

Tipi Generici

classi dipendenti da tipi generici

consentono di modellare strutture dati complesse in maniera indipendente dalla natura effettiva dei dati coinvolti

Couple

```
public class Couple {
```

```
    private final int first;
```

```
    private final int second;
```

```
    public Couple (int first, int second) {
```

```
        this.first = first;
```

```
        this.second = second;
```

```
    }
```

```
    public int getFirst () {
```

```
        return first;
```

```
    }
```

```
    public int getSecond () {
```

```
        return second;
```

```
    }
```

```
    @Override
```

```
    public String toString () {
```

```
        return "(" + first + " + " + second + ")";
```

```
    }
```

```
}
```

```
public class Couple <T> {
```

```
    private final T first;
```

```
    private final T second;
```

```
    public Couple (T first, T second) {
```

```
        this.first = first;
```

```
        this.second = second;
```

```
    }
```

```
    public T getFirst () {
```

```
        return first;
```

```
    }
```

```
    public T getSecond () {
```

```
        return second;
```

```
    }
```

```
    @Override
```

```
    public String toString () {
```

```
        return "(" + first + " + " + second + ")";
```

```
    }
```

```
}
```

Interfaccia Comparable

```
public interface Comparable <T>{           // 0 se o e this sono uguali

    public int compareTo (T o);           //valore negativo se this è più piccolo di o

}                                           //valore positivo se this è più grande di o
```

Comparable Couple

Sottoclasse di couple, confronti due ComparableCouple

```
public class ComparableCouple <T extends Comparable <T>> extends Couple <T> implements Comparable < ComparableCouple <T>>{

    public ComparableCouple (T first, Tsecond){

        super (first, second);

    }

}
```

@Override

```
public int compareTo ( ComparableCouple <T> other){

    int cmp = getFirst().compareTo (other. getFirst());

    if (cmp != 0){

        return cmp;

    }

    return getSecond ().compareTo (other. getSecond());

}
```

Stack

```
public interface Stack <T> {

    public void push ( T n);

    public T pop ();

}
```

```
public boolean isEmpty();  
  
}
```

```
public class StackArray<T> implements Stack<T> {
```

```
    private static final int DIM = 100;
```

```
    private int[] elems;
```

```
    private int top;
```

```
    public StackArray() {
```

```
        this.elems = new T[DIM];
```

```
        this.top = 0;
```

```
    }
```

non si possono istanziare tipi generici



@Override

```
    public void push ( T x ) {
```

```
        if ( top < DIM ) {
```

```
            elems [ top ] = x;
```

```
            top++; }
```

```
        else {
```

```
            throw new StackArrayFullException(); }
```

@Override

```
    public T pop () {
```

```
        if ( isEmpty () ) {
```

```
            throw new StackArrayEmptyException(); }
```

```
        else {
```

```
            top--;
```

```
return elems[top];
```

```
}
```

@Override

```
public boolean isEmpty() {
```

```
    if (top == 0) {
```

```
        return true; }
```

```
}
```

```
public int getElems (int i) {
```

```
    return elems[i];
```

```
    // check i
```

```
}
```

Iterator

Pattern di programmazione che permette di accedere agli elementi di un oggetto aggregato in maniera sequenziale, senza esporre la sua effettiva implementazione;

Iterable: qualcosa che può essere iterato

Albero → foglie

Insieme → elementi

Anno → date

Iterator: qualcosa che fornisce ogni elemento di sequenza di valori

