TEMA 1

Estructuras de Datos y Abstracción

Tema 1

Métodos de abstracción

Clases de abstracción

- Abstracción funcional
- Abstracción de datos
- Abstracción iterativa
- Jerarquía de objetos

The Java Collections Framework

Tema 1

- Disponible en el *campus virtual*
  - Apuntes
    - Con varios ejemplos completos de los conceptos que se van a abordar
    - Anexos complementarios
      - *Anexo I.* Manejo de errores utilizando excepciones (de los tutoriales online de Java)
      - *Anexo II.* Análisis de algoritmos (se ve en la asignatura de Algoritmia)
  - Diapositivas
  - Glosario de conceptos

Abstracción (1)

- Definición (RAE)
  - Acción o efecto de abstraer o abstraerse
    - Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción. *Olvidar diferencias, generalizar*
    - Hacer caso omiso de algo, o dejarlo a un lado. *Centrar la atención en lo esencial*

Centrar la atención en lo esencial, dejando al margen las diferencias

Generalizar

Abstracción (2)

- Métodos de abstracción en informática
  - Abstracción por especificación
  - Abstracción mediante uso de parámetros

Especificación de una función

Uso de parámetros (*generalizamos*)

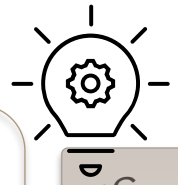
```
/**
 * Retorna el mayor de los dos enteros dados.
 *
 * @param a el primer entero
 * @param b el segundo entero
 * @return el mayor de {@code a} y {@code b}
 */
public static int maximo(final int a, final int b) {
    // Nos olvidamos de la implementación
}
```

Abstracción (3)

- La misma función en Python
  - Al no ser un lenguaje tipado, los parámetros *a* y *b* pueden ser cualquiera: *enteros*, *reales*, *caracteres*, *cadenas de caracteres*, e incluso, *listas*.
  - La función es más abstracta que la proporcionada para Java (más general)

```
def maximo(a, b):
    """Retorna el máximo de a y b"""
    if a >= b:
        max = a
    else:
        max = b

    return max
```



¿Se puede hacer algo similar en un lenguaje tipado?

Abstracción (4)

- Opción 1. Para LPOOs
  - Uso del *polimorfismo de inclusión* (herencia)

```
/**
 * Retorna el mayor de los dos objetos dados.
 * @param a el primer objeto
 * @param b el segundo objeto
 * @return el mayor de {@code a} y {@code b}
 * @throws ClassCastException si los objetos no son
 * comparables
 */
public static Object maximo(final Object a, final Object b) {
    return a >= b ? a : b;
}
```

Da error porque el operador `>=` no está definido para el tipo *Object*

Abstracción (5)

- En Java las clases de los objetos que se pueden comparar implementan la interfaz *Comparable*

```
public interface Comparable {
    /**
     * Compara este objeto con el especificado.
     *
     * @param o el objeto dado
     * @return  {@code >0}, {@code 0} o {@code <0}, según sea
     * este objeto mayor, igual o menor que {@code o}
     */
    int compare(Object o);
}
```

Abstracción (6)

```
/**
 * Retorna el mayor de los dos objetos dados.
 * @param a el primer objeto
 * @param b el segundo objeto
 * @return el mayor de {@code a} y {@code b}
 * @throws ClassCastException si los objetos no son
 * comparables
 */
public static Object maximo(final Object a, final Object b) {
    return a.compare(b) >= 0 ? a : b;  // solución aproximada
}
```

- Otros LPOOs proporcionan soluciones más o menos similares
- La solución no es válida si el LPOO carece de una clase base por defecto, por ejemplo, en C++ (C++ carece del equivalente de *Object*)
- No es aplicable a lenguajes de programación no orientados a objetos

Abstracción (7)

- Opción 2. Uso de *parámetros de tipo*
  - Es otra forma de *polimorfismo*, más general que el polimorfismo de inclusión, que se puede utilizar tanto en LPOOs como en lenguajes que no lo son.

Polimorfismo paramétrico

> Se utilizan ***parámetros de tipo*** para designar un tipo cualquiera (un tipo genérico)

  - Varios LPOOs, entre ellos Java y C++, soportan ambos tipos de polimorfismo:
    - *Polimorfismo de inclusión*
    - *Polimorfismo paramétrico*

Abstracción (8)

```
/**
 * Retorna el mayor de los dos objetos dados.
 * @param a el primer objeto
 * @param b el segundo objeto
 * @param <T> el tipo de los objetos dados
 * @return el mayor de {@code a} y {@code b}
 * @throws ClassCastException si el tipo T no es comparable
 * (no implementa la interfaz Comparable)
 */
public static <T> T maximo(final T a, final T b) {
    return compare(a, b) >= 0 ? a : b;
}
```

- T es un parámetro, al igual que a y b, pero a diferencia de éstos se reemplaza por un tipo de dato (*Integer*, *Character*, *String*, etc.), el tipo de los argumentos de la llamada

Abstracción (9)

- Clases de abstracción
  - *Abstracción funcional:* permite extender un lenguaje de programación añadiendo a éste nuevas operaciones
  - *Abstracción de datos:* permite extender un lenguaje de programación añadiendo a éste nuevos tipos de datos
  - *Abstracción iterativa:* permite iterar sobre los elementos de una colección ignorando el detalle de cómo se obtienen éstos
  - *Jerarquía de tipos:* permite abstraerse de tipos de datos individuales a conjuntos de tipos relacionados

Los métodos de abstracción: *especificación* y *uso de parámetros*, son aplicables a las tres primeras clases de abstracciones

Abstracción funcional (1)

- Operaciones
  - Extienden un lenguaje de programación añadiendo a éste nuevas operaciones
  - Uso de los métodos de abstracción
    - La *especificación* de la función, que nosotros haremos mediante cláusulas *JavaDoc*, establece lo que ésta hace. Permite invocar la función sin necesidad de conocer su definición.
    - La función puede utilizar parámetros que se sustituyen por los argumentos de la invocación, pero también puede utilizar *parámetros de tipo*. En este último caso, el parámetro de tipo se reemplaza por el tipo del argumento correspondiente
      - *maximo(3, -4): T* ⇒ tipo *Integer*
      - *maximo("hola", "adios"): T* ⇒ tipo *String*

Abstracción de datos (1)

- Tipo de datos
  - Extienden un lenguaje de programación añadiendo a éste nuevos tipos de datos.
    - Un *tipo de dato* consta de un conjunto de valores (u objetos) y un conjunto de operaciones (o métodos) que caracterizan su comportamiento.
  - Uso de los métodos de abstracción
    - La especificación del tipo de dato establece que son sus instancias y las características de éste, así como la especificación de cada una de las operaciones del tipo.
    - Las operaciones del tipo (abstracciones funcionales) podrán incluir *parámetros de tipo*. Estos mismos parámetros de tipo serán los parámetros de tipo de la abstracción de datos.

Abstracción de datos (2)

- Tanto las abstracciones funcionales, como las abstracciones de datos separan la especificación y uso de la implementación.

Abstracción funcional

Especificación
- Perfil
- *¿Qué hace la función?*

Implementación
- Código de la definición
- *¿Cómo lo hace?*

```
/**
 * Retorna el mayor de los dos objetos dados.
 * @param a el primer objeto
 * @param b el segundo objeto
 * @param <T> el tipo de los objetos
 * @return el mayor de {@code a} y {@code b}
 * @throws ClassCastException si el tipo T no es
 * comparable
 */
public static <T> T maximo(final T a, final T b);
```

```
public static <T> T maximo(final T a, final T b) {
    Comparable<T> ta = (Comparable<T>) a;
    return ta.compare(b) >= 0 ? a : b;
}
```

Abstracción de datos (3)

Especificación (en Java **interface**, en C++ una clase abstracta)

- Nombre del tipo
- Significado de las instancias (¿qué son?)
- Características del tipo
- Especificación de cada operación

Forma conceptual

Relevante para el desarrollador de los programas cliente y del tipo de dato

Tipo de Dato Abstracto (TDA)

Implementación (en Java **class**, una clase que implementa la interfaz)

- Almacenamiento de la información (*representación*)
- Código de las operaciones

Forma estructural

Relevante para el diseñador del tipo de dato y **oculta** para los programas cliente

Abstracción de datos (4)

- En algunos lenguajes de programación, como por ejemplo Java y C++, los nombres de los tipos de datos que tienen *parámetros de tipo* incluyen éstos como sufijo, encerrados entres los caracteres < y > y separados por comas

```
public interface nombre_del_TDA<T1, T2, ...>
public class nombre_del_tipo<T1, T2, ...>
```

- Una buena parte de las interfaces y clases de la biblioteca de Java tienen *parámetros de tipo*, mismamente la interfaz *Comparable<T>*

Abstracción de datos (5)

- ¿Cómo y cuándo se reemplazan los parámetros de tipo?
  - En una función cuando se invoca ésta y los parámetros de tipo se sustituyen por los tipos de sus argumentos
    - *maximo(3, -4): T* ⇒ tipo *Integer*
    - *maximo("hola", "adios"): T* ⇒ tipo *String*
  - En un tipo de dato cuando se instancia el objeto del tipo. Por ejemplo para el TDA *Pair<K, V>* (interfaz), implementado mediante la clase *PairImp<K, V>*:
    - *Pair<String, Integer> p1;*
    - *Pair<String, Integer> p2 =* ***new*** *PairImp<>("hola", 4);*

Cuando se crea la instancia se puede prescindir de indicar los tipos, ya que Java los infiere de la declaración, pero los caracteres < y > son necesarios.

Abstracción de datos. Ejemplo (1)

- Especificación (TDA)

```
/**
 * Tipo de dato abstracto de pares de elementos {@code (a, b)}
 *
 * @param <K> el tipo de la primera componente del par
 * @param <V> el tipo de la segunda componente del par
 */
public interface Pair<K, V> {

    /**
     * Retorna el primer elemento de este par.
     *
     * @return la primera componente del par
     */
    K first();
```

Abstracción de datos. Ejemplo (2)

```
    /**
     * Retorna el segundo elemento de este par.
     *
     * @return la segunda componente del par
     */
    V second();

    . . .
}
```

Abstracción de datos. Ejemplo (3)

- Implementación

```
/**
 * ¡Faltan las características del tipo de dato PairImp<K, V>!
 *
 * @param <K> el tipo de la primera componente del par
 * @param <V> el tipo de la segunda componente del par
 */
public class PairImp<K, V> implements Pair<K, V> {

    private K first;     // primera componente del par
    private V second;    // segunda componente del par
```

Almacenamiento (representación o estructura de datos)

Abstracción de datos. Ejemplo (4)

```
    /**
     * Crea un par con los objetos especificados.
     * @param k la primera componente del par
     * @param v la segunda componente del par
     */
    public PairImp(final K k, final V v) {
        this.first = k;
        this.second = v;
    }

    @Override
    public K first() {
        return this.first;
    }

    . . .
}
```

Abstracción de datos. Clases de tipos (1)

- Clasificación de los tipos de datos
  1. *Según la clase de operaciones soportadas*:
     - **Tipos de datos no modificables (o inmutables)**. Sus casos no se pueden modificar
       - *No ofrecen operaciones* que cambien la representación (pueden tenerlas pero ocultas)
       - La representación de estos tipos puede ser modificable o no
     - **Tipos de datos modificables (o mutables)**. Sus casos se pueden modificar
       - *Ofrecen alguna operación* que modifica la representación
       - La representación de estos tipos debe ser modificable

Abstracción de datos. Clases de tipos (2)

2. *Según su estructura (almacenamiento)*:
   - **Tipos de datos simples**. Cambian su valor pero no su estructura. El espacio de almacenamiento se mantiene constante.
     - Por ejemplo: *booleanos*, *enteros*, *caracteres*
   - **Tipos de datos contenedores (colecciones o, agregados)**. Cambian su valor y su estructura.
     - Por ejemplo: *listas*, *pilas*, *colas*, *árboles*

Abstracción de datos. Clases de operaciones (1)

- Clasificación de las operaciones
  - **Operaciones básicas**. Sólo se pueden implementar una vez elegida la *representación* (o *estructura de datos*) del tipo.

Op. Básicas	Descripción
Constructoras	Crean una nueva instancia del tipo
Observadoras	Retornan un valor u objeto que no es del tipo
Modificadoras	Modifican un valor u objeto del tipo
Destructoras	*Eliminan una instancia del tipo (rec. espacio)*

  - **Operaciones no básicas**. Son operaciones que se pueden, y deben, implementar en base a otras operaciones del tipo.

Abstracción de datos. Clases de operaciones (2)

- ¿Qué operaciones se incluyen en un TDA?
  - Operaciones básicas
    - En los LPOOs que nombran los constructores igual que la clase que implementa la interfaz, por ejemplo en Java, éstos quedan excluidos, pero podrían incluir otras operaciones constructoras.
    - En Java tampoco se incluyen *destructores*, ya que de la recuperación de memoria (almacenamiento) se encarga el recolector de basura (*garbage collection*)
  - Operaciones no básicas que proporcionen funcionalidad al TDA

Abstracción de datos. Características del tipo (1)

- Características del tipo de dato
  - Como ya se ha indicado, proporcionar las características del tipo de dato es una parte de su especificación
  - Las características de un tipo, están muy relacionadas con las clases de tipos dadas previamente.
    - Habitualmente deberá indicarse para todos los tipos de datos la característica de mutabilidad (*mutable* o *inmutable*)
    - Además, si el tipo de dato es un *contenedor*, deberán indicarse características como, por ejemplo:
      - Si puede contener objetos nulos (***null***)
      - Si son de capacidad fija o variable
      - Si admite repeticiones de elementos

Abstracción de datos. Características del tipo (2)

- ¿Uno o varios TDAs, según características del tipo?
  - En C++ es habitual disponer de una clase abstracta por cada posible característica del tipo, por ejemplo, según la característica de mutabilidad
  - En Java lo habitual es que las interfaces (TDAs) sean lo más generalistas posible e independientes de las características del tipo. Estas características se concretan en cada implementación. Así, por ejemplo, un mismo TDA (interfaz) sirve para:
    - Tipos de datos mutables e inmutables
    - Contenedores que admiten o no objetos nulos
    - Contenedores que admiten o no elementos repetidos

Abstracción de datos. Pair<K, V> revisado (1)

```
/**
 * Tipo de dato abstracto de pares de elementos {@code (a, b)}
 * @param <K> el tipo de la primera componente del par
 * @param <V> el tipo de la segunda componente del par
 */
public interface Pair<K, V> {
    /**
     * Retorna el primer elemento de este par.
     * @return la primera componente del par
     */
    K first();   // operación observadora
    /**
     * Retorna el segundo elemento de este par.
     * @return la segunda componente del par
     */
    V second();  // operación observadora
```

Abstracción de datos. Pair<K, V> revisado (2)

```
    /**
     * Cambia el primer elemento de este par por el objeto
     * especificado (operación opcional).
     * @param k el objeto dado
     * @throws UnsupportedOperationException si esta operación
     * no está soportada para este par
     * @throws NullPointerException si este par no admite
     * {@code null} como primer elemento
     */
    default void setFirst(final K k) {    // operación modificadora
        throw new UnsupportedOperationException();
    }

    . . .
}
```

De forma análoga se especificaría la operación:

```
void setSecond(final V v)
```

Abstracción de datos. Sobre las operaciones (1)

- Consideraciones

	Tipos de datos inmutables	Tipos de datos mutables
¿Qué ocurre al compartir la representación entre instancias del tipo?	No hay problema, ya que la información, esté compartida o no entre sus instancias, no se puede modificar	Se producirían efectos incontrolados sobre varias instancias si se modifica la información compartida de cualquiera de ellas, y esto es posible
Espacio de almacenamiento (*gestión de memoria*)	Más eficiente si se comparte información entre las instancias porque sus operaciones o son observadoras o crean nuevas instancias (que requieren espacio)	No compartir la información entre instancias no es crítico, como lo es para los tipos inmutables, porque las instancias se modifican cuando es necesario (no se crea una nueva)

Abstracción de datos. Sobre las operaciones (2)

- ¿En qué operaciones deben tenerse en cuenta las consideraciones previas?
  - En todas las operaciones que creen un objeto del tipo de dato a partir de una o más instancias de éste. Por ejemplo, en el *constructor de conversión*

```
/**
 * Crea un par copia del objeto especificado.
 * @param p el par a copiar
 */
public PairImp(final Pair<K, V> p) {
    // ¿código?
}
```

Es un *constructor de conversión* porque el parámetro p puede ser de tipo *PairImp<K, V>* (copiando p) o de cualquier otro tipo que implemente el TDA *Pair<K, V>* (convirtiendo p)

Abstracción de datos. Sobre las operaciones (3)

- En verdad, con una representación tan simple como la del tipo *PairImp<K, V>*, donde no se requiere utilizar el operador *new* para crear objetos (y por tanto almacenamiento), no se van apreciar diferencias por el hecho de que el tipo de dato sea mutable o inmutable. Sin embargo, esto no va a ser lo habitual, ya que fundamentalmente vamos a ver tipos de datos contenedores.
- Para poner de manifiesto el problema de operaciones como, por ejemplo, el *constructor de conversión*, haremos una nueva implementación del TDA *Pair<K, V>*, el tipo *PairImpAlt<K, V>*.

Abstracción de datos. Sobre las operaciones (4)

```
/**
 * @param <K> el tipo de la primera componente del par
 * @param <V> el tipo de la segunda componente del par
 */
public class PairImpAlt<K, V> implements Pair<K, V> {
    private Data<K, V> data;
    private static class Data<K, V> {
        K first;    // primera componente
        V second;   // segunda componente
        Data(K k, V v) {
            this.first = k;
            this.second = v;
        }
    }
    public PairImpAlt(K k, V v) {
        this.data = new Data<>(k, v);
    }
```

Abstracción de datos. Sobre las operaciones (3)

- Caso 1. *PairImpAlt<K, V>* es mutable
  - Por tanto, las operaciones modificadoras *setFirst(k)* y *setSecond(v)* están soportadas (redefinidas en la clase *PairImpAlt<K, V>*)

```
/**
 * Crea un par copia del par especificado.
 * @param p el par a copiar
 * @throws NullPointerException si el par dado es
 * {@code null}
 */
public PairImpAlt(final Pair<K, V> p) {
    this.data = new Data<>(p.first(), p.second());
}
```

Abstracción de datos. Sobre las operaciones (4)

- Caso 2. *PairImpAlt<K, V>* es inmutable
  - Por tanto, las operaciones modificadoras *setFirst(k)* y *setSecond(v)* no están soportadas (no redefinidas en la clase *PairImpAlt<K, V>*)

```
/**
 * Crea un par copia del par especificado.
 * @param p el par a copiar
 * @throws NullPointerException si el par dado es
 * {@code null}
 */
public PairImpAlt(final Pair<K, V> p) {
    if (p instanceof PairImpAlt<?, ?>) {
        PairImpAlt<K, V> other = (PairImpAlt<K,V>)p;
        this.data = other.data; // datos compartidos
    } else {
        this.data = new Data<>(p.first(), p.second());
    }
}
```

Abstracción de datos. Sobre las operaciones (2)

- Consideraciones adicionales para el lenguaje Java
  - Debe tenerse en cuenta la posible redefinición de algunas de las operaciones que se heredan de *Object*
    - *toString*, *clone*, *equals* y *hashCode*
      - Para más información consúltese el apartado 2.3.2.1 de los apuntes
  - También ha de tenerse en cuenta si las instancias del tipo han de poder compararse
    - En este caso la clase que define el tipo de dato debe implementar una de las siguientes interfaces:
      - *Comparable<T>*
      - *Comparator<T>*

Comparadores (1)

- Las interfaces *Comparable<T>* y *Comparator<T>*
  - La interfaz *Comparable<T>* impone un orden total sobre los objetos de las clases que implementen ésta. Este orden es el *orden natural* de las clases.
  - La interfaz *Comparator<T>* es una interfaz funcional
    - Una **interfaz funcional** define un único método abstracto.
      - Adicionalmente pueden incorporar otros métodos implementados por defecto o estáticos, y que, por tanto, no son abstractos (como así ocurre en *Comparator<T>*)
    - En particular, el método abstracto de la interfaz *Comparator<T>* es una *función de comparación* que impone un orden total sobre una colección de objetos de tipo *T*.

Comparadores (4)

- Interfaz $Comparable<T>$
  - Especifica el método abstracto $int\ compareTo(T\ x)$ para el tipo T, como se indica a continuación:

$$a.compareTo(b) = \begin{cases} < 0 & \text{si } a < b \\ 0 & \text{si } a = b \\ > 0 & \text{si } a > b \end{cases}$$

- Interfaz $Comparator<T>$
  - Especifica el método abstracto, $int\ compare(T\ a, T\ b)$, como se indica a continuación:

$$cmp.compare(a,b) = \begin{cases} < 0 & \text{si } a < b \\ 0 & \text{si } a = b \\ > 0 & \text{si } a > b \end{cases}$$

Un comparador

Comparadores (5)

- Diferencias entre ambas interfaces
  - *Comparable<T>* permite comparar instancias de tipos de datos para los que exista un *orden natural* (*Integer*, *Character*, *String*, etc.)
  - *Comparator<T>* permite comparar instancias de tipos de datos para los que no exista un *orden natural*, o bien, si existiendo tal orden se pretende utilizar un orden distinto al natural

	Comparator<T>	***Comparable<T>***
Método especificado	$cmp.compare(a,b)$	$a.compareTo(b)$
¿Método del tipo T?	no	si
Requisitos del tipo T	ninguno	*orden natural*
Generalidad	+	–

Comparadores (6)

- ¿Qué utilizar en las abstracciones *Comparable<T>* o *Comparator<T>*?
  - Normalmente, será preferible utilizar la interface *Comparator<T>* porque es más general

```
/**
 * Retorna el mayor de los dos objetos dados.
 * @param a el primer objeto
 * @param b el segundo objeto
 * @param <T> el tipo de los objetos
 * @return el mayor de {@code a} y {@code b}
 * @throws ClassCastException si el tipo T no es
 * comparable
 */
public static <T> T maximo(final T a, final T b) {
    Comparable<T> ta = (Comparable<T>) a;
    return ta.compare(b) >= 0 ? a : b;
}
```

Comparadores (7)

- Mejor opción para la abstracción funcional *maximo*
  - Es una función más abstracta

```
public static <T> T maximo(final T a,
                           final T b,
                           final Comparator<T> cmp) {
    return cmp.compare(a, b) >= 0 ? a : b;
}
```

- ¿Cómo crear un comparador para una abstracción, por ejemplo, para invocar la función previa?
  - El comparador se establece en la invocación de la función y se conoce por tanto el tipo *T* (por ejemplo, supondremos que éste es el tipo *String*)

Comparadores (8)

1. *Método tradicional:* crear una clase que implemente la interfaz para crear una instancia de ésta.

prescindible

```
class UnComparador<String> implements Comparator<String> {
   public UnComparador() {}  // constructor por defecto

   @Override
   public int compare(String str1, String str2) {
      if (str1 == null) { return str2 == null ? 0 : -1 }
      return -str1.compareTo(str2);
   }
}
```

El código depende de cómo se deban ordenar los *String*, en este caso, en sentido inverso al *orden natural* (orden descendente).

Comparadores (9)

2. *Utilizar una clase anónima*

- Opción válida para las versiones 6 y 7 de Java, pero que carece de ventaja alguna a partir de la versión 8 del lenguaje.
- No se recomienda, es preferible utilizar la primera opción o la siguiente

```
m = maximo(s1, s2, new Comparator<String>() {
    @Override
    public int compare(String str1, String str2) {
        if (str1 == null) { return str2 == null ? 0 : -1 }
        return -str1.compareTo(str2);
    }
});
```

No se está creando una instancia de una interfaz (esto no es posible, bajo ninguna circunstancia). Se utiliza esta sintaxis porque la clase en la que se instancia el objeto, la clase anónima, no tiene nombre.

Comparadores (10)

3. *Utilizar una expresión lambda*

- Es el método más simple (recomendado)
- Una *expresión lambda* es la definición de *una función anónima*. Este tipo de funciones se pueden pasar como argumento a otra función y también pueden ser el retorno de una función.
  - A diferencia de las funciones con nombre estas expresiones utilizan el símbolo de mapeo (→) para separar los parámetros de la función de su definición. Por ejemplo, $(x, y) \rightarrow x + y$.

```
(x, y) -> {
   return x + y;
}
```

- En Java el símbolo de mapeo se representa mediante el par de caracteres: ->. Para el resto de la expresión (declaración de parámetros y cuerpo de la definición) se utiliza exactamente la misma notación que para cualquier otra función, pero sin nombre ni tipo de retorno. Bajo determinadas circunstancias se pueden realizar ciertas simplificaciones en la expresión, pero una que siempre se puede hacer es prescindir de los tipos de los parámetros

Comparadores (11)

- En Java, una *expresión lambda*, es la implementación del método abstracto de una *interfaz funcional*.
  - Por eso no es necesario especificar los tipos los parámetros de una expresión lambda, se infieren de la especificación del método abstracto de la interfaz funcional correspondiente
- Es más, como cualquier otra expresión del lenguaje, una expresión lambda tiene que tener un tipo. Este tipo es una **interfaz funcional**.

```
m = maximo(s1, s2, (str1, str2) -> {
    if (str1 == null) {
        return str2 == null ? 0 : -1;
    }
    return -str1.compareTo(str2))
});
```

Abstracción iterativa (1)

- Iteradores
  - Un **iterador** (*iterator*) es un *generador* que permite abstraer el proceso de obtener, uno a uno, los elementos de un tipo de dato contenedor (colección o agregado). El proceso se lleva a cabo sin exponer la representación interna de éste.

Genera elementos

Produce los ítems uno a uno

Esquema del procesamiento de los elementos de una colección a con un iterador

```
para cada item producido por el iterador hacer
    f(item);
fin para;
```

El cuerpo del bucle define la acción a realizar con cada ítem

Abstracción iterativa (2)

Objeto Iterable
(Java, Python, etc.)

Definen el comportamiento de su iteración. En Java, un objeto es iterable si la clase que lo instancia implementa la interfaz `Iterable<T>`

- Lenguajes que ofrecen soporte (Python)
  - **Generador**. Un tipo especial de procedimiento que es también un iterador
- Lenguajes que no ofrecen soporte (Java)
  - Un iterador es un *objeto cursor*. Una abstracción de datos basada en el patrón de diseño *iterator* (y éste, a su vez, en *memento*)
    - Con la incorporación de la expresiones lambda (Java 8):
      1. Se simplifica algo el desarrollo de *generadores*, pero éstos no se integran como una parte más del lenguaje
      2. El *`for-each`* se añade como *función de orden superior* a la interfaz `Iterable<T>` (más adelante se incidirá en este aspecto)

Abstracción iterativa (3)

- Ejemplo:
  - Obtener la secuencia: 0, 1, 3, 6, 10, 15, 21, ...

Funciones Generadoras (generadores)

```
def sucesion():
    term, index = 0, 1
    while True:
        yield term
        term += index
        index = index + 1

suc = sucesion()
for i in range(20):
    print(next(suc))
```

```
def sucesion(n):
    term, index = 0, 1
    for index in range(0, n, 1):
        yield term
        term += index

for valor in sucesion(20)
    print(valor)
print()
```

Abstracción iterativa (4)

- Una mala opción en Java

```
public static long enesimo(final int n) {
   long term = 0;
   for (int index = 1; index <= n; index++) {
       term += index;
   }
   return termino;
}
```

```
public static void main(final String[] args) {
   int index = 0;
   // bucle ineficiente, de O(NUM_TERMINOS^2)
   while (index < NUM_TERMINOS) {
      System.out.printf("%d ", enesimo(index));
      index++;
   }
   System.out.println();
}
```

Iteradores en Java (1)

Interfaz Iterable<T>

```
/**
 * Retorna un iterador externo para este Iterable.
 * @return un iterador externo
 */
public Iterator<T> iterator ();
```

```
for (T t: c) {
    f(t);
}
```

Cliente de la abstracción

Bucle *for–each*

Utilizable por objetos iterables (su clase implementa la interfaz *Iterable<T>*)

Iteradores en Java (2)

Interfaz Iterator<T>

```
/**
 * Retorna cierto si la iteración tiene más elementos.
 * @return cierto si la iteración tiene más elementos
 */
public boolean hasNext();

/**
 * Retorna el siguiente elemento en la iteración.
 * @return el siguiente elemento en la iteración
 * @throws NoSuchElementException si la iteración
 * no tiene más elementos
 */
public T next();
```

También incluye la operación opcional `remove()` (ver la API de Java)

Iteradores en Java (3)

- Interfaz `Iterator<T>`
  - Un TDA que incluye las operaciones necesarias para facilitar la construcción del esquema de procesamiento de un iterador en los clientes (iterador externo)

Esquema de un iterador externo

```
para cada item producido por el iterador hacer
    f(item);
fin para;
```

Java

```
Iterator<T> itr = obj.iterator();
while (itr.hastNext()) {
   f(itr.next());
}
```

objeto iterable

Python

```
for item in generador(args):
   f(item)
```

Iteradores en Java (4)

- El iterador implícito de Java
  - En Java, el bucle *for-each* no es una construcción del lenguaje e internamente se transforma en un `for` clásico que utiliza un iterador (un objeto `Iterator<T>`) como variable de control.

Iterador implícito

```
for (T t: c) {
    f(t);
}
```

objeto iterable

```
for (Iterator<T> itr = obj.iterator(); itr.hasNext(); ) {
    f(itr.next());
}
```

Iteradores en Java (5)

- Ejemplo:
  - Obtener la secuencia: 0, 1, 3, 6, 10, 15, 21, ...

```
public class Sucesion implements Iterable<Long> {
   private int numTerminos;

   public Sucesion(final int n) {
      if (n <= 0) {
         throw new IllegalArgumentException();
      }
      this.numTerminos = n;
   }

   @Override
   public Iterator<Long> iterator() {
      return new SucesionIterator();
   }
```

Iteradores en Java (6)

```
    private final class SucesionIterator implements Iterator<Long> {
        private long term = 0;
        private int index = 0;

        public boolean hasNext() {
            return this.index < numTerminos;
        }

        public Long next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            Long temp = this.term;
            this.index++;
            this.term += this.index;

            return temp;
        }
    }
}
```

Iteradores en Java (7)

- Equivalencia entre un bucle y un iterador externo

```
public static void main(final String[] args) {
   int index = 0;
   long term = 0;

   // bucle eficiente, de O(NUM_TERMINOS)

   while (index < NUM_TERMINOS) {
      System.out.printf("%d ", term);
      term += index;
      index++;
   }

   System.out.println();
}
```

hasNext()

next()

Abstracción iterativa (4)

- Alternativa a los iteradores externos
  - Uso de *funciones de orden superior* (FOS).

Esquema de un iterador externo

```
para cada item producido por el iterador hacer
    f(item);
fin para;
```

Alternativa (iterador interno)

```
obj.paraTodosLosItemsHacer(f);
obj.siElItemCumpleHacer(f, predicado)
...
```

objeto iterable

Iteradores en Java (5)

- Iteradores internos

Método de Iterable<T>

```
/**
 * Realiza la acción dada para cada elemento del Iterable
 * hasta que todos los elementos se hayan procesado o la
 * acción lance una excepción.
 * @param action la acción a realizar con cada elemento
 * @throws NullPointerException si la acción especificada
 * es null
 */
default void forEach(Consumer<? super T> action)
```

Interfaz funcional

FOS

Función *f* a realizar con cada ítem de un objeto iterable. Como en el caso de los *comparadores*, el argumento que puede recibir la FOS puede ser:

1. Una instancia de una clase que implemente la interfaz funcional y que podría ser anónima
2. Una *expresión lambda* que, en este caso, sería la implementación del método abstracto: `void action(T t)`

Iteradores (comparación)

- Iteradores externos (*función generadora u objeto cursor*)
  - Más flexibles
  - Menos seguros
  - Tratamiento secuencial de los ítems

El control de la iteración es del cliente

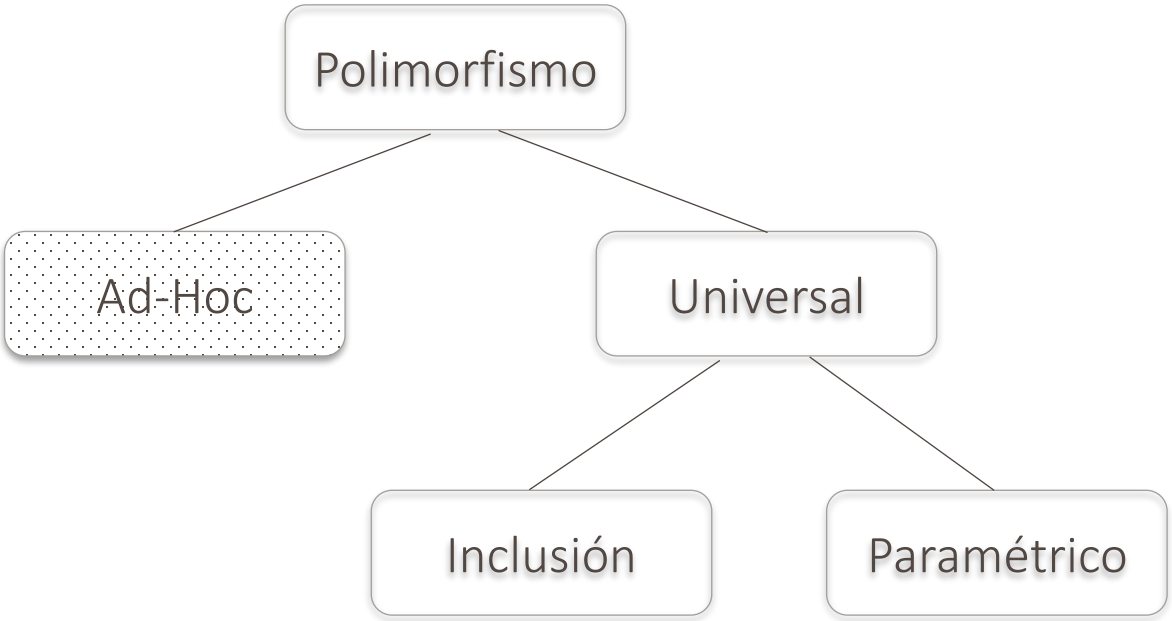
- Iteradores internos (*función de orden superior*)
  - Menos flexibles, pero más simples de utilizar
  - Más seguros
  - Tratamiento secuencial y posibilidad de procesamiento paralelo de los ítems

El control de la iteración es del propio iterador

Polimorfismo (1)

- Definición RAE
  - Cualidad de lo que tiene o puede tener distintas formas.
- En informática
  - Se refiere a la multiplicidad de significados asociados con un nombre.
  - Concepto introducido en la década de los 60.
- Chistopher Strachey (1967) distinguió de manera informal dos clases de polimorfismo: **polimorfismo ad-hoc** y **polimorfismo paramétrico**

Polimorfismo (2)



Clasificación de Cardelli y Wegner (1985)

Abstracciones polimórficas (1)

- Genéricos
  - En los lenguajes de programación orientados a objetos las abstracciones con parámetros de tipo se suelen denotar como *abstracciones genéricas* y a la programación en torno a éstas **programación genérica**.
- Restricciones de los tipos genéricos en Java
  - Son debidas al problema del ***borrado de tipo*** en la MVJ, dando lugar al tipo *raw*.

Abstracciones polimórficas (2)

- No se pueden instanciar tipos genéricos con tipos primitivos
- No se pueden crear instancias de parámetros de tipo
- No se pueden declarar campos estáticos (*static*) de parámetros de tipo
- No se puede invocar el operador *instanceof* con tipos que contengan parámetros de tipo

Abstracciones polimórficas (3)

- No se pueden crear *arrays* de tipos que contengan parámetros de tipo
- No se pueden crear, capturar o lanzar excepciones de objetos de tipos que contengan parámetros de tipo
- No se puede sobrecargar un método cuando los tipos de sus parámetros dan lugar a la misma signatura de tipos *raw*

```
datos = new T[CAPACIDAD]
```

```
datos = (T[]) new Object[CAPACIDAD]
```

Abstracciones polimórficas (4)

- Subtipos de abstracciones polimórficas
  - En Java el tipo *Object* es la raíz de la jerarquía de objetos
    - ¿Es posible extender dicha relación a una abstracción polimórfica como *Collection<T>*?
    - De otro modo, ¿es *Collection<Object>* la raíz de todos los tipos posibles de *Collection<T>*?
      - No, no lo es. La demostración, por reducción a lo absurdo, la tenéis disponible en el apartado 5.1.4.1.3 de los apuntes.

En general, si `A` es un subtipo de `B` y `C<T>` es un tipo genérico, `C<A>` no es un subtipo de `C<B>`.

Abstracciones polimórficas (5)

- Comodines (?)

```
public static void imprimirTodo(Collection<Object> b) {
    for (Object o: b) {
        System.out.println(o.toString());
    }
}
```

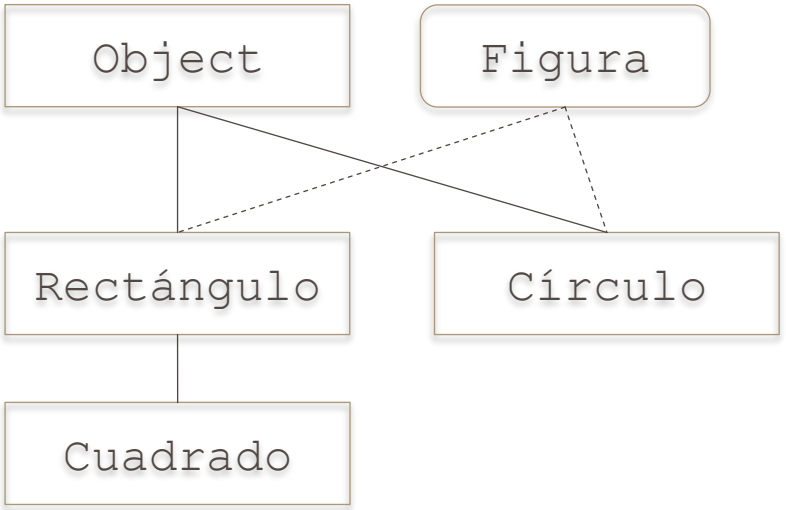
```
public static void imprimirTodo(Collection<?> b) {
    for (Object o: b) {
        System.out.println(o.toString());
    }
}
```

Abstracciones polimórficas (6)

- Comodines delimitados

```
public interface Figura {

    /**
     * Retorna el centro
     * de esta figura.
     * @return el centro
     * de esta figura
     */
    Punto centro();
     /**
     * Retorna el área
     * de esta figura.
     * @return el área
     * de esta figura
     */
    double area();
}
```



Abstracciones polimórficas (7)

```
public static double areaTotal(Collection<?> b) {
    double total = 0;
    for (Object o: b) {
        Figura f = (Figura) o;
        total += f.area();
    }
    return total;
}
```

```
Collection<? extends Figura>
```

```
Collection<? super B>

Collection<A extends B>
```

Abstracciones polimórficas (8)

- Sobre los comodines delimitados
  1. En una jerarquía de tipos la herencia funciona en sentido descendente (algo que ya es sabido y se supone completamente asumido)
     - Por eso en funciones, como la anteriormente vista, lo usual es que el comodín se delimite por la parte superior.
  2. Pero en cualquier situación en la que se use un comparador (en general un predicado) es todo lo contrario, hay que delimitar el comodín por la parte inferior.
     - En una jerarquía de tipos las comparaciones funcionan en sentido ascendente. Si un tipo no dispone directamente de un método de comparación (incluido *equals*) sólo tiene sentido que se busque éste entre sus supertipos.

Análisis de algoritmos (1)

- Comparar la eficiencia de los algoritmos
  - Factores que influyen:
    - El **coste** o **complejidad espacial**
    - El **coste** o **complejidad temporal**
- Coste
  - Depende de la talla del problema.
  - En la práctica, la mayor parte de los algoritmos incluyen alguna sentencia condicional y, en consecuencia, el coste computacional temporal también va a depender de los datos concretos que se le presenten (*casos*).

Análisis de algoritmo (2)

- Orden (notación O)
  - Estima una cota superior del tiempo de ejecución de un algoritmo para entradas de talla *n*.
- Omega (notación Ω)
  - Estima una cota inferior del tiempo de ejecución de un algoritmo para entradas de talla *n*.
- Orden exacto (notación Θ)

- Órdenes de complejidad
  - $O(f(n))$ define un orden de complejidad. Para representar los distintos órdenes se utiliza la función $f: N \to R^+$ más sencilla de cada uno de ellos:

$$O(1) \subset O(\log n) \quad O(\sqrt{n}) \subset O(n) \subset O(n \cdot \log n) \quad O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$$

$$g(n) \in O(f(n)) \Rightarrow 0 < \lim_{n \to \infty} \frac{g(n)}{f(n)} < \infty$$

Análisis de algoritmos (3)

Nombres de funciones de comparación			Orden
Sublineales		Constantes	$O(1)$
		Logarítmicas	$O(\log n)$
			$O(\sqrt{n})$
Lineales			$O(n)$
Superlineales			$O(n \log n)$
	Polinómicas	Cuadráticas	$O(n^2)$
		Cúbicas	$O(n^3)$
	Exponencial		$O(2^n)$
	Factorial		$O(n!)$

Análisis de algoritmos (4)

Talla	Funciones de complejidad						
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
5	3	5	12	25	125	32	120
10	4	10	33	100	1.000	1.024	$3{,}63\ 10^6$
100	7	100	664	10^4	10^6	$1{,}27\ 10^{30}$	$> 10^{100}$
200	8	200	1.529	$4\ 10^4$	$8\ 10^6$	$1{,}6\ 10^{60}$	$> 10^{100}$
1.000	10	1.000	9.965	10^6	10^9	$> 10^{100}$	$> 10^{100}$
2.000	11	2.000	$2{,}2\ 10^4$	$4\ 10^6$	$8\ 10^9$	$> 10^{100}$	$> 10^{100}$
10.000	14	10^4	$1{,}33\ 10^5$	10^8	10^{12}	$> 10^{100}$	$> 10^{100}$