UT10 Utilización avanzada de clases

PROGRAMACIÓN DAM/DAW/Distancia

María Pérez Durán

Ribera del Tajo
instituto de educación secundaria

Herencia

Es un mecanismo que permite definir unas clases a partir de otras heredando todas sus propiedades.

El resultado es una nueva clase que extiende la funcionalidad de una clase existente sin tener que reescribir el código asociado a esta última.

Herencia

La nueva clase, a la que se conoce como **subclase**, puede poseer atributos y métodos que no existan en la clase original

Los objetos de la nueva clase heredan los atributos y los métodos de la clase original, que se denomina **superclase**

Herencia: La clase Object

Es la raíz de toda la jerarquía de Java. Todas las clases de Java derivan de Object

Tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase.

Herencia: Clase Object

Los más importantes son los siguientes:

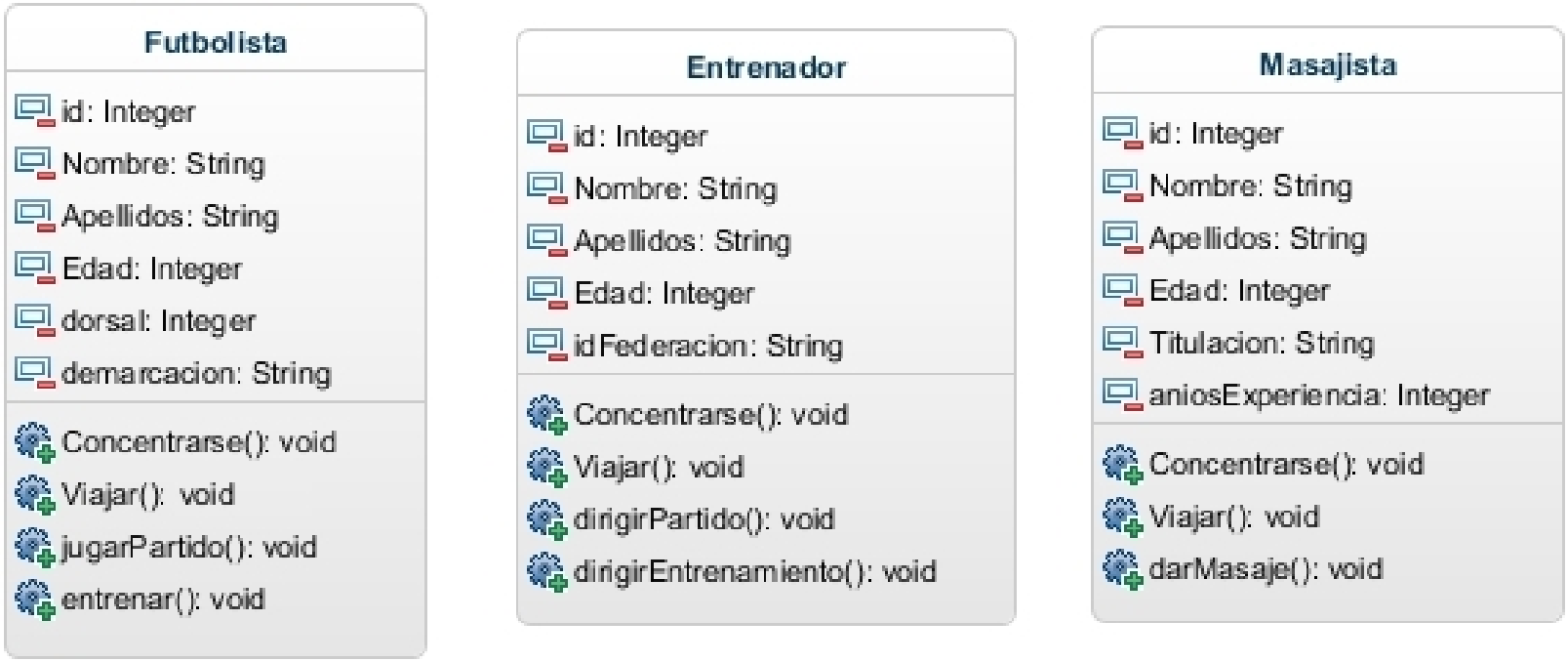
- clone() crea un objeto a partir de otro de la misma clase. No debería llamar al operador new ni a los constructores
- equals() Indica si dos objetos son o no iguales. Devuelve true si son iguales
- toString() Devuelve un String que contiene la representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.
- finalize() se llama automáticamente cuando se vaya a destruir el objeto

Ejemplo de Herencia

Vamos a simular el comportamiento que tendrían los diferentes integrantes de la selección española de futbol; tanto los futbolistas como el cuerpo técnico.

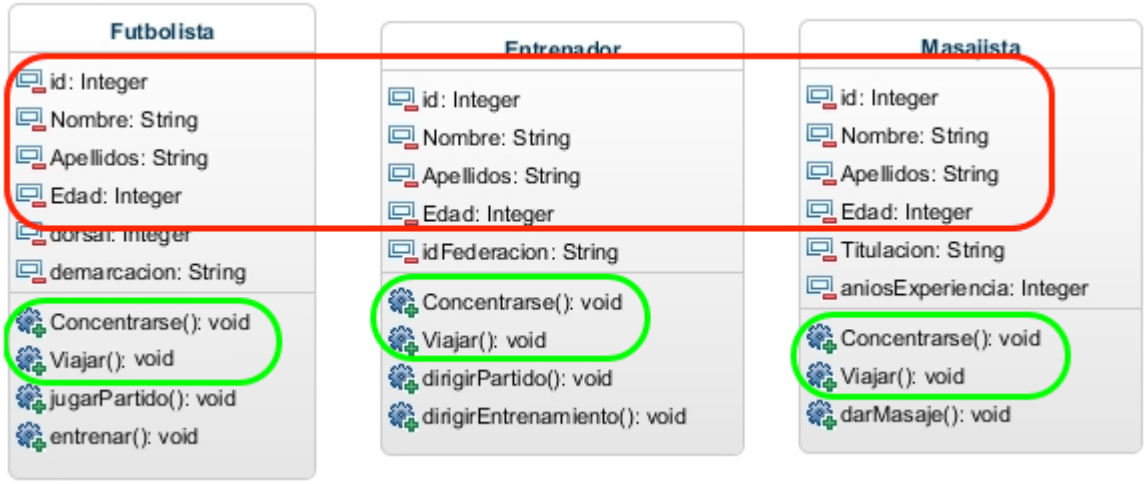
De cada unos de ellos vamos a necesitar algunos datos que reflejaremos en los ***atributos*** y una serie de acciones que reflejaremos en sus ***métodos***. Estos atributos y métodos los mostramos en el siguiente diagrama de clases:

Ejemplo de Herencia



Ejemplo de Herencia

Como se puede observar, vemos que en las tres clases tenemos atributos y métodos que son iguales ya que los tres tienen los atributos *id, Nombre, Apellidos y Edad*; y los tres tienen los métodos de *Viajar* y *Concentrarse:*



Ejemplo Herencia

```
public class Futbolista
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private int dorsal;
        private String demarcacion;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void jugarPartido() {
                ...
        }

        public void entrenar() {
                ...
        }
}
```

```
public class Entrenador
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String idFederacion;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void dirigirPartido() {
                ...
        }

        public void dirigirEntreno() {
                ...
        }
}
```

```
public class Masajista
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String Titulacion;
        private int aniosExperiencia;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void darMasaje() {
                ...
        }
}
```

Herencia

El objetivo será:

Crear una clase con el código que es común a las tres clases que será la Clase Padre o SuperClase

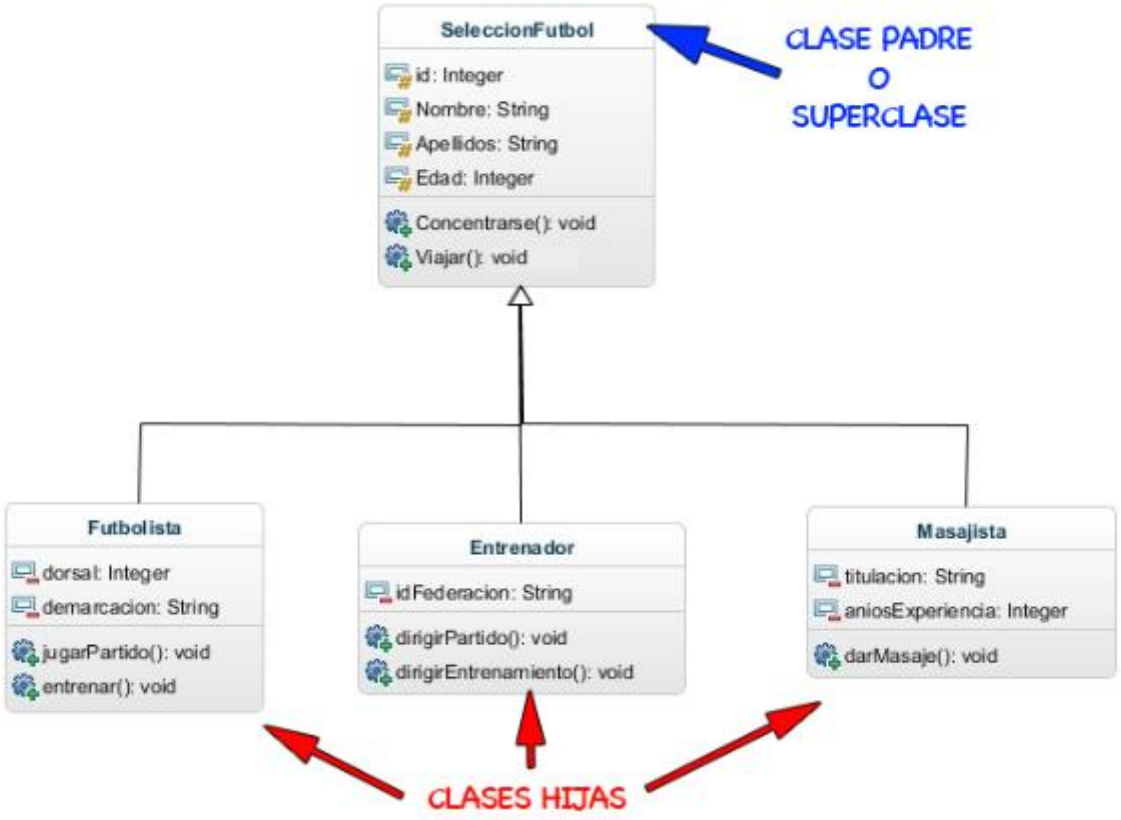
Con ***el código que es especifico de cada clase***, lo dejaremos en ella, siendo ***denominadas estas clases como Clases Hijas o subclases***

Herencia

Las clases hijas o subclases ***heredan de la clase padre todos los atributos y métodos públicos o protegidos***.

Es muy importante decir que las clases hijas ***no van a heredar nunca los atributos y métodos privados de la clase padre***

Ejemplo de Herencia



```
public class SeleccionFutbol
{

        protected int id;
        protected String Nombre;
        protected String Apellidos;
        protected int Edad;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }
}
```

```
public class Futbolista extends SeleccionFutbol
{
        private int dorsal;
        private String demarcacion;

        public Futbolista() {
                super();
        }

        // getter y setter

        public void jugarPartido() {
                ...
        }

        public void entrenar() {
                ...
        }
}
```

```
public class Entrenador extends SeleccionFutbol
{

        private String idFederacion;

        public Entrenador() {
                super();
        }

        // getter y setter

        public void dirigirPartido() {
                ...
        }

        public void dirigirEntreno() {
                ...
        }
}
```

```
public class Masajista extends SeleccionFutbol
{

        private String Titulacion;
        private int aniosExperiencia;

        public Masajista() {
                super();
        }

        // getter y setter

        public void darMasaje() {
                ...
        }
}
```

Operador intanceof

El operador **instanceof** nos permite comprobar si un objeto es de una **clase concreta.**

Operador intanceof

Por ejemplo, tenemos 3 clases:

Empleado (clase padre),

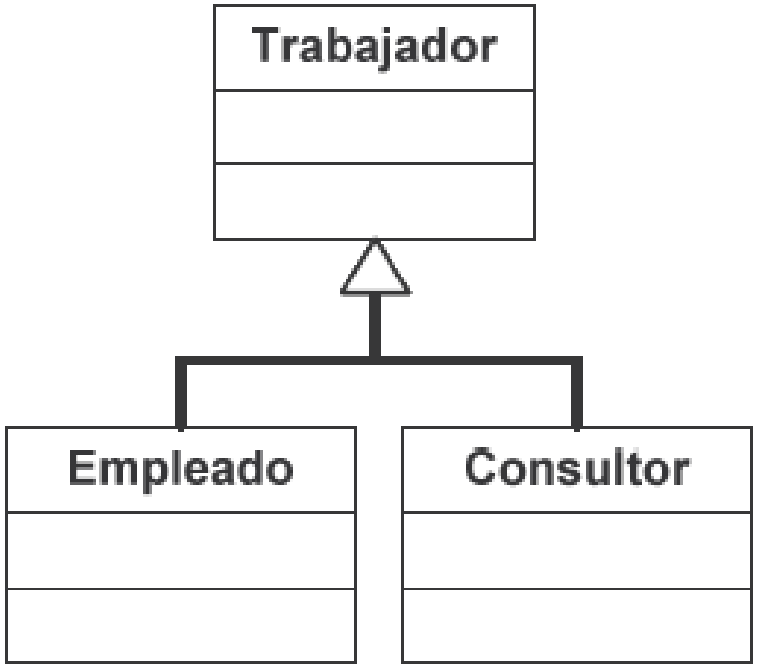
Comercial (clase hija de Empleado)

Repartidor (clase hija Empleado).

Operador intanceof

```
public class EmpleadoApp {

    public static void main(String[] args) {

        Empleado empleados[]=new Empleado[3];
        empleados[0]=new Empleado();
        empleados[1]=new Comercial();
        empleados[2]=new Repartidor();

        for(int i=0;i<empleados.length;i++){
            if(empleados[i] instanceof Empleado){
                System.out.println("El objeto en el indice "+i+" es de la clase Empleado");
            }
            if(empleados[i] instanceof Comercial){
                System.out.println("El objeto en el indice "+i+" es de la clase Comercial");
            }
            if(empleados[i] instanceof Repartidor){
                System.out.println("El objeto en el indice "+i+" es de la clase Repartidor");
            }

        }

    }

}
```

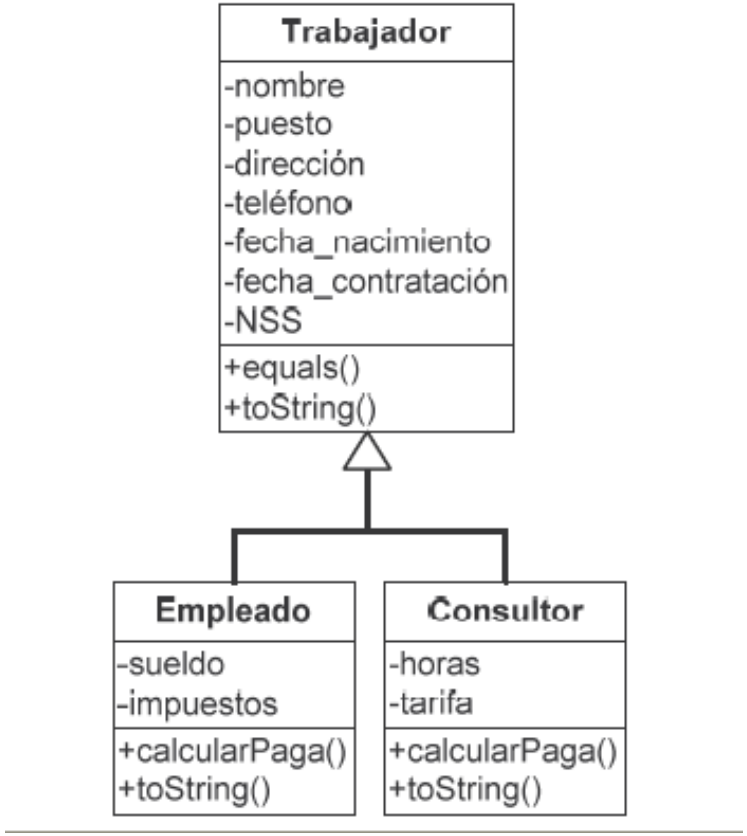
Herencia



Herencia

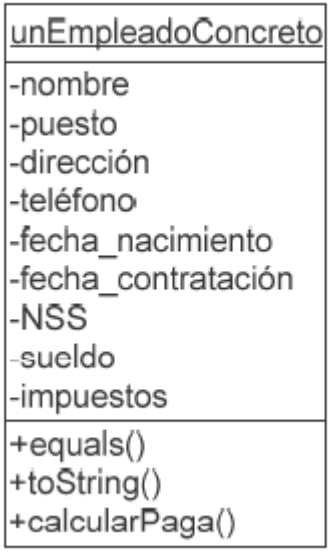
- Trabajador es una clase genérica que sirve para almacenar datos como el nombre, la dirección, el número de teléfono o el número de la seguridad social de un trabajador.
- Empleado es una clase especializada para representar los empleados que tienen una nómina mensual (encapsula datos como su salario anual o las retenciones del IRPF).
- Consultor es una clase especializada para representar a aquellos trabajadores que cobran por horas (por ejemplo, registra el número de horas que ha trabajado un consultor y su tarifa horaria).

Herencia


Trabajador
-nombre
-puesto
-dirección
-teléfono
-fecha_nacimiento
-fecha_contratación
-NSS
+equals()
+toString()
Empleado
-sueldo
-impuestos
+calcularPaga()
+toString()
Consultor
-horas
-tarifa
+calcularPaga()
+toString()

Herencia

Las clases Empleado y Consultor además de los atributos y operaciones que definen, heredan de Trabajador todos sus atributos y operaciones.



Ejercicio

Crear la clase Trabajador con todos sus atributos, los métodos de acceso y modificación y definiendo los métodos toString() y equals() (siempre deberemos definirlos).

El constructor tiene sólo 2 parámetros que son el nombre y el NSS.

El método toString() debe imprimir el nombre del Trabajador seguido de su NSS entre paréntesis. P.e. *Marta (NSS 123)*

Ejercicio

Crear la clase Empleado que hereda de trabajador.

public class Empleado **extends** Trabajador

Con la palabra reservada **super** accedemos a los miembros de la superclase desde la subclase

- En el constructor lo primero que encontramos es una llamada al constructor de la clase padre con super(…). Si no ponemos nada, se llama al constructor por defecto de la superclase antes de ejecutar el constructor de la subclase. Debe tener los siguientes parámetros.

  public Empleado(String nombre, String NSS, double sueldo)

  La variable impuestos=0.3*sueldo;

El método calcularPaga() realiza la siguiente operación:

(sueldo-impuestos)/PAGAS

donde PAGAS es una variable privada de la clase Empleado que tiene un valor constante de 14.

Redefinir el método toString para que imprima lo siguiente:

Empleado Marta (NSS 123)

Ejercicio

Crea la clase Consultor que hereda de Trabajador.

public class Consultor extends Trabajador

Crea un constructor con los siguientes parámetros:

public Consultor(String nombre, String NSS, int horas, double tarifa)

El método calcularPaga() realiza la siguiente operación: *horas*tarifa*

Redefinir el método toString para que imprima lo siguiente:

Consultor Marta (NSS 123)

Ejercicio

```
System.out.println(t);
System.out.println(e);
System.out.println(c);
```

Crea en la clase principal los siguientes objetos:

Trabajador t=new Trabajador ("Marta","123");

Empleado e=new Empleado("Juan", "456",24000.0);

Consultor c= new Consultor("Marta", "123", 10, 50.0);

Imprímelos por pantalla

Comprueba si el trabajador es un empleado o un consultor utilizando el método equals()

```
if (t.equals(e))
   System.out.println("El trabajador "+t.GetNombre() +" es un empleado");
```

Acceso a miembros heredados

Cuadro de niveles accesibilidad a los atributos de una clase

	Misma clase	Subclase	Mismo paquete	Otro paquete
Sin modificador (paquete)	X		X	X
public	X	X	X	X
private	X			
protected	X	X	X	

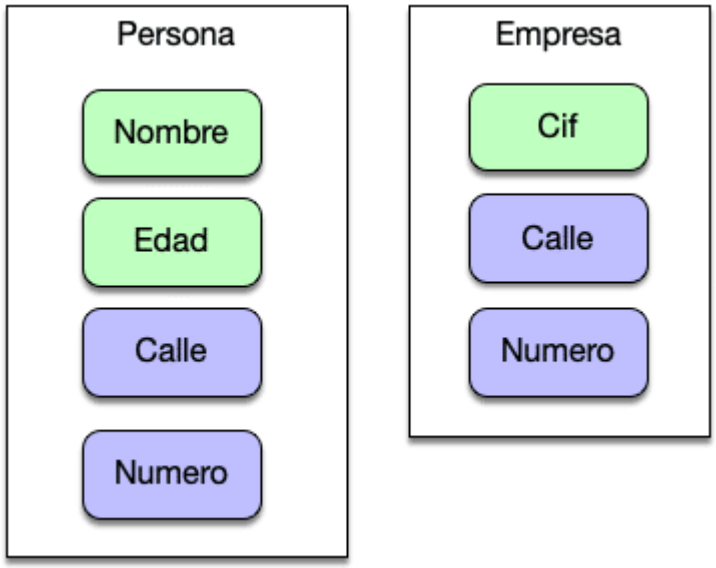
Composición de clases

La composición es el agrupamiento de uno o varios objetos y valores, como atributos, que conforman el valor de los distintos objetos de una clase.

Normalmente, los atributos contenidos se declaran con acceso privado *(private)* y se inicializan en el constructor de la nueva clase.

Composición de clases

En este ejemplo la Persona como la Empresa contienen ellas mismas la información de la dirección como propiedades individuales.

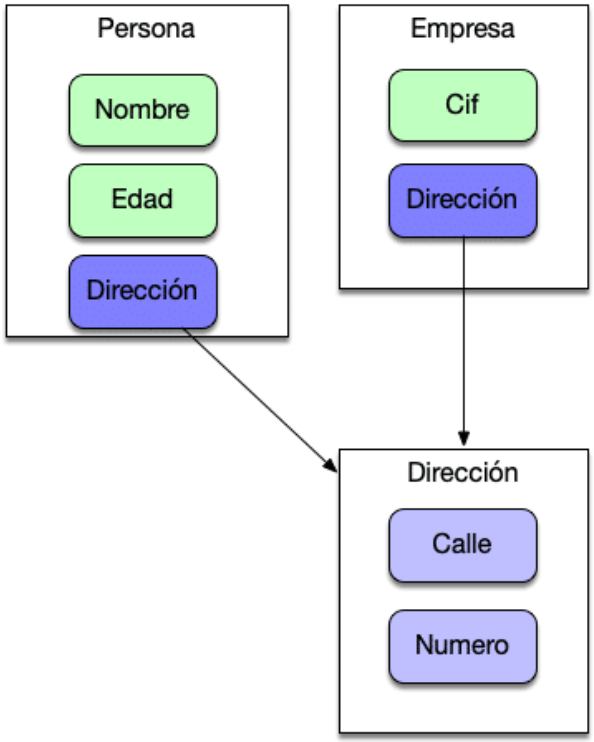


Este enfoque no favorece la reutilización ya que ambas clases comparten las pr
v nos encontramos ante una situación sencilla de repetición de código

Composición de clases

Para solventar este problema debemos apostar por un enfoque de Java Composicion .

En donde tanto Persona como Empresa se apoyan en la clase Dirección para gestionar el concepto de calle y numero



Composición de clases

```java
public class Persona {

    private String nombre;
    private int edad;

    private Direccion dirección;
```

Clases anidadas o internas

Se puede definir una clase dentro de otra clase (clases internas)

```
class claseContenedora {

    // Cuerpo de la clase

…

    class claseInterna {

        // Cuerpo de la clase interna

            …

    }

}
```

Clases anidadas o internas

Se pueden distinguir varios tipos de clases internas:

- Clases internas estáticas (o clases anidadas), declaradas con el modificador static.
- Clases internas miembro, conocidas habitualmente como clases internas. Declaradas al máximo nivel de la clase contenedora y no estáticas.
- Clases internas locales, que se declaran en el interior de un bloque de código (normalmente dentro de un método).
- Clases anónimas, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la gestión de eventos en los interfaces gráficos.

Clases abstractas

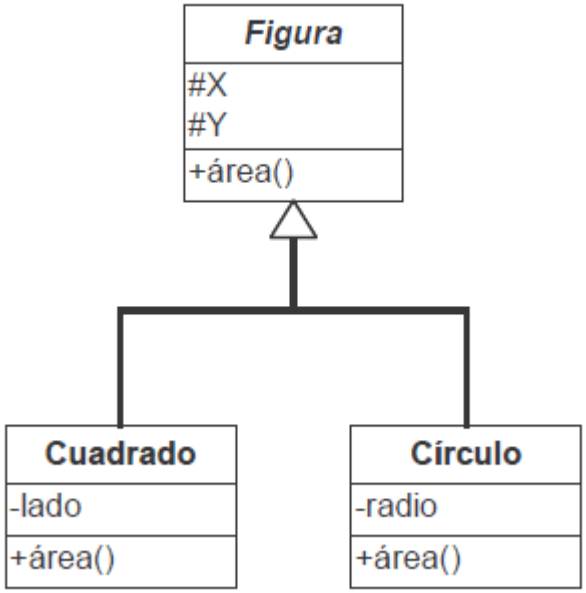
Es una clase de la que no se pueden crear objetos.

Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general

public abstract class Miclase {}

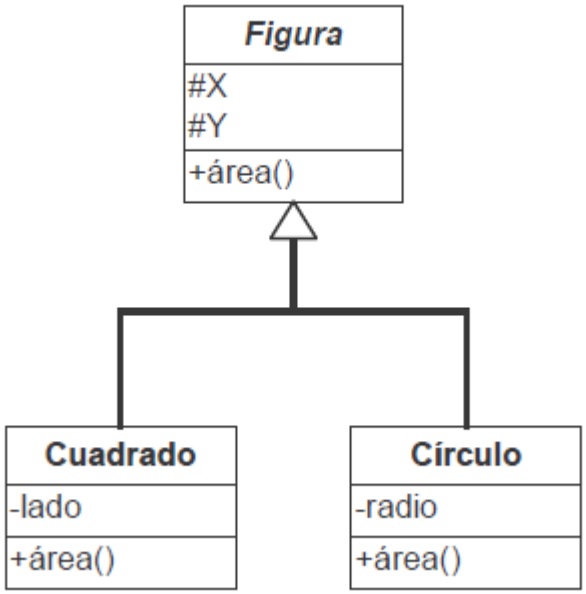
¿Cuándo se utilizan las clases abstractas?

Cuando deseamos definir una clase que englobe objetos de distintos tipos y queremos hacer uso del polimorfismo



Clase abstracta

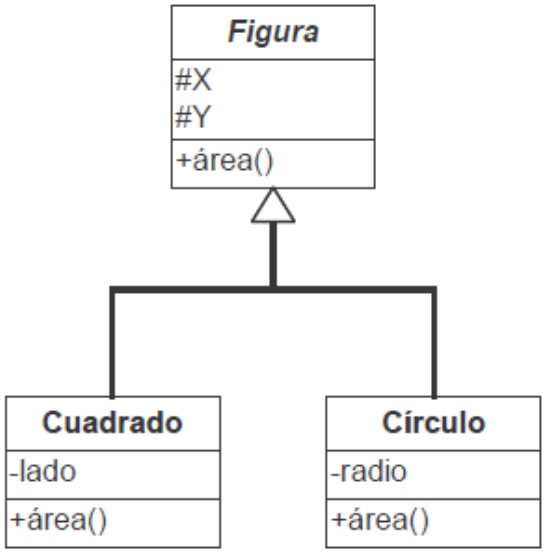
Figura es una clase abstracta porque no tiene sentido calcular su área, pero sí la de un cuadrado o un círculo.



Ejercicio

Implementa en Java estas 3 clases, sabiendo que el área del cuadrado es lado* lado y la del círculo es PI*radio*radio

Las variables X e Y indican la posición y sólo pueden acceder a ellas las clases hijas de Figura.



Métodos abstractos

Es un método cuya implementación no se define, sino que se declara únicamente su interfaz (cabecera) para que su cuerpo sea implementado más adelante en una clase derivada.

Un método se declara como abstracto mediante el uso del modificador abstract (como en las clases abstractas):

```
[modificador_acceso] abstract  <tipo> <nombreMetodo> ([parámetros]) [excepciones];
```

Métodos abstractos

Estos métodos tendrán que ser obligatoriamente redefinidos (en realidad "definidos", pues aún no tienen contenido) en las clases derivadas. Si en una clase derivada se deja algún método abstracto sin implementar, esa clase derivada será también una clase abstracta.

Debes tener en cuenta al trabajar con métodos abstractos:

1) Un método abstracto implica que la clase a la que pertenece tiene que ser abstracta, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.

2) Un método abstracto no puede ser privado (no se podría implementar, dado que las clases derivadas no tendrían acceso a él).

3) Los métodos abstractos no pueden ser estáticos, pues los métodos estáticos no pueden ser redefinidos (y los métodos abstractos necesitan ser redefinidos).

Polimorfismo

Al redefinir métodos, objetos de diferentes tipos pueden responder de forma distinta a la misma llamada (y podemos escribir código de forma general sin preocuparnos del método concreto que se ejecutará en cada momento).

Polimorfismo

Ejemplo:

Podemos añadirle a la clase Trabajador un método calcularPaga genérico (que no haga nada):

```
public class Trabajador…

  public double calcularPaga ()
  {
    return 0.0;                    // Nada por defecto
  }
```

Polimorfismo

En las subclases de Trabajador, no obstante, sí que definimos el método calcularPaga() para que calcule el importe del pago que hay que efectuarle al trabajador (en función de su tipo).

```
public class Empleado extends Trabajador…
  public double calcularPaga ()        // Nómina
  {
    return (sueldo-impuestos)/PAGAS;
  }

  class Consultor extends Trabajador…
    public double calcularPaga ()        // Por horas
    {
      return horas*tarifa;
    }
```

Polimorfismo

Como los consultores y los empleados son trabajadores, podemos crear un array de trabajadores con consultores y empleados:

```
Trabajador trabajadores[]=new Trabajador[2];
trabajadores[0]=new Empleado("Jose","135", 24000.0);
trabajadores[1]=new Consultor("Juan","246",10,50.0);
```

Polimorfismo

Una vez que tengamos un vector con todos los trabajadores de una empresa, podríamos crear un programa que realizase los pagos correspondientes a cada trabajador.

```
Trabajador trabajadores[]=new Trabajador[2];
trabajadores[0]=new Empleado("Jose","135", 24000.0);
trabajadores[1]=new Consultor("Juan","246",10,50.0);
```

Polimorfismo

La salida por pantalla sería la siguiente:

```
El trabajador Jose tiene una paga de 1200.0
El trabajador Juan tiene una paga de 500.0
BUILD SUCCESSFUL (total time: 1 second)
```

Polimorfismo

Cada vez que se invoca el método calcularPaga(), se busca automáticamente el código que en cada momento se ha de ejecutar en función del tipo de trabajador (enlace dinámico)

La búsqueda del método que se ha de invocar como respuesta a un mensaje dado se inicia con la clase receptor. Si no se encuentra un método apropiado en esta clase, se busca en su clase padre y así sucesivamente hasta encontrar la implementación adecuada del método que se ha de ejecutar como respuesta a la invocación original

Polimorfismo

No hay que confundir el polimorfismo con la sobrecarga de métodos (distintos métodos con el mismo nombre pero con diferentes parámetros) p.e. la sobrecarga de constructores dentro de una misma clase

```
public Punto (){
        this.x = 0.0;
        this.y = 0.0;
}

public Punto ( double coordenadaX, double coordenadaY){
        x = coordenadaX;
        y = coordenadaY;
}
```

Polimorfismo

Una clase puede redefinir algunos de los métodos que ha heredado de su clase base. En tal caso, el nuevo método (especializado) sustituye al heredado. Este procedimiento también es conocido como de sobrescritura de métodos.

Cuando sobrescribas un método heredado en Java puedes incluir la anotación @Override. Esto indicará al compilador que tu intención es sobrescribir el método de la clase padre.

```
@Override

public String getApellidos (){}
```

La palabra reservada final

En Java, usando la palabra reservada final, podemos:

Evitar que un método se pueda redefinir en una subclase

```
class Consultor extends Trabajador
{
  …
  public final double calcularPaga ()
  {
    return horas*tarifa;
  }
  …
}
```

Aunque creemos subclases de Consultor, el dinero que se le pague siempre será en función de las horas que trabaje y de su tarifa horaria (y eso no podremos cambiarlo aunque queramos)

La palabra reservada final

Evitar que se puedan crear subclases de una clase dada:

```
public final class Circulo extends Figura
…

public final class Cuadrado extends Figura
…
```

Al usar “final”, tanto Circulo como Cuadrado son ahora clases de las que no se puede crear subclases.

En ocasiones será “final” porque no tenga sentido crear subclases o, simplemente porque no deseemos que la clase se pueda extender.

RECORDATORIO: final también sirve para definir constantes

Ampliación de métodos heredados

Para poder hacer esto necesitas poder preservar el comportamiento antiguo (el de la superclase) y añadir el nuevo (el de la subclase)

```
public void mostrar () {
    super.mostrar ();     // Llamada al método "mostrar" de la superclase
    // A continuación mostramos la información "especializada" de esta subclase
    System.out.printf ("Grupo: %s\n", this.grupo);
    System.out.printf ("Nota media: %5.2f\n", this.notaMedia);
}
```

Interfaz

Una interfaz en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.

Estos métodos sin implementar indican un comportamiento, un tipo de conducta, aunque no especifican cómo será ese comportamiento (implementación), pues eso dependerá de las características específicas de cada clase que decida implementar esa interfaz. Podría decirse que una interfaz se encarga de establecer qué comportamientos hay que tener (qué métodos), pero no dice nada de cómo deben llevarse a cabo esos comportamientos (implementación). Se indica sólo la forma, no la implementación.

Interfaz funcional

Una interfaz funcional será una interfaz que solamente tenga la definición de un método abstracto. En realidad, puede tener varios métodos abstractos, siempre que todos menos uno sobrescriban a un método público de la clase Object. Además, pueden tener uno o varios métodos por defecto o estáticos.

```
public interface Interfaz {

    public void metodo();

    default public void metodoPorDefecto() {
            System.out.println("Este es uno de los nuevos
                                        métodos por defecto");
    }

    public static void metodoEstatico() {
            System.out.println("Método estático en
                                        un interfaz");
```

Abstracto

Por defecto

Estático

Interfaz funcional

```java
@FunctionalInterface
public interface MiInterfaz {

    void print(String str);

    boolean equals(Object o);

    default void defecto() {
        System.out.println("Método con implementación por defecto");
    }

    static void estatico() {
        System.out.println("Método estático");
    }

}
```

Paradigma de programación funcional

Antes de continuar con las expresiones lambda vamos a introducir el paradigma de programación funcional

Un paradigma de programación es un conjunto de normas, patrones o estilos de programación.

Los paradigmas que hemos usado hasta ahora han sido

1) Paradigma imperativo (basado en sentencias)

2) Paradigma orientado a objetos

Paradigma de programación funcional

Basado en el concepto matemático de función

La salida de una función depende exclusivamente de sus parámetros de entrada

Utiliza menos código y es más elegante

No es una novedad (origen 1930)

Paradigma de programación funcional

Las funciones se convierten en entidades de primer nivel como las variables o los objetos

Por tanto, podremos pasar una función como parámetro a un método, como argumento o devolver una función

Paradigma de programación funcional

Old Imperative Style	New Declarative Style
Internal iteration, using new default method `Iterable#forEach(Consumer<? super T> action)`	
`List<String> list = Arrays.asList("Apple", "Orange", "Banana"); for (String s : list) { System.out.println(s); }`	`List<String> list = Arrays.asList("Apple", "Orange", "Banana"); //using lambda expression list.forEach(s -> System.out.println(s)); //or using method reference on System.out instance list.forEach(System.out::println);`

Paradigma de programación funcional

Counting even numbers in a list, using `Collection#stream()` and `java.util.stream.Stream`

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int count = 0;
for (Integer i : list) {
    if (i % 2 == 0) {
        count++;
    }
}
System.out.println(count);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
long count = list.stream()
                 .filter(i -> i % 2 == 0)
                 .count();
System.out.println(count);
```

Paradigma de programación funcional

Retrieving even number list

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
List<Integer> evenList = new ArrayList<>();
for (Integer i : list) {
    if (i % 2 == 0) {
        evenList.add(i);
    }
}
System.out.println(evenList);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
List<Integer> evenList =
                list.stream()
                    .filter(i -> i % 2 == 0)
                    .collect(Collectors.toList());
System.out.println(evenList);
```

Or if we are only interested in printing:

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
list.stream().filter(i -> i % 2 == 0)
            .forEach(System.out::println);
```

Paradigma de programación funcional

Finding sum of all even numbers

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int sum = 0;
for (Integer i : list) {
    if (i % 2 == 0) {
        sum += i;
    }
}
System.out.println(sum);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int sum = list.stream()
              .filter(i -> i % 2 == 0)
              .mapToInt(Integer::intValue)
              .sum();
System.out.println(sum);
```

Alternatively

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int sum = list.stream()
              .filter(i -> i % 2 == 0)
              .reduce(0, (i, c) -> i + c);
System.out.println(sum);
```

Paradigma de programación funcional

Finding whether all integers are less than 10 in the list

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
boolean b = true;
for (Integer i : list) {
    if (i >= 10) {
        b = false;
        break;
    }
}
System.out.println(b);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
boolean b = list.stream()
                .allMatch(i -> i < 10);
System.out.println(b);
```

Also look at `Stream#anyMatch(...)` method

Expresión lamba

Una expresión lambda es una función sin nombre.

Su estructura es

```
() -> expresión

(parámetros) -> expresión

(parámetros) -> { sentencias; }
```

Expresión lamba

Por ejemplo:

```
() -> new ArrayList<>()

(int a, int b) -> a+b

(a)-> {
      System.out.println(a);
      return true;
 }
```

Interfaz funcional y expresiones lambda

En lugar de implementar una interfaz funcional en una clase, o en una clase anónima, podemos utilizar una expresión lambda.

INTERFACES FUNCIONALES MUY ÚTILES

1) PREDICATE
2) CONSUMER
3) FUNCTION
4) SUPPLIER

1) Predicate

Método boolean test(T t)

Comprueba si se cumple o no una determinada condición

Muy utilizado con expresiones lambda para filtrar

```
listaPersonas
    .stream()
    .filter((p) -> p.getEdad() >= 351)
    .forEach(System.out::println);
```

```
Stream<T>        filter(Predicate<? super T> predicate)
                 Returns a stream consisting of the elements of this stream that match the
                 given predicate.
```

1) Predicate

Se pueden construir predicados complejos utilizando and, or o negate

```
Predicate<Persona> edad = (p) -> p.getEdad() >= 35l;
Predicate<Persona> nombre = (p) ->
                            p.getApellidos().contains("e");
Predicate<Persona> complejo = edad.or(nombre);
```

2) Consumer

Método void accept(T t)

Sirve para consumir objetos, el ejemplo más común es imprimir

Se utiliza con expresiones lambda para imprimir

```
listaPersonas
    .stream()
    .filter((p) -> p.getEdad() >= 35l)
    .forEach(System.out::println);
```

3) Function

Método R apply(T t)

Sirve para aplicar una transformación sobre un objeto, es decir, para pasar un objeto a otro.

El ejemplo más claro es el mapeo de un objeto en otro

Se utiliza con expresiones lambda para mapear

```
lista
    .stream()
    .map((p) -> p.getNombre())
    .forEach(System.out::println);
```

3) Function

Sirve para aplicar una transformación sobre un objeto, es decir, para pasar un objeto a otro.

El ejemplo más claro es el mapeo de un objeto en otro

Se utiliza con expresiones lambda para mapear

```
lista
    .stream()
    .map((p) -> p.getNombre())
    .forEach(System.out::println);
```

4) Supplier

Método T get()

No recibe parámetros

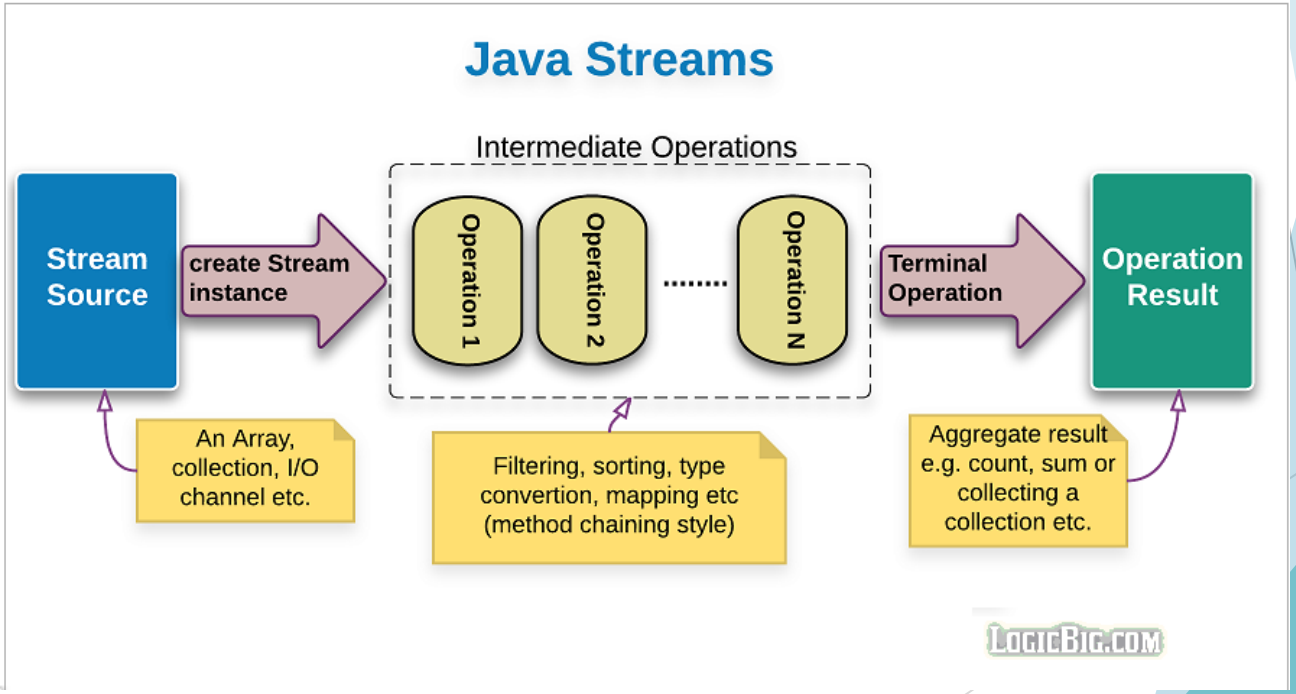
Sirve para obtener objetos

Su sintaxis como expresión lambda sería

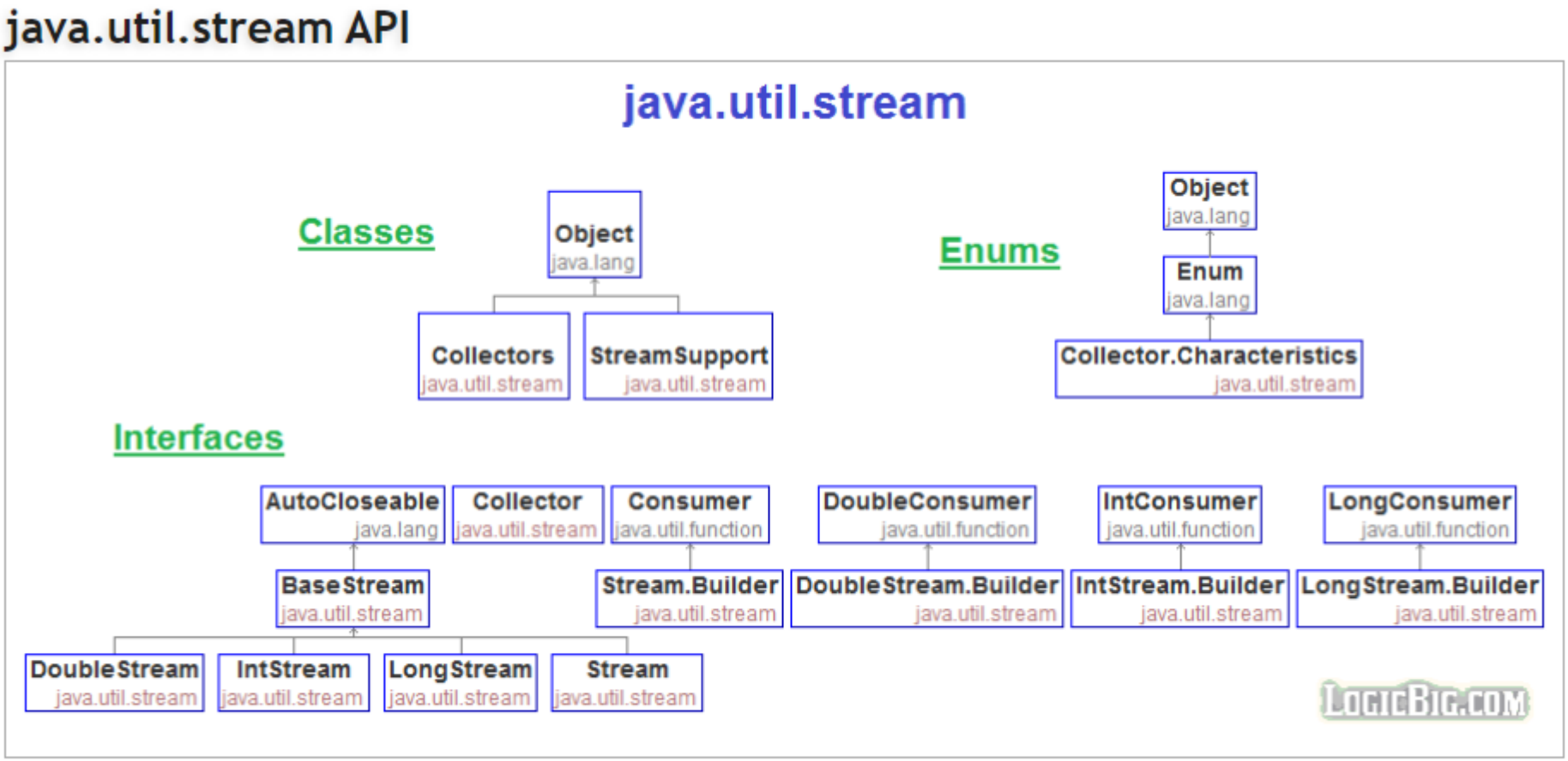
```
() -> something
() -> { return something; }
```

Interface Stream<T>

Un stream es una secuencia de objetos que soportan varios métodos y éstos pueden ser canalizados para producir el resultado esperado.



Interface Stream<T>



Interface Stream<T>

Las principales características de un Java stream son:
1) No es una estructura de datos pero si toma como entrada colecciones o arrays
2) Los streams no cambian la estructura de datos, solo proporcionan la salida con un conjunto de métodos que tienen como entrada la salida del anterior (pipeline)
3) Cada operación intermedia de un stream devuelve un stream
4) Las operaciones terminales marcan el final del stream y devuelven el resultado

Interface Stream<T>

Operaciones intermedias:

1. map: Devuelve un stream que consiste en aplicar una función dada a los elementos del stream

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x-
>x*x).collect(Collectors.toList());
```

Interface Stream<T>

Operaciones intermedias:

1. map:

a) Es muy usada

b) Permite aplicar una transformación a una serie de objetos

c) Recibe como argumento una Function<T,R> para realizar la transformación

d) Se invoca sobre un stream<T> y devuelve un stream <R>

e) Se pueden realizar transformaciones sucesivas

```
lista
    .stream()                          Stream<Persona>
    .map(Persona::getNombre)           Stream<String>
    .map(String::toUpperCase)          Stream<String>
    .forEach(System.out::println);
```

Interface Stream<T>

Operaciones intermedias:

2. filter: se usa para seleccionar elementos aplicando la expresión lambda que se pasa como parámetro

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s-
>s.startsWith("S")).collect(Collectors.toList());
```

Interface Stream<T>

Operaciones intermedias:

3. sorted: se usa para ordenar el stream()

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());
```

Interface Stream<T>

Operaciones terminales

1. collect: Este método se usa para devolver el resultado de un stream como un determinado objeto

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());


List number = Arrays.asList(2,3,4,5,3);

Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

Interface Stream<T>

Operaciones terminales

2. forEach: Se usa para iterar a través de cada elemento del stream

```
List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

Interface Stream<T>

Otras operaciones terminales son:

findFirst()
- La primer coincidencia

findAny()
- Trabaja igual que findFirst(), pero actúa sobre un stream paralelo
- boolean allMatch(Predicate p)
  - Si todos los elementos del stream coinciden utilizando el Predicado p
- boolean anyMatch(Predicate p)
  - Si al menos uno de los elmeentos coincide con la condición del predicado p
- boolean noneMatch(Predicate p)
  - ninguno de los elementos coincide de acuerdo al predicado

Interface Stream<T>

Otras operaciones terminales son:

A medida que van pasando

collect(Collector c)
- Realiza una reducción mutable sobre el stream

toArray()
- regresa un array que contiene los elementos del stream

Interface Stream<T>

Otras operaciones terminales son:

count()
- Regresa el numero de elementos que hay en el stream

max (Comparator c)
- El elemento con el máximo valor que está dentro del stream en base al comparador

min(Comparador c)
- El elemento con el valor más pequeño que está dentro del stream en base al comparador
- Regresa un Optional, si el stream esta vacío

Interface Stream<T>

Otras operaciones terminales son:

Streams de tipo primitivo (IntStream, DoubleStream(), LongStream)

average()
- Regresa la media aritmética del stream
- Si el stream está vacío. regresa un Optional

sum()
- Regresa la suma de los elementos del stream

Fechas

Para utilizar fechas en java es necesario importar el paquete java.time. Entre las clases que se incluyen destacan

Class	Description
`LocalDate`	Represents a date (year, month, day (yyyy-MM-dd))
`LocalTime`	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
`LocalDateTime`	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
`DateTimeFormatter`	Formatter for displaying and parsing date-time objects

Fechas

Por ejemplo, para ver la fecha de hoy

```java
package fechas;

import java.time.LocalDate;

public class PruebaFechas {
    public static void main(String[] args) {
        LocalDate hoy = LocalDate.now();
        System.out.println(hoy);
    }
}

```

Console

<terminated> PruebaFechas [Java Application] C:\Program Files\Java\jdk-16\bin\javaw.exe (15 feb 2023 17:05:53 – 17:05:56) [pic

```
2023-02-15
```

Fechas

Para mostrar la hora actual (hora, minuto, segundo y nanosegundos)

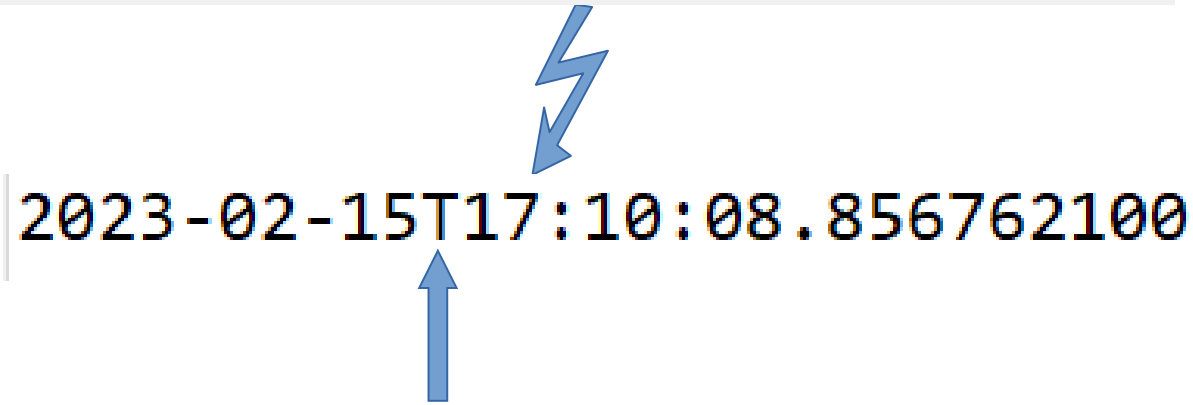
```
LocalTime ahora = LocalTime.now();
System.out.println(ahora);
```

```
17:07:28.630330700
```

Fechas

La fecha y la hora

```
LocalDateTime hoyAhora = LocalDateTime.now();
System.out.println(hoyAhora);
```



La "T" en el ejemplo anterior se usa para separar la fecha de la hora.

Fechas

Para formatear la fecha y la hora se puede usar la clase DateTimeFormatter con el método ofPattern(). Entre los valores que acepta este método encontramos

Value	Example
yyyy-MM-dd	"1988-09-29"
dd/MM/yyyy	"29/09/1988"
dd-MMM-yyyy	"29-Sep-1988"
E, MMM dd yyyy	"Thu, Sep 29 1988"

Fechas

```
LocalDateTime fechaHora = LocalDateTime.now();
System.out.println("Sin formatear: " + fechaHora);
DateTimeFormatter fechaFormato = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDate = fechaHora.format(fechaFormato);
System.out.println("Con formato: " + formattedDate);
```

```
Sin formatear: 2023-02-15T17:17:39.877618900
Con formato: 15-02-2023 17:17:39
```

Fechas

Para extraer el día, mes o año de una fecha utilizaremos los getters correspondientes:

```
int dia=LocalDate.now().getDayOfMonth();
int mes=LocalDate.now().getMonthValue();
int año=LocalDate.now().getYear();
System.out.println(dia+"/"+mes+"/"+año);
```

Fechas

Si queremos comparar fechas podemos utilizar los métodos de LocalDate isAfter() o isBefore()

```
LocalDate today = LocalDate.now();
LocalDate otraFecha = LocalDate.of(2023, 2, 24);
if (otraFecha.isAfter(today)) {
    System.out.println("La otra fecha es posterior a la de hoy");
}
LocalDate ayer = today.minus(1, ChronoUnit.DAYS);
if (ayer.isBefore(today)) {
    System.out.println("Ayer es anterior a hoy");
}
```

Fechas

También podremos ver si son iguales dos fechas utilizando el método equals()

```
LocalDate fecha = LocalDate.of(2022, 11, 11);

if (fecha.equals(hoy)) {
    System.out.printf("Hoy %s y la fecha %s son la misma fecha", hoy, fecha);
} else {
    System.out.println("Las fechas no son iguales");
}
```

Fechas

Si queremos ver fechas anteriores o posteriores podemos sumar o restar días, años, siglos, ...

```
LocalDate previousYear = hoy.minus(1, ChronoUnit.YEARS);
System.out.println("Fecha hace un año: " + previousYear);
LocalDate nextYear = hoy.plus(1, ChronoUnit.YEARS);
System.out.println("Fecha dentro de 1 año : " + nextYear);
```