1)Scrivi una classe ENUM + test (Direzione)

```java
public enum Direzione {

    NORD(90) {
        @Override public Direzione opposta() { return SUD; }
    },
    EST(0) {
        @Override public Direzione opposta() { return OVEST; }
    },
    SUD(270) {
        @Override public Direzione opposta() { return NORD; }
    },
    OVEST(180) {
        @Override public Direzione opposta() { return EST; }
    };

    private final int grado;

    private Direzione(int grado) {
        this.grado = grado;
    }

    public int getGrado() {
        return this.grado;
    }

    public abstract Direzione opposta();  //return la direzione oppo
}
```

```java
public class TestEsame {

    @Test
    public void test() {
        assertSame(Direzione.SUD, Direzione.NORD.opposta());
        assertSame(Direzione.NORD, Direzione.SUD.opposta());
        assertSame(Direzione.OVEST, Direzione.EST.opposta());
    }

    @Test
    public void testGradi() {
        assertEquals(90, Direzione.NORD.getGrado());
        assertEquals(270, Direzione.SUD.getGrado());
        assertEquals(0 , Direzione.EST.getGrado());
    }
}
```

1.2) Scrivi un metodo che somma tutti gli elementi positivi di una lista + test

```java
public class TestLista {

    private List<Integer> lista = new ArrayList<>();

    public int contaPositivi(List<Integer> lista ) {
        int cont=0;
        for(Integer i : lista) {
            if(i>0)
                cont=cont+i.intValue();
        }
        return cont;
    }

    @Test
    public void test() {
        lista.add(new Integer(-1));
        lista.add(new Integer(-2));
        lista.add(new Integer(3));
        lista.add(new Integer(-3));
        lista.add(new Integer(4));
        lista.add(new Integer(6));
        assertEquals(6, lista.size());
        assertEquals(13, this.contaPositivi(lista));
    }
}
```

2)Crea una lista di Studenti (ordinati per MATRICOLA) e di Persone (ordinate per NOME) con comparatore ESTERNO.

```java
public class Persona{

    private String nome;

    public Persona(String nome) {
        this.nome=nome;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    @Override
    public String toString() {
        return this.getNome();
    }
}

public class Studente extends Persona{

    private int matricola;

    public Studente(String nome, int matricola) {
        super(nome);
        this.matricola = matricola;
    }
    public int getMatricola() {
        return matricola;
    }
    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }
}
```

```java
import java.util.Comparator;

public class Comparatore<T> implements Comparator<Persona>{

    @Override
    public int compare(Persona o1, Persona o2) {
        return o1.getNome().compareTo(o2.getNome());
    }

}
```

```java
import static org.junit.Assert.assertEquals;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

public class SommaIndiciPariSomma {

    private List<Persona> lista;

    @Before
    public <T> void setUp() {
        this.lista= new ArrayList<Persona>();
        this.lista.add(new Persona("Mario"));
        this.lista.add(new Persona("Giacomo"));
        this.lista.add(new Studente("Antonio", 123));
        Collections.sort(this.lista, new Comparatore<T>() );
    }

    @Test
    public void test() {
        assertEquals("Antonio", this.lista.get(0).getNome());
        assertEquals("Giacomo",this.lista.get(1).getNome());
        assertEquals("Mario", this.lista.get(2).getNome());
    }

}
```

3)uguale ma con ordinamento INTERNO

```java
package esame2020;

public class Persona implements Comparable<Persona>{

    private String nome;

    public Persona(String nome) {
        this.nome=nome;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public int compareTo(Persona p) {
        return this.getNome().compareTo(p.getNome());

    }
}
```

```java
import static org.junit.Assert.assertEquals;

public class SommaIndiciPariSomma {

    private List<Persona> lista;

    @Before
    public void setUp() {
        this.lista= new ArrayList<Persona>();
        this.lista.add(new Persona("Mario"));
        this.lista.add(new Persona("Giacomo"));
        this.lista.add(new Persona("Antonio"));
        Collections.sort(this.lista);
    }

    @Test
    public void test() {
        assertEquals("Antonio", this.lista.get(0).getNome());
        assertEquals("Giacomo",this.lista.get(1).getNome());
        assertEquals("Mario", this.lista.get(2).getNome());
    }

}
```

3.1) con Generics?

Un generics è uno strumento che permette la definizione di un tipo parametrizzato, che viene esplicitato successivamente in fase di compilazione secondo le necessità; i generics permettono di definire delle astrazioni sui tipi di dati definiti nel linguaggio.

```
class ComparatoreNome<T extends Comparable<T>> implements Comparator<T> {

    @Override
    public int compare(T a, T b) {
        return a.compareTo(b);
    }
}
```

```
public class Persona implements Comparable<Persona>{

    private String nome;

    public Persona(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public int compareTo(Persona o) {
        return this.getNome().compareTo(o.getNome());
    }
}
```

```
public class Studente extends Persona {

    private String matricola;

    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }

    public String getMatricola() {
        return matricola;
    }

    public void setMatricola(String matricola) {
        this.matricola = matricola;
    }
}
```

```
public class PersonaTest {


    @Test //nomi uguali ma tipo diverso in Lista larga
    public void testListaDueOmonimiTipoDiversoListaLarga() {
        List<Persona> persone = new ArrayList<>();
        Persona pers = new Persona("Paolo");
        Studente stud = new Studente("Rita", "4564");
        Studente stud2 = new Studente("Antonio", "1234");
        persone.add(pers);
        persone.add(stud);
        persone.add(stud2);

        assertTrue(persone.size()==3);
        Collections.sort(persone, new ComparatoreNome<Persona>());

        for(Persona p : persone) {
            System.out.println(p.getNome() + p.getClass());
        }
    }
}
```

- Definizione di una classe generica:

```
public class Gen<X,Y> {

      private final X var1;
      private final Y var2;

      public Gen(X x,Y y) {
        var1 = x;
        var2 = y;
      }

      public X getVar1() {
        return var1;
      }

      public Y getVar2() {
        return var2;
      }

      public String toString() {
        return "(" + var1 + ", " + var2 + ")";
      }
    }
```

```
Gen<String, String> esempio1 = new Gen<String, String>("esempio", "uno");
Gen<String, Integer> esempio2 = new Gen<String, Integer>("esempio", 2);
```

4) Esempio di classe nidificata + test

Classe INTERNA:
- non può esistere senza la classe esterna.
- La sua visibilità dipende dalla sua definizione (es. private non può essere utilizzata all' esterno dalla classe Esterna)
- Può accedere alle var di istanza e ai metodi della classe Esterna anche se privati (non servono getter & setter).
- Se la classe Interna dichiara una var con lo stesso nome di una var della classe Esterna, quest'ultima risulterà non più accessibile (*Shadowing*) -> per accedere a quella esterna bisogna scrivere *Esterna.this.var*

```
public class OuterClass {

        private int x = 7;

        public void creaInnerClass() {
            InnerClass in = new InnerClass ();
            in.stampaX();
        }

        class InnerClass {
            public void stampaX () {
                System.out.println("Il valore di x è " + x);
            }
        }
    }
```

```
- Per chiamarla:

Esterna oggetto = new Esterna();
Esterna.Interna oggettoInterno = oggetto.new Interna();
```

```
public static void main(String[] args) {
    OuterClass mo = new OuterClass ();
    OuterClass.InnerClass inner = mo.new InnerClass();
    inner.stampaX();
}
```

- Per ottenere un riferimento all'istanza dell'*InnerClass* all'interno di quest'ultima è possibile utilizzare **this.**
- Per avere un riferimento "outer this" all'interno dell'*Inner Class*, si utilizza **NomedellOuterClass.this**

```
class ClasseEsterna {

    private int x = 5;

    public  void creaInnerClass() {
        ClasseInterna in = new ClasseInterna ();
        in.stampaX();
    }

    class ClasseInterna {
        public void stampaX() {
            System.out.println("Il valore di x è " + x);
            System.out.println("Il riferimento della Inner Class " + this);
            System.out.println("Il riferimento della Outer Class " + ClasseEsterna.this);
        }
    }
```

Classe ANONIMA:
- classe usa e getta, a volte risulta eccessivo anche definire una nuova classe
- viene definita in un'unica espressione
- solitamente definita per implementare supertipi già noti (Interface-ClasseAstratta-SuperClasse)

```
Class A {

    void metodoDiTest() {

        B b = new B();
        b.metodo(new In() {

            public void faiQualcosa() {
                System.out.println("fatto");
            }
        });
    }
}
```

```
Comparator<Studente> comp = new Comparator<Studente>() {

    @Override
    public int compare(Studente arg0, Studente arg1) {
        return arg0.getNome().compareTo(arg1.getNome());
    }
};
```

Classe LOCALE: (usa e getta)

- viene definita all'interno di un metodo e non può essere utilizzata al di fuori.
- l'oggetto della ClasseInterna non può usare le variabili locali del metodo in cui è inserita l'Inner Class a meno che questi non siano dichiarati FINAL! (in questo modo si crea una copia della variabile per non andare a modificare quella iniziale)
- Può accedere alle variabili di istanza.

```
class ClassEsterna
{
    private String x = "Classe Esterna";

    public void metodo()
    {
        String z = "variabile locale";

        class ClasseInterna
        {
            public void stampaXZ()
            {
                System.out.println("Il valore di x è " + x);
                System.out.println("La variabile locale è " + z); // Non compila!
            }
        }
```

```
// metodo che ordina per intero
public static Map<String, Integer> classificaPerOccorrenze(final Map<String, Integer> mappa) {

    class ComparatoreOccorrenze implements Comparator<String> {

        private Map<String, Integer> map;

        public ComparatoreOccorrenze(final Map<String, Integer> mappa) {
            this.map = mappa;
        }

        @Override
        public int compare(String o1, String o2) {
            if(-map.get(o1) + map.get(o2)==0)
                return o2.compareTo(o1);
            return -map.get(o1) + map.get(o2);
        }
    }

    ComparatoreOccorrenze comp = new ComparatoreOccorrenze(mappa);
    SortedMap<String, Integer> risultato = new TreeMap<>(comp);
    risultato.putAll(mappa);
    return risultato;

}
```

Classe STATICA:

```
class A {

    static class ClasseAnnidata1 {

        void stampaQualcosa() {
            System.out.println("stampa annidata 1");
        }
    }
}
class B {

    static class ClasseAnnidata2 {

        void stampaQualcosa2() {
            System.out.println("stampa annidata 2");
        }
    }

    public static void main(String[] args) {

        A.ClasseAnnidata1 n = new A.ClasseAnnidata1();
        n.stampaQualcosa();
        ClasseAnnidata2 b2 = new ClasseAnnidata2();
        b2.stampaQualcosa2();
    }
}
```

- sono dichiarate all'interno di altre classi usando il modificatore STATIC.
- a differenza della altre NON sono istanziate a partire da un oggetto della classe Esterna.
- possono essere create:

- dall'esterno della classe della classe stessa:
Esterna.Interna ogg = new Esterna.Interna();

- dall'interno della classe della classe stessa:
Interna ogg1 = new Interna();

5) Metodo che somma gli elementi di <u>posizione pari</u> di una lista di interi + test

SommaIndiciPariSomma.java

```
10
11 public class SommaIndiciPariSomma {
12
13     private List<Integer> lista;
14
15     @Before
16     public void setUp() {
17         this.lista= new ArrayList<Integer>();
18     }
19
20     public int contaIndiciPositivi(List<Integer> lista) {
21         int somma=0;
22         for(int i=0; i<lista.size();i++) {
23             somma = somma + lista.get(i);
24             i++;
25         }
26         return somma;
27     }
28
29
30     @Test
31     public void test() {
32         lista.add(new Integer(1));
33         lista.add(new Integer(1));
34         lista.add(new Integer(1));
35         lista.add(new Integer(1));
36         lista.add(new Integer(1));
37         lista.add(new Integer(1));
38
39         int somma = contaIndiciPositivi(lista);
40         assertEquals(3, somma);
41     }
42
43 }
```

1)Lista di oggetti Persona ordinati per nome

Persona.java | SommaIndiciPariSomma.java

```
1  package esame2020;
2
3  import static org.junit.Assert.assertEquals;
12
13 public class SommaIndiciPariSomma {
14
15     private List<Persona> lista;
16
17     @Before
18     public void setUp() {
19         this.lista= new ArrayList<Persona>();
20         this.lista.add(new Persona("Mario"));
21         this.lista.add(new Persona("Giacomo"));
22         this.lista.add(new Persona("Antonio"));
23         Collections.sort(this.lista, new Comparator<Persona>(){
24
25             @Override
26             public int compare(Persona p1, Persona p2) {
27                 return p1.getNome().compareTo(p2.getNome());
28             }
29
30         });
31
32     }
33
34
35
36     @Test
37     public void test() {
38         assertEquals("Antonio", this.lista.get(0).getNome());
39         assertEquals("Giacomo",this.lista.get(1).getNome());
40         assertEquals("Mario", this.lista.get(2).getNome());
41     }
42
```

2) scrivi 2 esempi di codice che sollevino (usando javadoc) : - unchecked exception - checked exception

- checked: esprimono eccezioni che si riferiscono a condizioni recuperabili e che quindi possono essere gestite dal metodo chiamante;
- unchecked: esprimono condizioni non recuperabili, generalmente dovute ad errori di programmazione e quindi non gestibili dal metodo chiamante.

[se non vengono gestite le eccezioni la JVM termina con la stampa stack-trace e messaggio di errore]

1) CHECKED
Esempio di eccezioni java del primo tipo sono IOException e FileNotFoundException.
Il client è in grado di gestirli ad esempio prospettandole all'utente che può quindi intervenire. Se non vengono gestiti non compila!

```
public class UserNotFoundException extends Exception {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

- Per creare una eccezione di questo tipo è necessario estendere java.lang.Exception.

Per questo tipo di eccezioni il compilatore richiede la loro gestione o attraverso l'utilizzo di un blocco *try...catch* oppure differendo la gestione al metodo chiamante attraverso la clausola *trows*.

```
public class UserService {

    public User findUser(String username) throws UserNotFoundException {
        if ( "".equals( username ) ) {
            throw new UserNotFoundException("User does not exists!");
        }
        return new User( username );
    }

    public static void main(String[] args) {
        UserService service = new UserService();
        try {
            User user = service.findByName("");
        } catch (UserNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

(THROW crea un oggetto eccezione)

2) UNCHECKED
Esempi di eccezioni del secondo tipo sono, invece, NullPointerException o IndexOutOfBoundsException.
Il client non può fare nulla per gestire l'eccezione che può essere eliminata solamente programmaticamente, ad esempio testando i valori nulli prima di utilizzarli.
- Per creare una eccezione unchecked è necessario estendere java.lang.RuntimeException.

```
public class TooManyDataException extends RuntimeException {
    public TooManyDataException(String message) {
        super(message);
    }
}
```

Per queste tipologie di eccezione il compilatore non ne richiede la gestione, l'utilizzo del blocco try...catch è infatti opzionale.

```
public class UserService {
    public void analyze(List<User> users) {
        if ( users.size() > 50 ) {
            throw new TooManyDataException("Too many users in the list");
        }
    }

    public static void main(String[] args) {
        UserService service = new UserService();

        // Create 100 users
        List<User> users = new ArrayList<>( Collections.nCopies( 100, new User() ) );

        obj.analyze(users);
    }
}
```

2.1) Scrivi la classe Studente e Persona con hashCode e Equals <u>senza ripetizione di codice:</u>

```java
public class Persona{

    private String nome;

    public Persona(String nome) {
        this.nome=nome;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    @Override
    public String toString() {
        return this.getNome();
    }

    @Override
    public boolean equals(Object o) {
        Persona that = (Persona)o;
        return this.getNome().equals(that.getNome());
    }
    @Override
    public int hashCode() {
        return this.getNome().hashCode();
    }
}
```

```java
public class Studente extends Persona{

    private int matricola;

    public Studente(String nome, int matricola) {
        super(nome);
        this.matricola = matricola;
    }

    public int getMatricola() {
        return matricola;
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    @Override
    public boolean equals(Object o) {
        Studente that = (Studente)o;
        if(super.equals(o))   //se hanno lo stesso nome confronta matricola
            return this.getMatricola()==that.getMatricola();
        return false;  //se non hanno lo stesso nome sono diversi
    }
    @Override
    public int hashCode() {
        return this.matricola;
    }
}
```

3) Come la 2. ma usa Collections.max

Throws:
ClassCastException – se la collezione contiene elementi che non sono mutualmente comparabili
NoSuchElementException – se la collezione è vuota.
NullPointerException – se sto chiamando su una lista nulla (dichiarata ma non inizializzata)

3.2) cosa cambia se all'interno di Collections.max(…) passo una variabile inizializzata vuota o Collections.emptyList()? e Collections.EMPTYLIST?

-- **Collections.EMPTY_LIST** ritorna una raw-list (NOT SAFE)
-- **Collections.emptyList()** ritorna una List<T>

```java
List<String> collection1 = Collections.EMPTY_LIST;  // Noncompliant
```

- **<u>Collections.emptyList():</u>** - ritorna una lista vuota di natura IMMUTABILE.

Collections.emptyList() è stato aggiunto da java 1.5 ed è SEMPRE PREFERIBILE, in questo modo non devi necessariamente fare operazioni di cast.

Per ottenere un tipo sicuro si chiama:

```java
List<String> collection1 = Collections.emptyList();
Collections.<String>emptyList();
```

Throws:
java.lang.UnsupportedOperationException (poiché immutabili non posso svolgere .add)

- **<u>Collections.EMPTY_LIST</u>** – ha le stesse proprietà ma ritorna un raw-type (NON USARLO NON E' SICURO!!)

```java
List collection2 = Collections.EMPTY_LIST;
```

4) scrivere una lista di strumenti per poi ordinarla tramite un comparatore di chitarra dove chitarra è un sottotipo di strumenti(interfaccia)

```
public class Chitarra implements Strumento, Comparable<Chitarra>{

    private int modello;

    public Chitarra(int modello) {
     this.modello = modello;
    }

    public int getModello() {
     return modello;
    }

    public static List<Strumento> ordina(List<Strumento> lista) {
     Collections.sort(lista, new Comparator<Strumento>() {

      @Override
      public int compare(Strumento o1, Strumento o2) {
      Chitarra c1 = (Chitarra)o1;
      Chitarra c2 = (Chitarra)o2;
      return c1.compareTo(c2);
      }
     });

     return lista;
    }

    @Override
    public int compareTo(Chitarra o) {
     return this.getModello()-o.getModello();
    }
   }
```

- Comparare diversi Strumenti di una lista List<Strumento> per nome:

```
public interface Strumento {

    public int getModello();
}

public class Chitarra implements Strumento {

    private int modello;

    public Chitarra(int modello) {
        this.modello = modello;
    }

    public int getModello() {
        return modello;
    }
}
public class Violino implements Strumento {

    private int modello;

    public Violino(int modello) {
        this.modello = modello;
    }

    public int getModello() {
        return modello;
    }
```

```
public class ChitarraTest {

    private Strumento c1;
    private Strumento c2;
    private Strumento v;
    private List<Strumento> lista;

    @Before
    public void setUp() throws Exception {
        c1 = new Chitarra(123);
        c2 = new Chitarra(999);
        v = new Violino(345);
        lista = new LinkedList<>();
    }

    @Test
    public void test() {
        lista.add(c2);
        lista.add(c1);
        lista.add(v);
        Collections.sort(lista, new Comparator<Strumento>() {

            @Override
            public int compare(Strumento o1, Strumento o2) {
                return o1.getModello()-o2.getModello();
            }
        });

        assertEquals(3, lista.size());
        assertEquals(c1, lista.get(0));
        assertEquals(v, lista.get(1));
        assertEquals(c2, lista.get(2));

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

1) Esame giugno 19

1.1) Principio di sostituzione:
oggetti dichiarati in un SuperTipo possono essere sostituiti con oggetti di un suo SottoTipo senza alterare la correttezza dei risultati del programma.

2) Crea una classe Persona con nome e età.
Scrivere u metodo che data una lista di persone restituisce una map<Integer,Set<Persone>>

*me.java *Persona.java

```
private int eta;
private String nome;

public Persona(int eta, String nome) {
    this.nome=nome;
    this.eta=eta;
}

public int getEta() {
    return eta;
}

public void setEta(int eta) {
    this.eta = eta;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public Map<Integer, List<Persona>> eta2persone (List<Persona> lista){
    Map<Integer, List<Persona>> eta2persone = new HashMap<>();
    for(Persona p : lista) {
        if(eta2persone.containsKey(p.getEta())) {
            eta2persone.get(p.getEta()).add(p);
        }
        else {
            List<Persona> nuova = new ArrayList<>();
            nuova.add(p);
            eta2persone.put(new Integer(p.getEta()), nuova);
        }
    }
    return eta2persone;
}
```

```
public class TestEsame {

    private Persona p1 = new Persona(32,"Antonio");
    private Persona p2 = new Persona(32,"Gianni");
    private Persona p3 = new Persona(12,"Antonio");

    private List<Persona> persone = new ArrayList<>();
    private Map<Integer, List<Persona>> mappa = new HashMap<>();

    @Test
    public void test() {
        persone.add(p1);
        persone.add(p2);
        persone.add(p3);

        mappa = p1.eta2persone(persone);
        assertEquals(2, mappa.keySet().size());
        assertEquals(2, mappa.get(new Integer(32)).size());
        assertTrue(mappa.containsKey(new Integer(32)));
        assertTrue(mappa.containsKey(new Integer(12)));
    }
}
```

3) Data una stringa scrivere un metodo che calcoli per ogni parola quante volte occorre + test

```
public class TestEsame {

    private String stringa;

    private Map<String, Integer> ordinaOcc(String stringa) {
        Map<String, Integer> mappa = new HashMap<>();
        String[] split = stringa.split(" ");
        for(String s : split) {
            if(mappa.containsKey(s)) {
                mappa.put(s, mappa.get(s)+1);
            }
            else{
                mappa.put(s, new Integer(1));
            }
        }
        return mappa;
    }
```

```
@Test
public void test() {

    stringa = "ciao mondo poo ciao mondo";
    Map<String, Integer> mappatest = this.ordinaOcc(stringa);
    assertEquals(3, mappatest.keySet().size());
    assertTrue(mappatest.containsKey("ciao"));
    assertTrue(mappatest.containsKey("poo"));
    assertTrue(mappatest.containsKey("mondo"));
    String ciao = "ciao";
    assertEquals(2, mappatest.get(ciao).intValue());
```

4) se volessi ordinarle per numero di occorrenze?

// .split

```java
public class TestEsame {

    private String stringa;

    private Map<String, Integer> ordinaOcc(String stringa) {
        Map<String, Integer> mappa = new HashMap<>();
        String[] split = stringa.split(" ");
        for(String s : split) {
            if(mappa.containsKey(s)) {
                mappa.put(s, mappa.get(s)+1);
            }
            else{
                mappa.put(s, new Integer(1));
            }
        }
        SortedMap<String, Integer> mappaOrdinata = new TreeMap<>(new ComparatoreValori(mappa));
        mappaOrdinata.putAll(mappa);
        return mappaOrdinata;
    }

    @Test
    public void test() {
        stringa = "ciao ciao mondo poo poo poo";
        assertEquals(this.ordinaOcc(stringa).keySet().size(), 3);
        String ciao = "ciao";
        String mondo = "mondo";
        String poo = "poo";
        assertEquals(this.ordinaOcc(stringa).get(ciao), new Integer(2));
        assertEquals(this.ordinaOcc(stringa).get(mondo), new Integer(1));
        assertEquals(this.ordinaOcc(stringa).get(poo), new Integer(3));
    }
}
```

```java
public class ComparatoreValori implements Comparator<String> {

    private Map<String, Integer> mappa;

    public ComparatoreValori(Map<String, Integer> mappa) {
        this.mappa = mappa;
    }
    @Override
    public int compare(String o1, String o2) {
        if( this.mappa.get(o1).intValue()- this.mappa.get(o2).intValue()==0) {
            return o1.compareTo(o2);
        }
        return this.mappa.get(o1).intValue()- this.mappa.get(o2).intValue();
    }
```

//Metodo con SCANNER:

//scanner

```java
public class splitStringhe {

    public static Map<String, Integer> contaOccorrenze(String string) {

        Scanner scanner = new Scanner(string);
        Map<String, Integer> parola2occorrenze = new HashMap<>();

        while(scanner.hasNext()) {
            String parola = scanner.next().toLowerCase();
            if(parola2occorrenze.containsKey(parola)) {
                parola2occorrenze.put(parola, parola2occorrenze.get(parola)+1);
            } else {
                parola2occorrenze.put(parola, 1);
            }
        }
        return parola2occorrenze;
    }
```

```java
public static Map<String, Integer> classificaPerOccorrenze(final Map<String, Integer> mappa) {

    class ComparatoreOccorrenze implements Comparator<String>{

        private Map<String, Integer> mappa;

        public ComparatoreOccorrenze(Map<String, Integer> mappa) {
            this.mappa = mappa;
        }
        public int compare(String o1, String o2) {
            return mappa.get(o1).compareTo(mappa.get(o2));
        }
    }
    SortedMap<String, Integer> risultato = new TreeMap<>(new ComparatoreOccorrenze(mappa));
    risultato.putAll(mappa);
    return risultato;
}
```

5) Scrivi un metodo che data una lista di persone ritorna una mappa Map<String, Set<Persona>>

```java
public class Persona {

    private String nome;

    public Persona(String nome) {
        this.nome=nome;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public static Map<String, Set<Persona>> occorrenze(List<Persona> persone){
        Map<String, Set<Persona>> mappa = new HashMap<>();
        for(Persona p : persone) {
            if(mappa.containsKey(p.getNome())) {
                mappa.get(p.getNome()).add(p);
            }
            else {
                mappa.put(p.getNome(), new HashSet<Persona>());
                mappa.get(p.getNome()).add(p);
            }
        }
        return mappa;
    }
```

```java
public class TestEsame {

    private Persona p;
    private Persona p1;
    private Persona p2;
    private List<Persona> lista;

    @Before
    public void setUp() {
        p = new Persona("Gianni");
        p2 = new Persona("Gianni");
        p1= new Persona("Mario");

        lista = new ArrayList<>();
        lista.add(p);
        lista.add(p1);
        lista.add(p2);
    }

    @Test
    public void test() {
        Map<String, Set<Persona>> mappa = Persona.occorrenze(lista);
        assertEquals(2, mappa.keySet().size());
        assertEquals(2, mappa.get(p.getNome()).size());
        assertTrue(mappa.containsKey(p1.getNome()));
        assertTrue(mappa.containsKey(p.getNome()));

    }
}
```

5.1) Utilizza una interface per la mappa ordinata e ordinarla secondo le occorrenze:

```java
  public static Map<String, Set<Persona>> occorrenze(List<Persona> persone){
      final Map<String, Set<Persona>> mappa = new HashMap<>();
      for(Persona p : persone) {
          if(mappa.containsKey(p.getNome())) {
              mappa.get(p.getNome()).add(p);
          }
          else {
              mappa.put(p.getNome(), new HashSet<Persona>());
              mappa.get(p.getNome()).add(p);
          }
      }

      class ComparatoreMappa implements Comparator<String>{

          @Override
          public int compare(String o1, String o2) {
              if(mappa.get(o1).size()-mappa.get(o2).size()==0)
                  return o1.compareTo(o2);
              return mappa.get(o1).size()-mappa.get(o2).size();
          }

      }

      SortedMap<String, Set<Persona>> nuova = new TreeMap<>(new ComparatoreMappa());
      nuova.putAll(mappa);
      return nuova;
  }
```

1) Scrivi un test su Collections.max

```java
public class TestEsame {

    private List<Integer> listavuota = new LinkedList<>();
    private List<Integer> listaNulla;

    @Test (expected=NoSuchElementException.class)
    public void test_Vuoto() {
        Collections.max(listavuota);
    }

    @Test (expected=NullPointerException.class)
    public void test_Nullo() {
        Collections.max(listaNulla);
    }
}
```

2)Scrivi una classe ENUM con i giorni della settimana e metodo polimorfo che torna il giorno successivo

```
public enum Giorni {

    LUNEDI(){
        @Override
        public Giorni successivo() {
            return MARTEDI;
        }
    },
    MARTEDI(){
        @Override
        public Giorni successivo() {
            return MERCOLEDI;
        }
    },
    MERCOLEDI(){
        @Override
        public Giorni successivo() {
            return GIOVEDI;
        }
    },
    GIOVEDI(){
        @Override
        public Giorni successivo() {
            return VENERDI;
        }
    },
    VENERDI(){
        @Override
        public Giorni successivo() {
            return SABATO;
        }
    },
    SABATO(){
        @Override
        public Giorni successivo() {
            return DOMENICA;
        }
    },
    DOMENICA(){
        @Override
        public Giorni successivo() {
            return LUNEDI;
        }
    };

    public abstract Giorni successivo();
```

```
public class TestEsame {

    @Test
    public void test() {
        assertSame(Giorni.DOMENICA.successivo(), Giorni.LUNEDI);
        assertSame(Giorni.LUNEDI.successivo(), Giorni.MARTEDI);
        assertSame(Giorni.MARTEDI.successivo(), Giorni.MERCOLEDI);
        assertSame(Giorni.MERCOLEDI.successivo(), Giorni.GIOVEDI);
        assertSame(Giorni.GIOVEDI.successivo(), Giorni.VENERDI);
    }

}
```

2.1) Come fare per evitare tutti gli override?

```
public enum Giorni {

    LUNEDI(1),
    MARTEDI(2),
    MERCOLEDI(3),
    GIOVEDI(4),
    VENERDI(5),
    SABATO(6),
    DOMENICA(7);

    private int giorno;

    public int getGiorno() {
        return this.giorno;
    }

    private Giorni(int giorno) {
        this.giorno=giorno;
    }

    public int successivo() {
        if(this.equals(DOMENICA))
            return LUNEDI.getGiorno();
        return this.giorno+1;
    }
}
```

```
public class TestEsame {

    @Test
    public void test() {
        assertEquals(Giorni.LUNEDI.successivo(), Giorni.MARTEDI.getGiorno());
        assertEquals(Giorni.MARTEDI.successivo(), Giorni.MERCOLEDI.getGiorno());
        assertEquals(Giorni.MERCOLEDI.successivo(), Giorni.GIOVEDI.getGiorno());
        assertEquals(Giorni.GIOVEDI.successivo(), Giorni.VENERDI.getGiorno());
        assertEquals(Giorni.VENERDI.successivo(), Giorni.SABATO.getGiorno());
        assertEquals(Giorni.SABATO.successivo(), Giorni.DOMENICA.getGiorno());
        assertEquals(Giorni.DOMENICA.successivo(), Giorni.LUNEDI.getGiorno());
    }
```

```
//se volessi confrontare i giorni senza int
public static GiorniSettimana successivo(GiorniSettimana sett) {
      switch(sett) {
      case LUNEDI: return MARTEDI;
      case MARTEDI: return MERCOLEDI;
      case MERCOLEDI: return GIOVEDI;
      case GIOVEDI: return VENERDI;
      case VENERDI: return SABATO;
      case SABATO: return DOMENICA;
      case DOMENICA: return LUNEDI;
      default: return null;
      }

}
```

3) Scrivi una lista di studenti ordinata per nome, e a parità di nome, per Matricola

```java
public class Studente implements Comparable<Studente>{

    private String nome;
    private int matricola;

    public Studente(String nome, int matricola) {
        this.nome=nome;
        this.matricola=matricola;
    }

    public String getNome() {
        return nome;
    }
    public int getMatricola() {
        return matricola;
    }

    public static List<Studente> ordinaStud(List<Studente> lista){
        List<Studente> nuova = new LinkedList<>();
        nuova.addAll(lista);
        Collections.sort(nuova);
        return nuova;
    }

    @Override
    public int compareTo(Studente s) {
        if(this.getNome().compareTo(s.getNome())==0)
            return this.matricola-s.matricola;
        return this.getNome().compareTo(s.getNome());
    }
```

```java
public class TestEsame {

    private Studente s;
    private Studente s1;
    private Studente s2;
    private List<Studente> lista;

    @Before
    public void setUp() {
        this.lista = new ArrayList<>();
        s = new Studente("Mario",123);
        s1 = new Studente("Mario", 837);
        s2 = new Studente("Valentina",424);
        lista.add(s); lista.add(s2); lista.add(s1);
    }

    @Test
    public void test() {
        lista = Studente.ordinaStud(lista);
        assertSame(s, lista.get(0));
        assertSame(s1, lista.get(1));
        assertSame(s2, lista.get(2));
    }

}
```

3.1) Fare un metodo OrdinaPerNome nella classe Persona estesa da Studente

```java
public class Persona {

    private String nome;

    public Persona(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    static class ComparatorePersone implements Comparator<Persona> {

        @Override
        public int compare(Persona o1, Persona o2) {
            return o1.getNome().compareTo(o2.getNome());
        }
    }

    @Override
    public boolean equals(Object o) {
        if(o==null || o.getClass()!=this.getClass()) return false;
        Persona that = (Persona)o;
        return this.getNome().equals(that.getNome());
    }
    @Override
    public int hashCode() {
        return this.getClass().hashCode()+this.getNome().hashCode();
    }
}
```

```java
public class Studente extends Persona implements Comparable<Studente>{

    private int matricola;

    public Studente(String nome, int i) {
        super(nome);
        this.matricola = i;
    }

    public int getMatricola() {
        return matricola;
    }

    public int compareTo(Studente o) {
        if(this.getNome().compareTo(o.getNome())==0)
            return this.getMatricola()-(o.getMatricola());
        return this.getNome().compareTo(o.getNome());
    }

    @Override
    public boolean equals(Object o) {
        if(o==null || o.getClass()!=this.getClass()) return false;
        Studente that = (Studente)o;
        return this.getNome().equals(that.getNome()) && this.getMatricola()==that.getMatricola();
    }
    @Override
    public int hashCode() {
        return this.getClass().hashCode()+this.getNome().hashCode()+this.getMatricola();
    }
}
```

<u>Test SET:</u>

- **TreeSet<>(comp)** ordinano in base al comparatore passato per parametro
- **TreeSet<>()** ordina in base al comparatore NATURALE

```java
public Set<Persona> ordinaSetNome (Set<Persona> set){
    ComparatorePersone comp = new ComparatorePersone();
    SortedSet<Persona> ordinato = new TreeSet<>(comp);
    ordinato.addAll(set);
    return ordinato;
}

@Test
public void test() {

    set.add(new Persona("Gianni"));
    Persona gianni2 = new Persona("Gianni");
    set.add(gianni2);  //NO
    set.add(new Persona("Antonio"));
    Studente antonio = new Studente("Antonio",123);
    set.add(antonio);
    Studente antonio2 = new Studente("Antonio", 456);
    set.add(antonio2);
    set.add(new Studente("Francesco", 678));

    Set<Persona> ordinato = ordinaSetNome(set);

    assertEquals(3, ordinato.size());
    assertFalse(ordinato.contains(gianni2));
}
```

Test LISTA:

- **Collections.sort(comp)** ordina in base al comparatore passato per parametro
- **Collections.sort()** ordina in base al comp naturale

```java
public void ordinaNomi(List<Persona> lista) {
    if(!lista.isEmpty()) {
    List<Persona> ordinata = lista;
    ComparatorePersone comp = new ComparatorePersone();
    Collections.sort(ordinata, comp);
    }
}

@Test
public void test() {
    lista.add(new Persona("Gianni"));
    lista.add(new Persona("Gianni"));
    lista.add(new Persona("Antonio"));
    Studente antonio = new Studente("Antonio",123);
    lista.add(antonio);
    Studente antonio2 = new Studente("Antonio", 456);
    lista.add(antonio2);
    lista.add(new Studente("Francesco", 678));

    ordinaNomi(lista);

    assertEquals("Antonio", lista.get(0).getNome());
    assertEquals(antonio, lista.get(1));
    assertEquals(antonio2, lista.get(2));
    assertEquals("Francesco", lista.get(3).getNome());
    assertEquals("Gianni", lista.get(4).getNome());
    assertEquals("Gianni", lista.get(5).getNome());
}
```

```java
public class PersonaTest {

	@Test  //TIPO DIVERSO, NOMI UGUALI, CON HASH SET
	public void testHASHSETTipoDiversoNomiUguali() {
		Set<Persona> setPersone = new HashSet<>();
		setPersone.add(new Persona("Persona"));
		setPersone.add(new Studente("Persona", "matricola"));
		System.out.println("test1" + setPersone + "\n");
		assertTrue(setPersone.size() == 2);
				// aggiunge entrambi perchè equals e HashCode riconoscono due classi diverse
	}


	/* NON COMPILA PERCHE PERSONA NON IMPLEMENTA COMPARABLE E NON PASSO UN COMP. ESTERNO
	 * E IL TREE SET RICHIEDE PER FORZA UN COMPARATORE PER MANTENERSI ORDINATO.
	 * SI GENERA QUINDI CLASS CAST EXCEPTION */
	@Test (expected = ClassCastException.class)
	//TIPO DIVERSO, NOMI UGUALI, CON TREE SET SENZA ALCUN COMPARATORE
	public void testTreeSetNomiUgualiTipoDiverso() {
		Set<Persona> setPersone = new TreeSet<>();
		setPersone.add(new Persona("Persona"));
		setPersone.add(new Studente("Persona", "matricola"));
		System.out.println("test2" + setPersone + "\n");
		assertTrue(setPersone.size() == 2);
				//OK perchè equals e HashCode riconoscono due classi diverse
	}

	@Test //TIPO DIVERSO, NOMI UGUALI CON TREE SET LARGO E COMPARATORE ESTERNO
	public void testAggiungoUnoStudenteEUnaPersonaOmonimiTreeSetLargo() {
		Set<Persona> setPersoneOmonime = new TreeSet<>(new Persona.ComparaNomePersone());
		setPersoneOmonime.add(new Persona("Giulio"));  //c61
		setPersoneOmonime.add(new Studente("Giulio", "matricola"));  //45bf
		System.out.println("test3" + setPersoneOmonime + "\n");
		assertTrue(setPersoneOmonime.size() == 1);
		//Nel TreeSet ci sarà una sola persona: la prima,
		//Poichè l'inserimento della seconda sarà rifiutato
		//In quanto il comparatore si basa sul nome ed è già presente un omonimo.
	}

	@Test //TIPO DIVERSO, NOMI UGUALI CON TREE SET STRETTO E COMPARATORE ESTERNO
	public void testAggiungoUnoStudenteEUnaPersonaOmonimiTreeSetStretto() {
		Set<Studente> setPersoneOmonime = new TreeSet<>(new Persona.ComparaNomePersone());
		setPersoneOmonime.add(new Persona("Giulio"));  // NON COMPILA!!
		//Il compilatore non sa come rendere più specifico un qualcosa di più generico.
		setPersoneOmonime.add(new Studente("Giulio", "matricola"));
		System.out.println("test3" + setPersoneOmonime + "\n");
		assertTrue(setPersoneOmonime.size() == 1);
		//Nel TreeSet ci sarà una sola persona: la prima,
		//Poichè l'inserimento della seconda sarà rifiutato
		//In quanto il comparatore si basa sul nome ed è già presente un omonimo.
	}

	@Test //TIPO DIVERSO, NOMI DIVERSI CON TREE SET E COMPARATORE ESTERNO
	public void testAggiungoUnoStudenteEUnaPersonaNomiDiversi() {
		Persona.ComparaNomePersone comp = new Persona.ComparaNomePersone();
		Set<Persona> setPersoneOmonime = new TreeSet<>(comp);    //E' ordinato?
		setPersoneOmonime.add(new Persona("Giulio"));  //c61
		setPersoneOmonime.add(new Studente("Lucia", "matricola"));  //45bf
		System.out.println("test3" + setPersoneOmonime + "\n");
		assertTrue(setPersoneOmonime.size() == 2);
		//vengono aggiunti entrambi, poichè il comparatore si basa sul nome ed
		//hanno nome diverso. Per il principio di sostituzione posso inserire
		//uno Studente dove è prevista una Persona, poichè Studente è sottotipo di Persona
		List<Persona> setList = new ArrayList<Persona>();
		setList.addAll(setPersoneOmonime);
		assertEquals("Giulio", setList.get(0).getNome());
		assertEquals("Lucia", setList.get(1).getNome());
		//E' ordinata secondo il criterio di comparazione del comp passato come parametro.
		// ovvero è ordinato secondo il nome.

	}
```

```java
	@Test //nomi uguali, tipo diverso, HashMap
	public void testDueOmonimiTipoDiversoInMappa() {
		Map<String,
		String> nome2matricola = new HashMap<>();
		nome2matricola.put(new Persona("Paolo").getNome(), null);
		Studente stud = new Studente("Paolo", "4564");
		nome2matricola.put(stud.getNome(), stud.getMatricola());
		assertTrue(nome2matricola.keySet().size()==1);
		System.out.println("test4" + nome2matricola + "\n");
		//Ci sarà solo lo Studente Paolo che avrà sovrascritto il primo Paolo (persona)
	}

	@Test //nomi uguali ma tipo diverso in hash map
	//N.B.:  la mappa ha tipo generico Persona
	public void testMappaDueOmonimiTipoDiversoMappaChiaveTipoLargo() {
		Map<Persona, String> tipo2nome = new HashMap<>();
		Persona pers = new Persona("Paolo");
		Studente stud = new Studente("Paolo", "4564");
		tipo2nome.put(pers, pers.getNome());
		tipo2nome.put(stud, stud.getNome());
		assertTrue(tipo2nome.keySet().size()==2);
		System.out.println("test5" + tipo2nome + "\n");
		//Poiche la mappa ha come chiave il tipo e i tipi sono diversi, coesistono entrambi i Paolo
	}

	@Test //nomi uguali ma tipo diverso in hash map
	//N.B.:  la mappa ha tipo più stretto Studente
	public void testMappaDueOmonimiTipoDiversoMappaChiaveTipoStretto() {
		Map<Studente, String> tipo2nome = new HashMap<>();
		Persona pers = new Persona("Paolo");
		Studente stud = new Studente("Paolo", "4564");
		tipo2nome.put(pers, pers.getNome());   //NON COMPILA!!
		// non posso mettere un tipo piu grande al posto di uno piu stretto.
		// il compilatore non saprebbe come renderlo più specifico da largo quale è
		tipo2nome.put(stud, stud.getNome());
		assertTrue(tipo2nome.keySet().size()==2);
		System.out.println("test5" + tipo2nome + "\n");
		//Poiche la mappa ha come chiave il tipo e i tipi sono diversi, coesistono entrambi i Paolo
	}

	@Test //nomi uguali ma tipo diverso in Lista larga
	public void testListaDueOmonimiTipoDiversoListaLarga() {
		List<Persona> persone = new ArrayList<>();
		Persona pers = new Persona("Paolo");
		Studente stud = new Studente("Paolo", "4564");
		persone.add(pers);
		persone.add(stud);
		assertTrue(persone.size()==2);
		System.out.println("test5" + persone + "\n");
	}

	@Test //nomi uguali ma tipo diverso in Lista larga
	public void testListaDueOmonimiTipoDiversoListaStretta() {
		List<Studente> persone = new ArrayList<>();
		Persona pers = new Persona("Paolo");
		Studente stud = new Studente("Paolo", "4564");
		persone.add(pers); //NON COMPILA!!
		//il compilatore non sa come rendere più specifico qualcosa di più generico
		persone.add(stud);
		assertTrue(persone.size()==2);
		System.out.println("test5" + persone + "\n");
	}
}
```

4) Crea una classe Chitarra e un metodo statico in un'altra classe per ordinare le chitarre.

5) Crea una classe Studente che estende Persona e scrivere i rispettivi equals e hashcode.
Prova a fare un test con una Persona e uno Studente con lo stesso nome.

5.2) Come potrei risolvere il problema del downcasting in equals?
- aggiungo un controllo if(o==null || o.getClass()!=this.getClass()) return false;

```java
//PERSONA
@Override
public boolean equals(Object o) {
 if(o==null || o.getClass()!= this.getClass()) return false;
 Persona that = (Persona)o;
 return this.getNome().equals(that.getNome());
}

@Override
public int hashCode() {
 return this.getClass().hashCode() + this.getNome().hashCode();
}


//STUDENTE
@Override
public boolean equals(Object o) {
    Studente that = (Studente)o;
    return super.equals(o) && this.getMatricola().equals(that.getMatricola());
}

@Override
public int hashCode() {
    return super.hashCode() + this.getMatricola().hashCode();
}
```

5.3) Se imposto il metodo hashCode() return 0; cosa succede?
- se sia Studente che Persona ritornano lo stesso hashCode, solleva un ClassCastException nelle Hash (HashMap e HashSet)
- questo avviene perché gli hashCode vengono creati per allocare l'oggetto in uno spazio di memoria di una tabella hash, due oggetti con uno stesso hashCode vengono allocati nello steso spazio ed avviene un conflitto.
- Per questione di ORDINAMENTO o UGUAGLIANZA (su liste o treeset o treemap) il metodo usato principale è equals() quindi hashcode non va ad intaccare la scelta.