Programación 4

Implementación

Colecciones

Contenido

- Introducción
- Colecciones de Objetos
  - Colecciones Genéricas
  - Realización de una Colección Genérica
- Iteradores
- Diccionarios
- Búsquedas
- Contenedores STL

Introducción

- La implementación de asociaciones usualmente requiere del uso de colecciones para permitir links con muchos objetos
- El tipo de los elementos de las colecciones depende de la clase correspondiente al extremo de asociación navegable
- Por tratarlas de manera uniforme éstas comparten una misma estructura que puede ser reutilizada para generarlas

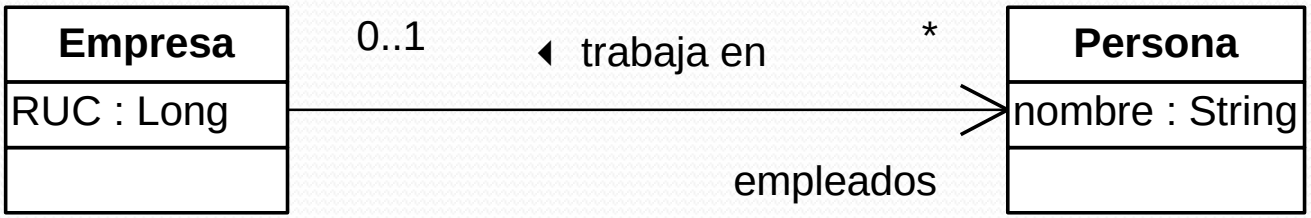
Introducción (2)

- Se distinguen dos tipos de colecciones dependiendo de si los elementos contenidos poseen una clave que los identifique o no
- La definición de las colecciones a utilizar en la implementación se estudiará incrementalmente
- Se comenzará definiendo una colección genérica de elementos sin clave la cual será aumentada para:
  - Permitir iteraciones sobre sus elementos
  - Soportar el uso de claves
  - Soportar diferentes tipos de búsquedas

Colecciones

Colecciones de Objetos

- Las colecciones de objetos son una herramienta fundamental para la implementación de muchas de las asociaciones presentes en un diseño



El pseudoatributo **`empleados`** de la clase **`Empresa`** introduce la necesidad de una colección de personas

Colecciones de Objetos (2)

- Las colecciones deben permitir:
  - Realizar iteraciones sobre sus elementos
  - Realizar búsquedas de elementos por clave (en caso de que los elementos tengan una)
  - Realizar búsquedas diversas

Colecciones de Objetos (3)

- Desarrollar cada colección en forma íntegra cada vez que se necesita resulta poco práctico
- Es posible definir una única vez una infraestructura común que sirva de base para todas las colecciones específicas:
  - **Colecciones paramétricas** (templates): el tipo del elemento a almacenar es declarado como parámetro que será instanciado al generar la colección particular
  - **Colecciones genéricas**: pueden almacenar directamente cualquier tipo de elemento

Colecciones Genéricas

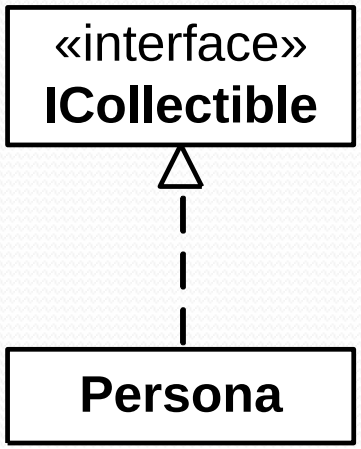
- Una colección genérica está definida de forma tal que pueda contener a cualquier tipo de elemento
- Aspectos a considerar:
  - ¿Cómo lograr que un elemento de una clase cualquiera pueda ser almacenado en la colección genérica?
  - ¿Cómo se define la colección genérica?

Genericidad de la Colección

- ¿Cómo lograr que un elemento de una clase cualquiera pueda ser almacenado en la colección genérica?
- Se define la interfaz de marca **`ICollectible`**
- Cuando se desea que los elementos de una cierta clase puedan ser almacenados en una colección genérica se solicita que dicha clase realice la interfaz **`ICollectible`**
- De esta forma la colección genérica contendrá elementos “coleccionables” (es decir, que implementarán la interfaz **`ICollectible`**)

Genericidad de la Colección (2)

- Ejemplo:
  - La clase `Persona` debe realizar la interfaz de marca `ICollectible` para poder agregar personas a una colección genérica



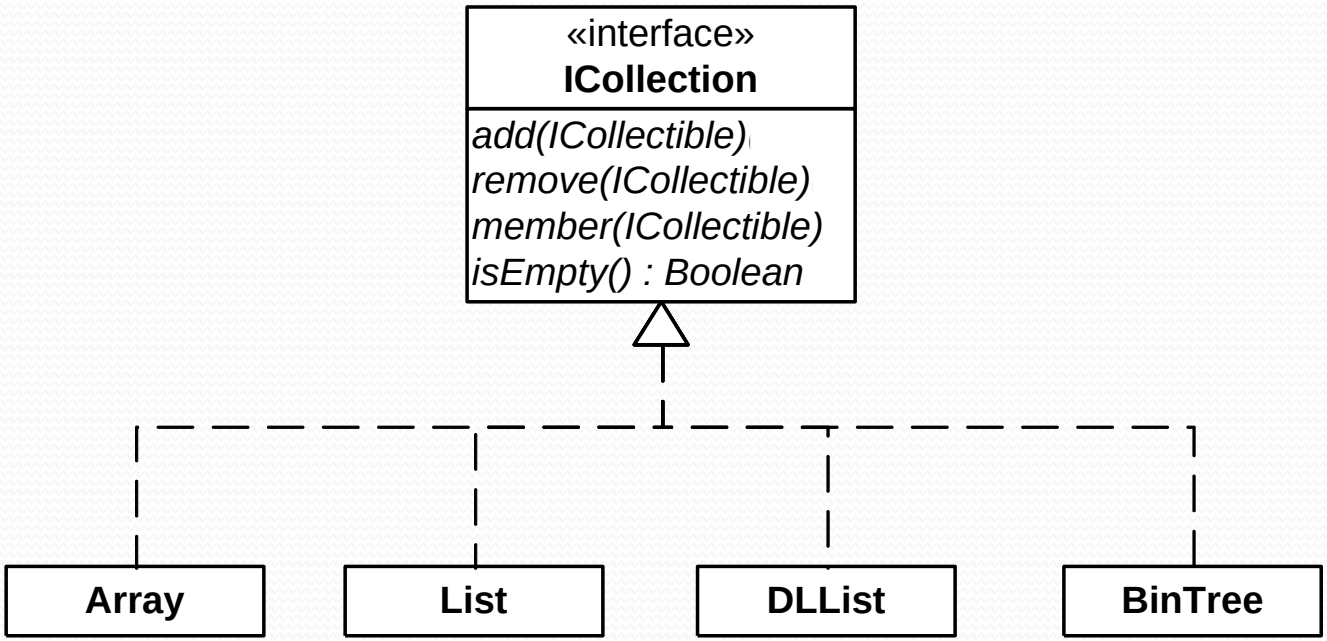
Una interfaz de marca no posee ninguna operación, por lo que no obliga a las clases que la implementan a presentar ningún servicio.

Encapsulamiento

- ¿Cómo se define una colección genérica?
- La noción de colección es independiente de su implementación
- Se separa la especificación de la implementación:
  - Se define una interfaz `ICollection`
  - Una cierta colección genérica será una implementación que realice esta interfaz

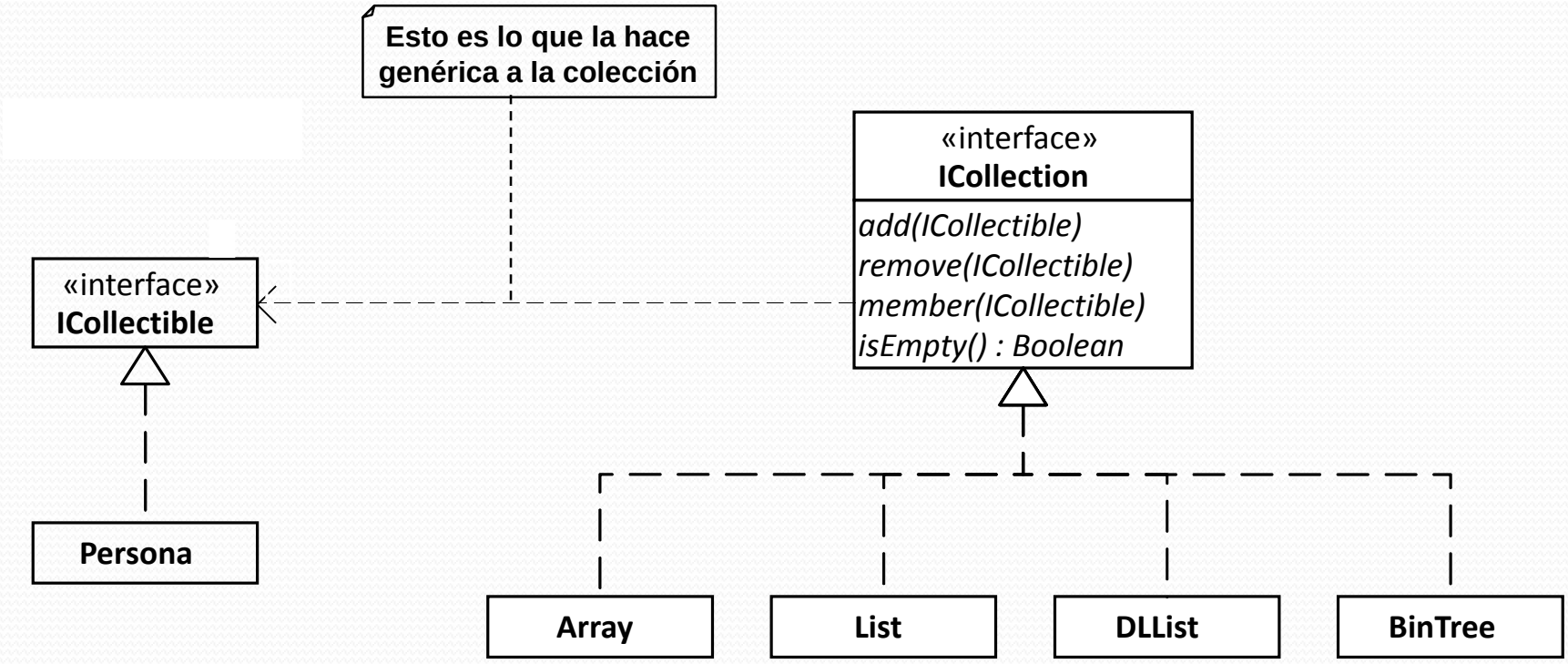
Realizaciones

- Podrán existir diferentes realizaciones (cada una con su estructura de datos particular) de la colección genérica



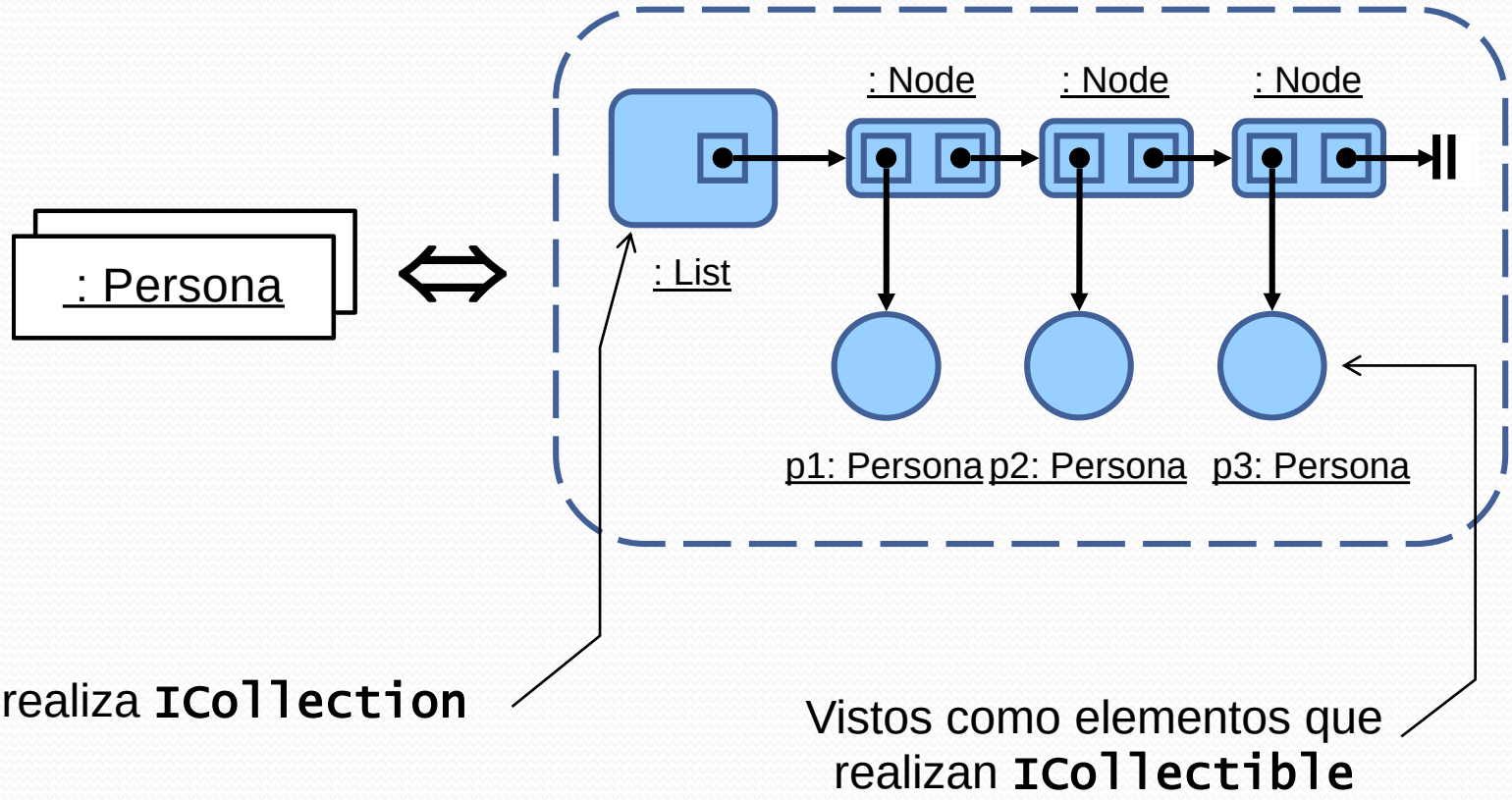
Colecciones Genéricas

Realizaciones (2)



Realizaciones (3)

- Estructura de una colección genérica:

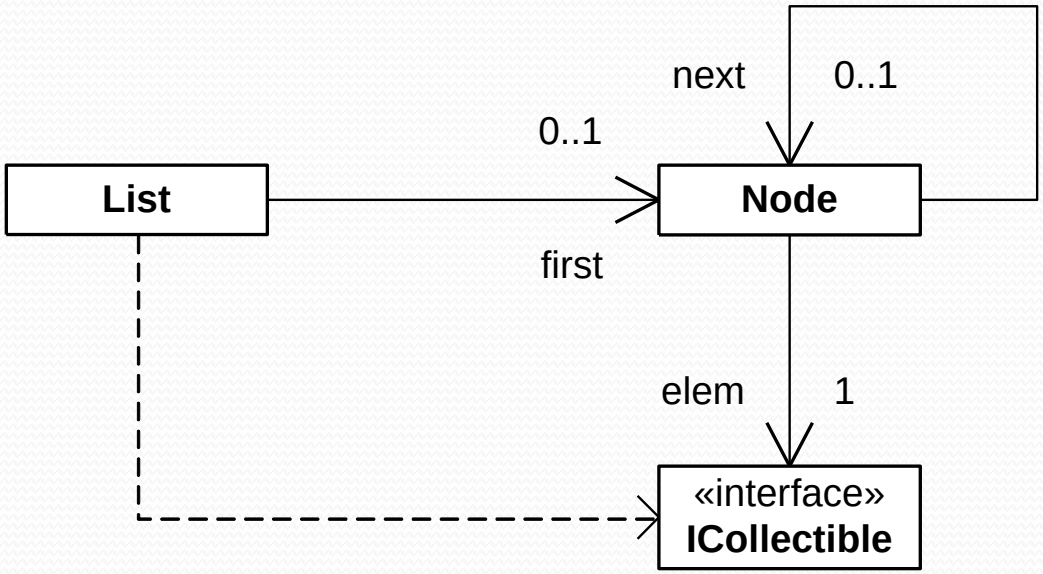


Realización de una Col. Genérica

- La interfaz **`ICollection`** declara qué servicios debe proveer una colección
- Es posible realizar dicha interfaz de diferentes maneras mediante diferentes estructuras de datos
- Realizaciones posibles:
  - Array o Vector
  - Lista común o doblemente enlazada
  - Arbol binario
  - Etc.

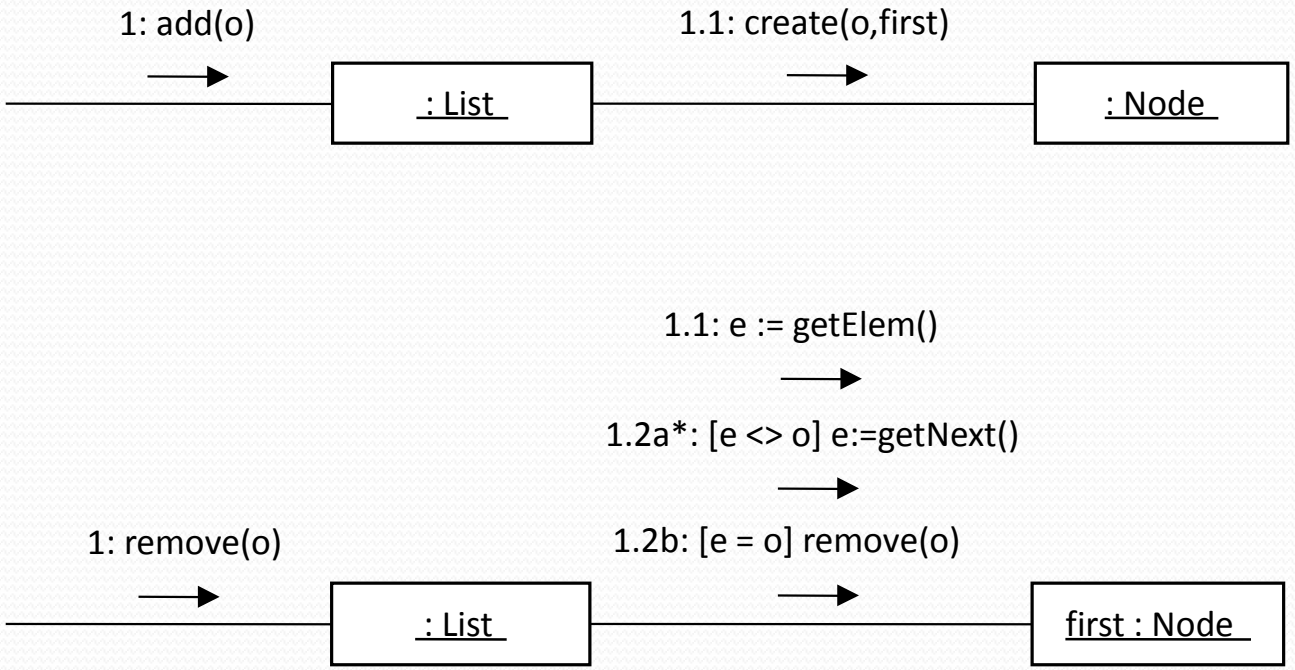
Común

- El diseño de una lista común utilizando clases no difiere significativamente del diseño usual



Lista Común (2)

- Las operaciones no se resuelven en forma completa en la clase **`List`**



Iteradores

Iteradores

- Es muy común necesitar realizar iteraciones sobre los elementos de una colección
- La interfaz **`ICollection`** es aumentada con la siguiente operación:

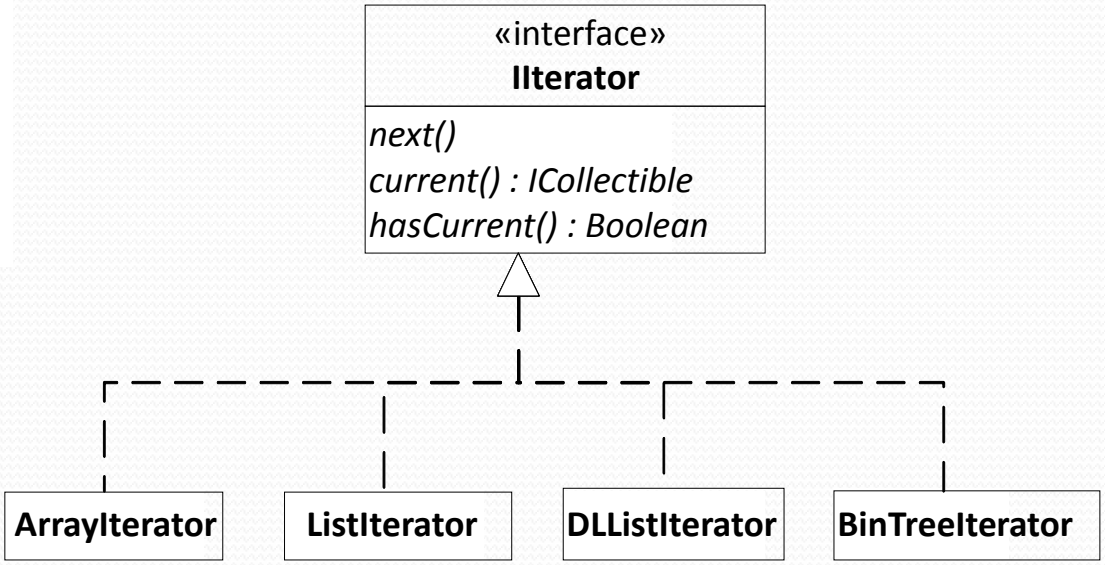
```
getIterator():IIterator
```

Iteradores (2)

- Un iterador (sugerido en el patrón de diseño Iterator) es un objeto que permite recorrer uno a uno los elementos de una colección
- Un iterador es un observador externo de la colección (no en el sentido del patrón Observer)
- Una colección puede tener diferentes iteradores realizando diferentes iteraciones simultáneamente

Iteradores

Estructura



Uso de Iteradores

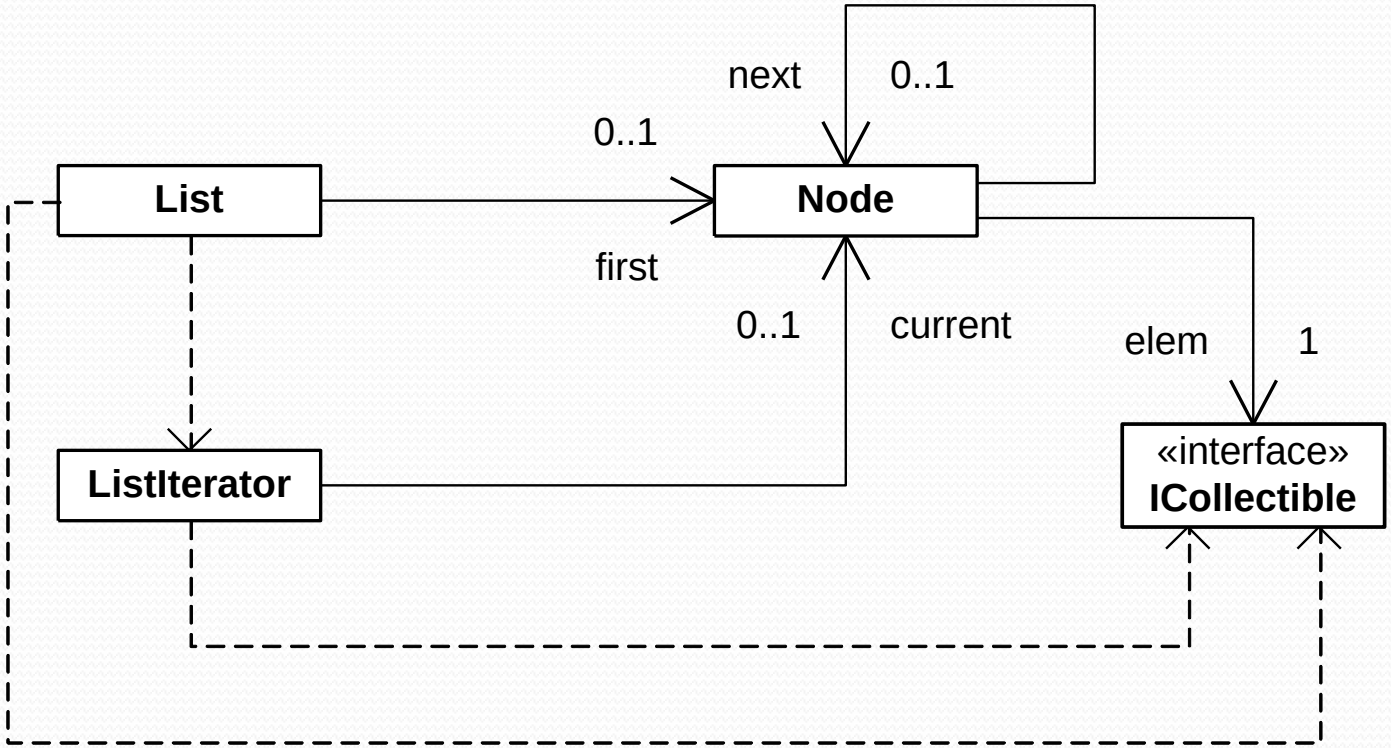
```
class Venta {
private:
  ICollection *lineas;
public:
  float totalVenta();
  virtual ~Venta();
};

float Venta::totalVenta() {
  float total = 0;

  for(IIterator *it = lineas->getIterator(); it->hasCurrent(); it->next()) {
   LineaDeVenta *ldv = dynamic_cast<LineaDeVenta *>(it->getCurrent());
   total += ldv->subtotal();
  }
  return total;
}
```

Realización de Iteradores

- Como ejemplo de realización de iteradores se presenta el diseño de un iterador sobre listas comunes
- La clase **`ListIterator`** realiza la interfaz **`IIterator`**

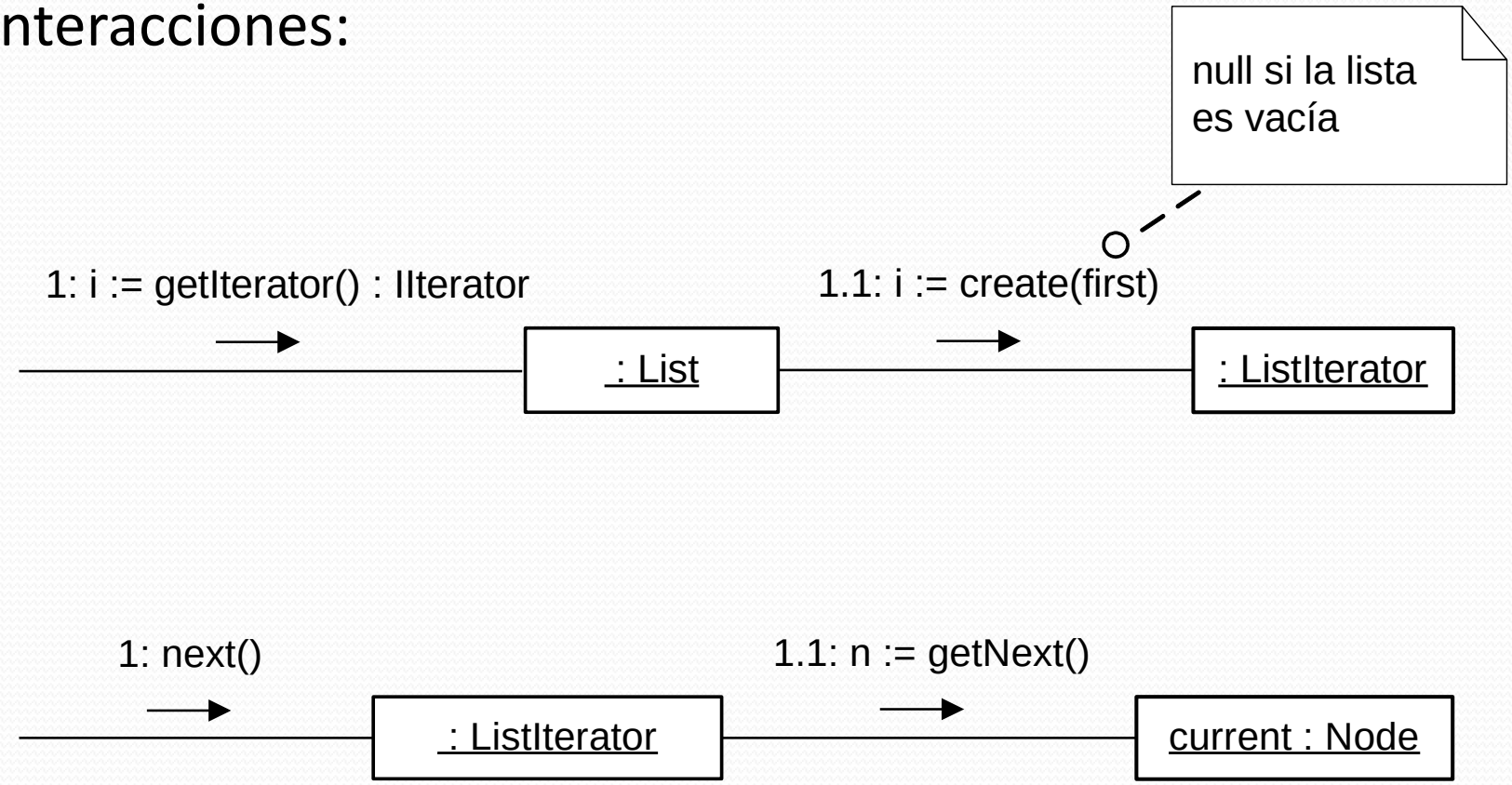
Realización de Iteradores (2)

- Estructura:



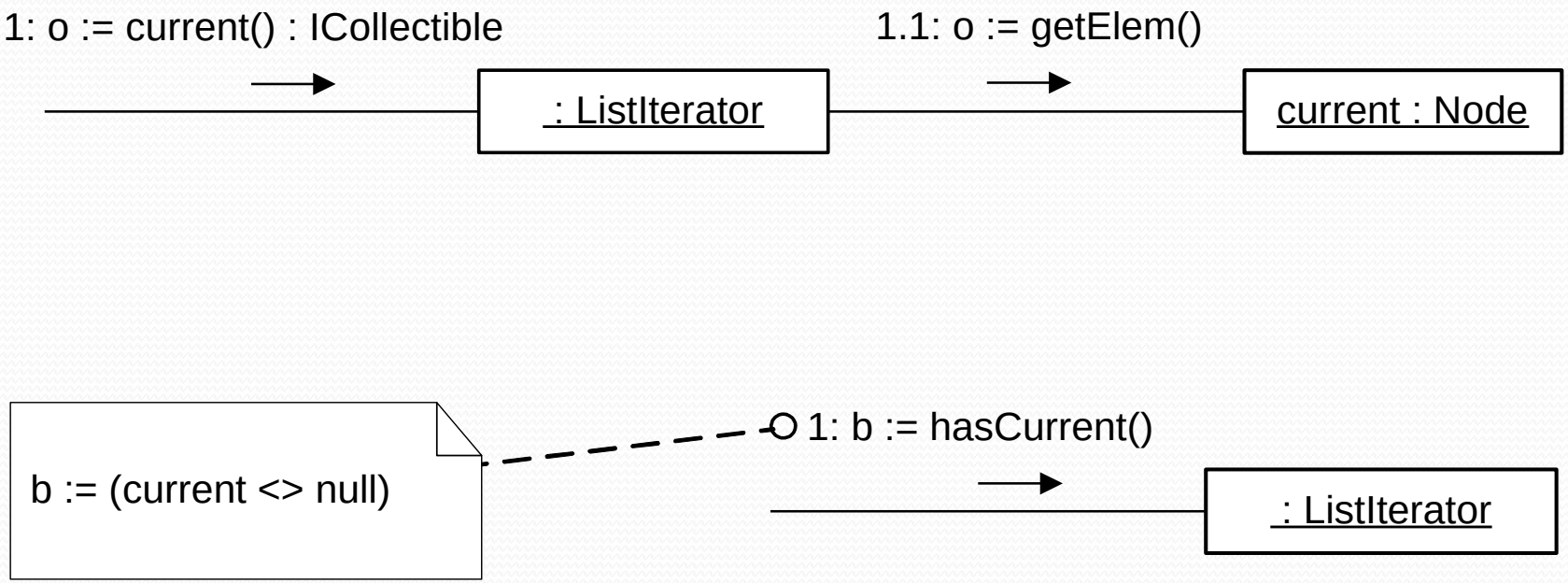
Realización de Iteradores (3)

- Interacciones:



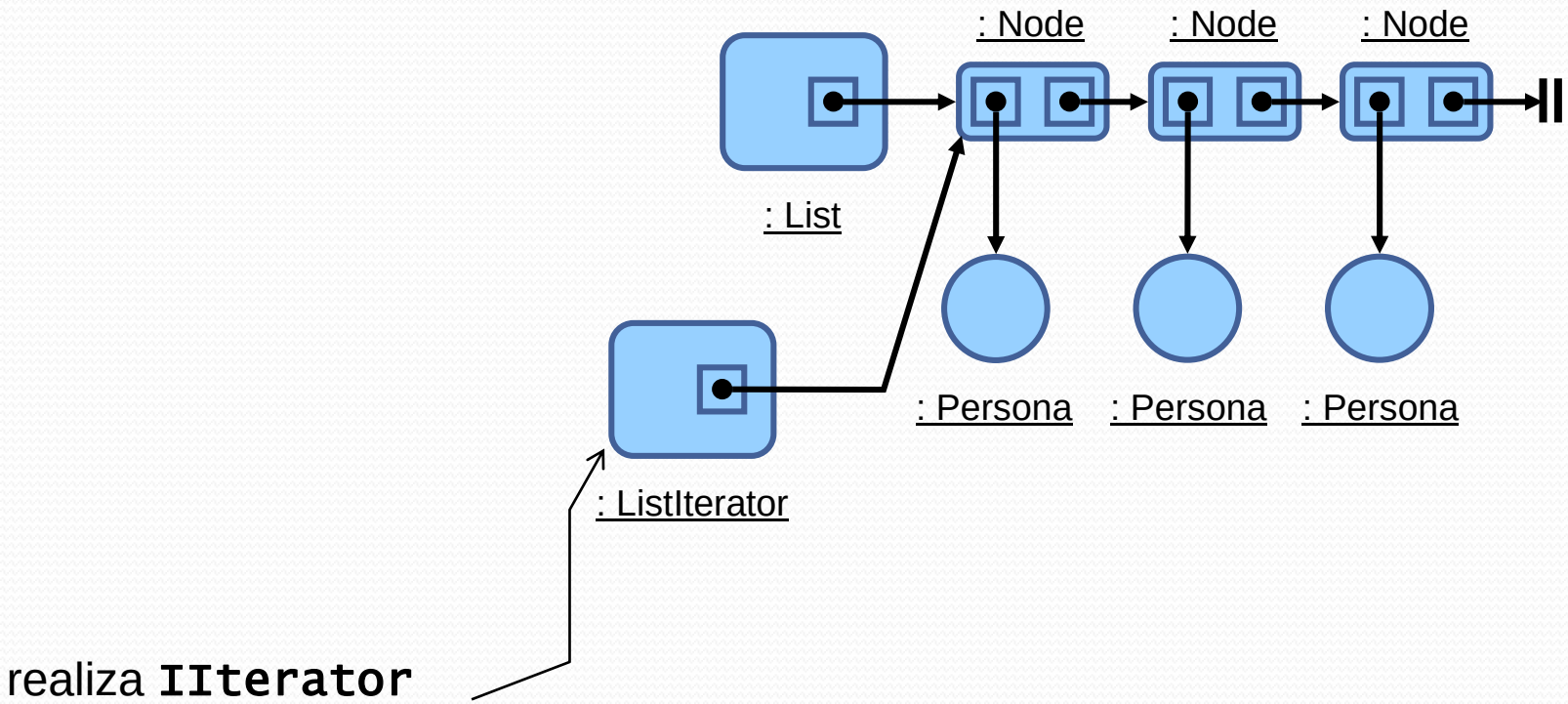
Realización de Iteradores (4)

- Interacciones (cont.):



Realización de Iteradores (5)

- Ejemplo:



Diccionarios

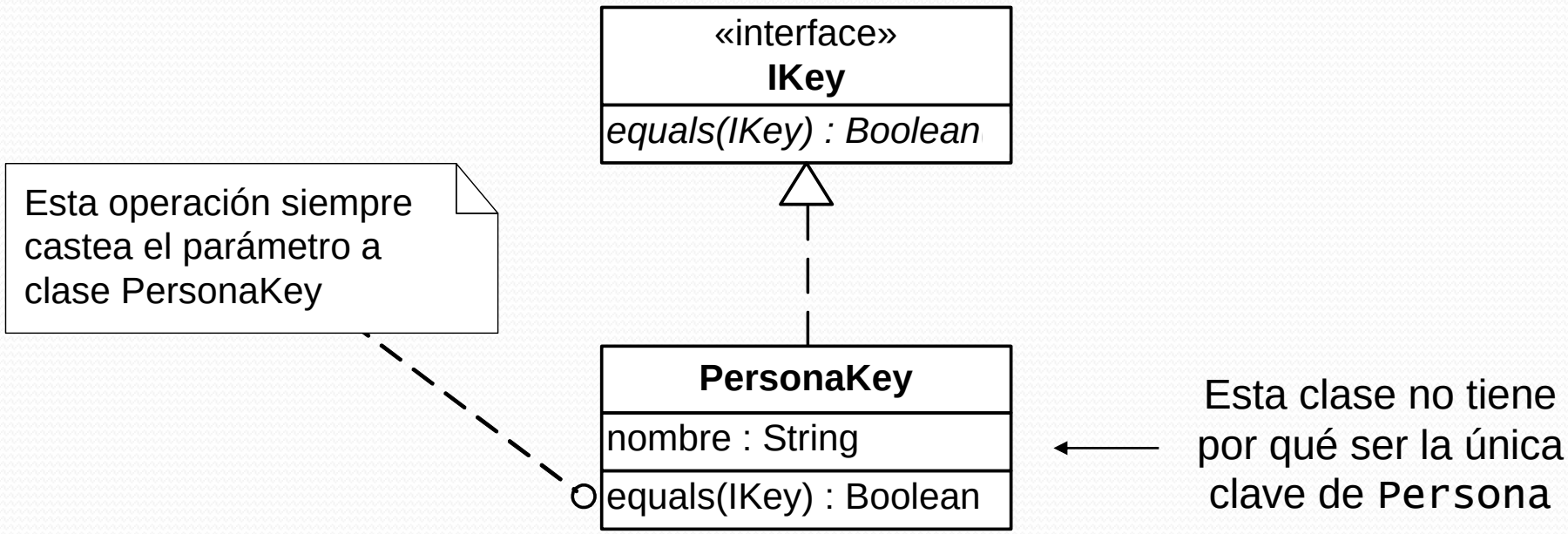
Diccionarios

- Un diccionario es un tipo particular de colección en el cual se almacenan objetos que pueden ser identificados por una clave
- Se define la interfaz **`IDictionary`** y se utiliza en forma análoga a la interfaz **`ICollection:`**
  - Existirán diferentes realizaciones de **`IDictionary`**
  - Las mismas contendrán elementos que realicen la interfaz **`ICollectible`**

Uso de Claves

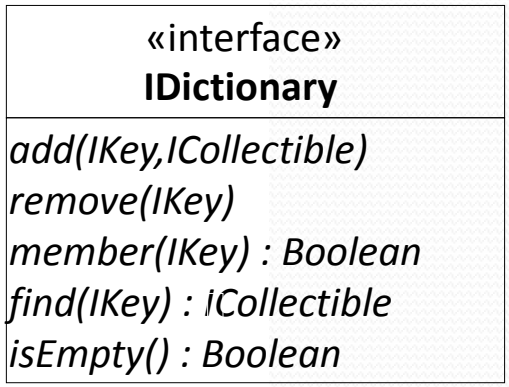
- Se está tratando la noción de diccionario genérico por lo que la clave que identifica a los elementos debe ser también genérica
- Se define la interfaz **`IKey:`**
  - Debe ser realizada por una clase que representa la clave de los elementos a incluir en el diccionario
  - Contiene únicamente la operación **`equals(IKey):boolean`** utilizada para comparar claves concretas

Uso de Claves (2)



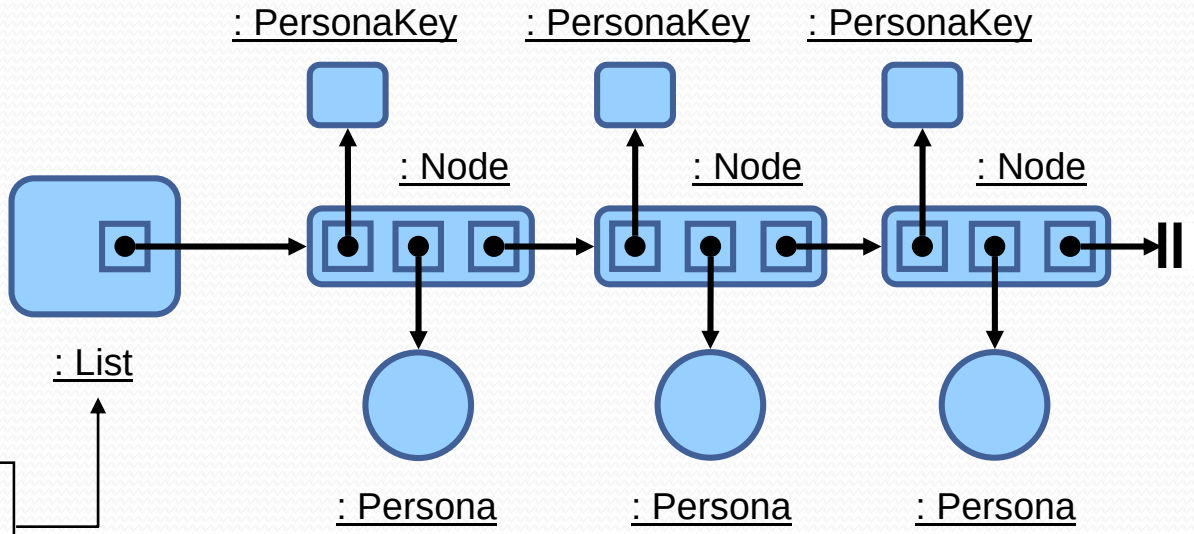
Los atributos de la clave concreta son una combinación de atributos de la clase

Diccionarios Genéricos



Diccionarios Genéricos (2)

- Ejemplo:



`List` realiza `IDictionary` y ya no `ICollection`

La relación entre una persona y su clave es particular a cada diccionario. La clave que le corresponde a una persona es la referenciada por el nodo que referencia a la persona (determinada al momento del **`add()`**).

Iteraciones en Diccionarios

- Un diccionario es una colección por lo que tiene sentido necesitar iterar sobre sus elementos
- Se incorpora a la interfaz **`IDictionary:`**
  - **`getElemIterator():IIterator`** que devuelve un iterador sobre los elementos contenidos en el diccionario
  - **`getKeyIterator():IIterator`** que devuelve un iterador sobre las claves de los elementos contenidos en el diccionario

Búsquedas

Búsquedas

- Las búsquedas por clave no son el único tipo de búsqueda que se suele requerir
- Existe otro tipo de búsquedas que no involucran necesariamente una clave:
  - Buscar todos los empleados menores de una cierta edad
  - Buscar todos los empleados contratados antes de una fecha dada

Búsquedas (2)

- Dado que este tipo de búsquedas dependen de cada colección no se implementan en las colecciones genéricas
- De esta forma se define una operación por cada búsqueda necesaria:
  - Por ejemplo para buscar los empleados contratados antes de una fecha:

```
selectContratadosAntes(Fecha):ICollection
```

Búsquedas (3)

```
List * ClaseA::selectContratadosAntes(Fecha f) {
   List * result = new List();
   ListIterator * it = coleccion->getIterator();

   while(it.hasCurrent()) {
      if(((Persona*)it.current())->getFecha()<f)
         result->add(it.current());
      it.next();
   }
   return result;
}
```

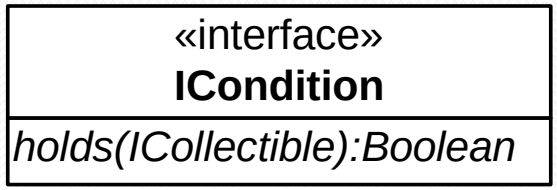
Notar que todas las variantes de **`select()`** de **`ColPersona`** serán exactamente iguales entre sí a menos de esta porción del código

Búsquedas (4)

- Las operaciones de búsqueda son muy similares entre sí
- Las búsquedas en todas las colecciones son similares salvo:
  - La condición que determina la inclusión de un elemento en el resultado
  - Los parámetros
  - El tipo del iterador y la colección resultado

Búsquedas (5)

- Sería posible incorporar a las interfaces **`ICollection`** e **`IDictionary`** respectivamente las operaciones:
  - **`select(ICondition) : ICollection`**
  - **`select(ICondition) : IDictionary`**
- La interfaz **`ICondition`** se define como



en cada realización **`holds()`** indicará si un cierto objeto debe formar parte del resultado del **`select()`**

Búsquedas (6)

- ¿Cómo manejar las diferencias mencionadas entre las diferentes implementaciones?
  - El tipo del iterador sería **`IIterator`**
  - El tipo del resultado sería **`ICollection`** o **`IDictionary`** respectivamente
  - La condición encapsula:
    - El o los parámetros de la búsqueda (en sus atributos)
    - El algoritmo que determina si un elemento de la colección debe pertenecer además al resultado (en el método asociado a **`holds()`**)

Búsquedas (7)

- Una posible implementación de **`select()`** en una realización de **`ICollection`** sería:

```
ICollection * List::select(ICondition * cond) {
   ICollection * result = new List();
   IIterator * it = coleccion->getIterator();

   while(it->hasCurrent()) {
      if(cond->holds(it->current()))
         result->add(it->current());
      it->next();
   }
   return result;
}
```

Búsquedas (8)

- De esta forma las clases que implementan **`ICondition`** son estrategias concretas que el **`select()`** utiliza para construir la colección resultado
- En esta aplicación de Strategy se dan las siguientes correspondencias:
  - **`List`** → Context
  - **`ICondition`** → Estrategia
  - **`select()`** → solicitud()
  - **`holds()`** → algoritmo()

Búsquedas (9)

- Ejemplo de condición concreta:

```
class Persona : ICollectible {
private:
        String nombre;
        int edad;
public:
        Persona();
        String getNombre();
        int getEdad();
}
```

```
// CondEdad.h
class CondEdad : ICondition {
private:
        int valorEdad;
public:
        CondEdad(int i);
        bool holds(ICollectible *ic);
}
// CondEdad.cpp
CondEdad::CondEdad(int i) {
         valorEdad = i;
}
bool CondEdad::holds(ICollectible *ic) {
         Persona * p = (Persona *)ic;
         return (p->getEdad() == valorEdad);
}
```

Contenedores STL

Introducción

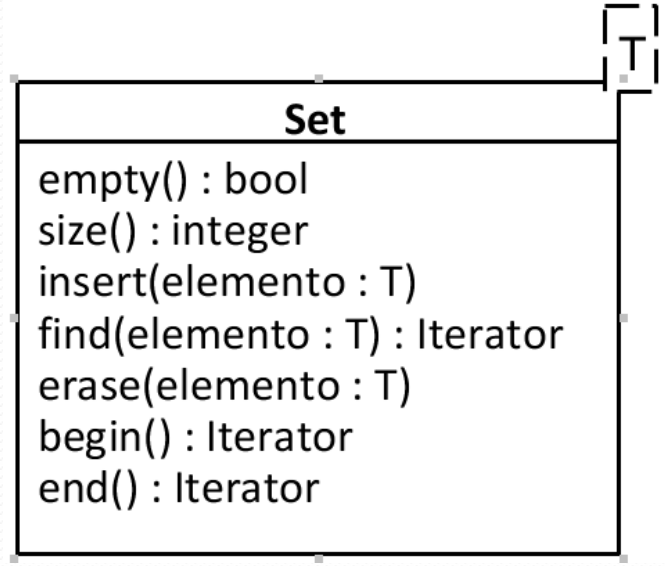
- Usualmente los lenguajes de programación ofrecen bibliotecas para el manejo de colecciones
- C++ ofrece la Standard Template Library, un conjunto de contenedores, iteradores y otras utilidades para manejar conjuntos de objetos.
- Es una alternativa a implementar colecciones genéricas (no se reinventa la rueda)

Introducción (2)

- La STL provee *contenedores*
- Los contenedores son clases cuyo propósito son almacenar otros objetos
- Algunos contenedores
  - `set<T>`: Una colección de objetos. Permite opcionalmente ordenamiento
  - `map<K,V>`:Un Diccionario de claves de tipo K y valores de tipo V
  - Otros: `vector`, `queue`, `list`, `stack`
  - Todos los contendedores son clases paramétricas

Set

- Permite el almacenamiento de una colección de elementos
- Permite ordenamiento (por defecto se usa el operador <)
- Permite iterar sobre los elementos



Ejemplo

```
set<int> col;
set<int>::iterator it;     // iterador para set<int>

for (int i=1; i<=5; i++)   // inicializa
  col.insert(i*10);        // col: 10 20 30 40 50

it = col.find(20);         // busca un elemento
col.erase (it);            // borra 20
col.erase (col.find(40)); // borra 40

cout << "col contiene:";
for (it=col.begin(); it!=col.end(); ++it){
 int current = *it;
 cout << ' ' << current;
}
```

Iteradores

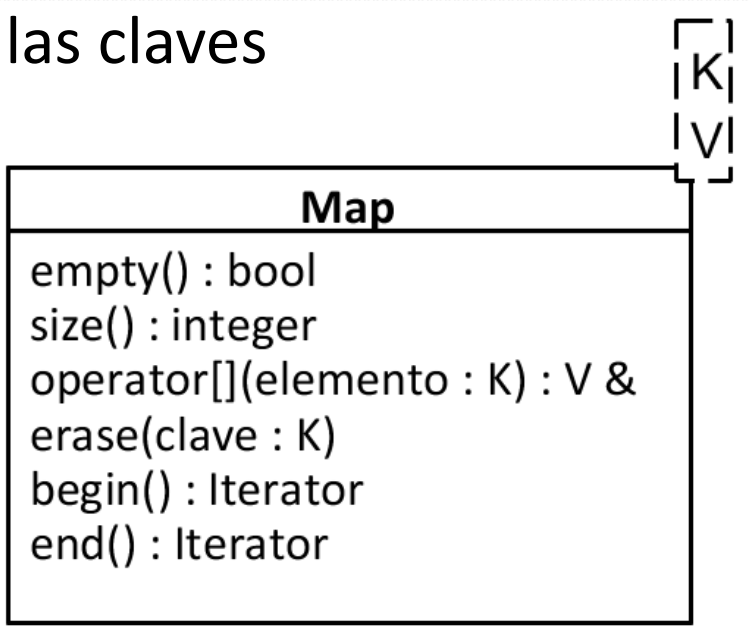
- La operación `begin()` devuelve un iterador al principio de la colección
- `end()` devuelve un iterador un lugar después del final de la colección. Se utiliza
  - Para marcar el final en las iteraciones for()
  - En la operación `find(e)`, para indicar que no encuentra el elemento e
- El tipo del iterador es `set<T>::iterator`
  - `*it` devuelve el elemento actual (de tipo T)
  - `++it` avanza un lugar en la iteración

Iteradores (2)

```
set<Empleado *> col;
Empleado *e1 = new Empleado("pato");
Empleado *e2 = new Empleado("poto");
Empleado *e3 = new Empleado("carlitos");
Empleado *f  = new Empleado("peto");

col.insert(e1);
col.insert(e2);
col.insert(e3);

std::cout << "col contiene";
for (set<string>::iterator it=col.begin(); it!=col.end(); ++it){
  Empleado *current = *it;
  cout << ' ' << current->getNombre();
}
// imprime 'carlitos pato poto' (usa el orden <)

col.find(e1) // devuelve un iterador situado en el elemento pato
col.find(f)  // devuelve col.end()
```

Map<K,V>

- Brinda la misma funcionalidad que IDictionary
- Permite búsqueda por clave e iteraciones.
- Opcionalmente se puede cambiar la forma de comparación de las claves

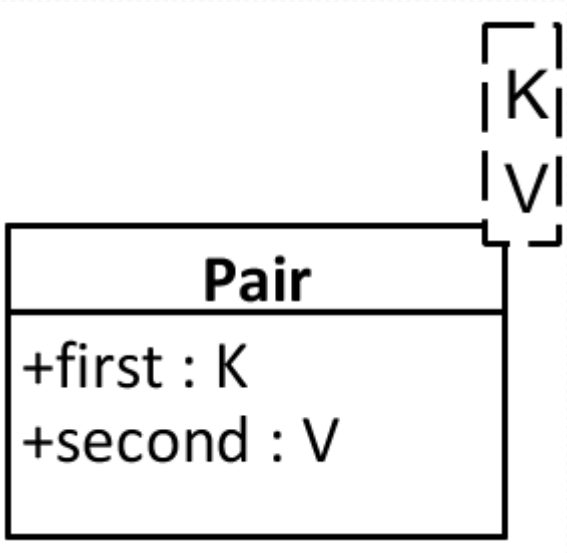


Map<K,V> (2)

```
map<string, Usuario *> col;
// agrega elementos
Usuario *u1 = new Usuario("Gustavo Cerati");
Usuario *u2 = new Usuario("León Gieco");

col["cerati"] = u1;
col["gieco"]  = u2;

col["gieco"];          // devuelve u2
col["sumo"];           // devuelve NULL

col.erase("gieco");    // borra elementos
col.erase("cerati");
col.empty();           // devuelve true
```

Iteración en map

- El iterador de map<K,V> itera de manera ordenada en las claves y cada elemento es de tipo Pair<K,V>
- Los elementos se iteran por el orden de la clave K de menor a mayor



Iteración en map(2)

```
map<char,int> dict;
map<char,int>::iterator it;

dict['b'] = 100;
dict['a'] = 200;
dict['c'] = 300;

for (it= dict.begin(); it!=dict.end(); ++it){
  char clave = it->fist;
  int  valor = it->second;
  cout << clave << " => " << valor << '\n';
}
```

Imprime

```
a => 200
b => 100
c => 300
```