

# Estructuras de datos Básicas

# ADT para representar colecciones

- Las colecciones de objetos se requieren frecuentemente en el desarrollo de software
- Operaciones frecuentes con colecciones:
  - agregar/eliminar items de la colección. Poder almacenar cantidades arbitrarias de items.
  - contener items de cualquier tipo: Genéricos
  - iterar sobre los items de la colección: Iteradores

# Tipos de colecciones

- Bolsa (Bag)  
Solo se agregan items  
El orden de los items no importa
- Cola (Queue / FIFO queue)  
Se agregan items y se remueven preservando el orden de llegada
- Pila (Stack / LIFO queue)  
Se agregan items y se remueven en el orden inverso de llegada

# APIs: Bag

public class <b>Bag</b> <Item> implements Iterable<Item>		
	Bag()	// Crear Bag vacío
void	add(Item item)	// Agregar un item
boolean	isEmpty()	// está vacío?
int	size()	// Número de items

# APIs: Queue

public class Queue<Item> implements Iterable<Item>		
	Queue()	// Crear Queue vacío
void	enqueue(Item item)	// Agregar un item
Item	dequeue()	// Remover item al comienzo de la cola
boolean	isEmpty()	// está vacía?
int	size()	// Número de items

# APIs: Stack

public class Stack<Item> implements Iterable<Item>		
	Stack()	// Crear una pila vacía
void	push(Item item)	// Agregar un item
Item	pop()	// Remover el último item agregado
boolean	isEmpty()	// está vacío?
int	size()	// Número de items

Nota:

Importar el paquete

**import** edu.princeton.cs.algs4.Stack;

Las bibliotecas de Java contienen una clase ligeramente distinta en el paquete java.util.Stack.

# Ejemplo: Pila de capacidad fija

```
public class PilaStringArreglo {  
  
    private String[] pila;  
    private int n;  
  
    public PilaStringArreglo(int max) {  
        pila = new String[max];  
    }  
  
    public void push(String s) {  
        pila[n++] = s;  
    }  
  
    public String pop() {  
        return pila[--n];  
    }  
  
    public boolean isEmpty() {  
        return n==0;  
    }  
  
    public int size() {  
        return n;  
    }  
  
}
```

Resolver:

- Que pasa si se añaden más de max elementos?
- Si se hace pop ( ) y la lista está vacía?
- Que hacer si se necesitan datos de otro tipo?

# Genéricos ó Tipos parametrizados

- Permiten indicar que las clases utilizan parámetros de un tipo variable. De esta forma, las colecciones se pueden implementar de forma “genérica”, para cualquier tipo de dato.
- El parámetro de tipo se indica entre <> al momento de invocar el constructor.
- Ventajas:
  - ♦ El compilador hace chequeo de tipo con los datos contenidos en la colección
  - ♦ Se elimina el casting al sacar datos de la colección



# Pila Genérica de tamaño fijo

```
public class PilaGenericaArreglo<T> {  
  
    private T[] pila;  
    private int n;  
  
    @SuppressWarnings("unchecked")  
    public PilaGenericaArreglo(int n) {  
        // NOTA: Java no permite arreglos de tipo generico  
        pila = (T[]) new Object[n];  
    }  
  
    public void push(T s) {  
        pila[n++] = s;  
    }  
  
    public T pop() {  
        return pila[--n];  
    }  
  
    public boolean isEmpty() {  
        return n==0;  
    }  
  
    public int size() {  
        return n;  
    }  
  
}
```

# Genéricos: Ejemplo de uso

```
Stack<String> pila = new Stack<>();  
pila.push("Hola");  
...  
String dato = pila.pop();
```

# Autoboxing

- Los tipos primitivos no son tipos de referencias.
- Java define clases que contienen datos de tipo primitivo: Int, Long, Short, Byte, Boolean, Double, Float, Character.
- El compilador automáticamente convierte tipos primitivos a tipos de referencia y viceversa.

Por qué no manejar entonces todas las variables usando las wrapper classes?

# Ejemplo autoboxing

```
Queue<Integer> cola = new Queue<>();  
cola.enqueue(Integer.valueOf(1));  
cola.enqueue(1);
```

...

```
int a = cola.dequeue();  
int b = cola.dequeue();
```

# ADT iterables

Implementan la interfaz:

```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

y devuelven un iterador sobre la colección:

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();
}
```

# Iteración mediante la sentencia foreach

- Iterar sobre todos los items de una colección es una operación muy común.
- Para facilitar esta operación se implementa la interfaz `Iterable`.
- Cuando un ADT es `Iterable` se puede utilizar esta sintaxis simplificada

```
for(Item i: coleccion) {  
    StdOut.println(i);  
}
```

# Equivalencia de la sentencia foreach

```
for(Iterator i=coleccion.iterator(); i.hasNext(); ) {  
    Item x = i.next();  
    // Usar el item x  
}
```

# Iterador reverso para la pila

```
public class PilaGenericaArreglo<T> implements Iterable<T> {  
  
    ...  
  
    public Iterator<T> iterator() {  
        return new IteradorReverso();  
    }  
  
    private class IteradorReverso implements Iterator<T> {  
  
        private int pos=n;  
  
        @Override  
        public boolean hasNext() {  
            return pos>0;  
        }  
  
        @Override  
        public T next() {  
            return pila[--pos];  
        }  
  
    }  
  
}
```

```
public static void main(String[] args) {  
    PilaGenericaArreglo<Integer> p = new PilaGenericaArreglo<>(10);  
    ...  
    for(Integer item: p) {  
        StdOut.println(item);  
    }  
  
}
```



# Implementación de colecciones

- Varias alternativas
  - ♦ Con arreglos
  - ♦ Con tipos genéricos
  - ♦ Con listas