Diseño OO: Nombres, Polimorfismo tipos, genéricos y expresiones lambda

Por: Luis Daniel Benavides Navarro

3-06-2020

Java Primer

Clases e interfaces

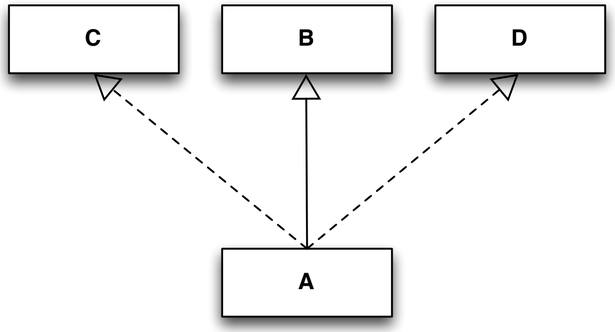
- En Java existen:
  - Clases: Definen tipo de objeto (Interface) e implementación.
  - Interfaces: Definen solo la interface de objetos (tipo)
  - Clases Abstractas: Definen la interface de objetos (tipo) y la implementación de algunos métodos, los otros son métodos abstractos.

Herencia y subtipado

- Las clases pueden heredar de una clase o de una clase abstracta (Heredan la interface(tipo) y la implementación)
- Las clases pueden implementar múltiples interfaces (múltiples tipos)
- las clases abstractas pueden heredar de una clase
- La interfaces pueden heredar de otras interfaces (implementar otros tipos)

Polimorfismo primer

- Objetos de tipo A se puede comportar como 4 tipos diferentes
- **`public class A extends B implements C, D{ … }`**
  - **`A a = new B();`** //No válido
  - **`B b = new A();`** //Válido
  - **`C c = new A();`** //Válido
  - **`D d = new A();`** //Válido
  - **`C c = new B();`** //Depende
  - **`B b = new C();`** //Inválido
  - **`A a = new C();`** //Inválido
  - **`C c = new C(){// Se definen todos los métodos de C};`** //Válido clase anónima



Tipos Genéricos

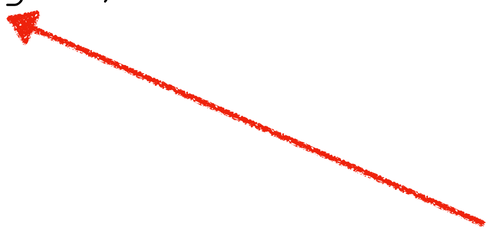
- Esta presentación está basada en la excelente guía desarrollada por
  - **Gilad Bracha**
- La puede encontrar en:
  - https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Los errores son inevitables

- Durante el diseño de software los errores son inevitables.
- Los peores son los errores en tiempo de ejecución, ya que se detectan muy tarde y son costosos de corregir.
- Los que se detectan en tiempo de compilación son menos costosos y se corrigen a tiempo

Un problema típico en Java antiguo

```
List intList = new LinkedList();
intList.add (new Integer(0));
Integer x = (Integer) intList.iterator().next();
```



Casting. El compilador no sabe el tipo de la colección. Propenso a errores de runtime. El programador puede equivocarse. El compilador no puede verificar una decisión de diseño

Una versión mejorada usando genéricos

Expresamos una decisión de diseño. El tipo de la colección. El compilador puede verificar y reforzar esta decisión.

```
List<Integer> intList = new LinkedList<Integer>();
intList.add (new Integer(0));
Integer x = intList.iterator().next();
```

Los tipos genéricos (tipos parametrizados) permiten expresar desiciones de diseño sobre los tipos para ser chequeadas por el compilador

Definiendo Tipos Genéricos Simples

Del módulo java.base, paquete java.util

```
public interface List <E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Ojo: no se crean múltiples interfaces. Solo hay una interface List que es una interface genérica

Subtipado

Si B es subtipo de A, List<B> no es subtipo de List<A>

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;

//Imagine que si lo permitiera, entonces:
lo.add(new Object());
String s = ls.get(0);
```

Ojo: Compilador no deja hacer esto

Le asignaría un Object a un string

Wildcards

Intentemos imprimir todos los elementos de una colección

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

Esto es en Java antiguo. Imprime colecciones de cualquier tipo

```
void printCollection(Collection<Object> c)
{
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Esta versión es peor que la anterior. ¿Por qué?

Wildcards

Intentemos imprimir todos los elementos de una colección

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

Esto es en Java antiguo. Imprime colecciones de cualquier tipo

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Esta versión si permite imprimir colecciones de cualquier tipo

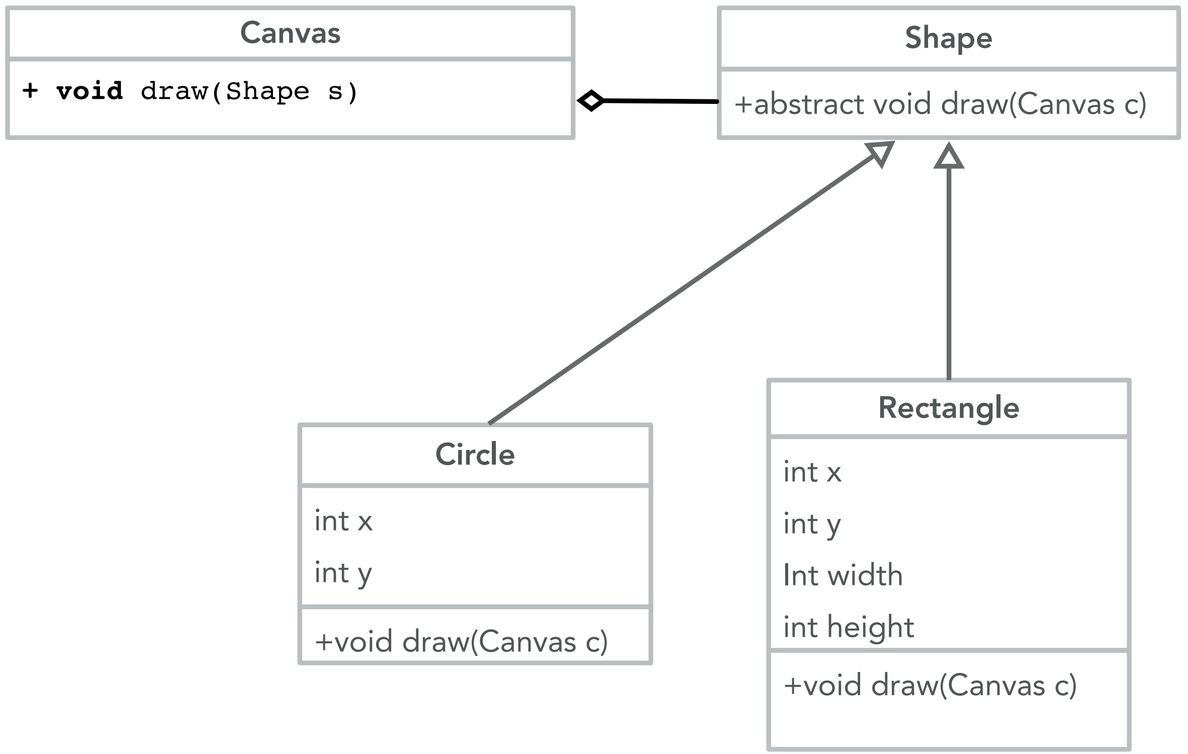
Wildcards y escritura

```
void printCollection(Collection<?> c) {
    C.add(new Object());
    for (Object e : c) {
        System.out.println(e);
    }
}
```

El tipo de la colección es desconocido. Así el compilador me permitirá leer pero no escribir en la colección. No puede inferir el tipo.

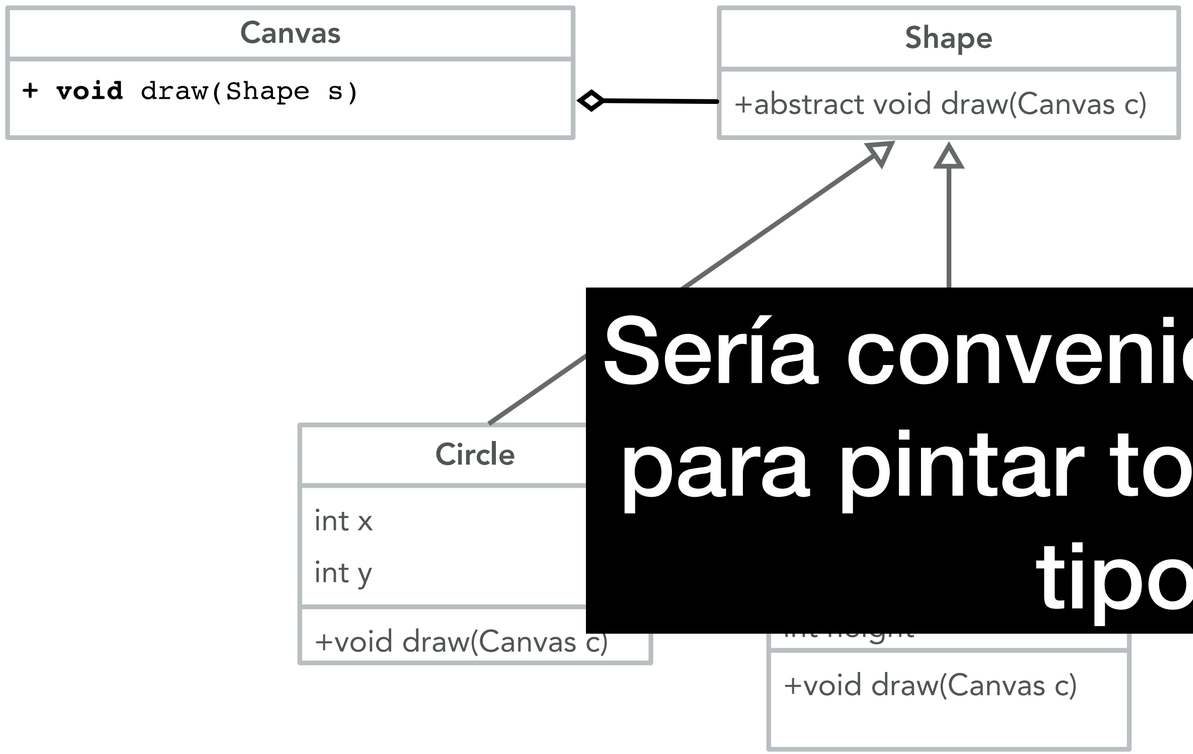
Error

Wildcards con límites




```
public abstract class Shape {
    public abstract void draw(Canvas c);}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {
        ...
    }}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {
        ...
    }}

public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
   }}
```

Wildcards con límites



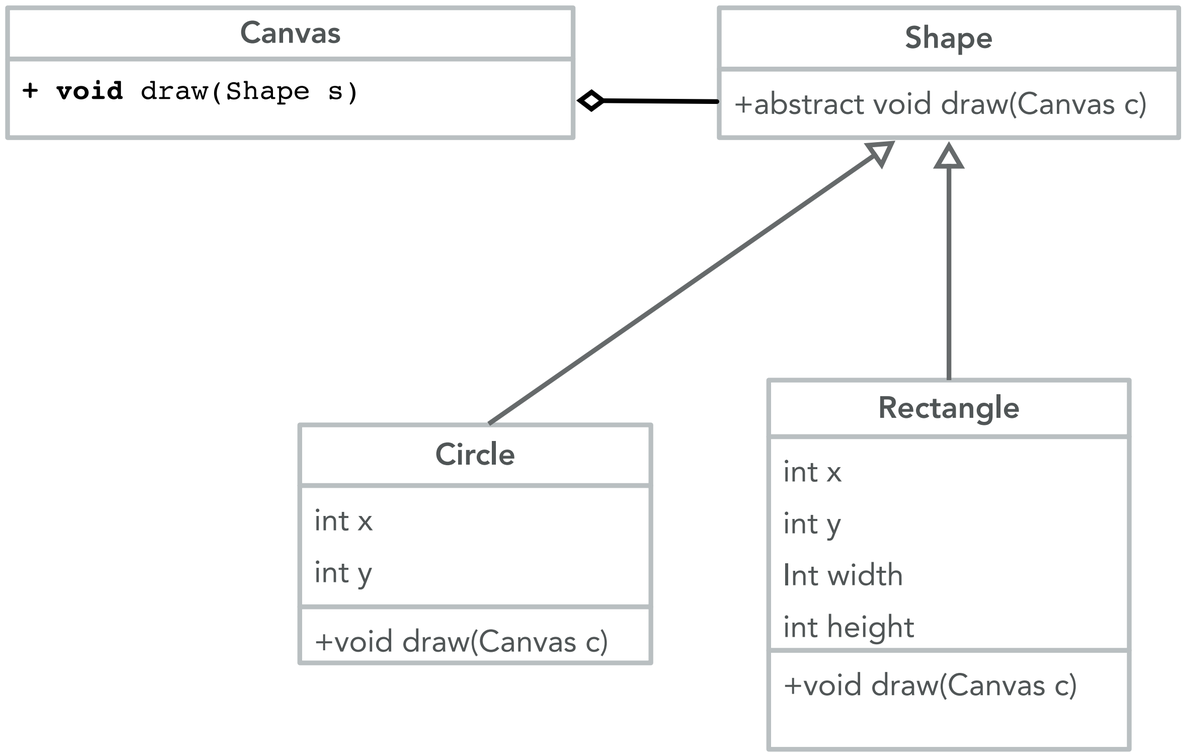
Sería conveniente tener un método para pintar todas las figuras de un tipo específico

```
public abstract class Shape {
    public abstract void draw(Canvas c);}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {
```

```
s Shape {
eight;
 {
```

```
    }}

public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
   }}
```

Wildcards con límites

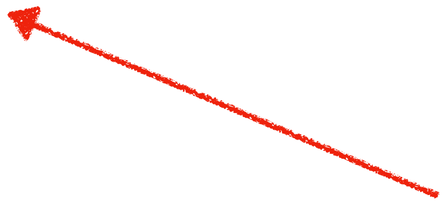



```
// No funciona, recuerde si B es
// subtype de A
// No implica que List<B> sea subtipo de
// List<A>
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
    }
}

// Mejor
// Aunque el precio a pagar es que no
// puedo escribir en la colección
public void
drawAll(List<? extends Shape> shapes) {
    ...
}
```

Métodos genéricos

Suponga el problema de copiar un arreglo de objetos en una colección

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // compile-time error
    }
}
```



No me permite copiar objetos en c. El tipo de c es desconocido

Métodos genéricos

Suponga el problema de copiar un arreglo de objetos en una colección

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
```

Ahora si usamos
métodos genéricos

Más información en

- https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Expresiones Lambda en Java

Referencias

- Basada en el Tutorial de expresiones Lambda que presenta ORACLE en:

- http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

¿Qué son ?

- Son un nuevo mecanismo en Lambda 8 que permite tratar funcionalidad como un argumento de métodos, o código como datos.

- Antes de la programación objetos existía la programación funcional (e.g., LISP), pero su utilidad no era muy apreciada.

- Ahora, por las ventajas que ofrece para la programación concurrente y de eventos, los lenguajes OO las han incorporado en su conjunto de herramientas.

Problema

- En una red social un administrador desea ejecutar acciones sobre un grupo de usuarios que cumple con un criterio.
    - El sistema le permite seleccionar de una lista un conjunto de usuarios que cumple con un criterio.
    - El sistema ejecuta acciones sobre elementos de una lista que cumple con un criterio específico

Imagine una Lista de Personas

```
public class Person {

    public enum Sex {MALE, FEMALE}

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

List<Person> personas = new ArrayList<Person>();

Sol. 1: Métodos de búsqueda y acción

```
public static void printPersonsOlderThan(List<Person> lista, int age) {
        for (Person p : lista) {
            if (p.getAge() >= age) {
                p.printPerson();
            }
        }
}
```

Nota: Un método por cada criterio

Sol2: Métodos más genéricos

```
public static void printPersonsWithinAgeRange(List<Person> lista, int low, int high) {
        for (Person p : lista) {
            if (low <= p.getAge() && p.getAge() < high) {
                p.printPerson();
            }
        }
    }
```

Nota: Un método por cada tipo de criterio

Sol3: Especificar criterios en una clase

```
public interface CheckPerson {
    boolean test(Person p);
}
```

```
public class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.FEMALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;}}
```

```
public static void printPersons(
            List<Person> roster, CheckPerson tester) {
        for (Person p : roster) {
            if (tester.test(p)) {
                p.printPerson();
            }
        }
    }
```

```
printPersons(personas, new CheckPersonEligibleForSelectiveService());
```

Nota: Una clase por cada criterio o tipo de criterio

Sol4: Igual a la tres pero con clase anónima

```
printPersons(
                personas,
                new CheckPerson() {
                    public boolean test(Person p) {
                        return p.gender == Person.Sex.MALE
                        && p.getAge() >= 18
                        && p.getAge() <= 30;
                    }
                }
        );
```

Nota: Una clase anónima por cada criterio o tipo de criterio

Sol5: Use expresiones lambda

Ojo el método *printPersons* tiene un parámetro de tipo *CheckPerson*, que es una interfaz de un solo método y esto se llama interface funcional. Gracias a las interfaces funcionales se pueden pasar las expresiones lambda como parámetro.

```
printPersons(
                personas,
                (Person p) -> p.gender == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 29
        );
```

Nota: Una interfaz funcional por cada tipo de criterio

Sol6: Use una interfaz funcional estándar

Java provee múltiples interfaces funcionales que puede usar para no tener que crearlas. Por ejemplo la interfaz java.util.function.Predicate<T> define un método test(T t)

```
interface Predicate<T> {
    boolean test(T t);        }

public static void printPersonsWithPredicate(
            List<Person> lista, Predicate<Person> tester) {
        for (Person p : lista) {
            if (tester.test(p)) {
                p.printPerson();
            }}}


printPersonsWithPredicate(personas,  p -> p.gender == Person.Sex.FEMALE
                && p.getAge() >= 18
                && p.getAge() <= 25);
```

Nota: La acción es siempre la misma, imprimir

Sol7: Use lambda expr. para la ación

Ojo use la interfaz funcional estándar *Consumer<T>* que contiene le método *void accept(T t)*

```
public static void processPersons(List<Person>lista,
Predicate<Person> tester,
Consumer<Person> block) {
        for (Person p : lista) {
            if (tester.test(p)) {
                block.accept(p);
            }}}
```

```
processPersons(personas,  p -> p.gender == Person.Sex.MALE
                            && p.getAge() >= 18
                           && p.getAge() <= 29,
                          p -> p.printPerson());
```

Nota: Mis acciones con lambda están limitadas a Consumer y Predicate, no puedo retornar valores

Sol7: Use interfaz funcional con retorno

Ojo use la interfaz funcional estándar *Function<T, R>* que contiene le método *R apply(T t)*

```
public static void processPersonsWithFunction(List<Person> roster,Predicate<Person> tester,
            Function<Person, String> mapper,
            Consumer<String> block) {

        for (Person p : roster) {
            if (tester.test(p)) {
                String data = mapper.apply(p);
                block.accept(data);
            }
        }
    }
```

```
processPersonsWithFunction(
                personas,
                p -> p.gender == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 29,
                p -> p.emailAddress,
                email -> System.out.println(email)
        );
```

Sol. 8. Uso intensivo de Genéricos

Generalice el método processWithFunctions para aceptar una colección con elementos de cualquier tipo.

```
public static <X, Y>
void processElements(Iterable<X> source, Predicate<X> tester,
                     Function<X, Y> mapper,Consumer<Y> block) {
        for (X p : source) {
          if (tester.test(p)) {
            Y data = mapper.apply(p);
             block.accept(data);
          }
        }
     }
```

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Sol. 9 Use métodos agregados

```
personas.stream()
.filter(
      p -> p.gender == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 40)
.map(p -> p.emailAddress)
.forEach(email -> System.out.println(email));
```

Sintaxis de lambda expressions

- Lista de parámetros entre paréntesis separadas por coma. Se puede omitir el paréntesis cuando hay un solo parámetro y se pueden omitir los tipos.
- Una flecha ->
- Un cuerpo que puede ser una sentencia o un bloque.

Ejemplo calculadora

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

Sobre tipos y alcance

- La funciones lambda tiene alcance léxico, es decir no crean un nuevo entorno donde se usan nuevas referencias a variables, su alcance es del entorno que las contiene.

- Las funciones lambda solo se pueden usar en entornos en los que se puede determinar el tipo que representan.

- Más información… sigan el tutorial.