6) The following classes are given:

```
public class Ring { protected String owner = "who knows"; }

public class Hobbit extends Thread {
  private String name; private Ring ring;

  Hobbit (String name, Ring ring) {
    this.name = name; this.ring = ring;
  }
  public void run() {
    ring.owner = name;
    System.out.println(name + " says: the precious is owned by myself, " + ring.owner);
  }
}

public class Main {
  public static void main(String[] args) {
    Ring ring = new Ring();
    Hobbit gollum, bilbo, frodo;

    gollum = new Hobbit("Gollum",ring);
    bilbo = new Hobbit("Bilbo Baggins",ring);
    frodo = new Hobbit("Frodo Baggins",ring);

    gollum.start();
    bilbo.start();
    frodo.start();
  }
}
```

Which of the following statements about the method run() is true? (8 Points) 8

☐ The implementation of run() is thread-safe.

☒ Changing the code of the method run() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
public void run() {
  synchronized(ring) {
    ring.owner = name;
    System.out.println(name + " says: the precious is owned by myself, " + ring.owner);
  }
}
```

☐ Changing the code of the method main() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
synchronized (ring) {
    gollum = new Hobbit("Gollum",ring);
    bilbo = new Hobbit("Bilbo Baggins",ring);
    frodo = new Hobbit("Frodo Baggins",ring);
}
```

☐ Changing the code of the method main() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
gollum.run ();
bilbo.run ();
frodo.run ();
```

3) Which of the following definitions of Function can be implemented by the lambda expression given below? (4 Points)

```
Function f = () -> System.out.println("I am a cool function!");
```

☐
```
interface Function {
    void func1();
    int func2();
}
```
Funktioniert nicht, weil eine Lambda-Expression nur EINE abstrakte Methode implementieren kann, da sind aber zwei abstrakte methoden

☒
```
interface Function {
    void func1();
}
```
Gültig, weil das Interface nur eine abstrakte Methode (func1()) hat, die mit der Lambda-Expression kompatibel ist.

☐
```
abstract class Function {
    abstract void func1();
}
```
Geht nicht, weil eine Lambda-Expression keine abstrakte Klasse implementieren kann. Lambda-Expressions funktionieren nur mit funktionalen Interfaces (mit einer einzigen abstrakten Methode).

☒
```
interface Function {
    void func1();
    default int func2() {
        return 42;
    }
}
```
Gültig, weil func2() als default-Methode deklariert ist.
Lambda-Expressions ignorieren default-Methoden, da diese eine Standardimplementierung haben.
Die Lambda-Expression implementiert nur func1(), was erlaubt ist.

4) The following classes and objects are given for an Adapter Pattern:

```
public interface Vehicle { public void drive (int km); }
public class Vespa implements Vehicle { public void drive (int km) { } }
public class VespaClassic extends Vespa { public void drive (double km) { } }
public class Bike { public void ride (int km) { } }
public class BikeAdapter implements Vehicle {
    private Bike bike;
    public BikeAdapter (Bike bike) { this.bike = bike; }
    public void drive (int km) { bike.ride(km); }
}
Bike bike = new Bike();
Vehicle v = new BikeAdapter(bike);
```

Which of the following statements about the method drive is true? (4 Points)

☐ Calling bike.drive(10) works.
Die Klasse Bike hat keine drive(int km)-Methode, sondern nur ride(int km).

☒ Method drive in class VespaClassic is a case of overloading.
Da sich die Parameter unterscheiden, handelt es sich um Überladung (Method Overloading).

☒ Method drive in class BikeAdapter is a case of overriding.
BikeAdapter implementiert Vehicle, das eine drive(int km)-Methode definiert. BikeAdapter überschreibt (override) diese Methode mit einer eigenen Implementierung.

☒ Calling v.drive(10) works.
v ist vom Typ Vehicle, aber eine Instanz von BikeAdapter.
BikeAdapter implementiert drive(int km), das intern bike.ride(km) aufruft.

A 2 - Applying Knowledge (24 Points)

Note that for each wrong answer, there is a reduction of points. Multiple answers are possible.

1) The following classes and interfaces are given:

```
public class Robot { public void manageEnergy() { } }
public interface CleaningRobot { public void clean(); }
public interface DiscoveryRobot { public void search(); }

public class Eve extends Robot implements DiscoveryRobot {public void search() {} }
public class WallE extends Robot implements CleaningRobot {public void clean() {} }
```

Which of the following statements is correct? (4 Points)

- ☐ DiscoveryRobot discovery = new Robot();
- ☒ Eve eve = new DiscoveryRobot();
- ☐ WallE wallE = new CleaningRobot();
- ☐ CleaningRobot cleaning = new WallE();

2) The following method is given:

```
import java.util.function.Predicate;

void someMethod (int a, Predicate<Integer> pred) {
    System.out.println("Result: " + pred.test(a));
}
```

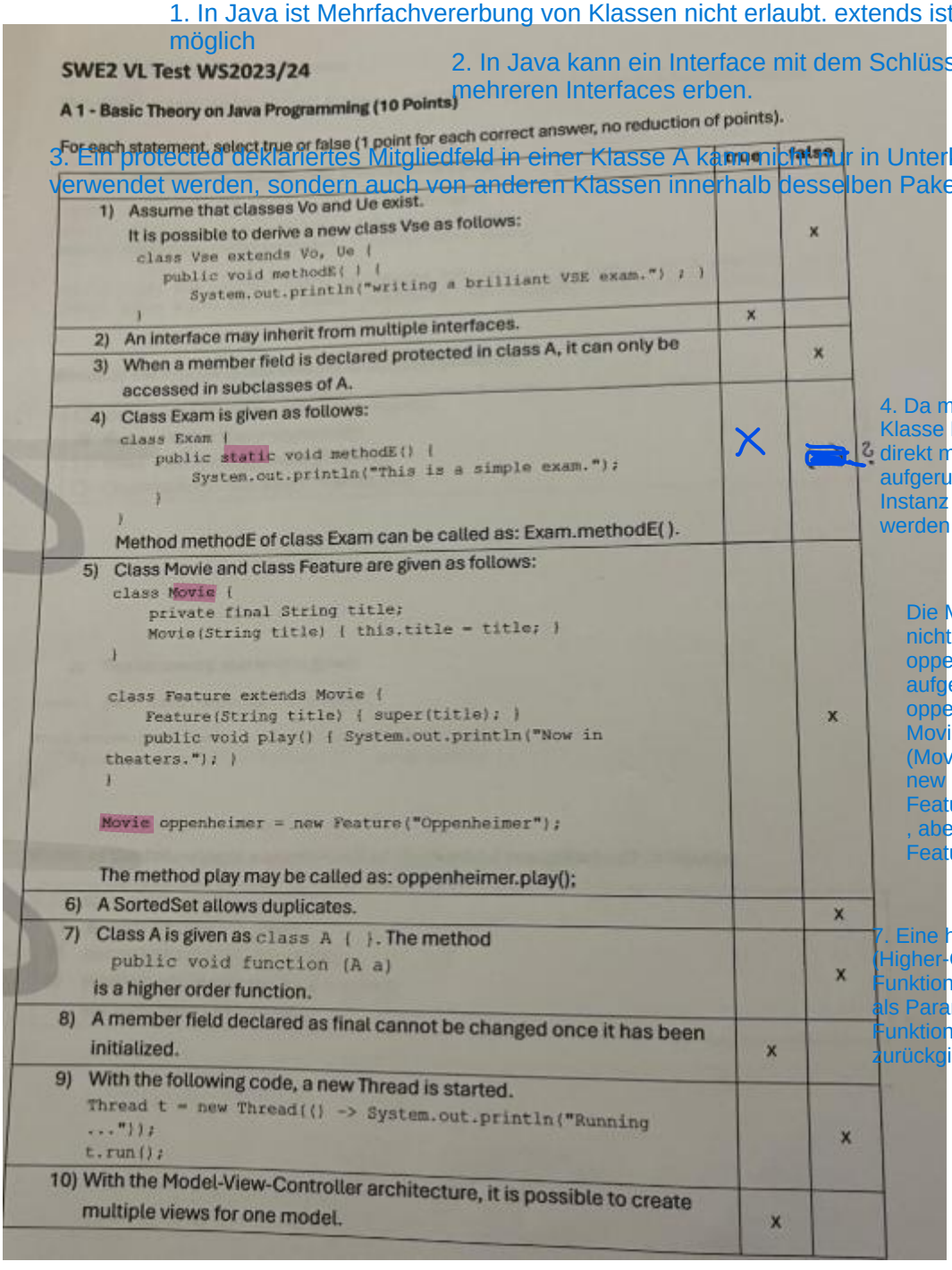
Which of the following is a correct call of the method someMethod? (4 Points)

- ☐ someMethod(4711, a -> a+1);
- ☐ someMethod(4711, true);
- ☒ someMethod(4711, a -> a > 0);
- ☒ Predicate<Integer> pred = a -> a == 0;
  someMethod(4711, pred);

1. In Java ist Mehrfachvererbung von Klassen nicht erlaubt. extends ist nur für eine Klasse möglich

2. In Java kann ein Interface mit dem Schlüsselwort extends von mehreren Interfaces erben.

3. Ein protected deklariertes Mitgliedfeld in einer Klasse A kann nicht nur in Unterklassen von A verwendet werden, sondern auch von anderen Klassen innerhalb desselben Pakets.



SWE2 VL Test WS2023/24

A 1 - Basic Theory on Java Programming (10 Points)

For each statement, select true or false (1 point for each correct answer, no reduction of points).

	true	false
1) Assume that classes Vo and Ue exist. It is possible to derive a new class Vse as follows: `class Vse extends Vo, Ue { public void methodE( ) { System.out.println("writing a brilliant VSE exam."); } }`		x
2) An interface may inherit from multiple interfaces.	x	
3) When a member field is declared protected in class A, it can only be accessed in subclasses of A.		x
4) Class Exam is given as follows: `class Exam { public static void methodE() { System.out.println("This is a simple exam."); } }` Method methodE of class Exam can be called as: Exam.methodE().	X	
5) Class Movie and class Feature are given as follows: `class Movie { private final String title; Movie(String title) { this.title = title; } } class Feature extends Movie { Feature(String title) { super(title); } public void play() { System.out.println("Now in theaters."); } } Movie oppenheimer = new Feature("Oppenheimer");` The method play may be called as: oppenheimer.play();		x
6) A SortedSet allows duplicates.		x
7) Class A is given as `class A { }`. The method `public void function (A a)` is a higher order function.		x
8) A member field declared as final cannot be changed once it has been initialized.	x	
9) With the following code, a new Thread is started. `Thread t = new Thread(() -> System.out.println("Running ..."));` `t.run();`		x
10) With the Model-View-Controller architecture, it is possible to create multiple views for one model.	x	

4. Da methodE() als static in der Klasse Exam deklariert ist, kann sie direkt mit Exam.methodE(); aufgerufen werden, ohne dass eine Instanz der Klasse Exam erstellt werden muss.

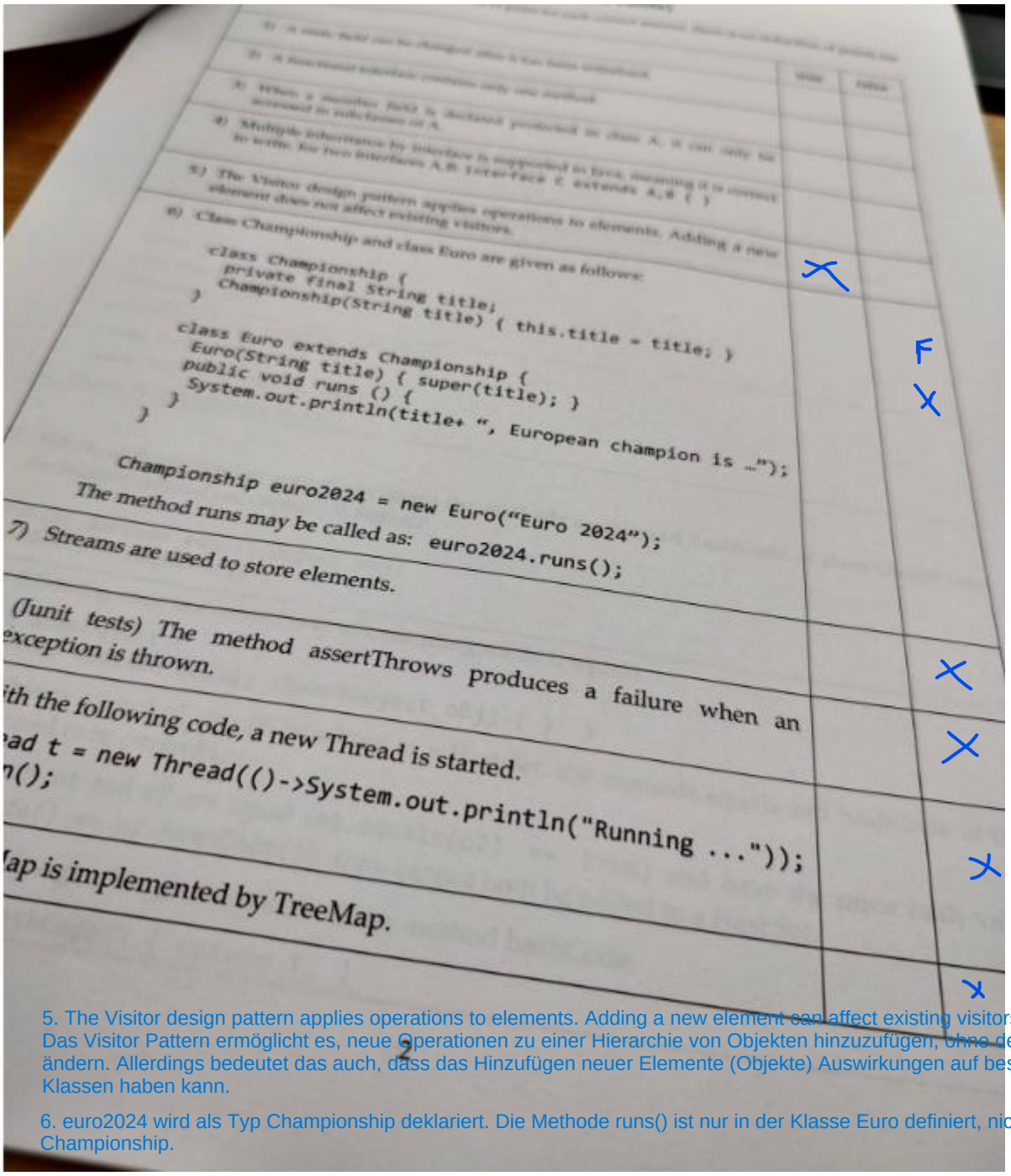
Die Methode play() kann nicht mit oppenheimer.play(); aufgerufen werden, weil oppenheimer als Typ Movie deklariert ist (Movie oppenheimer = new Feature("Oppenheimer");) , aber play() ist nur in der Feature-Klasse definiert.

7. Eine höhere Ordnung Funktion (Higher-Order Function) ist eine Funktion, die entweder eine Funktion als Parameter nimmt oder eine Funktion als Rückgabewert zurückgibt.

6. Ein SortedSet in Java (z. B. TreeSet) erlaubt keine doppelten Elemente. Es stellt sicher, dass alle Elemente in sortierter Reihenfolge gespeichert werden und dass keine Duplikate vorhanden sind.

9. Der Code erstellt zwar ein Thread-Objekt mit einer Lambda-Funktion als Runnable, aber der Thread wird nicht tatsächlich gestartet, weil t.run(); anstelle von t.start(); aufgerufen wird.
t.run(); führt die run()-Methode im Hauptthread aus, ohne einen neuen Thread zu starten.
t.start(); hätte einen neuen Thread gestartet und die run()-Methode in diesem neuen Thread ausgeführt.

10. Das Model repräsentiert die Daten und Geschäftslogik und ist unabhängig von der Darstellung.
Mehrere Views können dasselbe Model verwenden, um Daten auf unterschiedliche Weise darzustellen
Der Controller verwaltet die Interaktion zwischen View und Model.



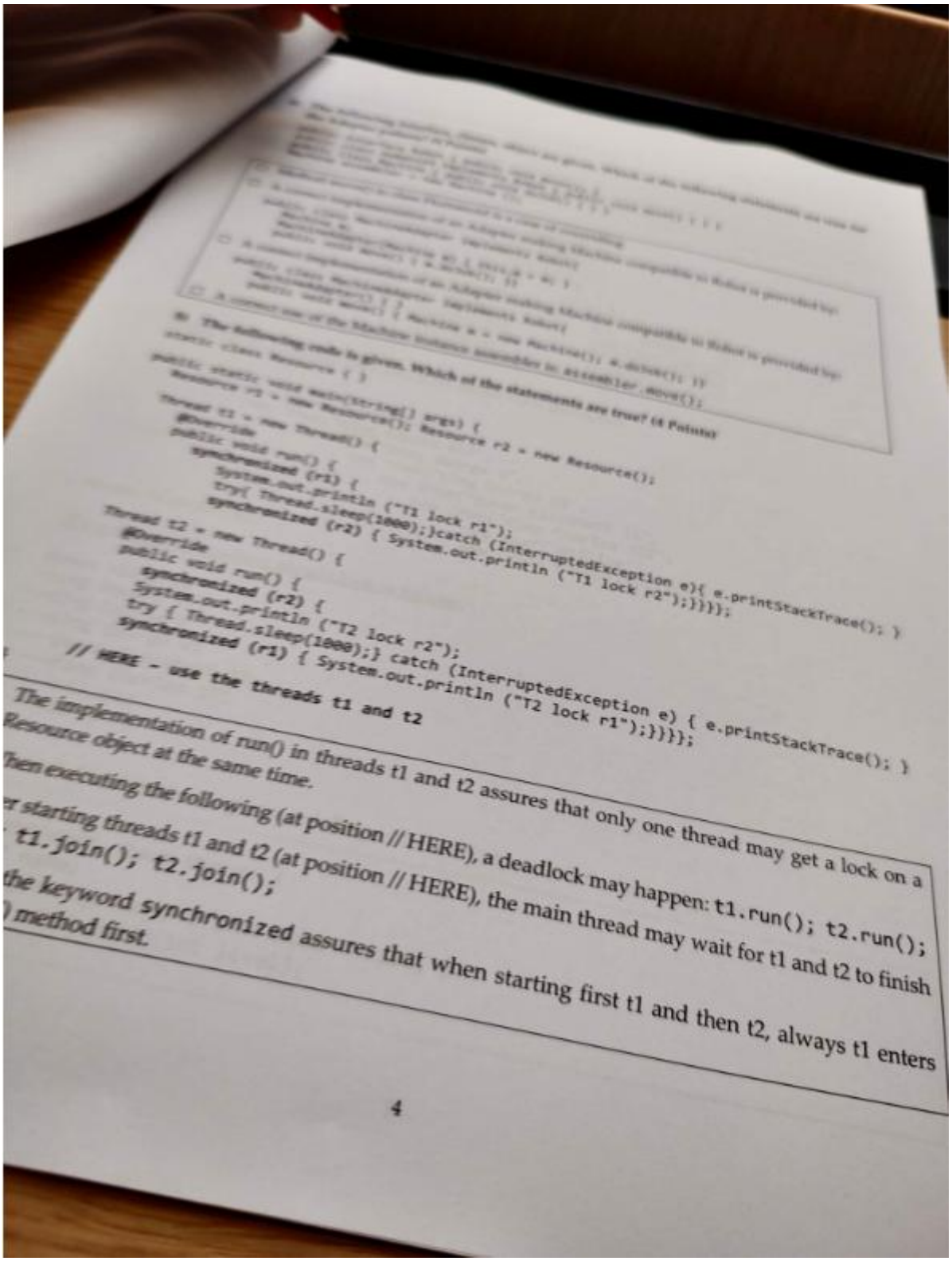
5. The Visitor design pattern applies operations to elements. Adding a new element can affect existing visitors (?) Wahr
Das Visitor Pattern ermöglicht es, neue Operationen zu einer Hierarchie von Objekten hinzuzufügen, ohne deren Klassen zu ändern. Allerdings bedeutet das auch, dass das Hinzufügen neuer Elemente (Objekte) Auswirkungen auf bestehende Visitor-Klassen haben kann.

6. euro2024 wird als Typ Championship deklariert. Die Methode runs() ist nur in der Klasse Euro definiert, nicht in Championship.

7. Streams in Java speichern keine Elemente, sondern verarbeiten sie in einer Pipeline. Ein Stream ist eine Sequenz von Daten, die einmal verarbeitet und dann verbraucht wird (z. B. gefiltert, gemappt, reduziert).

8. Die Methode assertThrows in JUnit überprüft, ob eine bestimmte Exception tatsächlich geworfen wird. Wenn die erwartete Exception geworfen wird, dann besteht der Test. Wenn keine Exception oder eine falsche Exception geworfen wird, dann schlägt der Test fehl.

10. HashMap und TreeMap sind zwei verschiedene Implementierungen der Map-Schnittstelle in Java.



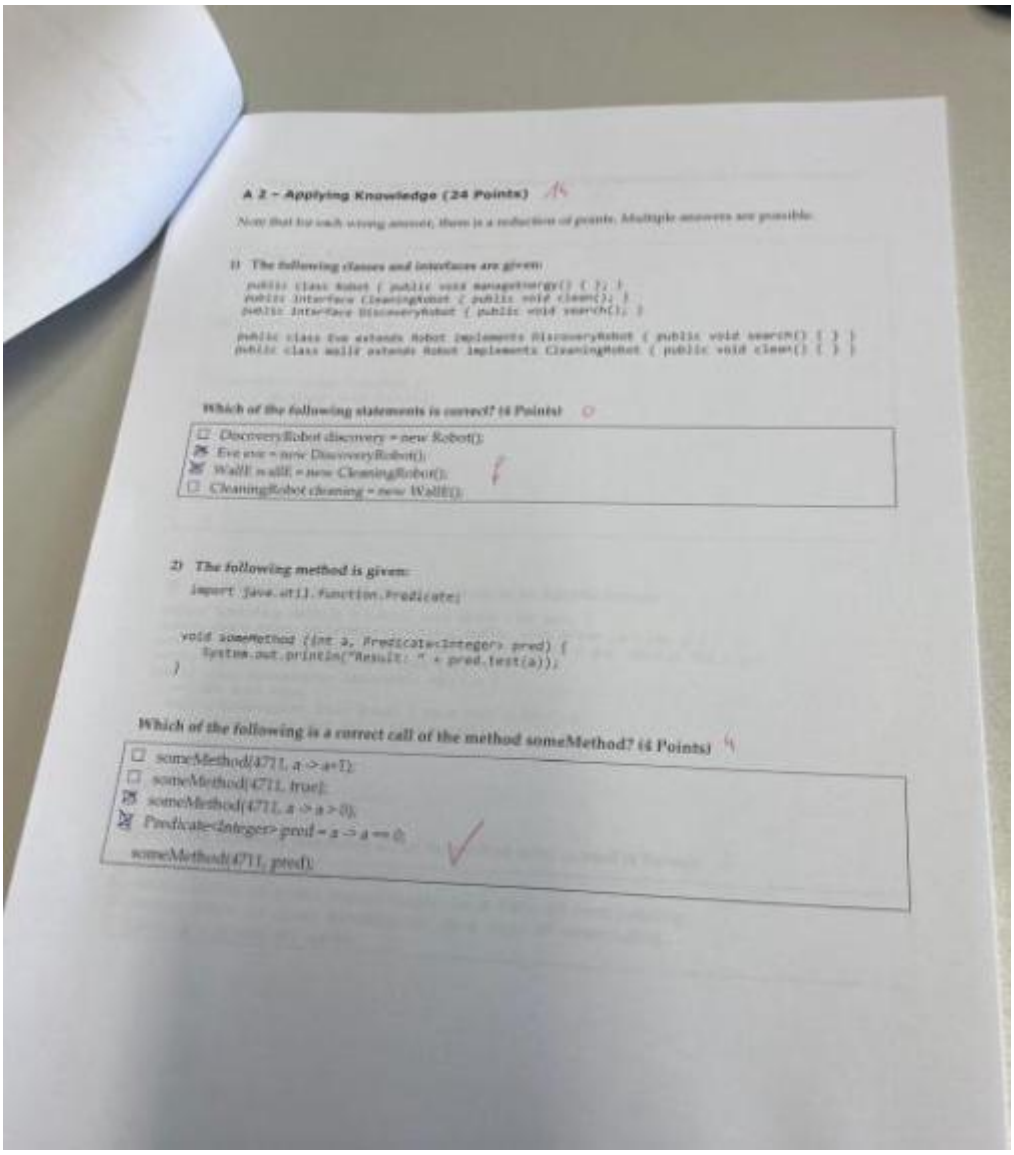
```
Resource r1 = new Resource(); Resource r2 = new Resource();
Thread t1 = new Thread() {
@Override
public void run() {
synchronized (r1) {
System.out.println ("T1 lock r1");
try{ Thread.sleep(1000);}catch (InterruptedException e){ e.printStackTrace(); }
synchronized (r2) { System.out.println ("T1 lock r2");}}}};
Thread t2 = new Thread() {
@Override
public void run() {
synchronized (r2) {
System.out.println ("T2 lock r2");
try { Thread.sleep(1000);} catch (InterruptedException e) { e.printStackTrace(); }
synchronized (r1) { System.out.println ("T2 lock r1");}}}};
// HERE - use the threads t1 and t2
```

The implementation of run() in threads t1 and t2 assures that only one thread may get a lock on a Resource object at the same time.

hen executing the following (at position // HERE), a deadlock may happen: `t1.run(); t2.run();`

r starting threads t1 and t2 (at position // HERE), the main thread may wait for t1 and t2 to finish `t1.join(); t2.join();`

the keyword synchronized assures that when starting first t1 and then t2, always t1 enters) method first.



A 2 – Applying Knowledge (24 Points)

Note that for each wrong answer, there is a reduction of points. Multiple answers are possible.

1) The following classes and interfaces are given:

```
public class Robot { public void manageEnergy() { }; }
public interface CleaningRobot { public void clean(); }
public interface DiscoveryRobot { public void search(); }

public class Eve extends Robot implements DiscoveryRobot { public void search() { } }
public class Wall-E extends Robot implements CleaningRobot { public void clean() { } }
```

Which of the following statements is correct? (4 Points)

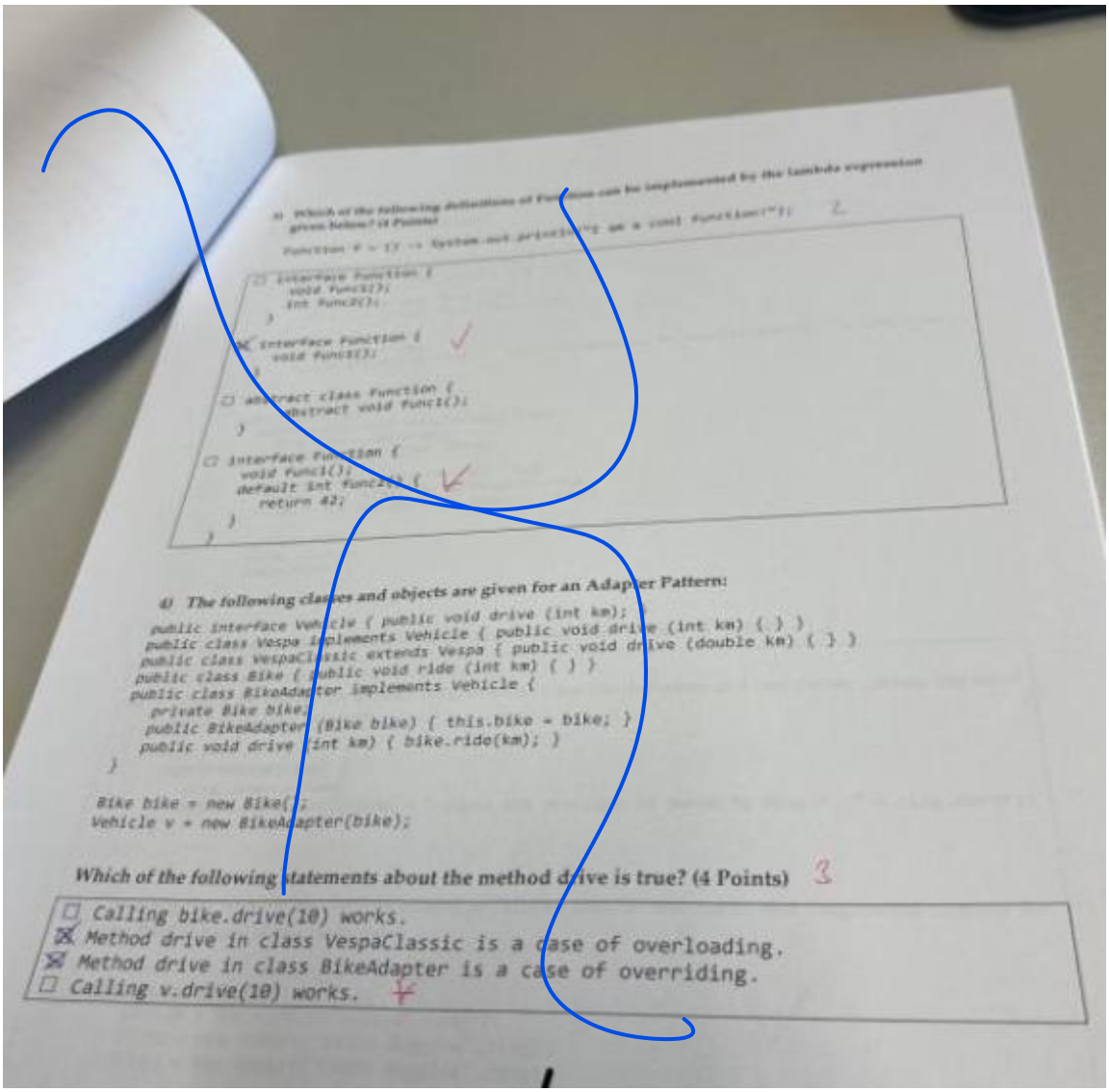
- ☐ DiscoveryRobot discovery = new Robot();
- ☒ Eve eve = new DiscoveryRobot();
- ☒ Wall-E wallE = new CleaningRobot();
- ☐ CleaningRobot cleaning = new Wall-E();

2) The following method is given:

```
import java.util.function.Predicate;

void someMethod (int a, Predicate<Integer> pred) {
    System.out.println("Result: " + pred.test(a));
}
```

Which of the following is a correct call of the method someMethod? (4 Points)

- ☐ someMethod(4711, a -> a+1);
- ☐ someMethod(4711, true);
- ☒ someMethod(4711, a -> a > 0);
- ☒ Predicate<Integer> pred = a -> a == 0;
  someMethod(4711, pred);



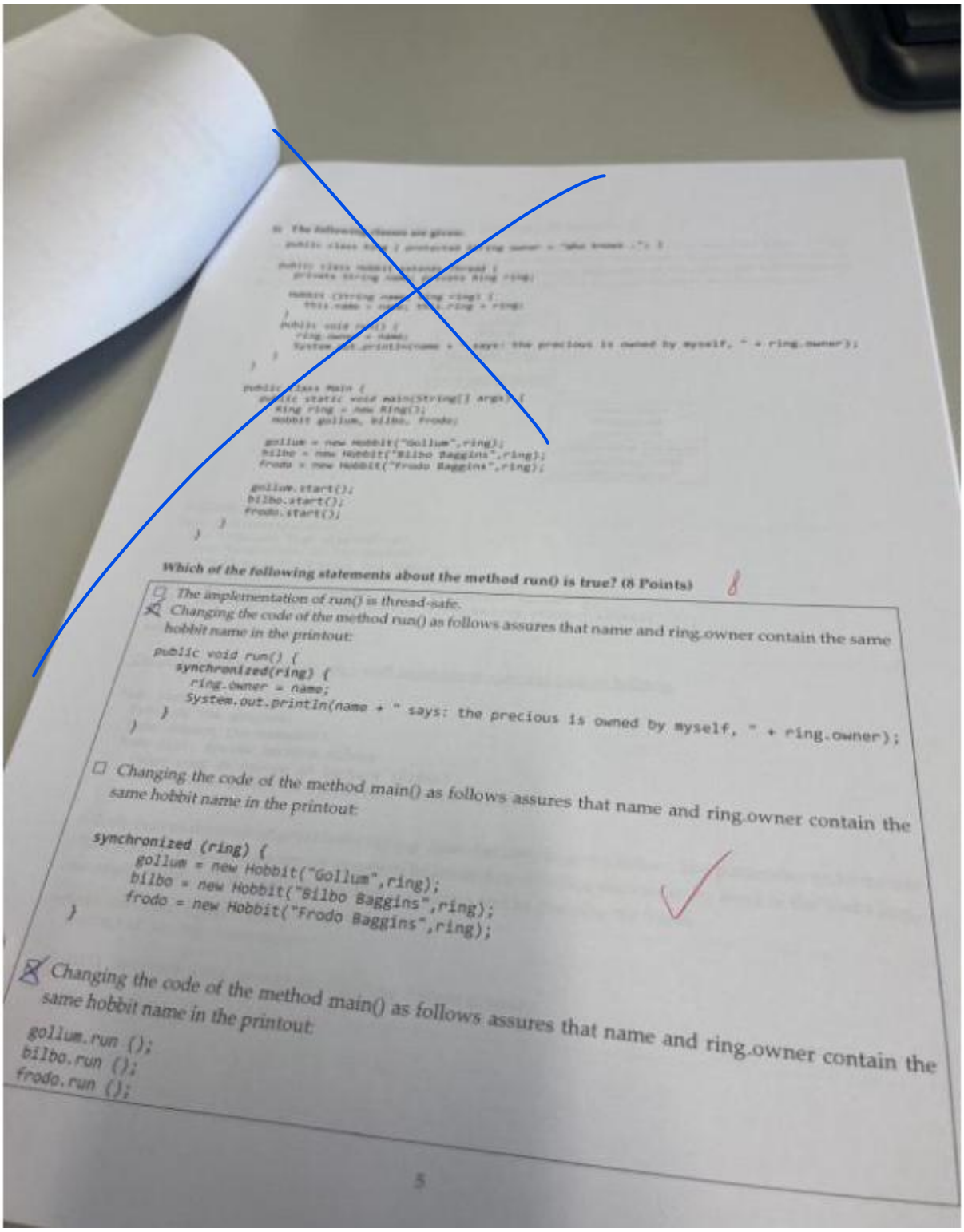
```
interface Function {
  void func1();
  int func2();
}

interface Function {
  void func1();
}

abstract class Function {
  abstract void func1();
}

interface Function {
  void func1();
  default int func2() {
    return 42;
  }
}
```

4) The following classes and objects are given for an Adapter Pattern:

```
public interface Vehicle { public void drive (int km); }
public class Vespa implements Vehicle { public void drive (int km) { } }
public class VespaClassic extends Vespa { public void drive (double km) { } }
public class Bike { public void ride (int km) { } }
public class BikeAdapter implements Vehicle {
  private Bike bike;
  public BikeAdapter (Bike bike) { this.bike = bike; }
  public void drive (int km) { bike.ride(km); }
}

Bike bike = new Bike();
Vehicle v = new BikeAdapter(bike);
```

Which of the following statements about the method drive is true? (4 Points)

- ☐ Calling bike.drive(10) works.
- ☒ Method drive in class VespaClassic is a case of overloading.
- ☒ Method drive in class BikeAdapter is a case of overriding.
- ☐ Calling v.drive(10) works.



Which of the following statements about the method run() is true? (8 Points)

☐ The implementation of run() is thread-safe.

☒ Changing the code of the method run() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
public void run() {
  synchronized(ring) {
    ring.owner = name;
    System.out.println(name + " says: the precious is owned by myself, " + ring.owner);
  }
}
```

☐ Changing the code of the method main() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
synchronized (ring) {
    gollum = new Hobbit("Gollum",ring);
    bilbo = new Hobbit("Bilbo Baggins",ring);
    frodo = new Hobbit("Frodo Baggins",ring);
}
```

☒ Changing the code of the method main() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
gollum.run ();
bilbo.run ();
frodo.run ();
```

A 1 – Basic Theory on Java Programming (10 Points)

For each statement, select true or false (1 point for each correct answer, no reduction of points).

	true	false

1) Assume that classes Vo and Uo exist.
It is possible to derive a new class Vse as follows:
`class Vse extends Vo, Uo {`
`  public void methodVse() {`
`  System.out.println("Writing a brilliant VSE exam.");}`
`}`
	X	

2) An interface may inherit from multiple interfaces.		X
3) When a member field is declared protected in class A, it can only be accessed in subclasses of A.	X	

4) Class Exam is given as follows:
`class Exam {`
`  public static void methodE() {`
`  System.out.println("This is a simple exam.");}`
`}`
Method methodE of class Exam can be called as: `Exam.methodE()`.
	X	

5) Class Movie and class Feature are given as follows:
`class Movie {`
`  private final String title;`
`  Movie(String title) { this.title = title; }`
`}`
`class Feature extends Movie {`
`  Feature(String title) { super(title); }`
`  public void play () {System.out.println("Now in theaters.");}`
`}`
`Movie oppenheimer = new Feature ("Oppenheimer");`
The method play may be called as: `oppenheimer.play ();`
	X	

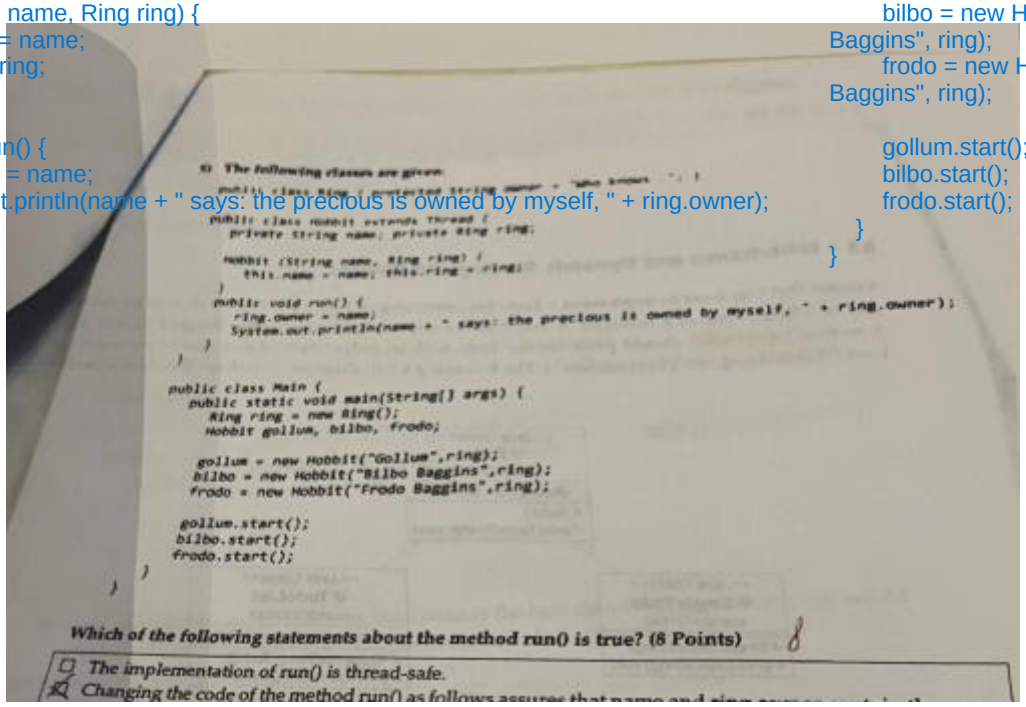
6) A SortedSet allows duplicates.	X	
7) Class A is given as `class A { }`. The method `public void function (A a)` is a higher order function.		X
8) A member field declared as final cannot be changed once it has been initialized.	X	

9) With the following code, a new Thread is started.
`Thread t = new Thread(()->System.out.println("Running ..."));`
`t.run();`
		X

With the Model-View-Controller architecture, it is possible to create multiple views for one model.	X	

```
public class Ring {
    protected String owner = "who knows";
}

public class Hobbit extends Thread {
    private String name;
    private Ring ring;

    Hobbit(String name, Ring ring) {
        this.name = name;
        this.ring = ring;
    }

    public void run() {
        ring.owner = name;
        System.out.println(name + " says: the precious is owned by myself, " + ring.owner);
    }
}
```

```
public class Main {
    public static void main(String[] args)
{
        Ring ring = new Ring();
        Hobbit gollum, bilbo, frodo;

        gollum = new Hobbit("Gollum",
ring);
        bilbo = new Hobbit("Bilbo
Baggins", ring);
        frodo = new Hobbit("Frodo
Baggins", ring);

        gollum.start();
        bilbo.start();
        frodo.start();
    }
}
```



Which of the following statements about the method run() is true? (8 Points) 8

☐ The implementation of run() is thread-safe.

☒ Changing the code of the method run() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
public void run() {
    synchronized(ring) {
        ring.owner = name;
        System.out.println(name + " says: the precious is owned by myself, " + ring.owner);
    }
}
```

☐ Changing the code of the method main() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
synchronized (ring) {
    gollum = new Hobbit("Gollum",ring);
    bilbo = new Hobbit("Bilbo Baggins",ring);
    frodo = new Hobbit("Frodo Baggins",ring);
}
```

Die Synchronisation in main() alleine stellt nicht sicher, dass name und ring.owner während der Ausgabe immer übereinstimmen.

☒ Changing the code of the method main() as follows assures that name and ring.owner contain the same hobbit name in the printout:

```
gollum.run ();
bilbo.run ();
frodo.run ();
```

Wenn du statt start() die Methode run() direkt aufrufst, wird die run()-Methode sequenziell (nacheinander) ausgeführt und nicht in parallelen Threads. Hier wird kein Multithreading mehr genutzt. Jeder run()-Aufruf wird vollständig beendet, bevor der nächste run()-Aufruf startet.Dadurch gibt es keine Race Conditions mehr und ring.owner wird für jeden Hobbit korrekt gesetzt und ausgegeben.
Daher stimmt ring.owner in jeder Ausgabe mit name überein.

1. Mehrere Threads (Hobbit-Instanzen) greifen gleichzeitig auf ring.owner zu und ändern den Wert. Dies führt zu Race Conditions, da ring.owner nicht synchronisiert ist.
2. Die synchronized(ring)-Anweisung sorgt dafür, dass nur ein Thread gleichzeitig ring.owner ändern kann. Dadurch wird sichergestellt, dass die Ausgabe konsistent bleibt. richtig

3) Which of the following definitions of Function can be implemented by the lambda expression given below? (4 Points)

```
Function f = () -> System.out.println("I am a cool function!");
```

☐
```
interface Function {
    void func1();
    int func2();
}
```

☒
```
interface Function {
    void func1();
}
```

☐
```
abstract class Function {
    abstract void func1();
}
```

☒
```
interface Function {
    void func1();
    default int func2() {
        return 42;
    }
}
```

4) The following classes and objects are given for an Adapter Pattern:

```
public interface Vehicle { public void drive (int km); }
public class Vespa implements Vehicle { public void drive (int km) { } }
public class VespaClassic extends Vespa { public void drive (double km) { } }
public class Bike { public void ride (int km) { } }
public class BikeAdapter implements Vehicle {
    private Bike bike;
    public BikeAdapter (Bike bike) { this.bike = bike; }
    public void drive (int km) { bike.ride(km); }
}
Bike bike = new Bike();
Vehicle v = new BikeAdapter(bike);
```

Which of the following statements about the method drive is true? (4 Points)

- ☐ Calling bike.drive(10) works.
- ☒ Method drive in class VespaClassic is a case of overloading.
- ☒ Method drive in class BikeAdapter is a case of overriding.
- ☒ Calling v.drive(10) works.

A 2 - Applying Knowledge (24 Points)

Note that for each wrong answer, there is a reduction of points. Multiple answers are possible.

1) The following classes and interfaces are given:

```
public class Robot { public void manageEnergy() { } }
public interface CleaningRobot { public void clean(); }
public interface DiscoveryRobot { public void search(); }

public class Eve extends Robot implements DiscoveryRobot {public void search() {} }
public class WallE extends Robot implements CleaningRobot {public void clean() {} }
```

Robot implementiert nicht DiscoveryRobot, daher ist dies nicht möglich.

Which of the following statements is correct? (4 Points)

- ☐ DiscoveryRobot discovery = new Robot();
- ☒ Eve eve = new DiscoveryRobot();
- ☐ WallE wallE = new CleaningRobot();
- ☒ CleaningRobot cleaning = new WallE();

DiscoveryRobot ist ein Interface, und man kann keine Objekte von Interfaces erstellen.
Es müsste stattdessen DiscoveryRobot eve = new Eve(); sein.

CleaningRobot ist ein Interface und kann nicht direkt instanziiert werden.

2) The following method is given:

```
import java.util.function.Predicate;

void someMethod (int a, Predicate<Integer> pred) {
    System.out.println("Result: " + pred.test(a));
}
```

Predicate boolean

Which of the following is a correct call of the method someMethod? (4 Points)

- ☐ someMethod(4711, a -> a+1);
- ☐ someMethod(4711, true);
- ☒ someMethod(4711, a -> a > 0);
- ☒ Predicate<Integer> pred = a -> a == 0;
  someMethod(4711, pred);

Predicate<Integer> erwartet eine Lambda-Funktion, die einen boolean zurückgibt. a -> a+1 gibt jedoch einen int zurück, nicht boolean.

2. Der zweite Parameter muss ein Predicate<Integer> sein, aber true ist ein boolean-Wert.
Richtige Lösung wäre: someMethod(4711, a -> true);

3. Richtig: a -> a > 0 ist eine gültige Predicate<Integer>-Lambda-Funktion, die prüft, ob a größer als 0 ist.

4. Richtig:
pred ist eine gültige Predicate<Integer>-Definition.
pred ist vom Typ Predicate<Integer>, daher kann es als zweiter Parameter verwendet werden.

SWE2 VL Test WS2023/24

A 1 - Basic Theory on Java Programming (10 Points)

For each statement, select true or false (1 point for each correct answer, no reduction of points).

	true	false
1) Assume that classes Vo and Ue exist. It is possible to derive a new class Vse as follows: `class Vse extends Vo, Ue { public void methodE( ) { System.out.println("writing a brilliant VSE exam."); } }`		x
2) An interface may inherit from multiple interfaces.	x	
3) When a member field is declared protected in class A, it can only be accessed in subclasses of A.		x
4) Class Exam is given as follows: `class Exam { public static void methodE() { System.out.println("This is a simple exam."); } }` Method methodE of class Exam can be called as: Exam.methodE( ).		(x)

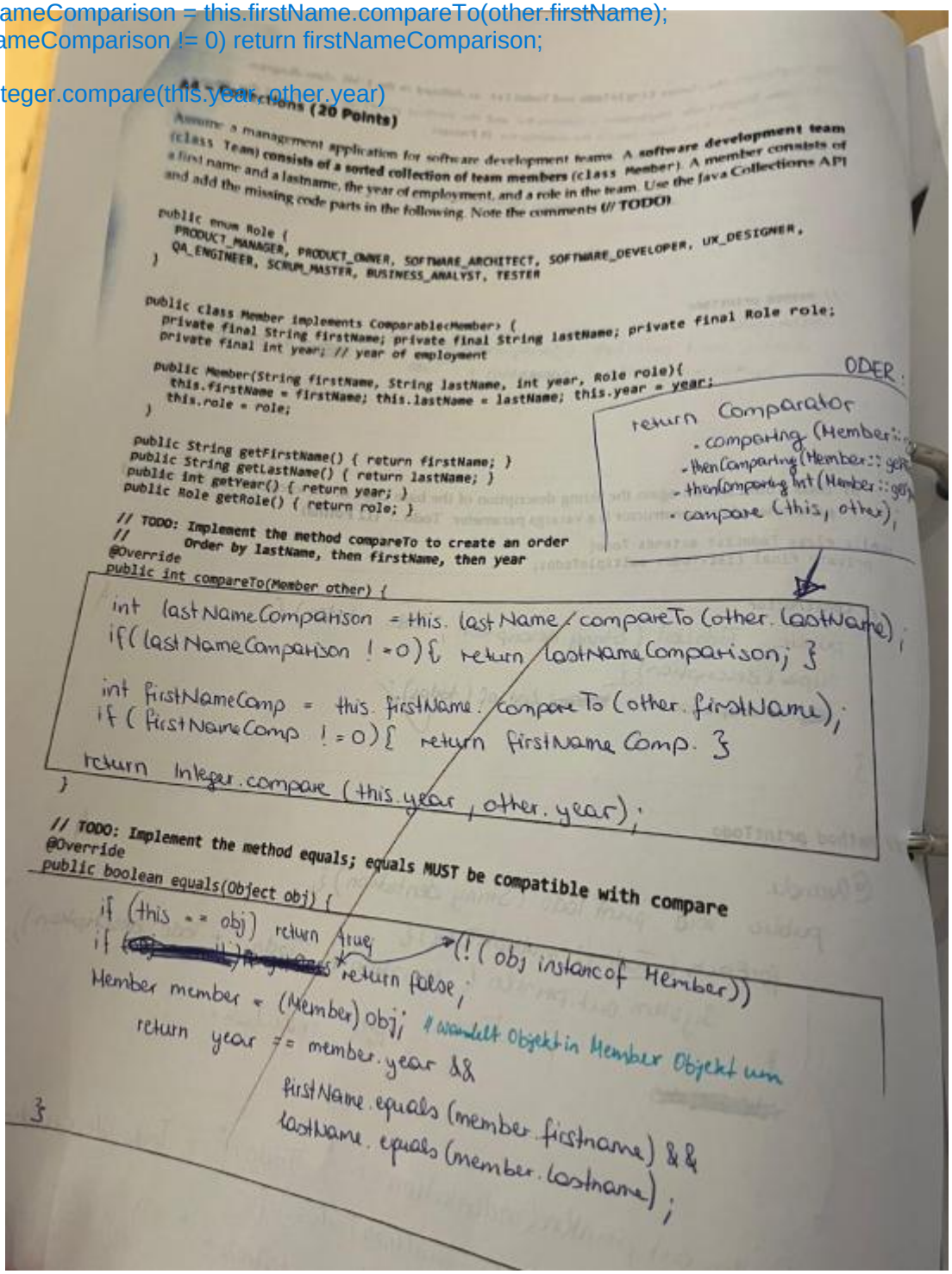
5) Class Movie and class Feature are given as follows:
`class Movie { private final String title; Movie(String title) { this.title = title; } }`
`class Feature extends Movie { Feature(String title) { super(title); } public void play() { System.out.println("Now in theaters."); } }`
`Movie oppenheimer = new Feature("Oppenheimer");`
The method play may be called as: oppenheimer.play();
		x

6) A SortedSet allows duplicates.		x
7) Class A is given as `class A { }`. The method `public void function (A a)` is a higher order function.		x
8) A member field declared as final cannot be changed once it has been initialized.	x	

9) With the following code, a new Thread is started.
`Thread t = new Thread(() -> System.out.println("Running ..."));`
`t.run();`
		x

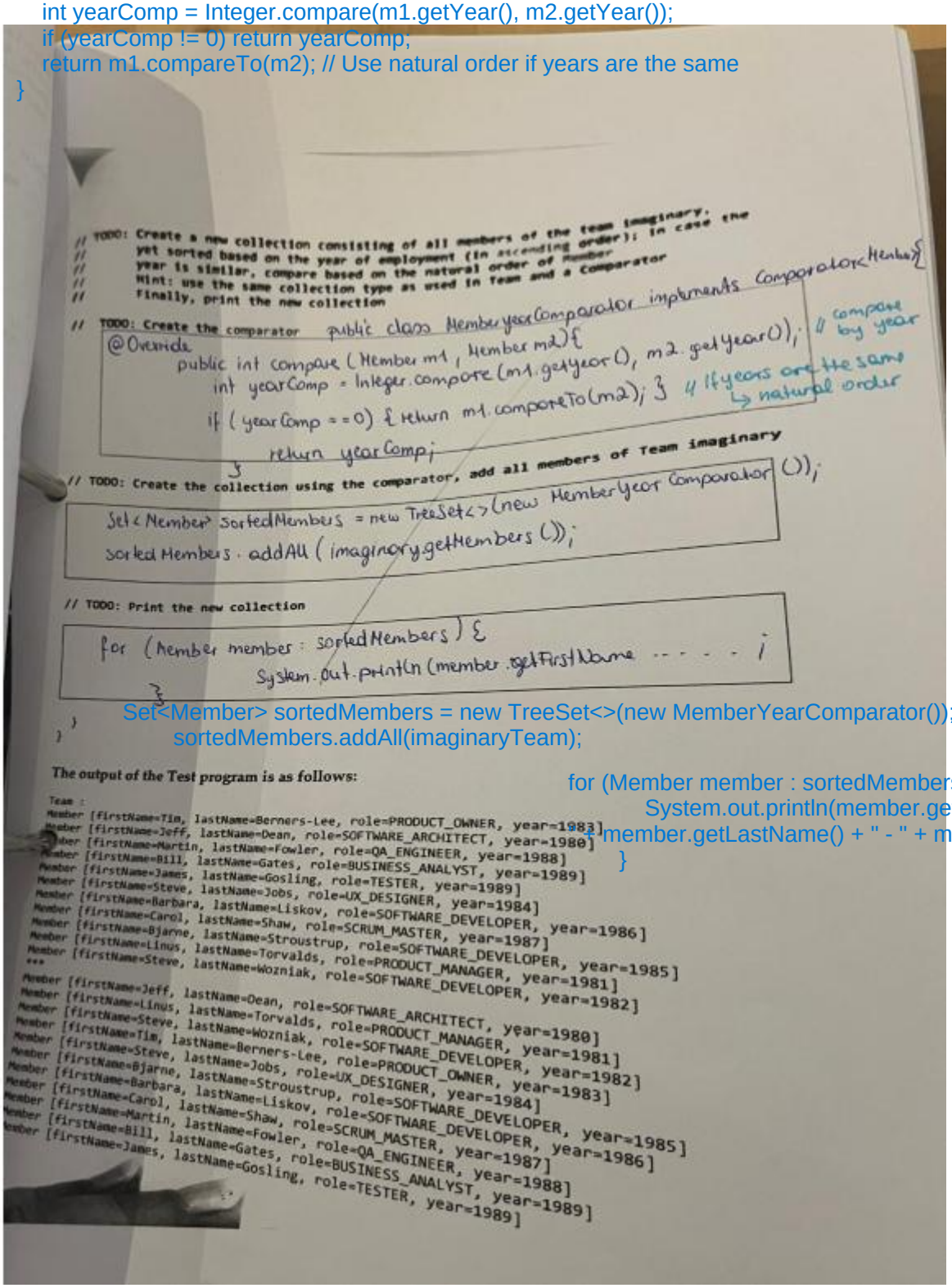
10) With the Model-View-Controller architecture, it is possible to create multiple views for one model.	x	

```java
@Override
public int compareTo(Member other) {
    int lastNameComparison = this.lastName.compareTo(other.lastName);
    if (lastNameComparison != 0) return lastNameComparison;

    int firstNameComparison = this.firstName.compareTo(other.firstName);
    if (firstNameComparison != 0) return firstNameComparison;

    return Integer.compare(this.year, other.year)
```




```java
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Member)) return false;
    Member member = (Member) obj;
    return year == member.year &&
           Objects.equals(firstName, member.firstName) &&
           Objects.equals(lastName, member.lastName);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName, year);
}
```

```
class MemberYearComparator implements Comparator<Member> {
    @Override
    public int compare(Member m1, Member m2) {
        int yearComp = Integer.compare(m1.getYear(), m2.getYear());
        if (yearComp != 0) return yearComp;
        return m1.compareTo(m2); // Use natural order if years are the same
    }
```




```
Set<Member> sortedMembers = new TreeSet<>(new MemberYearComparator());
        sortedMembers.addAll(imaginaryTeam);
```

```
for (Member member : sortedMembers) {
            System.out.println(member.getFirstName() + "
member.getLastName() + " - " + member.getYear());
        }
```

```
public class Team {
    // Definišemo odgovarajuću sortiranu kolekciju članova tima
    private final Set<Member> teamMembers;

    // Inicijalizujemo teamMembers u konstruktoru
    public Team() {
        this.teamMembers = new TreeSet<>();
    }
```

A 5 – Streams (16 Points)

The class Movie contains information about the title, year of production, director, and a set of actors of a movie:

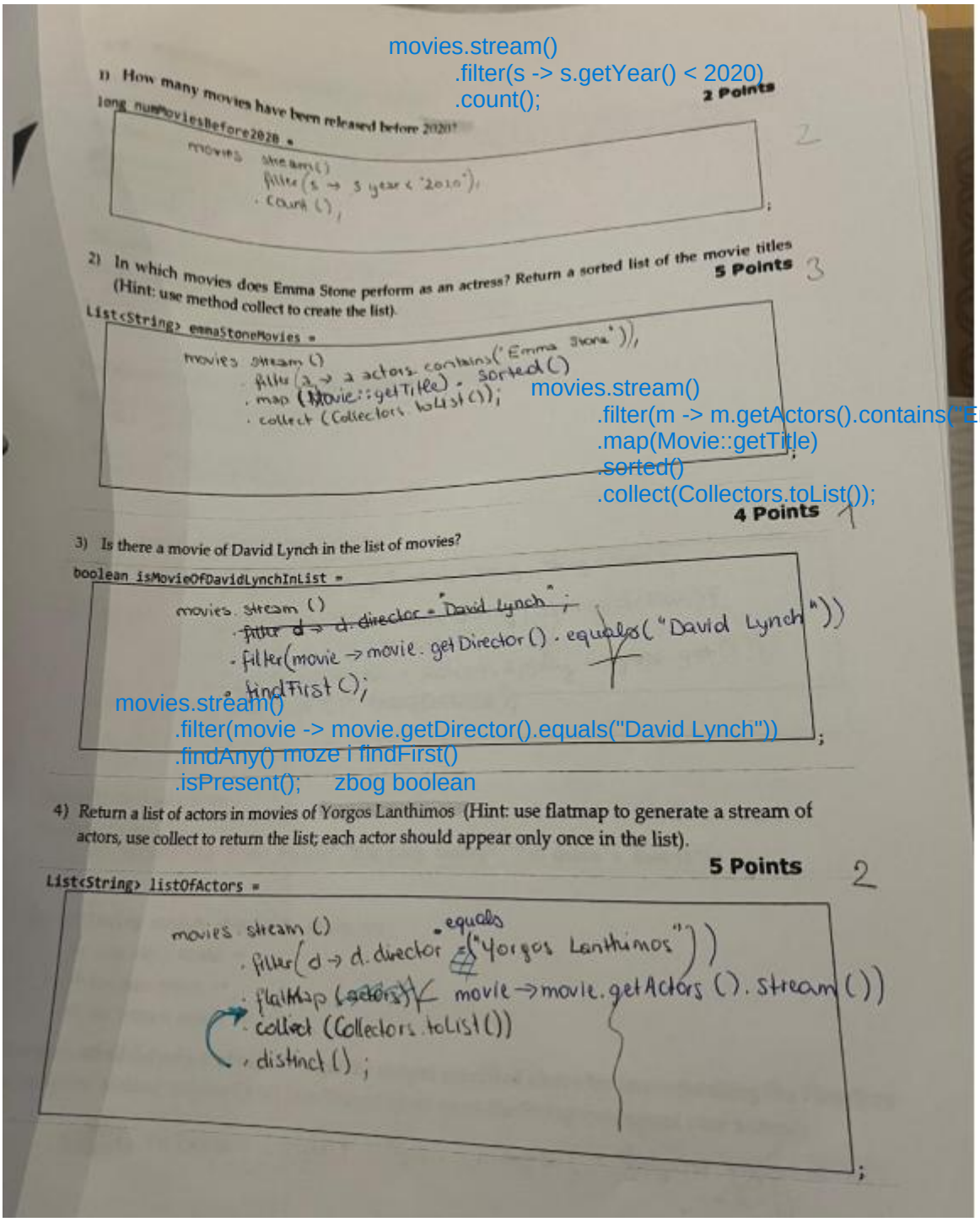
```
public class Movie {
        private String title;
        private int year;
        private String director;
        private Set<String> actors;

        public Movie (String title, int year, String director, Set<String> actors){
            this.title = title;
            this.year = year;
            this.director = director;
        }       this.actors = actors;

        String getTitle() { return title; }
        int getYear() { return year; }
        String getDirector () { return director; }
}       Set<String> getActors() { return actors; }
```

The list of movies contains the following movies:

```
List<Movie> movies = List.of(
  new Movie ("Evil Does Not Exist",2023,"Ruyusuke Hamaguchi", Set.of("Hitoshi Omika", "Ryo
Nishikawa", "Ryuji Kosaka")),
  new Movie ("Poor Things", 2023, "Yorgos Lanthimos", Set.of("Emma Stone","Willam Dafoe",
"Mark Ruffalo")),
  new Movie ("Tenet",2020, "Christopher Nolan", Set.of("Elizabeth Debicki","John David
Washington", "Robert Pattinson")),
  new Movie ("Dune", 2021, "Denis Villeneuve", Set.of("Timothee
Chalamet","Zendaya","Rebecca Ferguson")),
  new Movie ("La La Land", 2016, "Damien Chazelle", Set.of("Emma Stone","Ryan Gosling","Fin
Wittrock")),
  new Movie ("Casablanca", 1942, "Michael Curtiz", Set.of("Humphrey Bogart","Ingrid
Bergman", "Peter Lorre")),
  new Movie ("The Favorite", 2018, "Yorgos Lanthimos", Set.of("Emma Stone","Olivia Colman",
"Rachel Weisz")),
  new Movie ("Avengers: Infinity War", 2018, "Anthony Russo, Joe Russo",Set.of("Robert
Downey Jr.","Chris Hemsworth", "Scarlett Johansson", "Benedict Cumberbatch")));
```

Use the following STREAM OPERATIONS to answer the questions below:

- `Stream<T> filter(Predicate<? super T> predicate)`
- `Stream<T> sorted()`
- `Stream<T> distinct()`
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`
- `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
- `long count()`
- `Optional<T> findFirst()`
- `<R, A> R collect(Collector<? super T, A, R> collector)`
  - *To create a list:* `static Collectors<T, ?, List<T>> toList()`



movies.stream()
.filter(s -> s.getYear() < 2020)
.count();

movies.stream()
.filter(m -> m.getActors().contains("Emma Stone"))
.map(Movie::getTitle)
.sorted()
.collect(Collectors.toList());

movies.stream()
.filter(movie -> movie.getDirector().equals("David Lynch"))
.findAny() moze i findFirst()
.isPresent(); zbog boolean

movies.stream()
.filter(movie -> movie.getDirector().equals("Yorgos Lanthimos")) // Filtriramo filmove režisera
.flatMap(movie -> movie.getActors().stream()) // Ekstrahujemo stream glumaca iz filmova
.distinct() // Uklanjamo duplikate
.collect(Collectors.toList()); // Pravimo listu

A 6 – Higher Order Function (10 Points)

Define a method doForEach(Function<String,String> action) that allows to implement a transforming action for each String element of a class MyList. MyList should implement the interface DoForEach.

The functional interface Function is defined as follows:

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t);
}
```

The interface DoForEach is defined as follows:

```
public interface DoForEach{
  public void doForEach(Function<String,String> action);
}
```

Implement the class MyList by implementing the method doForEach:

```
public class MyList implements DoForEach {
  final List<String> myList;

  MyList (String...elems){
    myList = List.of(elems);
  }

// TODO: Implement method doForEach here
```

```
// Implement doForEach method
@Override
public void doForEach(Function<String, String> action) {
    for (int i = 0; i < myList.size(); i++) {
        String transformed = action.apply(myList.get(i));
        myList.set(i, transformed);
    }
}
```

```
public void doForEach (Function<String, String> action);
}
@Override
public void doForEach (Function<String, String> action){
    for (int i = 0; i < myList.size(); i++){
        String transformed = action.apply (myList.get(i));
        myList.set (i, transformed);
    }
}
```

The list is filled as follows:

```
MyList myList =
    new MyList("Star Trek: Picard","Big Bang Theory","The Queen's Gambit");
```

The following output should be produced:

```
** STAR TREK: PICARD **
** BIG BANG THEORY **
** THE QUEEN'S GAMBIT **
```

```
// Apply toUpperCase function using doForEach
myList.doForEach(String::toUpperCase);
// Print the modified list
myList.printList();
```

Use the method doForEach of myList to get the output specified above by implementing the Function action (Hint: Method toUpperCase() for String objects turns the String into upper case letters):

```
doForEach (myList, System.out.println("** "+myList.toUpperCase()
+" **"));
```

```
myList.doForEach(s -> System.out.println("** "+s.toUpperCase()+" **"));
myList.getMyList().forEach(System.out::println);
```