

```

public class ContoCorrente implements MetodoPagamento, implements Comparable<ContoCorrente> {
    private String IBAN;
    private double saldo;

    public ContoCorrente(String IBAN, double saldo) {
        this.IBAN = IBAN;
        this.saldo = saldo;
    }

    public double verificaSaldo() {
        return saldo;
    }

    public void incrementa(double value) {
        this.saldo += value;
    }

    public void decrementa(double value) throws SaldoNonSufficienteException {
        if (value > saldo)
            throw new SaldoNonSufficienteException(" ");
        this.saldo -= value;
    }

    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        else if (obj == null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        ContoCorrente other = (ContoCorrente) obj;
        return IBAN.equals(other.IBAN)
            && saldo == other.saldo;
    }

    public class SaldoNonSufficienteException extends Exception {
        public SaldoNonSufficienteException(String msg) {
            super(msg);
        }
    }
}

```

```

    public int compareTo(ContoCorrente other) {
        return Double.compare(saldo, other.saldo);
    }
}

```

```
public class Persona {
    private String nome;
    private String cognome;
    private Set<MetodoPagamento> metodiPag;
    public Persona (String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
        this.metodiPag = new HashSet<>();
    }
    public void aggiungiMetodoPagamento (MetodoPagamento metodo) {
        metodiPag.add (metodo);
    }
    public String paga (MetodoPagamento m, double importo) {
        if (!metodiPag.contains (m))
            return " pagamento non effettuato ";
        m.decrementa (importo);
        return " pagamento effettuato ";
    }
}
```

} → try {
 m.decrementa (importo);
 } catch (SaldoNonSufficienteException e) {
 return " saldo insufficiente ";
 }
 return " pagamento effettuato ";
 }

Ex. 1 02/09/22

```
template < class T >
class MultiSet {
private:
    T* container;
    int dim;
    int top;
    void resize() {
        T* tmp = new T[dim*2];
        for(int i=0; i<dim; i++)
            tmp[i] = container[i];
        delete [] this->container;
        this->dim *= 2;
        this->container = tmp;
    }
public:
    MultiSet() {
        this->dim = 10
        this->container = new T[dim];
        this->top = 0;
    }
    ~MultiSet() {
        delete [] this->container;
    }
    MultiSet(const MultiSet & other) {
        this->dim = other.dim;
        this->top = other.top;
        this->container = new T[dim];
        for(int i=0; i<top; i++)
            this->container[i] = other.container[i];
    }
    MultiSet& operator=(const MultiSet & other) {
        if (this != &other) {
            delete [] this->container;
            this->dim = other.dim;
            this->top = other.top;
        }
    }
}
```

```

this->container = new T[dim];
for (int i=0; i< top; i++)
    container[i] = other.container[i];
}

return *this;
}

void add (T elem) {
    if (top == dim)
        resize();
    this->container[top++] = elem;
}

void remove (T elem) {
    if (isEmpty())
        throw runtime_error(" ");
    T* newContainer = new T[dim];
    int count = 0;
    for (int i=0; i< top; i++) {
        if (container[i] != elem)
            newContainer[count++] = container[i];
    }
    delete [] this->container;
    this->container = newContainer;
    this->top = count;
}

bool isEmpty() const {
    return top == 0;
}

int cardinality (T elem) const {
    int count = 0;
    for (int i=0; i< top; i++)
        if (container[i] == elem)
            count++;
    return count;
}

void print(ostream& out) const {
    out << '[';
    for (int i=0; i< top; i++) {
}

```

```
        fout << container[i];
    if (i != top-1)
        fout << ", ";
}
fout << "]" << endl;
};

template<class T>
ostream& operator<< (ostream& fout, const MultiSet<T>& set){
    Set.print(fout);
    return fout;
}
```

```

public abstract class Aula {
    private String nome;
    private int maxCapienza;
    public Aula(String nome, int capienza) {
        this.nome = nome;
        this.maxCapienza = capienza;
    }
    public String getName() {
        return nome;
    }
    public int getMaxCapienza() {
        return maxCapienza;
    }
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        else if (obj == null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        Aula other = (Aula) obj;
        return nome.equals(other.nome)
            && maxCapienza == other.maxCapienza;
    }
}

public class AulaAttrezzata extends Aula {
    private int numeroPostazioniPc;
    public AulaAttrezzata(String nome, int maxCap, int numPc) {
        super(nome, maxCap);
        this.numeroPostazioniPc = numPc;
    }
    public boolean equals(Object obj) {
        if (super.equals(obj))
            return false;
        AulaAttrezzata other = (AulaAttrezzata) obj;
    }
}

```

```
        return numeroPostazioniPc >= other.numeroPostazioniPc;
    }

    public class Prenotazione {
        private Aula aula;
        private String dataPrenotazione;
        public Prenotazione(Aula aula, String data) {
            this.aula = aula;
            this.dataPrenotazione = data;
        }
        public Aula getAula() {
            return aula;
        }
        public String getData() {
            return dataPrenotazione;
        }
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            else if (obj == null)
                return false;
            else if (getClass() != obj.getClass())
                return false;
            Prenotazione other = (Prenotazione) obj;
            return aula.equals(other.aula)
                && dataPrenotazione.equals(other.dataPrenotazione);
        }
    }

    public class SistemaPrenotazioni implements Iterabile<Prenotazione> {
        private List<Prenotazione> prenotazioni;
        public SistemaPrenotazioni() {
            this.prenotazioni = new HashSet<>();
        }
        public void aggiungiPrenotazione(Prenotazione p) {
            if (prenotazioni.contains(p))
                throw new AulaOccupataException;
            prenotazioni.add(p);
        }
    }
}
```

```
public Set<AulaAttrezzata> getAuleAttrezzatePrenotateOm (String d) {  
    Set<AulaAttrezzata> auleAttrezzatePrenotateOm = new HashSet<()>;  
  
    for (Prenotazione prenotazione : prenotazioni) {  
        if (prenotazione.getAula() instanceof AulaAttrezzata)  
            && prenotazione.getData().equals(d))  
                auleAttrezzatePrenotateOm.add ((AulaAttrezzata) prenotazione.getAula());  
    }  
  
    return auleAttrezzatePrenotateOm;  
}  
  
public Iterator<Prenotazione> iterator() {  
    return prenotazioni.iterator();  
}  
}  
  
}  
  
public class AulaOccupataException extends RuntimeException {  
    public AulaOccupataException (String msg) {  
        super (msg);  
    }  
}
```

```

public abstract class Elegible {
    private String nome;
    protected int voti;
    public Elegible(String m, int v) {
        this.nome = m;
        this.voti = v;
    }
    public String getNome() {
        return nome;
    }
    public int getVoti() {
        return voti;
    }
    public boolean equals(Object obj) {
        if (this == obj) return true;
        else if (obj == null) return false;
        else if (getClass() != obj.getClass()) return false;
        Elegible other = (Elegible) obj;
        return nome.equals(other.nome) && voti == other.voti;
    }
}
public class Partito extends Elegible {
    public Partito(String m) {
        super(m, 0);
    }
    public void vota() {
        this.voti++;
    }
    public boolean equals(Object obj) {
        return super.equals(obj);
    }
}
public class Coalizione extends Elegible implements Iterable<Partito> {
    private Set<Partito> partiti;
    public Coalizione(String nome, Partito[] p) {

```

```
super(nome, 0);

this.partiti = new HashSet<P>();

for (Partito partito : p) {
    this.voti += partito.getVoti();
    this.partiti.add(partito);
}

}

public int getVoti() {
    return voti;
}

}

public boolean equals(Object obj) {
    if (!super.equals(obj)) return false;
    Coalizione other = (Coalizione) obj;
    return partiti.equals(other.partiti);
}

}

public Iterator<Partito> iterator() {
    return partiti.iterator();
}

}

public class Elezione {
    private Set<Coalizione> coalizioni;

    public Elezione() {
        this.coalizioni = new HashSet<C>();
    }

    public void add(Coalizione c) {
        if (coalizioni.contains(c))
            throw new RuntimeException(" ");
        coalizioni.add(c);
    }

    public Coalizione vincitore() {
        Coalizione vincitore = null;
        for (Coalizione c : coalizioni) {
            if (vincitore == null || vincitore.getVoti() < c.getVoti())
                vincitore = c;
        }
        return vincitore;
    }
}
```

ES. 2 12/10/123

```
public class Stack<T> implements Comparable<Stack<T>> {  
    private ArrayList<T> container;  
    public Stack() {  
        this.container = new ArrayList<T>();  
    }  
  
    public void push(T elem) {  
        container.add(elem);  
    }  
  
    public T pop() throws EmptyStackException {  
        if (isEmpty())  
            throw new EmptyStackException(" ");  
        T result = container.get(container.size() - 1);  
        container.remove(container.size() - 1);  
        return result;  
    }  
  
    public T peek() throws EmptyStackException {  
        if (isEmpty())  
            throw new EmptyStackException(" ");  
        return container.get(container.size() - 1);  
    }  
  
    public boolean isEmpty() {  
        return container.isEmpty();  
    }  
  
    public int size() {  
        return container.size();  
    }  
  
    public String toString() {  
        String result = "[";  
        for (int i = 0; i < size(); i++) {  
            result += container.get(i);  
            if (i != size() - 1)  
                result += ", ";  
        }  
        result += "]";  
        return container.toString();  
    }  
}
```

```
    return result;
}

public boolean equals (Object obj) {
    if (this == obj) return true;
    else if (obj == null) return false;
    else if (getClass() != obj.getClass()) return false;
    Stack<?> other = (Stack<?>) obj;
    return container.equals(other.container);
}

public int compareTo (Stack<?> other) {
    return Integer.compare(size(), other.size());
}

public class EmptyStackException extends Exception {
    public EmptyStackException (String msg) {
        super(msg));
    }
}
```

```

public interface User {
    String getUsername();
}

public class NormalUser implements User {
    private String nome;
    private String cognome;
    private Set<User> amici;

    public NormalUser (String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
        this.amici = new HashSet<>();
    }

    public void addFriend (User u) {
        if (this.equals(u))
            throw new SocialNetworkException(" ");
        amici.add(u);
    }

    public void follow (PremiumUser p) {
        if (p.follower.contains(this))
            throw SocialNetworkException(" ");
        p.addFollower(this);
    }

    public String getUsername() {
        return nome + cognome + amici.size();
    }

    public boolean equals (Object obj) {
        if (this == obj) return true;
        else if (obj == null) return false;
        else if (getClass() != obj.getClass()) return false;
        NormalUser other = (NormalUser) obj;
        return nome.equals(other.nome) &amp; cognome.equals(other.cognome);
    }

    public String toString () {
        return this.getUsername();
    }
}

```

```

public class PremiumUser implements User, Comparable<PremiumUser> {
    private String username;
    protected Set<User> follower;
    public PremiumUser(String u) {
        this.username = u;
        this.follower = new HashSet<>();
    }
    public void addFollower(User u) {
        if (follower.contains(u))
            throw new SocialNetworkException(" ");
        follower.add(u);
    }
    public boolean equals(Object obj) {
        if (this == obj) return true;
        else if (obj == null) return false;
        else if (getClass() != obj.getClass()) return false;
        PremiumUser other = (PremiumUser) obj;
        return username.equals(other.username);
    }
    public String getUsername() {
        return username;
    }
    public int compareTo(PremiumUser other) {
        return Integer.compare(follower.size(), other.follower.size());
    }
    public String toString() {
        return this.getUsername();
    }
}

public class SocialNetwork {
    private String nome;
    private Set<User> iscritti;
    public SocialNetwork(String nome) {
        this.nome = nome;
        this.iscritti = new HashSet<>();
    }
    public void addUser(User u) {
        if (iscritti.contains(u)) {
            new SocialNetworkException(" ");
        }
    }
}

```

```
    iscritti.add(u);
}

}

public class SocialNetworkException extends RuntimeException {
    public SocialNetworkException(String msg) {
        super(msg);
    }
}
```

ES. 1 25/05/22

```
template <class T>
class Inventory {
private:
    T* container;
    int dim;
    int size;
    void resize() {
        T* tmp = new T[dim*2];
        for (int i=0; i<dim; i++) {
            tmp[i] = container[i];
        }
        delete [] this->container;
        this->dim *= 2;
        this->container = tmp;
    }
public:
    Inventory() {
        this->dim = 10;
        this->container = new T[dim];
        this->size = 0;
    }
    ~Inventory() {
        delete [] this->container;
    }
    Inventory(const Inventory& other) {
        this->dim = other.dim;
        this->size = other.size;
        this->container = new T[dim];
    }
```

```

for (int i=0; i<size; i++) {
    this->container[i] = other.container[i];
}

Inventory& operator=(const Inventory& other) {
    if (this != &other) {
        delete[] this->container;
        this->dim = other.dim;
        this->size = other.size;
        this->container = new T[dim];
        for (int i=0; i<size; i++)
            this->container[i] = other.container[i];
    }
    return *this;
}

void add (T elem) {
    if (size == dim)
        resize();
    this->container[size++] = elem;
}

int count (T elem) const {
    int c=0;
    for (int i=0; i<size; i++) {
        if (container[i] == elem)
            c++;
    }
    return c;
}

T getMostCommon () const {
    T result = null;
    int c=0
    for (int i=0; i<size; i++) {
        if (count(container[i]) > c) {
            c = count(container[i]);
            result = container[i];
        }
    }
}

```

X non chiomere count 2 volte

```

T getResult() const {
    T result = container[0];
    int maxCount = count(result);
    for (int i=1; i<size; i++) {
        int currentCount = count(container[i]);
        if (currentCount > maxCount) {
            maxCount = currentCount;
            result = container[i];
        }
    }
    return result;
}

```

```
    return result;
```

```
}
```

ES. 2 23/05/22

```
public class Studente {  
    private final String nome;  
    private final String cognome;  
    private final String matricola;  
  
    public Studente (String n, String c, String mat) {  
        this.nome = n;  
        this.cognome = c;  
        this.matricola = mat;  
    }  
  
    public String toString() {  
        return "Nome: " + nome  
            + " Cognome: " + cognome;  
            + " Matricola: " + matricola;  
    }  
  
    public boolean equals (Object obj) {  
        if (this == obj) return true;  
        else if (obj == null) return false;  
        else if (getClass() != obj.getClass()) return false;  
        Studente other = (Studente) obj;  
        return matricola.equals (other.matricola);  
    }  
}  
  
public class Esame {  
    private Set<Studente> iscritti;  
    private Set<Studente> verbaliizzanti;  
  
    public Esame () {  
        this.iscritti = new HashSet<>();  
        this.verbaliizzanti = new HashSet<>();  
    }  
  
    public void iscrivi (Studente s) throws StudenteGiàIscrittoException {  
        if (iscritti.contains(s))  
            throw new StudenteGiàIscrittoException ("");  
    }
```

```

    iscritti.add(s);
}

public void verbalizza(Studente s, int voto) throws StudenteNonIscrittoException, StudenteGiàVerbalizzatoException {
    try {
        Verbalizzazione v = new Verbalizzazione(s, voto);
    } catch (RuntimeException e) {
        System.out.println(e);
    }
    if (!iscritti.contains(s))
        throw new StudenteNonIscrittoException("!");
    if (verbalizzati.contains(s))
        throw new StudenteGiàVerbalizzatoException("!");
}
}

```

ES. 2 30/06/13

```

public abstract class ElementoBibliografico {
    private String titolo;
    public ElementoBibliografico(String t) {
        this.titolo = t;
    }
    public abstract float getCosto();
    public boolean equals(Object obj) {
        if (this == obj) return true;
        else if (obj == null) return false;
        else if (getTitolo() != obj.getTitolo()) return false;
        ElementoBibliografico other = (ElementoBibliografico) obj;
        return titolo.equals(other.titolo);
    }
}

public class Libro extends ElementoBibliografico {
    private final String autore;
    private final float costolibro;
    public Libro(String t, String a, float c) {
        super(t);
        this.autore = a;
        this.costolibro = c;
    }
}

```

```
public float getCosto() {
    return costoLibro;
}

public boolean equals (object obj) {
    if (!super.equals(obj)) return false;
    Libro other = (Libro) obj;
    return autre.equals (other.autre);
}

?

public class DVD extends ElementoBibliografico {
    public final String regista;
    public final float costoDVD;
    public DVD (String r, String r, float c) {
        super();
        this.regista = r;
        this.costoDVD = c;
    }

    public float getCosto() {
        return costoDVD;
    }

    public boolean equals (object obj) {
        if (!super.equals(obj)) return false;
        DVD other = (DVD) obj;
        return regista.equals (other.regista);
    }
}

public class Biblioteca implements Comparable <Biblioteca> {
    private Set <ElementoBibliografico> disponibili;
    private Set <ElementoBibliografico> prestati;
    public Biblioteca () {
        this.disponibili = new HashSet<>();
        this.prestati = new HashSet<>();
    }

    public void add (ElementoBibliografico e) {
        disponibili.add (e);
    }
}
```

```
public void presta (ElementoBibliografico e) throws BibliotecaException {
    if (!disponibili.contains(e)) throw new BibliotecaException(" ");
    disponibili.remove(e);
    prestati.add(e);
}

public void restituisce (ElementoBibliografico e) throws BibliotecaException {
    if (!prestati.contains(e)) throw BibliotecaException(" ");
    prestati.remove(e);
    disponibili.add(e);
}

public int compareTo (Biblioteca b) {
    return Integer.compare (this.prestati.size(), b.prestati.size());
}

}

public class BibliotecaException extends Exception {
    public BibliotecaException (String msg) { super (msg); }
}
```

