

implementare una carta del gioco del poker

Card.java (classe concreta)

```
import java.util.Random;

public class Card {
    // Il valore della carta.
    private final int value;

    // Il seme della carta.
    private final int suit;

    // Genera una carta a caso con un valore da min (incluso) in su.
    // Param: min - il valore minimo (0-12) della carta che può essere generata
    public Card(int min) { //costruttore → assegna i valori alle proprietà e non ha return
        //ha lo stesso nome della classe e possono essercene più di uno,
        //si differenziano dal numero di parametri che ricevono in input

        Random random = new Random();
        int valore;
        do {
            valore = random.nextInt(bound: 13); //genera un numero random tra 0 e bound escluso
        } while (valore < min);

        value = valore;

        // value = random.nextInt(13 - min) + min;

        suit = random.nextInt(bound: 4);
    }

    // Genera una carta a caso con un valore da 0 (incluso) in su.
    public Card() {
        // this(0); //si può richiamare il primo costruttore passando il parametro più generico

        Random random = new Random();
        value = random.nextInt(bound: 13);

        suit = random.nextInt(bound: 4);
    }

    public int getValue() { return value; }

    public int getSuit() { return suit; }

    // Ritorna una descrizione della carta sotto forma di stringa, del tipo 10♠ oppure J♥.
    public String toString() {
        String[] valori = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "1"};
        String[] semi = {"♠", "♣", "♦", "♥"};

        return valori[value] + semi[suit];
    }
}
```

```

// Determina se questa carta è uguale ad other.
// Param: other - l'altra carta con cui confrontarsi
// Return: true se e solo se le due carte sono uguali
public boolean equals(Card other) {
    return value == other.value && this.suit == other.suit; //ritorna il risultato
                                                             della condizione
}

// Ordina le carte, prima in base al valore, poi in base al seme.
// Param: other - l'altra carta con cui confrontarsi
// Return: -1 se la carta è più piccola della carta other, 0 se sono uguali, 1 se la carta è più grande della carta other.
public int compareTo(Card other) {
    if (this.value != other.value) {
        if (this.value > other.value) {
            return 1;
        } else {
            return -1;
        }
    } else {
        if (this.suit != other.suit) {
            if (this.suit > other.suit) {
                return 1;
            } else {
                return -1;
            }
        } else {
            return 0;
        }
    }
}
}
}

```

Main.java

Crea una carta *card1* a caso, quindi crea ripetutamente una carta *card2* a caso finché non risulta che *card1* è *equals* con *card2*. A quel punto termina. Sia *card1* che tutte le *card2* dovranno venire stampate sul video man mano che vengono generate.

```

Public class Main {
    public static void main(String[] args) {
        String card1 = (new Card()).toString();
        System.out.println(card1);
        String card2;
        do {
            card2 = new Card().toString();
            System.out.println(card2);
        } while (!card1.equals(card2));

        System.out.println("-----");

        main2();
    }
}

```

```

public static void main2() {
    Card card1 = new Card(min: 12);
    System.out.println(card1);
    System.out.println(card1.toString()); //il toString() nel println è ridondante perché
                                           chiama già il toString della classe più vicina

    Card card2 = null;
    do {
        card2 = new Card(min: 12);
        System.out.println(card2);
    } while (!card1.equals(card2));

    System.out.println("-----");

    main3();
}

public static void main3() { //seconda parte del lab (Card2 - enumerazioni)
    Card2 card1 = new Card2(Value.DUE);
    System.out.println(card1);
    Card2 card2;
    do {
        card2 = new Card2(Value.DUE);
        System.out.println(card2);
    } while (!card1.equals(card2));
}
}

```

Definire due enumerazioni *Value* e *Suit*, che rappresentano rispettivamente il valore delle carte e il loro seme.

Value.java (enumerazione)

```

public enum Value {
    DUE,
    TRE,
    QUATTRO,
    CINQUE,
    SEI,
    SETTE,
    OTTO,
    NOVE,
    DIECI,
    J,
    Q,
    K,
    ASSO;
}

```

Suit.java (enumerazione)

```

public enum Suit {
    PICCHE,
    FIORI,
    QUADRI,
    CUORI;
}

```

Card2.java (classe concreta)

Usa le queste enumerazioni al posto degli interi come valore e seme delle carte.

```
import java.util.Random;

public class Card2 {
    // Il valore della carta.
    private final int value;

    // Il seme della carta.
    private final int suit;

    // Genera una carta a caso con un valore da min (incluso) in su.
    // Param: min - il valore minimo (0-12) della carta che può essere generata
    public Card2(int min) {
        Random random = new Random();
        int valore;
        do {
            valore = random.nextInt(bound: 13);
        } while (valore < min.ordinal()); //ordinal() ritorna l'indice della posizione del "segnaposto"
                                         all'interno dell'enumerazione

        value = (Value.values())[valore]; // values() ritorna a modi array i valori dentro Enum

        /*
        switch (valore) {
            case 0:
                value = Value.DUE;
                break;
            case 1:
                value = Value.TRE;
                break;
            // .....
        }
        */

        // value = random.nextInt(13 - min) + min;

        suit = Suit.values()[random.nextInt(bound: 4)];
    }

    // Genera una carta a caso con un valore da 0 (incluso) in su.
    public Card2() { this(Value.DUE); }

    public Value getValue() { return value; }

    public Suit getSuit() { return suit; }

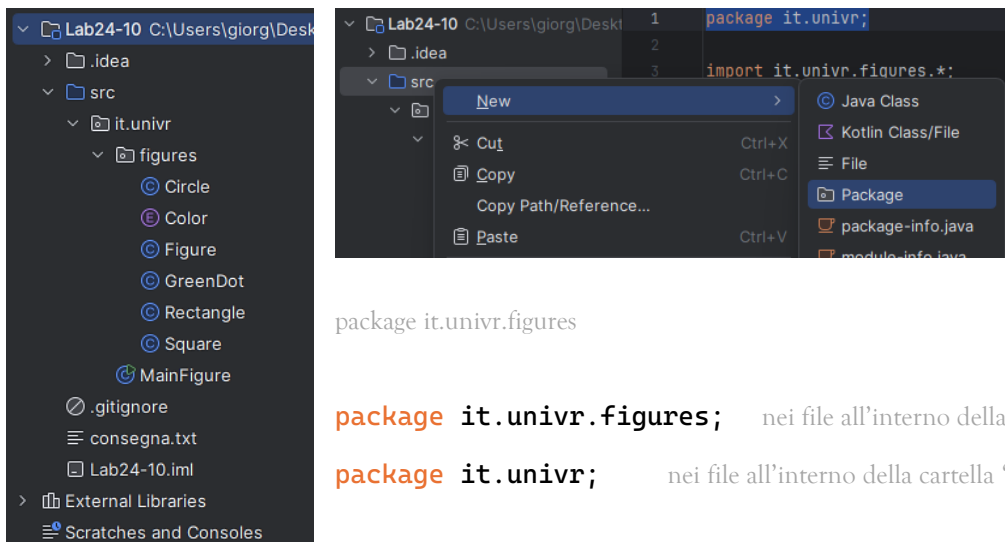
    // Ritorna una descrizione della carta sotto forma di stringa, del tipo 10♠ oppure J♥.
    public String toString() {
        String[] valori = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "1"};
        String[] semi = {"♠", "♣", "♦", "♥"};

        return valori[value.ordinal()] + semi[suit.ordinal()];
    }
}
```

```
// Determina se questa carta è uguale ad other.  
// Param: other - l'altra carta con cui confrontarsi  
// Return: true se e solo se le due carte sono uguali  
public boolean equals(Card other) {  
    return value == value.equals(other.value) && suit.equals(other.suit);  
}  
}
```

`==` → controlla se è lo stesso oggetto (punta alla stessa cella di memoria)

`equals()` → controlla se un oggetto è uguale a un altro



package it.univr.figures

package it.univr.figures; nei file all'interno della cartella "figures"

package it.univr; nei file all'interno della cartella "univr"

Color.java (enumerazione)

```
package it.univr.figures;
```

```
public enum Color {
    GIALLO,
    ROSSO,
    BLU,
    VERDE,
    NERO;
}
```

Figure.java (classe concreta)

Rappresenta una figura geometrica colorata.

```
package it.univr.figures;
```

```
public class Figure {
    private Color colore;

    public Figure (Color colore) { this.colore = colore; }

    public double perimeter() { return 0; }

    public double area() { return 0; }

    @Override     //indica che si sta andando a sovrascrivere il metodo di una classe superiore
    public String toString() {
        return "area: " + this.area() + ", perimeter: " + this.perimeter() +
            ", color: " + colore;
    }

    protected Color getColore() { return colore; }
}
```

Rectangle.java (classe concreta)

Rectangle di Figure, rappresenta un rettangolo.

```
package it.univr.figures;

public class Rectangle extends Figure{    //extends → Rectangle è figlia di Figure
    private double base;
    private double altezza;

    public Rectangle(Color colore, double base, double altezza) {
        super(colore);    //richiama il costruttore della superclasse e assegna il valore alla rispettiva proprietà,
                           //già presente nella superclasse (quindi non si deve ri-istanziare nella sottoclasse).
                           //va richiamato nella prima riga di codice del costruttore,
                           //se non si scrive, è implicito super();
        this.base = base;
        this.altezza = altezza;
    }

    @Override
    public double perimeter() { return (base + altezza)*2; }

    @Override
    public double area() { return base*altezza; }

    @Override
    Ridefinito per ritornare la stringa "Rectangle of " seguita dalla chiamata al toString() della superclasse.
    public String toString() { return "Rectangle of " + super.toString(); }
}
```

Square.java (classe concreta)

Square di Rectangle, rappresenta un quadrato. I metodi *double perimeter()* e *double area()* non vengono ridefiniti.

```
package it.univr.figures;

public class Square extends Rectangle{

    public Square(Color colore, double lato) { super(colore, lato, lato); }

    @Override
    public String toString() { return "Square, a " + super.toString(); }
}
```

Circle (classe concreta)

Circle di Figure, rappresenta un cerchio.

```
package it.univr.figures;

import it.univr.MainFigure;

public class Circle extends Figure{
    private double raggio;

    public Circle(Color colore, double raggio) {
        super(colore);
        this.raggio = raggio;
    }

    @Override
    public double perimeter() { return 2*Math.PI*raggio; }

    @Override
    public double area() { return Math.PI*raggio*raggio; }

    @Override
    public String toString() { return "Circle of " + super.toString(); }
}
```

GreenDot.java (classe concreta)

GreenDot di Circle, rappresenta un cerchio di raggio 1 e di colore verde.

```
package it.univr.figures;

public class GreenDot extends Circle{
    public GreenDot() { super(Color.VERDE, 1); }
}
```


MainFigure.java

```
package it.univr;

import it.univr.figures.*;

public class MainFigure {
    public static void main(String[] args) {
        Figure rettangolo = new Rectangle(Color.BLU, base: 10, altezza: 11);
        Figure quadrato = new Square(Color.BLU, lato: 10);
        print(rettangolo);

        Figure cerchio = new Circle(Color.GIALLO, raggio: 5);
        print(cerchio);

        //in questa classe non è possibile chiamare il metodo getColor() sulle figure,
        //perché il metodo è protected e la classe main non si trova nella stessa cartella del file che contiene il metodo
        //getColor() da richiamare
    }

    public static void print(Figure figure) { //l'oggetto più generico può contenere gli oggetti più
                                                //specifici, quindi posso passare Rectangle, che è una
                                                //sottoclasse di Figure,
                                                //ma viene poi utilizzata sotto il tipo di Figure,
        if (figure instanceof Rectangle) //per averla sotto il tipo di Rectangle, dopo aver controllato che
                                            //effettivamente è un'istanza di Rectangle,
                                            //si può castare, ovvero creare una nuova variabile, definendola col
                                            //tipo più specifico,
                                            //oppure si può dire di considerarla del tipo più specifico,
                                            //a si deve fare ogni volta che si utilizza

            return;

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

SoccerPlayer.java (*interfaccia*) comprende tutti i metodi di una classe, senza implementarli

Specifica un giocatore di calcio

```
public interface SoccerPlayer {
    String toString(); // ritorna il nome del giocatore
    boolean canUseHands(); // determina se il giocatore può usare le mani
}
```

AbstractSoccerPlayer.java (*classe astratta*) mette insieme la classe concreta e l'*interfaccia*, comprende quindi metodi già implementati, e metodi non implementati (da implementare nelle sottoclassi), contrassegnati dalla parola **abstract**

```
public abstract class AbstractSoccerPlayer implements SoccerPlayer { //implements → implementa l'interfaccia

    private String name;

    protected AbstractSoccerPlayer(String name) { this.name = name; }

    @Override
    public final String toString() { return name; }

    public abstract boolean canUseHands(); //metodo da implementare
}
```

Sottoclassi *Forward*, *Midfield*, *Defence* e *GoalKeeper*, (solo il *GoalKeeper* può usare le mani). ...

Forward.java (classe concreta)

```
public class Forward extends AbstractSoccerPlayer{

    protected Forward(String name) { super(name); } //anche il costruttore della superclasse è
                                                    protected, cambiando visibilità non si va a
                                                    sovrascrivere, ma se ne crea uno nuovo

    @Override
    public boolean canUseHands() { return false; }
}
```

Midfield.java (classe concreta)

```
public class Midfield extends AbstractSoccerPlayer{

    protected Midfield(String name) { super(name); }

    @Override
    public boolean canUseHands() { return false; }
}
```

Defence.java (classe concreta)

```
public class Defence extends AbstractSoccerPlayer{

    protected Defence(String name) { super(name); }

    @Override
    public boolean canUseHands() { return false; }
}
```

GoalKeeper.java (classe concreta)

```
public class GoalKeeper extends AbstractSoccerPlayer{

    protected GoalKeeper(String name) { super(name); }

    @Override
    public boolean canUseHands() { return true; }
}
```

Formation.java (classe concreta)

```
public class Formation {

    private SoccerPlayer[] players;

    public Formation(SoccerPlayer[] players) {
        this.players = players;
        if (!isValid())
            throw new IllegalArgumentException("invalid formation"); //lancia un errore (rosso)
    }
}
```

```
}
```

// ritorna true se e solo se la formazione è fatta da 11 giocatori, di cui esattamente uno è un portiere

```
protected boolean isValid() {
    int portieri = 0;

    /*
    for (int i = 0; i < players.length; i++) {
        if (players[i].canUseHands()) {
            portieri++;
        }
        if (portieri > 1) {
            return false;
        }
    }
    */

    for (SoccerPlayer player : players) { //for each → SoccerPlayer player = players[i];
                                            //viene istanziata una variabile, in questo caso di tipo
                                            SoccerPlayer
                                            (bisogna prendere il tipo della proprietà della singola
                                            cella dell'array → ad esempio con String si prende char),
                                            che passa una a una ogni cella dell'array

        if (player.canUseHands()) {
            portieri++;
        }
        if (portieri > 1) {
            return false;
        }
    }

    return players.length == 11 && portieri == 1;
}
```

// ritorna i giocatori di questa formazione

```
protected SoccerPlayer[] getPlayers() { return this.players; }
```

@Override

// ritorna i nomi dei giocatori della formazione, separati da virgola

```
public final String toString() {
    String giocatori = "";

    for (int i = 0; i < players.length; i++) {
        if (i == 0) {
            giocatori = players[i].toString();
        } else {
            giocatori = giocatori + ", " + players[i].toString();
        }
    }

    return giocatori;
}

}
```

Formation433.java (classe concreta)

Sottoclasse concreta di *Formation*.

```
public class Formation433 extends Formation{

    public Formation433(SoccerPlayer[] players) { super(players); }

    // ritorna true se e solo se la formazione è fatta da 11 giocatori,
    // di cui 4 difensori, 3 centrocampisti e 3 attaccanti, e un portiere
    protected boolean isValid() {
        if (!super.isValid()) { return false; }

        int difensori = 0;
        int centrocampisti = 0;
        int attaccanti = 0;

        /*
        for (SoccerPlayer player : getPlayers()) {
            if (player instanceof Defence) {
                difensori++;
                if (difensori > 4) { return false; }
            } else if (player instanceof Midfield) {
                centrocampisti++;
                if (centrocampisti > 3) { return false; }
            } else if (player instanceof Forward) {
                attaccanti++;
                if (attaccanti > 3) { return false; }
            }
        }

        return true;
        */

        for (SoccerPlayer player : getPlayers()) {
            if (player instanceof Defence) {
                difensori++;
            } else if (player instanceof Midfield) {
                centrocampisti++;
            } else if (player instanceof Forward) {
                attaccanti++;
            }
        }

        return difensori == 4 && centrocampisti == 3 && attaccanti == 3;
    }
}
```

Main.java

```
import java.util.LinkedList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        String[] difensori = {"Alex Sandro", "Rugani", "Chiellini", "Dani Alves"};
        String[] centrocampisti = {"Fabinho", "Iniesta", "Pjanic"};
        String[] attaccanti = {"Dybala", "Higuain", "Bernardeschi"};
        String[] portiere = {"Szczesny"};
        String[][] giocatori = {difensori, centrocampisti, attaccanti, portiere};

        SoccerPlayer[] players = new SoccerPlayer[12]; //array di SoccerPlayer
                                                         con dimensione definita
                                                         //12 → test per vedere se fallisce
                                                         (dimensione valida → 11)

        List<SoccerPlayer> playerList = new LinkedList<SoccerPlayer>(); //LinkedList
        test(playerList); //LinkedList

        int index = 0;
        for (int i = 0; i < giocatori.length; i++) {
            for (int j = 0; j < giocatori[i].length; j++) {
                switch (i) {
                    case 0:
                        players[index] = (new Defence (difensori[j]));
                        index++;
                        break;
                    case 1:
                        players[index] = (new Midfield (centrocampisti[j]));
                        index++;
                        break;
                    case 2:
                        players[index] = (new Forward (attaccanti[j]));
                        index++;
                        break;
                    case 3:
                        players[index] = (new GoalKeeper (portiere[j]));
                        index++;
                        break;
                }
            }
        }

        players[index] = (new GoalKeeper("Tizio")); //test per vedere se fallisce
    }
}
```

```

//try - catch - finally
try {          //prova il pezzo di codice all'interno,
               se durante l'esecuzione vengono generate delle eccezioni all'interno del try,
               e sono catturabili da uno dei catch, esegui il catch corrispondente all'eccezione (errore)
    System.out.println(new Formation433(players).isValid());
    System.out.println(new Formation433(players));
} catch (ExceptionInInitializerError e) {          //esegue in catch sse l'eccezione (errore)
                                                    generata è del tipo che riceve come parametro
                                                    o sua "figlia"

    System.out.println("Errore nella creazione della formazione.");
    //System.exit(-1);          //-1 → codice di uscita con errore
} catch (IllegalArgumentException e) {
    System.out.println(e);
    //System.exit(-1);
} finally {          //a prescindere esegui,
                    se non viene terminata prima l'esecuzione (System.exit, errori non catturati, return, ...)
    System.out.println("Try catch finito.");
    System.exit(-1);
}
}

public static void test (List<SoccerPlayer> playerList) {}          //LinkedList
                                                    //riesce a entrare perché la classe LinkedList è figlia
                                                    dell'interfaccia List
                                                    //e il metodo è static come il main
}

```

Number.java (interfaccia)

```
public interface Number extends Comparable<Number> { //Comparable<T>
    int getValue(); // restituisce il valore di questo numero
}
```

AbstractNumber.java (classe astratta)

Implementazione astratta di un Number. Fornisce le funzionalità comuni a tutti i numeri.

```
public abstract class AbstractNumber implements Number {
    private final int value;

    // se value è negativo, esegue throw new IllegalArgumentException(); altrimenti inizializza il campo value
    protected AbstractNumber(int value) {
        if (value < 0) {
            throw new IllegalArgumentException();
        }
        this.value = value;
    }

    // restituisce il valore di questo numero
    public final int getValue() { return this.value; }

    // restituisce la base di numerazione di questo numero
    protected abstract int getBase();

    // restituisce il carattere che rappresenta la cifra "digit" nella base di numerazione
    // di questo numero. Sarà sempre vero che 0 <= digit < getBase();
    // per esempio, in base sedici si avrà getCharForDigit(10) == 'A' e
    // in base otto si avrà getCharForDigit(7) == '7'
    protected abstract char getCharForDigit(int digit);

    // restituisce una stringa che rappresenta il numero nella sua base di numerazione
    @Override
    public String toString() {
        String string = "";

        int val = this.value;
        do {
            string = getCharForDigit(val % getBase()) + string;
            val = val/getBase();
        } while (val > 0);

        return string;
    }
}
```



```

// due numeri sono uguali se e solo se hanno lo stesso valore
@Override
public final boolean equals(Object other) {
    if (!(other instanceof Number)) { //controlla se other è istanza di Number,
                                     se non lo è si sa già che non è uguale
        return false;
    }

    //other = (Number) other;    NO → ridondante perché other resta di tipo Object

    //((Number) other).getValue();  scrivere ogni volta il tipo da considerare

    Number otherNumb = (Number) other; //cast
    return this.value == otherNumb.getValue();
}

// ordinamento fra i Number è quello crescente per valore
@Override
public final int compareTo(Number other) {
    if (this.value < other.getValue()) {
        return -1; //ritorna -1 se this < other
    } else if (this.value == other.getValue()) {
        return 0; //ritorna 0 se this = other
    } else {
        return 1; //ritorna 1 se this > other
    }
}
}

```

Sottoclassi concrete *DecimalNumber*, *BinaryNumber*, *OctalNumber*, *HexNumber* e *Base58Number* di *AbstractNumber*.
 Il metodo *toString()* ereditato da *AbstractNumber* funziona per tutte queste sottoclassi.

DecimalNumber.java (classe concreta)

```

public class DecimalNumber extends AbstractNumber{
    public DecimalNumber(int value) { super(value); }

    @Override
    protected int getBase() { return 10; }

    @Override
    protected char getCharForDigit(int digit) {
        return string.charAt(digit); //ritorna il char (della stringa string) che corrisponde all'indice in input (digit)
    }

    private static final String string = "0123456789";
}

```

BinaryNumber.java (classe concreta)

```
public class BinaryNumber extends AbstractNumber{
    public BinaryNumber(int value) { super(value); }

    @Override
    protected int getBase() { return 2; }

    @Override
    protected char getCharForDigit(int digit) { return string.charAt(digit); }

    private static final String string = "01";
}
```

-

OctalNumber.java (classe concreta)

```
public class OctalNumber extends AbstractNumber{
    public OctalNumber(int value) { super(value); }

    @Override
    protected int getBase() { return 8; }

    @Override
    protected char getCharForDigit(int digit) { return string.charAt(digit); }

    private static final String string = "012345678";
}
```

-

HexNumber.java (classe concreta)

```
public class HexNumber extends AbstractNumber{
    public HexNumber(int value) { super(value); }

    @Override
    protected int getBase() { return 16; }

    @Override
    protected char getCharForDigit(int digit) { return array[digit]; }

    private static final char[] array =
        {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
}
```

Base58Number.java (classe concreta)

```
public class Base58Number extends AbstractNumber{
    public Base58Number(int value) { super(value); }

    @Override
    protected int getBase() { return 58; }

    @Override
    protected char getCharForDigit(int digit) { return string.charAt(digit); }

    private static final String string =
        "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz";
}
```

BinaryNumberWithParity.java (classe concreta)

Sottoclasse concreta di *BinaryNumber*.

Il numero binario viene esteso con un'ulteriore cifra binaria di controllo, in modo da rendere pari il numero totale di cifre 1: se la quantità pari di 1, si aggiungerà uno 0, sennò si aggiungerà un 1.

```
public class BinaryNumberWithParity extends BinaryNumber{
    public BinaryNumberWithParity(int value) {
        super(value);
    }

    @Override
    public String toString(){
        String string = super.toString();
        int counter1 = 0;

        for (int i = 0; i < string.length(); i++) {
            //char c = string.charAt(i);
            if (string.charAt(i) == '1') {
                counter1++;
            }
        }
        /*
        for (char stringa : string) {           //non si può fare il for each sulle stringhe
            if (stringa == 1) {
                counter1++;
            }
        }
        */

        return string + (counter1 % 2);        //se dispari il resto è 1, senno è 0
                                              //viene concatenato alla stringa del numero binario già esistente
    }
}
```

Chiede all'utente di inserire un numero non negativo n , quindi crea il numero in base 10, 2, 2 con parità, 8, 16 e 58.

Crea un array[6] di Number:

8092 in base 58

```
import java.util.Arrays;

public class MainNumbersSort {
    public static void main(String[] args) {
        Number[] array = {new DecimalNumber(2024), new BinaryNumber(113),
            new BinaryNumberWithParity(158), new OctalNumber(827), new HexNumber(2066),
            new Base58Number(8092)};

        Arrays.sort(array);    //ordina l'array, secondo il metodo compareTo (ordinamento crescente) → Comparable<T>
        System.out.println(Arrays.toString(array));    // [1110001, 100111101, 1473, 2024, 812, 3QX]
        //qui non è ridondante perché bisogna chiamare il
        //toString degli array
    }
}
```

List.java (classe concreta)

Rappresenta una lista non vuota di T (può essere rimpiazzato con qualsiasi tipo (*obj*))
(≠ da interfaccia *List*, o dalle classi *LinkedList*, *ArrayList*, ...)

```
package it.univr.lists;

import java.io.IOException;
import java.io.PrintWriter;

public class List<T> {
    private final T head;          //testa della lista
    private final List<T> tail;    //il resto della lista

    // crea una lista con la testa e la coda indicate
    public List(T head, List<T> tail) {
        this.head = head;          //testa
        this.tail = tail;          //elemento successivo (next)
    }

    // crea una lista contenente la testa indicata, seguita dagli elementi indicati
    public List(T head, T... elements) { //passa in input in numero indefinito di elements di tipo T
        this.head = head;          //assegnata la testa ("value")

        /*
        T[] elementi = new T[elements.length]; //non si può istanziare un array di tipo T
        for (int i = 0; i < elements.length-1; i++) {
            elementi[i] = elements[i+1];
        }
        this.tail = new List<T>(elements[0], elementi);
        */

        List<T> list = null;

        for (int i = (elements.length - 1); i >= 0 ; i--) { //metodo ricorsivo
                                                    //abbiamo già la testa (this.head = head;)
                                                    //parte dall'ultimo elemento della lista, costruendo ogni nodo
                                                    //tramite il primo costruttore, e salvata mano a mano in list
            list = new List<T>(elements[i], list);    //prende in input "testa" e "elemento successivo"
        }

        this.tail = list;          //alla fine "collega" la testa alla coda
                                   (la coda è l'elemento successivo ("next"))
    }
}
```

```
public class List<T> { 10 usages  1 inheritor  ⚡ GiorgiaZanini
    private final List<T> tail; 10 usages      tail: null

    // crea una lista con la testa e la coda indicate
    public List(T head, List<T> tail) { 10 usages  ⚡ GiorgiaZanini      head: "are"      tail: "you?"
        this.head = head;  head: "are"      head: "are"
        this.tail = tail;  tail: "you?"      tail: null
    }
}
```

```

// restituisce una descrizione di questa lista, fatta dai toString()
// dei suoi elementi separati da virgole
public String toString() {
    String string = head.toString();

    if (tail == null)
        return string;

    List<T> tempTail = tail;
    while (tempTail != null) {
        string = string + ", " + tempTail.head;    //aggiunge il valore del nodo alla stringa
        tempTail = tempTail.tail;                //passa al nodo successivo
    }

    return string;
}

// restituisce il numero di elementi di questa lista
public int length() {
    int counterNodes = 1;
    if (tail == null)
        return counterNodes;    //se c'è solo un nodo ritorna i counter dei nodi a 1

    List<T> tempTail = tail;
    while (tempTail != null) {
        counterNodes++;    //conta il nodo corrente
        tempTail = tempTail.tail;    //passa al nodo successivo
    }

    return counterNodes;
}

// scrive gli elementi di questa lista (cioè il loro toString())
// dentro il file testuale col nome indicato (un PrintWriter vi aiuterà)
public void dump(String fileName) throws IOException{    //dump = "buttare fuori"
                                                         In questo caso scrivere su file
                                                         (struttura dati salvata nella ram)
    PrintWriter printWriter = new PrintWriter(fileName);    //per sola scrittura su file

    printWriter.print(this.head + " ");
    List<T> tempTail = tail;
    while (tempTail != null) {    //come il toString(), cicla sui nodi e li scrive mano a mano
        printWriter.print(tempTail.head + " ");
        tempTail = tempTail.tail;
    }

    printWriter.close();    //chiude il file (in scrittura)
}
}

```

IntList.java

Rappresenta una lista di interi (sottoclasse di *List<T>*)

```
package it.univr.lists;

import java.io.FileReader;
import java.io.IOException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class IntList extends List<Integer> {

    public IntList(Integer head, IntList tail) { super(head, tail); }
    public IntList(Integer head, Integer... elements) { super(head, elements); }

    // restituisce una lista di interi letta dal file testuale indicato;
    // in caso di errore di lettura, lancia una IOException; uno Scanner vi aiuterà
    public static IntList readFrom(String fileName) throws IOException {
        Scanner scanner = new Scanner(new FileReader(fileName)); //crea un oggetto Scanner che
                                                                    legge da file

        //return readFrom(scanner);
        try {
            return new IntList(scanner.nextInt(), readFrom(scanner)); // **
        } catch (NoSuchElementException e) { // *
            throw new IOException(e);
        }
    }

    private static IntList readFrom(Scanner scanner) throws IOException {
        try {
            //return new IntList(scanner.nextInt(), scanner.hasNextInt() ?
            //readFrom(scanner) : null); //new IntList(1, new IntList(2, new IntList(3, null)));
            //soluzione, ricorsiva, ma ritorna, il primo parametro (head) al
            //costruttore, prima di fare il controllo scanner.hasNextInt(),
            //quindi ritorna un errore, perché non controlla il primo ma
            //ritorna direttamente, e poi fa il controllo sui successivi,
            //e fa il controllo sul secondo parametro (tail)
            //aggiungo il ritorno del primo senza controllo nel readFrom(String) *
            //soluzione → ritorna il primo parametro (scanner.nextInt()),
            //poi per il secondo (tail) →
            //if il file ha un intero successivo (scanner.hasNextInt() ?)
            //se è true, ritorna leggi (readFrom(scanner)),
            //senno ( : ) ritorna null (null)

            //ricorsivo → implementata la soluzione per esteso **
            if (scanner.hasNextInt()) { //prima di leggere, controlla se quello che andrà a leggere è un intero
                return new IntList(scanner.nextInt(), readFrom(scanner));
            } else {
                return null;
            }
        } catch (NoSuchElementException e) { //se l'elemento letto non è un intero ...
            throw new IOException(e); //... lancia un'eccezione (di tipo IOException(e))
        }
    }
}
```

Main.java

```
package it.univr.lists;

import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try {
            List<String> l1 = new List<String>(head: "hello", ...elements: "how", "are", "you?");
            System.out.println(l1 + " di lunghezza " + l1.length());
            l1.dump(fileName: "l1.txt");

            IntList l2 = new IntList(head: 11, ...elements: 13, 42, 9, -5, 17, 13);
            System.out.println(l2 + " di lunghezza " + l2.length());
            l2.dump(fileName: "l2.txt");

            IntList l3 = IntList.readFrom(fileName: "l2.txt");
            System.out.println(l3 + " di lunghezza " + l3.length());

            IntList.readFrom(fileName: "l1.txt"); // fallisce perché l1.txt contiene stringhe, non interi
        }
        catch (IOException e) {
            //System.out.println(e);
            System.out.println("Errore di I/O");
        }
    }
}
```

Output main:

```
hello, how, are, you? di lunghezza 4
11, 13, 42, 9, -5, 17, 13 di lunghezza 7
11, 13, 42, 9, -5, 17, 13 di lunghezza 7
Errore di I/O
```