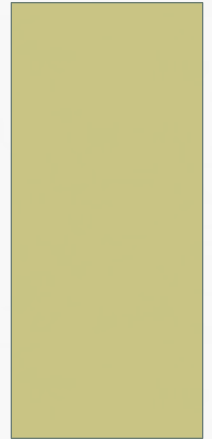




Universidad Tecnológica Nacional
Facultad Regional Córdoba

Paradigmas de Programación

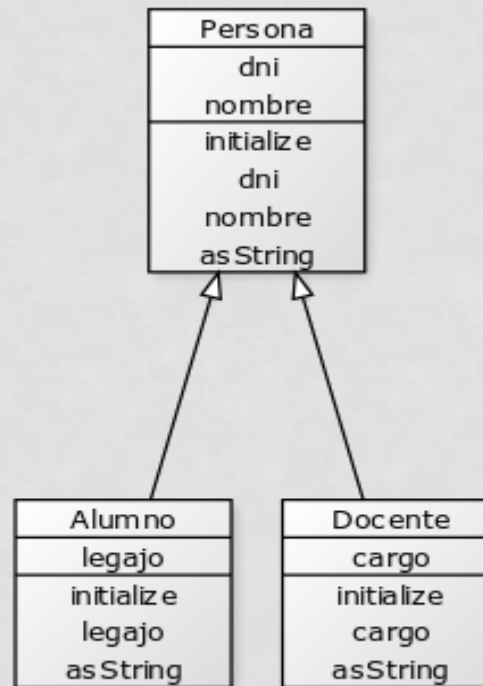
UNIDAD NRO. 3
PARADIGMA ORIENTADO A OBJETOS
(PARTE IV)



CONTENIDOS ABORDADOS

- Repaso de relaciones entre clases
- Revisión de Herencia
- Polimorfismo
- Colecciones en Smalltalk (Continuación)
 - SortedCollection
 - Conversión entre colecciones
 - Mensaje de recorrido para colecciones (Continuación)
 - detect:

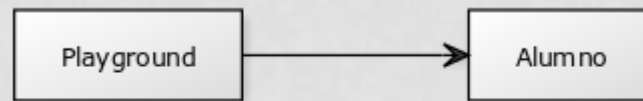
BLOQUE 1: REPASO DE HERENCIA



CREATED WITH YUML

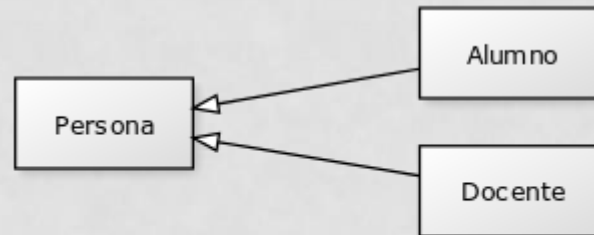
RELACIONES ENTRE CLASES

- Relación “**usa un**”: relaciones de asociación.



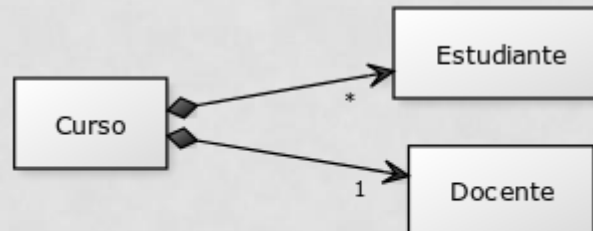
CREATED WITH YUML

- Relación “**es un**”: relaciones de especialización/generalización.(herencia)



CREATED WITH YUML

- Relación “**tiene un**”: relaciones de pertenencia



CREATED WITH YUML

HERENCIA

- Es una característica clave en los sistemas orientados a objetos.
- La herencia es un mecanismo que permite la definición de una clase a partir de otra/s ya existente/s.
- Por el cual los objetos comparten conocimiento común, atributos y comportamientos.
- Permite la reusabilidad.

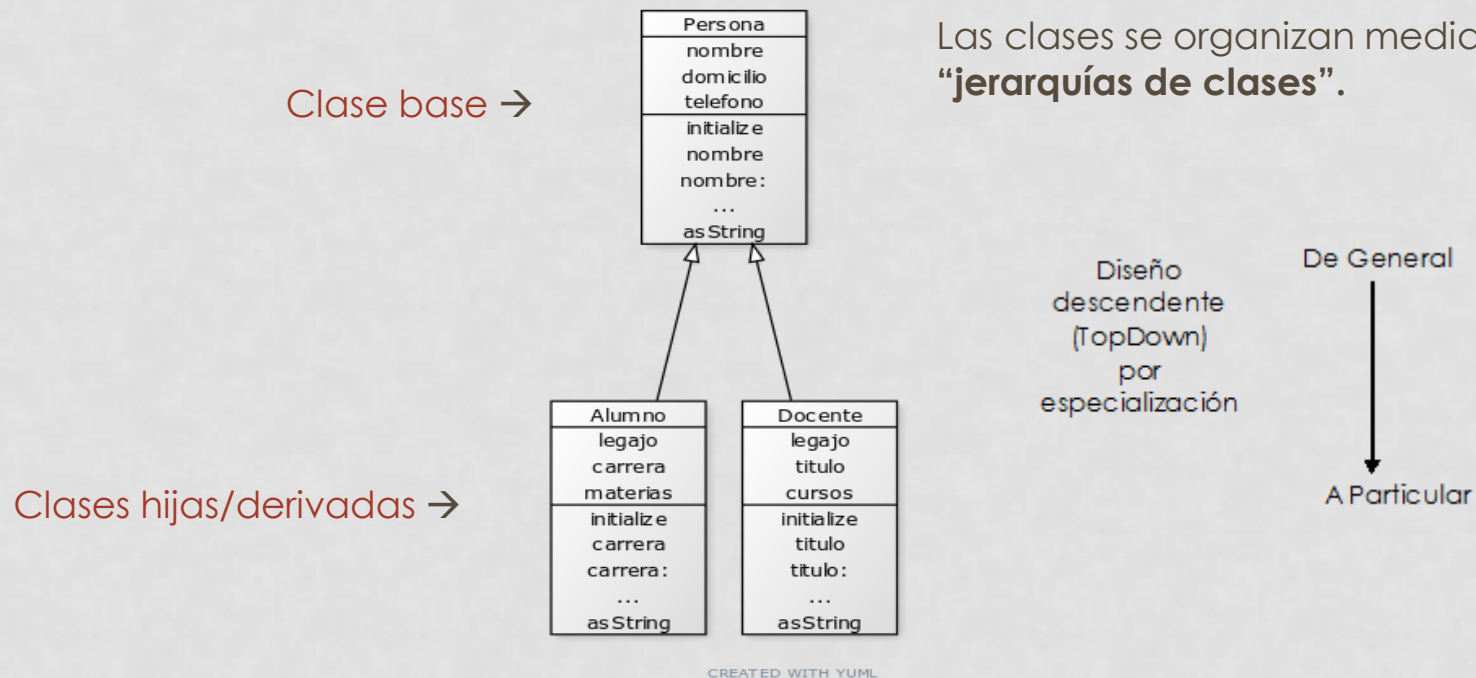
HERENCIA: BASADA EN CLASES

- Las clases se organizan jerárquicamente, y una nueva clase puede reutilizar la estructura y comportamiento de otras previamente definidas.
- Permite la reutilización de código, a través de la reutilización de los atributos y métodos de otras clases.



HERENCIA: BASADA EN CLASES

- Formada por una clase llamada base, padre o superclase y una clase llamada derivada, hija, subclase, descendiente, etc. que hereda las características y comportamientos de la clase base.



HERENCIA EN SMALLTALK

Herencia de Variables

- **Variables de Instancia:** Cada subclase tiene su propia copia, de esta forma hace a las variables de instancia propias de la instancia. Las variables de instancia de clase soportan el comportamiento de herencia, teniendo cada clase su propio estado de la variable.
- Por ejemplo: el objeto alumno tiene las variables de instancia: dni, nombre, teléfono, legajo y promedio. Estas variables incluyen las variables de instancia heredadas de Persona.

Instancia de Alumno



Variables hereda
de su antecesor

HERENCIA EN SMALLTALK

Herencia de Métodos

- La herencia de métodos es útil para permitir a una clase modificar su comportamiento respecto de su superclase. Esto puede ser hecho *“agregando nuevos métodos”*, o *“redefiniendo los métodos heredados”*.
- **Agregando Métodos:**
Se puede agregar métodos de manera muy simple, incorporándolos en la definición de la clase. Todo objeto de la subclase soporta los métodos de su superclase, más los nuevos métodos.

HERENCIA EN SMALLTALK

Herencia de Métodos

- **Redefinición de Métodos**

- Una subclase puede redefinir (volver a definir) algún método existente en la superclase, con el objeto de proveer una implementación diferente, por ejemplo: método **asString**.
- Para redefinir un método en la subclase, se tiene que declarar con la misma signature (nombre y parámetros).
- En la invocación de métodos, si existen dos métodos con la misma signature, uno en la subclase y otro en la superclase, se ejecutará siempre el de la subclase.

HERENCIA EN SMALLTALK

Clases Abstractas

- Son clases genéricas que sirven para agrupar clases del mismo género.
- No se pueden instanciar, es decir no se pueden crear objetos a partir de ellas, ya que representan conceptos tan generales que no se puede definir como será la implementación de los mismos.
- Definen comportamientos que se implementarán en las subclasses.
- Smalltalk no posee mecanismos formales para implementarlas, pero el desarrollador puede hacerlo definiendo métodos sin implementación (abstractos).

HERENCIA EN SMALLTALK

Métodos abstractos. Forma de Implementación en Smalltalk

- **Ejemplo:**

- Clase abstracta - Método abstracto:**

- size

- self subclassResponsability.

-

- size

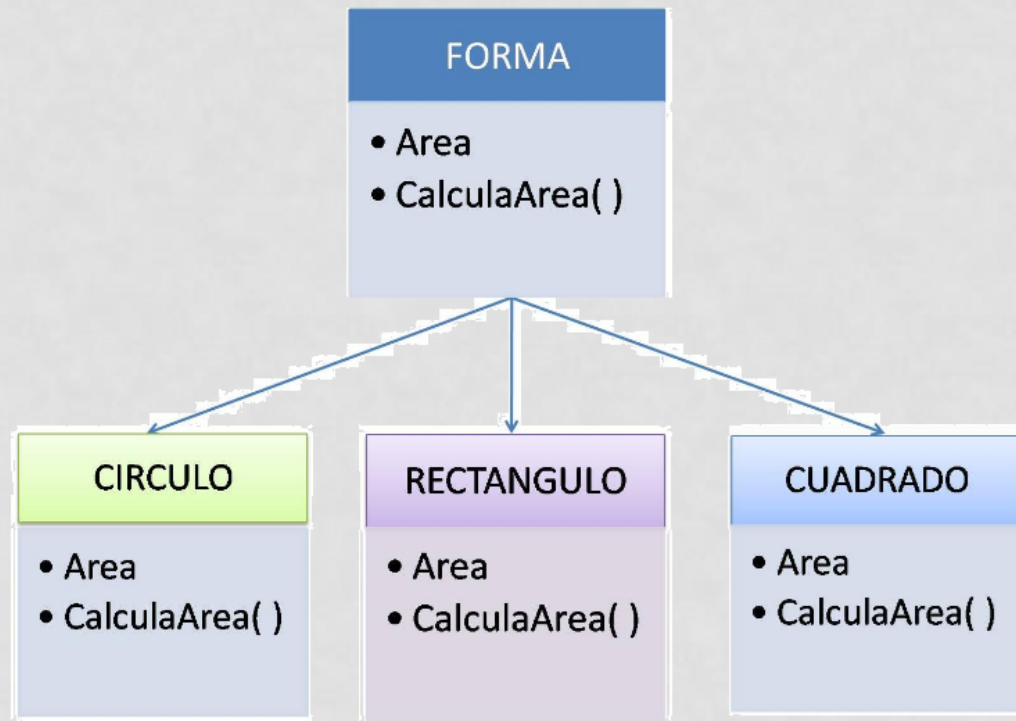
- “sin implementación”

- Clase derivada - Método implementado:**

- size

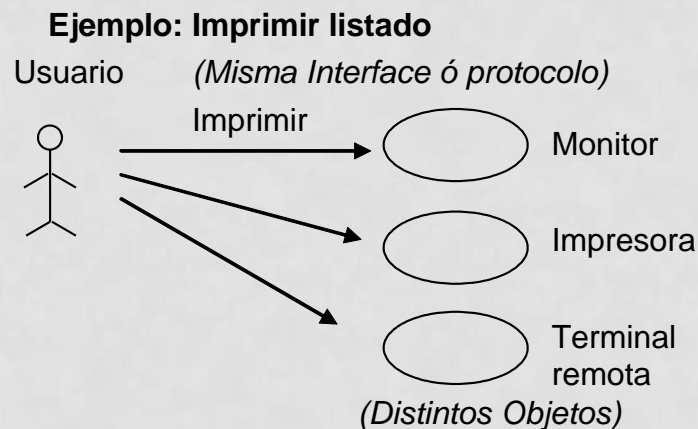
- ^size

BLOQUE 2: POLIMORFISMO



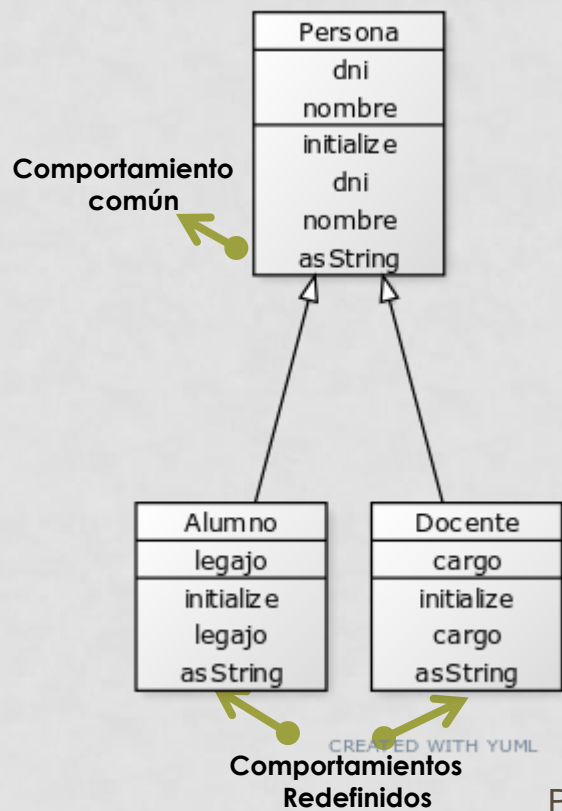
POLIMORFISMO

- Capacidad de que diferentes objetos reaccionen de distintas formas a un mismo mensaje.
- Posibilidad de acceder a un variado rango de comportamientos a través de la misma interfaz.
- Un mismo identificador puede tener distintas formas (distintos cuerpos de función, distintos comportamientos) dependiendo, en general, del contexto en el que se halle inserto.



POLIMORFISMO EN SMALLTALK

- Smalltalk permite implementar diferentes tipos de polimorfismo. Uno de ellos se denomina “**Polimorfismo Subtipado**”. A continuación un ejemplo:

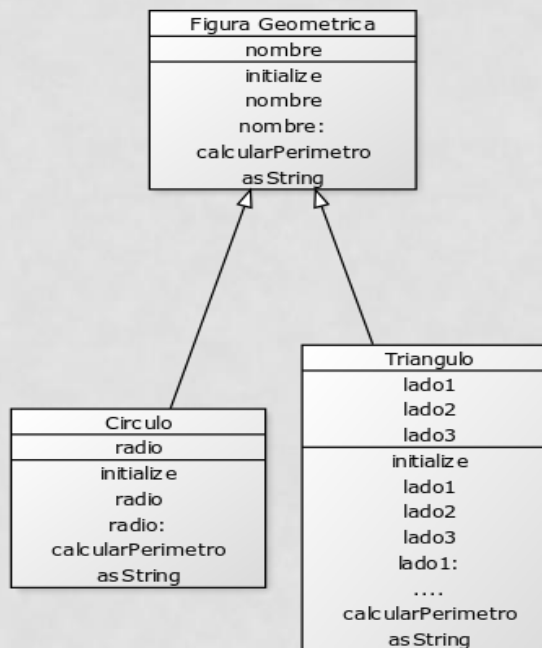


Tips para su implementación:

- Implementación de jerarquía de clases (Herencia).
- En la clase base se especifica/n los comportamientos comunes (polimórficos), estos pueden tener implementación o no.
- Los mensajes comunes se irán redefiniendo la/s clase/s derivadas, para aplicar el comportamiento correspondiente.
- En el ejemplo, el mensaje **asString** (polimórfico) puede ser respondido por cualquier instancia de la jerarquía.

POLIMORFISMO EN SMALLTALK

- **Caso de estudio:** Se necesita conocer el perímetro de dos figuras geométricas: Triangulo y Circulo. Las clases identificadas son: **Figura Geométrica**: tiene nombre; **Triángulo**: tiene lado1, lado2, lado3 y **Circulo**: tiene radio. A continuación se presenta el diagrama de clases correspondiente.

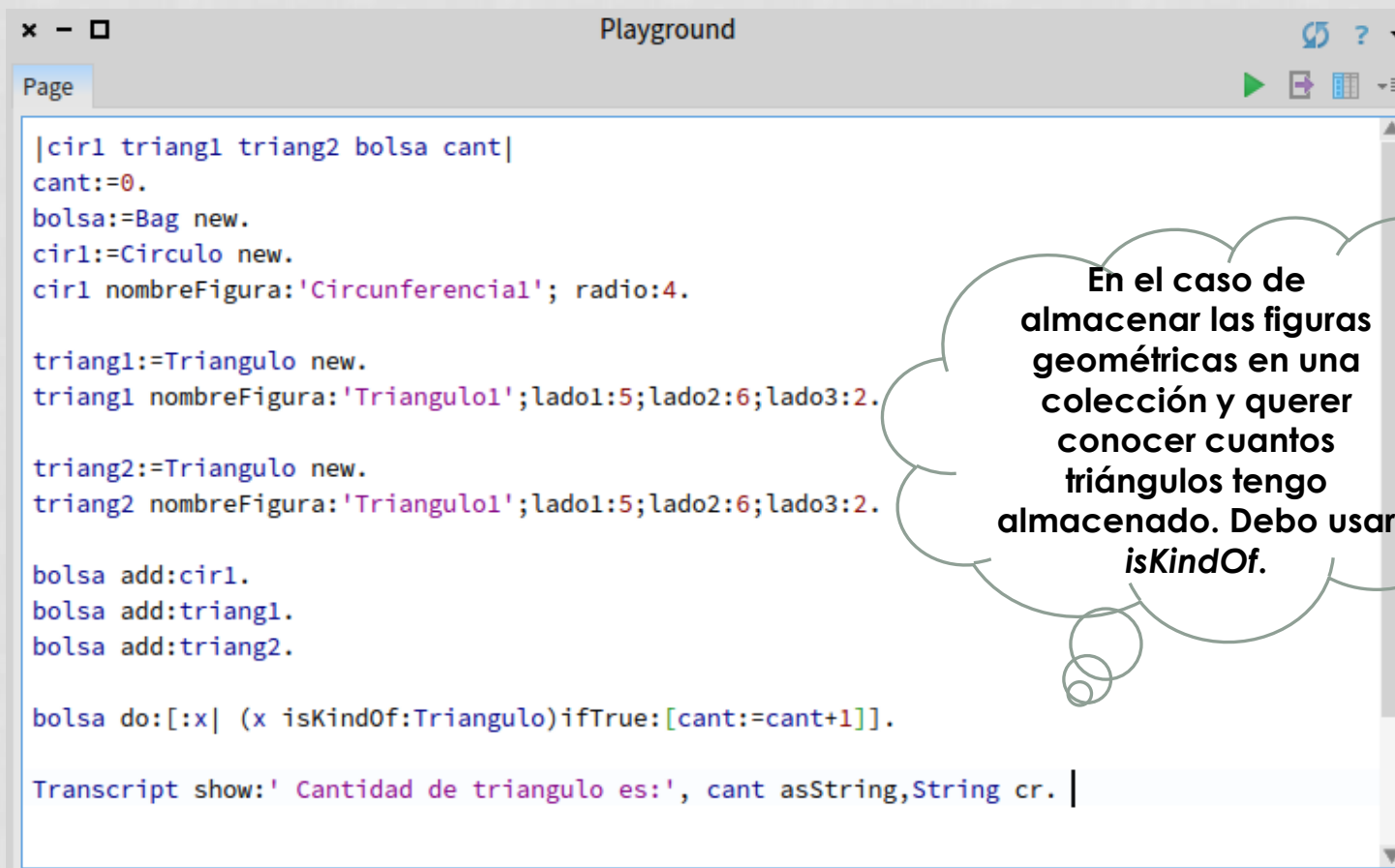


CREATED WITH YUML

La implementación completa disponible al finalizar la clase en UV.

POLIMORFISMO EN SMALLTALK

- Ejemplo Polimorfismo usando filtros:



```
|cir1 triang1 triang2 bolsa cant|
cant:=0.
bolsa:=Bag new.
cir1:=Circulo new.
cir1 nombreFigura:'Circunferencia1'; radio:4.

triang1:=Triangulo new.
triang1 nombreFigura:'Triangulo1';lado1:5;lado2:6;lado3:2.

triang2:=Triangulo new.
triang2 nombreFigura:'Triangulo1';lado1:5;lado2:6;lado3:2.

bolsa add:cir1.
bolsa add:triang1.
bolsa add:triang2.

bolsa do:[:x] (x isKindOf:Triangulo)ifTrue:[cant:=cant+1]].

Transcript show:' Cantidad de triangulo es:', cant asString,String cr. |
```

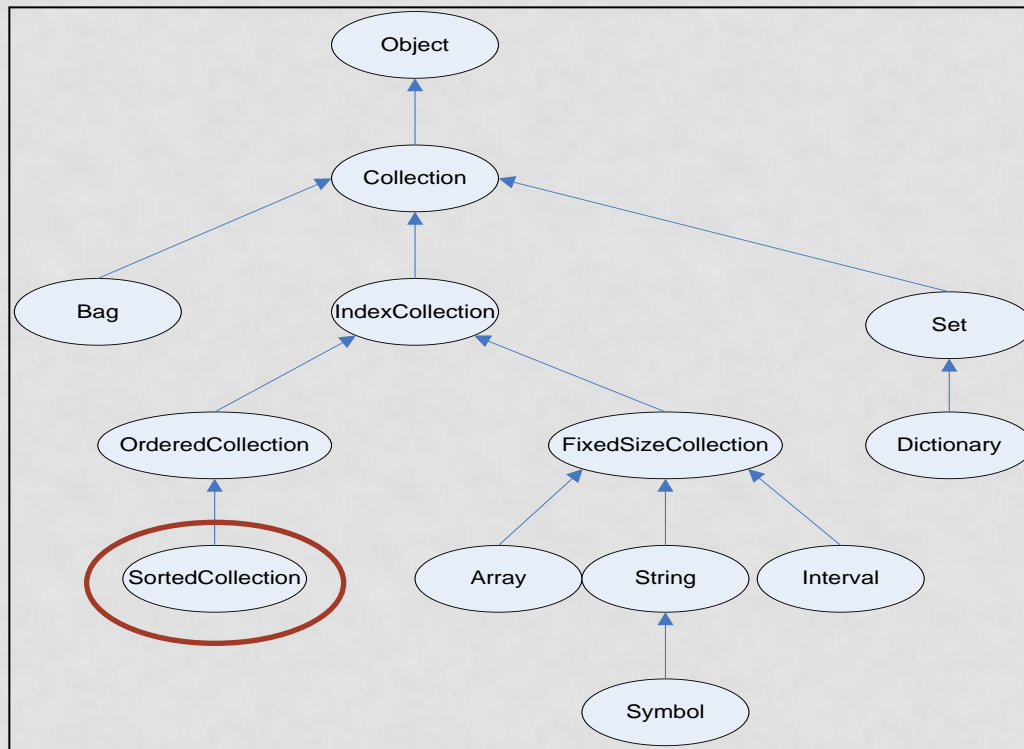
En el caso de almacenar las figuras geométricas en una colección y querer conocer cuantos triángulos tengo almacenado. Debo usar *isKindOf*.

BLOQUE 3: COLECCIONES EN SMALLTALK (CONTINUACIÓN)



COLECCIONES EN SMALLTALK

El siguiente cuadro muestra la jerarquía de colecciones disponibles en Smalltalk:



En esta jerarquía existen clases abstractas y concretas.

COLECCIÓN ORDENADA (SORTEDCOLLECTION)

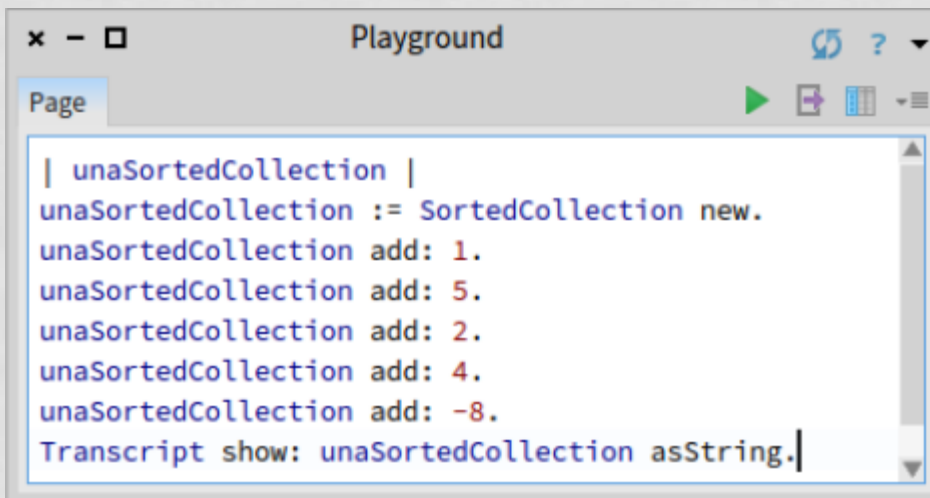
- Permite almacenar objetos teniendo en cuenta un criterio de orden específico. Por ejemplo: guardar alumnos “ordenados” por legajo.
- Esto es posible gracias a `sortBlock` .
- El bloque `sortBlock` es un bloque de dos argumentos que indica el orden en que los dos argumentos debieran ser almacenados en la colección respecto uno del otro.
- La colección guarda el primer argumento adelante del segundo argumento en la colección cuando la evaluación del bloque `sortBlock` es verdadera (`true`).
- El bloque `sortBlock` puede contener múltiples sentencias, pero debe retornar un valor de verdad: `true` o `false`.

COLECCIÓN ORDENADA (SORTEDCOLLECTION)

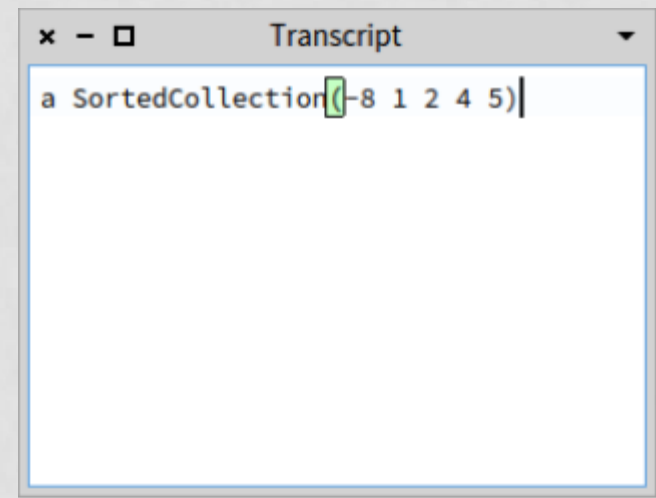
- Si se instancia una **SortedCollection** con la siguiente expresión:

unaSortedCollection := SortedCollection new.

- Se obtiene una colección que ordena sus elementos de **menor a mayor**. Implica que los objetos contenidos utiliza el criterio de ordenamiento por defecto. Ejemplo:



```
| unaSortedCollection |
unaSortedCollection := SortedCollection new.
unaSortedCollection add: 1.
unaSortedCollection add: 5.
unaSortedCollection add: 2.
unaSortedCollection add: 4.
unaSortedCollection add: -8.
Transcript show: unaSortedCollection asString.
```

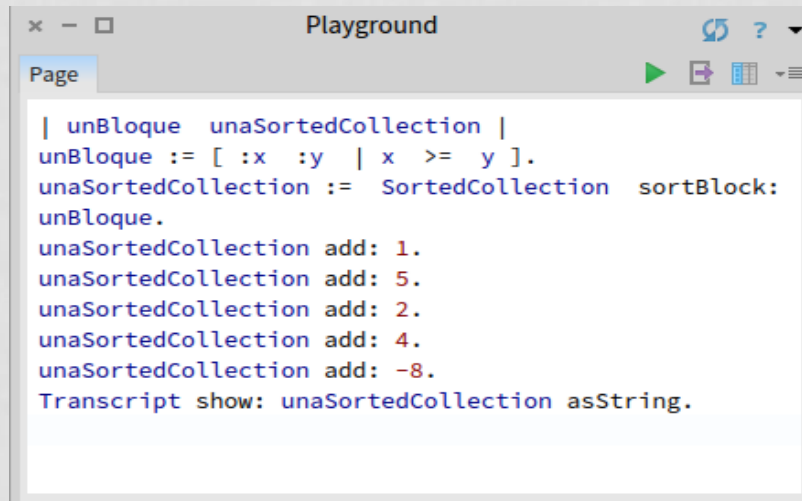


```
a SortedCollection(-8 1 2 4 5)|
```

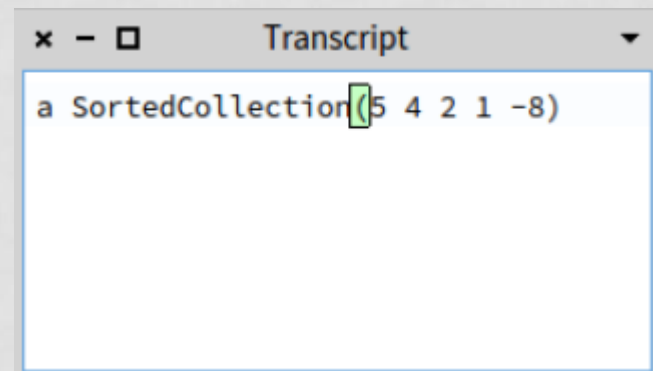
COLECCIÓN ORDENADA (SORTEDCOLLECTION)

- Para ordenar los elementos de la colección con un criterio en particular, se necesita el mensaje “**sortBlock**”, que es un bloque (objeto de la clase BlockClosure).
- Para especificar un ordenamiento diferente la colección debe instanciarse con la siguiente expresión:

unaSortedCollection := SortedCollection sortBlock: unBloqueBinario.



```
| unBloque unaSortedCollection |
unBloque := [ :x :y | x >= y ].
unaSortedCollection := SortedCollection sortBlock:
unBloque.
unaSortedCollection add: 1.
unaSortedCollection add: 5.
unaSortedCollection add: 2.
unaSortedCollection add: 4.
unaSortedCollection add: -8.
Transcript show: unaSortedCollection asString.
```



```
a SortedCollection(5 4 2 1 -8)
```

CONVERSIONES ENTRE COLECCIONES

- Todas las colecciones entienden una serie de mensajes que permiten obtener distintos tipos de colecciones con los mismos elementos que la colección receptora. Estos mensajes son de la forma “**as{ColeccionQueQuiero}**”.

Ejemplos:

Si tuviese una colección de la clase Bag y necesito sacarle los repetidos:

sinRepetidos := miCarrito asSet.

Si tuviese un Array, y lo quiero convertir en una colección de tamaño variable se podría hacer:

coleccionVariable := miVector asOrderedCollection.

CONVERSIONES ENTRE COLECCIONES

- Si quisiera ordenar mi carrito de compras del producto más caro al más barato, haría algo como:

ordenadosPorPrecio := miCarrito asSortedCollection:
[:unProd :otroProd | unProd precio > otroProd precio].

- También está el mensaje **asSortedCollection** sin parámetro que ordenará los elementos por el “orden natural”, dado por el mensaje <. En este caso, debemos tener en cuenta que todos los objetos que agreguemos a esta colección deberán entender el mensaje <, sino tendríamos un error de mensaje no entendido.

COLECCIONES. MENSAJES DE RECORRIDOS

- Smalltalk provee distintos mensajes para recorrer las diferentes colecciones que soporta. Algunos de ellos son:
- do:
- **detect:**
- select:
- collect:
- reject:
- inject:

COLECCIONES: MENSAJES RECORRIDOS

Mensaje

detect: unBloque

Evalúa un bloque para cada elemento de la colección. Retorna el primer elemento que haga que la condición evalúe **true**.

Sintaxis: retorno:=nombre_coleccion detect[:x|expresiones.].

```
Playground
Page
|sortColl pri|
sortColl:=SortedCollection new.
sortColl add:6.
sortColl add:10.
sortColl add:2.
sortColl add:3.
pri:=sortColl detect[:x|x odd]ifNone:[pri:='No encontrado'].

Transcript show:'El resultado de la búsqueda fue:', pri asString.
```

En el caso del ejemplo
retorna el objeto
retornado es 3 (primero
que cumple la
condición).

“TIPS PARA IDENTIFICAR CUANDO USAR DETECT”

- El mensaje **detect**: se utiliza mayormente cuando necesitamos buscar el “primer objeto” que cumpla una determinada condición o criterio.
- **Recordar**: El mensaje detect: “**tiene retorno**” y es el primer objeto que cumple con la condición especificada. En caso de que no exista ningún objeto que cumpla la condición, el detect: no funciona y genera un error en tiempo de ejecución. Debe usarse en conjunto con el mensaje ifNone:[].
Dentro del detect: solo se expresa la condición o criterio por el cual se efectúa la búsqueda.