1  List<Integer> marks = List.of(15, 18, 11);
2  List<String> words = List.of("pera", "uva", "mango");
3  System.out.println("max: " + ListUtils.max(marks) + ", " + ListUtils.max(words));
```

- Las palabras se comparan en orden alfabético.
- Devuelve un `T`: `max(marks)` es un `Integer`, no un `Comparable` al que haya que hacer casting.

> **Frente a `Comparable max(List list)`**
>
> El compilador revisa que los elementos se puedan comparar y conserva el tipo del resultado.

Wildcard para leer: `? extends`

ListUtils.java

```java
static double sum(
        List<? extends Number> numbers
    ) {
    double total = 0;
    for (Number n : numbers) {  // leer
        total += n.doubleValue();
    }
    return total;
}
```

- "Una lista de `Number` o de algún subtipo": un **wildcard**.

Wildcard para leer: `? extends`

ListUtils.java

```java
static double sum(
        List<? extends Number> numbers
    ) {
    double total = 0;
    for (Number n : numbers) {  // leer
        total += n.doubleValue();
    }
    return total;
}
```

- "Una lista de `Number` o de algún subtipo": un **wildcard**.
- Acepta `List<Integer>`, `List<Double>`... `sum(marks)` da `44.0`.

Wildcard para leer: `? extends`

ListUtils.java

```java
static double sum(
        List<? extends Number> numbers
    ) {
    double total = 0;
    for (Number n : numbers) {  // leer
        total += n.doubleValue();
    }
    return total;
}
```

- "Una lista de `Number` o de algún subtipo": un **wildcard**.
- Acepta `List<Integer>`, `List<Double>`... `sum(marks)` da `44.0`.
- Se puede leer: lo que sale es un `Number`.

Wildcard para leer: `? extends`

ListUtils.java

```java
static double sum(
        List<? extends Number> numbers
    ) {
    double total = 0;
    for (Number n : numbers) {  // leer
        total += n.doubleValue();
    }
    return total;
}
```

- "Una lista de `Number` o de algún subtipo": un **wildcard**.
- Acepta `List<Integer>`, `List<Double>`... `sum(marks)` da `44.0`.
- Se puede leer: lo que sale es un `Number`.
- No se puede escribir: ¿y si llegó una `List<Double>`? `numbers.add(1)` da `int cannot be converted to CAP#1`: "un subtipo desconocido de `Number`".

Wildcard para escribir: `? super`

ListUtils.java

```java
static void addCounting(
        List<? super Integer> out, int n)
    {
    for (int i = 1; i <= n; i++) {
        out.add(i);   // escribir
    }
}
```

WildcardDemo.java

```java
List<Number> numbers = new ArrayList<>();
ListUtils.addCounting(numbers, 3);
numbers.add(0.5);   // [1, 2, 3, 0.5]
```

- "Una lista de `Integer` o de algún supertipo".

Wildcard para escribir: `? super`

ListUtils.java

```java
static void addCounting(
        List<? super Integer> out, int n)
    {
    for (int i = 1; i <= n; i++) {
        out.add(i);   // escribir
    }
}
```

WildcardDemo.java

```java
List<Number> numbers = new ArrayList<>();
ListUtils.addCounting(numbers, 3);
numbers.add(0.5);   // [1, 2, 3, 0.5]
```

- "Una lista de `Integer` o de algún supertipo".
- Acepta `List<Integer>`, `List<Number>`, `List<Object>`.

Wildcard para escribir: `? super`

ListUtils.java

```java
static void addCounting(
        List<? super Integer> out, int n)
    {
    for (int i = 1; i <= n; i++) {
        out.add(i);   // escribir
    }
}
```

WildcardDemo.java

```java
List<Number> numbers = new ArrayList<>();
ListUtils.addCounting(numbers, 3);
numbers.add(0.5);   // [1, 2, 3, 0.5]
```

- "Una lista de `Integer` o de algún supertipo".
- Acepta `List<Integer>`, `List<Number>`, `List<Object>`.
- Se puede escribir un `Integer`: cabe en todas.

Wildcard para escribir: `? super`

ListUtils.java

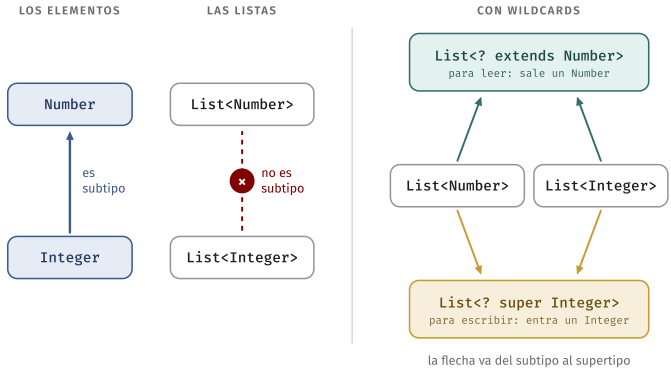
```java
static void addCounting(
        List<? super Integer> out, int n)
    {
    for (int i = 1; i <= n; i++) {
        out.add(i);   // escribir
    }
}
```

WildcardDemo.java

```java
List<Number> numbers = new ArrayList<>();
ListUtils.addCounting(numbers, 3);
numbers.add(0.5);   // [1, 2, 3, 0.5]
```

- "Una lista de `Integer` o de algún supertipo".
- Acepta `List<Integer>`, `List<Number>`, `List<Object>`.
- Se puede escribir un `Integer`: cabe en todas.
- Al leer, solo se sabe que sale un `Object`.

Las listas y los wildcards



Las dos juntas: `copy` y la regla PECS

ListUtils.java

```java
public static <T> void copy(List<? super T> dst, List<? extends T> src) {
    for (T x : src) {
        dst.add(x);
    }
}
```

PECS: ***producer extends, consumer super***

Las dos juntas: `copy` y la regla PECS

ListUtils.java

```java
public static <T> void copy(List<? super T> dst, List<? extends T> src) {
    for (T x : src) {
        dst.add(x);
    }
}
```

PECS: ***producer extends, consumer super***

- Del parámetro se sacan valores (productor): `? extends T`.

Las dos juntas: `copy` y la regla PECS

ListUtils.java

```java
public static <T> void copy(List<? super T> dst, List<? extends T> src) {
    for (T x : src) {
        dst.add(x);
    }
}
```

PECS: ***producer extends, consumer super***

- Del parámetro se sacan valores (productor): `? extends T`.
- En el parámetro se meten valores (consumidor): `? super T`.

Las dos juntas: `copy` y la regla PECS

ListUtils.java

```java
public static <T> void copy(List<? super T> dst, List<? extends T> src) {
    for (T x : src) {
        dst.add(x);
    }
}
```

PECS: *producer extends, consumer super*

- Del parámetro se sacan valores (productor): `? extends T`.
- En el parámetro se meten valores (consumidor): `? super T`.
- Las dos cosas: sin wildcard, `List<T>`.

`copy(all, marks)` paso a paso

```
static <T> void copy(List<? super T> dst, List<? extends T> src)
ListUtils.copy(all, marks);      // T = Integer
```

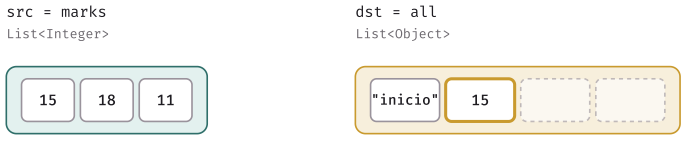



```
for (T x : src) dst.add(x);
```

Sin wildcards, `copy(List<T>, List<T>)` no compilaría: no hay un `T` que sea a la vez `Integer` y `Object`.

`copy(all, marks)` paso a paso

```
static <T> void copy(List<? super T> dst, List<? extends T> src)
ListUtils.copy(all, marks);      // T = Integer
```

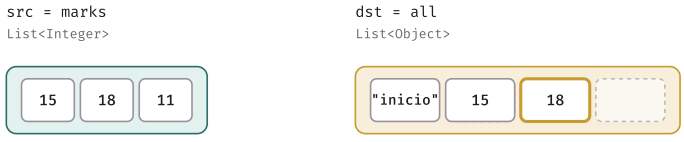


`x = 15 (se lee de src)  →  dst.add(15)`

Sin wildcards, `copy(List<T>, List<T>)` no compilaría: no hay un `T` que sea a la vez `Integer` y `Object`.

`copy(all, marks)` paso a paso

```
static <T> void copy(List<? super T> dst, List<? extends T> src)
ListUtils.copy(all, marks);      // T = Integer
```

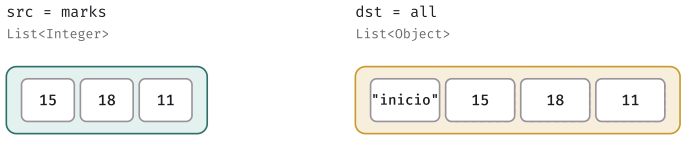


`x = 18 (se lee de src)  →  dst.add(18)`

Sin wildcards, `copy(List<T>, List<T>)` no compilaría: no hay un `T` que sea a la vez `Integer` y `Object`.

`copy(all, marks)` paso a paso

```
static <T> void copy(List<? super T> dst, List<? extends T> src)
ListUtils.copy(all, marks);      // T = Integer
```



`x = 11 (se lee de src)  →  dst.add(11)`

se leyó de src y se escribió en dst: ningún casting

Sin wildcards, `copy(List<T>, List<T>)` no compilaría: no hay un `T` que sea a la vez `Integer` y `Object`.

Error común: wildcards “por si acaso”

- Los wildcards hacen las firmas más difíciles de leer.

Error común: wildcards “por si acaso”

- Los wildcards hacen las firmas más difíciles de leer.
- Empiecen con tipos simples: `List<T>`.

Error común: wildcards “por si acaso”

- Los wildcards hacen las firmas más difíciles de leer.
- Empiecen con tipos simples: `List<T>`.
- Agreguen `? extends` o `? super` cuando una llamada real, o una prueba, muestre que hace falta aceptar listas de subtipos o de supertipos.

Error común: wildcards “por si acaso”

- Los wildcards hacen las firmas más difíciles de leer.
- Empiecen con tipos simples: `List<T>`.
- Agreguen `? extends` o `? super` cuando una llamada real, o una prueba, muestre que hace falta aceptar listas de subtipos o de supertipos.
- Nunca en el tipo de retorno: `List<? extends Number> f()` obliga a quien llama a lidiar con el wildcard.

Type erasure

ErasureDemo.java

```java
Box<String> a = new Box<>();
Box<Integer> b = new Box<>();
System.out.println(a.getClass() + " == " + b.getClass() + ": "
        + (a.getClass() == b.getClass()));
```

bash

```
class Box == class Box: true
```

Type erasure

ErasureDemo.java

```java
Box<String> a = new Box<>();
Box<Integer> b = new Box<>();
System.out.println(a.getClass() + " == " + b.getClass() + ": "
        + (a.getClass() == b.getClass()));
```

bash

```
class Box == class Box: true
```

- `javac` revisa todo con `T` y después lo borra: queda `Object` (o el bound).

Type erasure

ErasureDemo.java

```java
Box<String> a = new Box<>();
Box<Integer> b = new Box<>();
System.out.println(a.getClass() + " == " + b.getClass() + ": "
        + (a.getClass() == b.getClass()));
```

bash

```
class Box == class Box: true
```

- `javac` revisa todo con `T` y después lo borra: queda `Object` (o el bound).
- En ejecución, `Box<String>` y `Box<Integer>` son la misma clase.

Type erasure

ErasureDemo.java

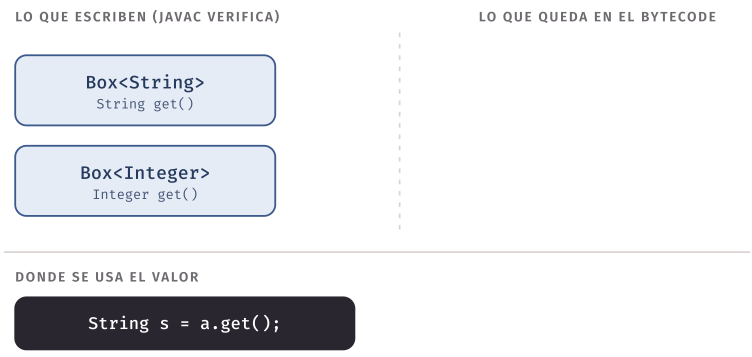
```java
Box<String> a = new Box<>();
Box<Integer> b = new Box<>();
System.out.println(a.getClass() + " == " + b.getClass() + ": "
        + (a.getClass() == b.getClass()));
```

bash

```
class Box == class Box: true
```

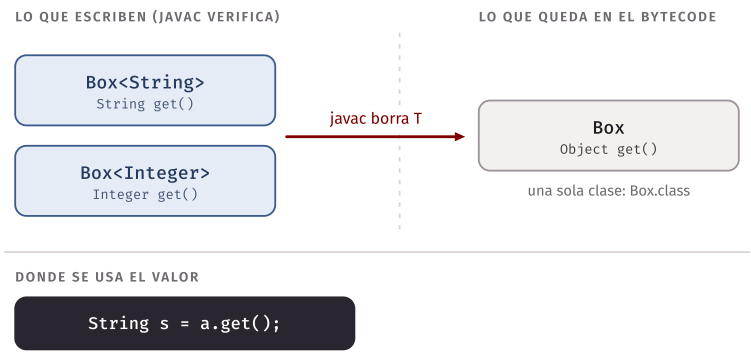
- `javac` revisa todo con `T` y después lo borra: queda `Object` (o el bound).
- En ejecución, `Box<String>` y `Box<Integer>` son la misma clase.
- Un objeto no sabe su type argument. El `.class` solo anota la declaración genérica, como metadato (`javap` muestra `T get()`).

Qué hace `javac` con `T`



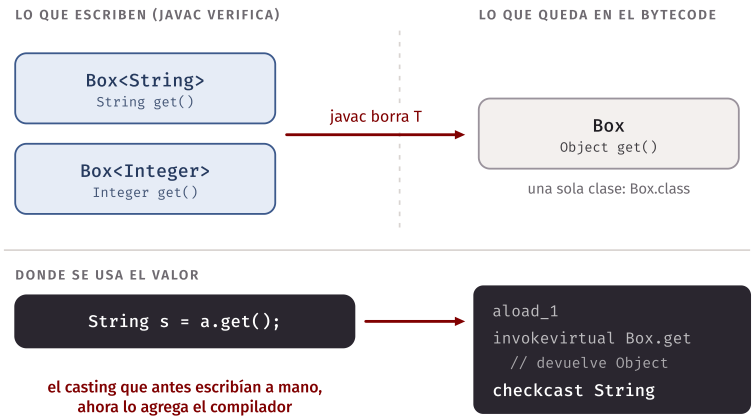
1. con T, javac revisa cada put y cada get

Qué hace `javac` con `T`



2. después borra T: queda Object

Qué hace `javac` con `T`



3. y agrega el casting donde hace falta

El `checkcast` en el bytecode

bash · salida recortada

```
$ javap -c -cp build Box
  public T get();
     1: getfield  // Field content:Object;
     4: areturn
$ javap -c -cp build BoxDemo
    15: invokevirtual // Method Box.get:()
    Object;
    18: checkcast     // class java/lang/
    String
```

- `get` devuelve un `Object`.

El `checkcast` en el bytecode

bash · salida recortada

```
$ javap -c -cp build Box
  public T get();
     1: getfield  // Field content:Object;
     4: areturn
$ javap -c -cp build BoxDemo
    15: invokevirtual // Method Box.get:()
    Object;
    18: checkcast     // class java/lang/
    String
```

- `get` devuelve un `Object`.
- Donde se usa como `String`, el compilador agregó `checkcast`: el casting de `ObjectBox`.

El `checkcast` en el bytecode

bash · salida recortada

```
$ javap -c -cp build Box
  public T get();
     1: getfield  // Field content:Object;
     4: areturn
$ javap -c -cp build BoxDemo
    15: invokevirtual // Method Box.get:()
    Object;
    18: checkcast     // class java/lang/
    String
```

- `get` devuelve un `Object`.
- Donde se usa como `String`, el compilador agregó `checkcast`: el casting de `ObjectBox`.
- En `max`, se borra al bound: devuelve un `Comparable`.

El `checkcast` en el bytecode

bash · salida recortada

```
$ javap -c -cp build Box
  public T get();
     1: getfield  // Field content:Object;
     4: areturn
$ javap -c -cp build BoxDemo
    15: invokevirtual // Method Box.get:()
    Object;
    18: checkcast     // class java/lang/
    String
```

- `get` devuelve un `Object`.
- Donde se usa como `String`, el compilador agregó `checkcast`: el casting de `ObjectBox`.
- En `max`, se borra al bound: devuelve un `Comparable`.
- En código sin warnings de `unchecked`, estos castings no fallan: el compilador ya verificó los tipos.

¿Por qué Java lo hizo así?

- Los genéricos llegaron en 2004, con millones de líneas que usaban `ArrayList` sin parámetros.

¿Por qué Java lo hizo así?

- Los genéricos llegaron en 2004, con millones de líneas que usaban `ArrayList` sin parámetros.
- Con el type erasure, el bytecode de una `ArrayList<String>` es el de siempre: el código viejo y el nuevo se pueden mezclar.

¿Por qué Java lo hizo así?

- Los genéricos llegaron en 2004, con millones de líneas que usaban `ArrayList` sin parámetros.
- Con el type erasure, el bytecode de una `ArrayList<String>` es el de siempre: el código viejo y el nuevo se pueden mezclar.
- Otros lenguajes, como C#, eligieron lo contrario: guardan el type argument en ejecución.

¿Por qué Java lo hizo así?

- Los genéricos llegaron en 2004, con millones de líneas que usaban `ArrayList` sin parámetros.
- Con el type erasure, el bytecode de una `ArrayList<String>` es el de siempre: el código viejo y el nuevo se pueden mezclar.
- Otros lenguajes, como C#, eligieron lo contrario: guardan el type argument en ejecución.
- El precio: todo lo que necesite saber cuánto vale `T` en ejecución está prohibido.

Lo que el type erasure no permite

errores/ErasureLimits.java

```java
class ErasureLimits<T> {
    T make() { return new T(); }
    T[] array(int n) { return new
     T[n]; }
    boolean isStrings(Object o) {
        return o instanceof List<
     String>;
    }
    void print(List<String> list)
     { }
    void print(List<Integer> list)
      { }
}
```

- `new T()`: `unexpected type`. ¿Qué constructor llamar?

Lo que el type erasure no permite

errores/ErasureLimits.java

```java
class ErasureLimits<T> {
    T make() { return new T(); }
    T[] array(int n) { return new
     T[n]; }
    boolean isStrings(Object o) {
        return o instanceof List<
     String>;
    }
    void print(List<String> list)
     { }
    void print(List<Integer> list)
      { }
}
```

- `new T()`: `unexpected type`. ¿Qué constructor llamar?
- `new T[n]`: `generic array creation`. ¿De qué tipo es el arreglo?

Lo que el type erasure no permite

errores/ErasureLimits.java

```java
class ErasureLimits<T> {
    T make() { return new T(); }
    T[] array(int n) { return new
     T[n]; }
    boolean isStrings(Object o) {
        return o instanceof List<
     String>;
    }
    void print(List<String> list)
     { }
    void print(List<Integer> list)
      { }
}
```

- `new T()`: `unexpected type`. ¿Qué constructor llamar?
- `new T[n]`: `generic array creation`. ¿De qué tipo es el arreglo?
- `instanceof List<String>`: en ejecución solo hay `List` (sí vale `List<?>`).

Lo que el type erasure no permite

errores/ErasureLimits.java

```java
class ErasureLimits<T> {
    T make() { return new T(); }
    T[] array(int n) { return new
     T[n]; }
    boolean isStrings(Object o) {
        return o instanceof List<
     String>;
    }
    void print(List<String> list)
     { }
    void print(List<Integer> list)
      { }
}
```

- `new T()`: `unexpected type`. ¿Qué constructor llamar?
- `new T[n]`: `generic array creation`. ¿De qué tipo es el arreglo?
- `instanceof List<String>`: en ejecución solo hay `List` (sí vale `List<?>`).
- `print`: `name clash ... same erasure`. Las dos se borran a `print(List)`.

`new T[n]` en la práctica

- Es la restricción que más se nota al escribir una colección propia.

`new T[n]` en la práctica

- Es la restricción que más se nota al escribir una colección propia.
- Opción 1: el arreglo interno es `Object[]` y se hace casting a `T` al sacar.

`new T[n]` en la práctica

- Es la restricción que más se nota al escribir una colección propia.
- Opción 1: el arreglo interno es `Object[]` y se hace casting a `T` al sacar.
- Opción 2: se crea un `Object[]` y se convierte a `T[]` una sola vez, en el constructor.

`new T[n]` en la práctica

- Es la restricción que más se nota al escribir una colección propia.
- Opción 1: el arreglo interno es `Object[]` y se hace casting a `T` al sacar.
- Opción 2: se crea un `Object[]` y se convierte a `T[]` una sola vez, en el constructor.

Advertencia unchecked

En los dos casos. Se silencia con `@SuppressWarnings("unchecked")` *solo* en ese método, y *solo* después de convencerse de que únicamente entran objetos de tipo `T`.

SECCIÓN 03

Laboratorio

03

Laboratorio: `BoundedStack<E>`

BoundedStack.java · esqueleto

```java
public class BoundedStack<E> {
    public BoundedStack(int capacity)
    public void push(E x)
    public E pop()
    public E peek()
    public int size()
    public boolean isEmpty()
    public boolean isFull()
}
```

1. Arreglo interno, invariante `0 <= size <= capacity` y `IllegalStateException`.

Laboratorio: `BoundedStack<E>`

BoundedStack.java · esqueleto

```java
public class BoundedStack<E> {
    public BoundedStack(int capacity)
    public void push(E x)
    public E pop()
    public E peek()
    public int size()
    public boolean isEmpty()
    public boolean isFull()
}
```

1. Arreglo interno, invariante `0 <= size <= capacity` y `IllegalStateException`.
2. Pruebas JUnit con dos tipos de elemento.

Laboratorio: `BoundedStack<E>`

BoundedStack.java · esqueleto

```java
public class BoundedStack<E> {
    public BoundedStack(int capacity)
    public void push(E x)
    public E pop()
    public E peek()
    public int size()
    public boolean isEmpty()
    public boolean isFull()
}
```

1. Arreglo interno, invariante `0 <= size <= capacity` y `IllegalStateException`.
2. Pruebas JUnit con dos tipos de elemento.
3. `max` con bound y `pushAll(List<? extends E>)`.

Laboratorio: BoundedStack<E>

BoundedStack.java · esqueleto

```java
public class BoundedStack<E> {
    public BoundedStack(int capacity)
    public void push(E x)
    public E pop()
    public E peek()
    public int size()
    public boolean isEmpty()
    public boolean isFull()
}
```

1. Arreglo interno, invariante `0 <= size <= capacity` y `IllegalStateException`.
2. Pruebas JUnit con dos tipos de elemento.
3. `max` con bound y `pushAll(List<? extends E>)`.
4. `javap -c`: encontrar el `checkcast`.

El mismo stack se vuelve iterable en la semana 11.

`make test` y `make errores`

bash · salida recortada

```
$ make test
== RawProblem
...
== WildcardDemo
max: 18, uva
sum: 44.0, 6.5
numbers: [1, 2, 3, 0.5]
all: [inicio, 15, 18, 11]
...
[        13 tests successful      ]
[         0 tests failed          ]
```

- Las demostraciones y las pruebas de `Box`, `Pair` y `ListUtils`.

`make test` y `make errores`

bash · salida recortada

```
$ make test
== RawProblem
...
== WildcardDemo
max: 18, uva
sum: 44.0, 6.5
numbers: [1, 2, 3, 0.5]
all: [inicio, 15, 18, 11]
...
[        13 tests successful      ]
[         0 tests failed          ]
```

- Las demostraciones y las pruebas de `Box`, `Pair` y `ListUtils`.
- `make errores`: compila uno por uno los archivos de `errores/` y verifica que ninguno compile con `-Xlint:all -Werror`.

`make test` y `make errores`

bash · salida recortada

```
$ make test
== RawProblem
...
== WildcardDemo
max: 18, uva
sum: 44.0, 6.5
numbers: [1, 2, 3, 0.5]
all: [inicio, 15, 18, 11]
...
[        13 tests successful      ]
[         0 tests failed          ]
```

- Las demostraciones y las pruebas de `Box`, `Pair` y `ListUtils`.
- `make errores`: compila uno por uno los archivos de `errores/` y verifica que ninguno compile con `-Xlint:all -Werror`.
- Así quedan escritos ejemplos de lo que el compilador rechaza.

Resumen

- Con `Object`, el error de tipo aparece **en ejecución**. Con genéricos, **al compilar**.
- **Clase genérica**: `Box<T>`, `Pair<A, B>`. **Método genérico**: `static <T> T f(...)`. El compilador **infiere** `T`; el diamante `<>` también.
- Sin genéricos de primitivos: wrappers, **boxing** y **unboxing**. Los `Integer`, con `equals`.
- **Raw types**: sin verificación. `-Xlint:all` los delata.
- Genéricos **invariantes**; arreglos covariantes (`ArrayStoreException`).
- **Bounded types**: `<T extends Comparable<T>`». **Wildcards**: `? extends` para leer, `? super` para escribir: **PECS**.
- **Type erasure**: `T` $\rightarrow$ `Object` (o el bound) + `checkcast`. Sin `new T()`, `new T[n]`, `instanceof List<String>` ni overloading por type argument.

Para la próxima clase

- Lecturas de la semana 11: Eck, 10.1.4, 10.2 y 10.3: el framework de colecciones, `List`, `Set` y `Map`.
- Laboratorio: `BoundedStack<E>` con sus pruebas.
- **P3** en curso (Ventas): donde el proyecto repita una misma estructura para distintos tipos, una clase genérica evita el código duplicado.

Pregunta para pensar

¿Qué diferencia hay entre `List<?>`, `List<Object>` y el raw type `List`? ¿Qué se puede meter en cada una y qué sale al leer?