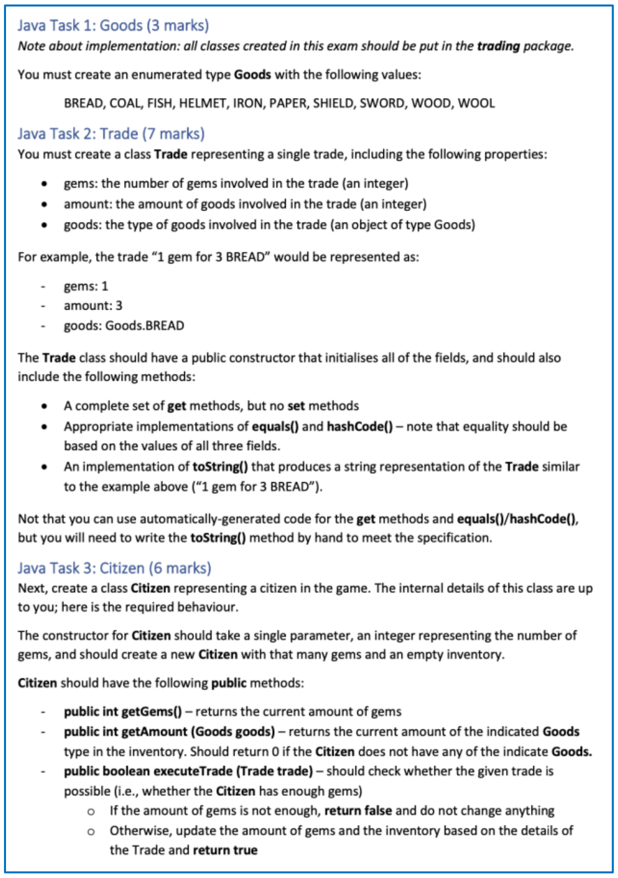


Java Task 1: Goods (3 marks)

Note about implementation: all classes created in this exam should be put in the ***trading*** *package.*

You must create an enumerated type **Goods** with the following values:

BREAD, COAL, FISH, HELMET, IRON, PAPER, SHIELD, SWORD, WOOD, WOOL

Java Task 2: Trade (7 marks)

You must create a class **Trade** representing a single trade, including the following properties:

- gems: the number of gems involved in the trade (an integer)
- amount: the amount of goods involved in the trade (an integer)
- goods: the type of goods involved in the trade (an object of type Goods)

For example, the trade "1 gem for 3 BREAD" would be represented as:

- gems: 1
- amount: 3
- goods: Goods.BREAD

The **Trade** class should have a public constructor that initialises all of the fields, and should also include the following methods:

- A complete set of **get** methods, but no **set** methods
- Appropriate implementations of **equals()** and **hashCode()** – note that equality should be based on the values of all three fields.
- An implementation of **toString()** that produces a string representation of the **Trade** similar to the example above ("1 gem for 3 BREAD").

Not that you can use automatically-generated code for the **get** methods and **equals()/hashCode()**, but you will need to write the **toString()** method by hand to meet the specification.

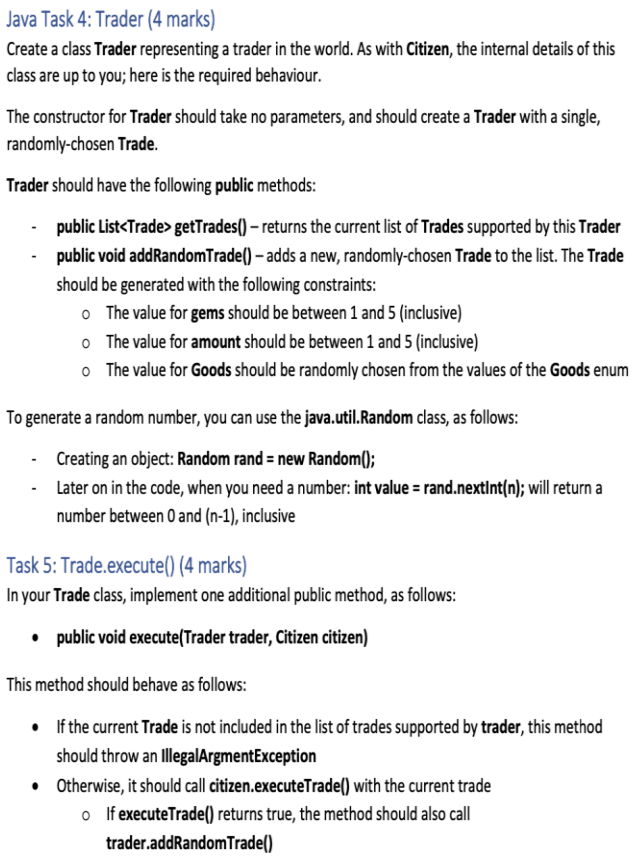
Java Task 3: Citizen (6 marks)

Next, create a class **Citizen** representing a citizen in the game. The internal details of this class are up to you; here is the required behaviour.

The constructor for **Citizen** should take a single parameter, an integer representing the number of gems, and should create a new **Citizen** with that many gems and an empty inventory.

Citizen should have the following **public** methods:

- **public int getGems()** – returns the current amount of gems
- **public int getAmount (Goods goods)** – returns the current amount of the indicated **Goods** type in the inventory. Should return 0 if the **Citizen** does not have any of the indicate **Goods.**
- **public boolean executeTrade (Trade trade)** – should check whether the given trade is possible (i.e., whether the **Citizen** has enough gems)
  - If the amount of gems is not enough, **return false** and do not change anything
  - Otherwise, update the amount of gems and the inventory based on the details of the Trade and **return true**



Java Task 4: Trader (4 marks)

Create a class **Trader** representing a trader in the world. As with **Citizen**, the internal details of this class are up to you; here is the required behaviour.

The constructor for **Trader** should take no parameters, and should create a **Trader** with a single, randomly-chosen **Trade**.

Trader should have the following **public** methods:

- **public List<Trade> getTrades()** – returns the current list of **Trades** supported by this **Trader**
- **public void addRandomTrade()** – adds a new, randomly-chosen **Trade** to the list. The **Trade** should be generated with the following constraints:
  - The value for **gems** should be between 1 and 5 (inclusive)
  - The value for **amount** should be between 1 and 5 (inclusive)
  - The value for **Goods** should be randomly chosen from the values of the **Goods** enum

To generate a random number, you can use the **java.util.Random** class, as follows:

- Creating an object: **Random rand = new Random();**
- Later on in the code, when you need a number: **int value = rand.nextInt(n);** will return a number between 0 and (n-1), inclusive

Task 5: Trade.execute() (4 marks)

In your **Trade** class, implement one additional public method, as follows:

- **public void execute(Trader trader, Citizen citizen)**

This method should behave as follows:

- If the current **Trade** is not included in the list of trades supported by **trader**, this method should throw an **IllegalArgmentException**
- Otherwise, it should call **citizen.executeTrade()** with the current trade
  - If **executeTrade()** returns true, the method should also call **trader.addRandomTrade()**

Lab Exam 2021:

Task 1:

```
package trading;

public enum Goods {

BREAD, COAL, FISH, HELMET, IRON, PAPER, SHIELD, SWORD, WOOD, WOOL;

}
```

Task 2:

```
package trading;


public class Trade {
private final int gems; // Number of gems required for the trade
private final int amount; // Amount of goods provided in the trade
private final Goods goods;
```

```
public Trade(int gems, int amount, Goods goods) {
this.gems = gems;
this.amount = amount;
this.goods = goods;
}



public int getGems() {
return gems;
}

public int getAmount() {
return amount;
}

public Goods getGoods() {
return goods;
}

@Override
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + gems;
result = prime * result + amount;
result = prime * result + ((goods == null) ? 0 : goods.hashCode());
return result;
}

@Override
public boolean equals(Object obj) {
if (this == obj)
return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
Trade other = (Trade) obj;
if (gems != other.gems)
return false;
if (amount != other.amount)
return false;
if (goods != other.goods)
return false;
return true;
}

@Override
public String toString() {
return gems + " gem for" + amount +" "+goods ;
}}
```

Task 3:

```
package trading;

import java.util.HashMap;
import java.util.Map;


public class Citizen {
private int gems; // Gems the citizen has
private final Map<Goods, Integer> inventory; // Inventory of goods
public Citizen(int gems) {
this.gems = gems;
inventory = new HashMap<>();
}
public int getGems() {
return this.gems;
}
public int getAmount(Goods goods) {
return inventory.get(goods);
}

public boolean executeTrade (Trade trade)
{
if (gems < trade.getGems())
{
return false;
}
else{
this.gems-= trade.getGems();
inventory.put(trade.getGoods(), inventory.getOrDefault(trade.getGoods(), 0) + trade.getAmount());
return true;
}
}
}
```

Task 4:

```
package trading;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Trader {
List <Trade> Trades;

public Trader() {
Trades = new ArrayList<>();
addRandomTrade();
}
public List<Trade> getTrades()
{
return this.Trades;
}

public void addRandomTrade()
{
Random rand = new Random();
int gems = rand.nextInt(5) + 1; // Gems between 1 and 5
int amount = rand.nextInt(5) + 1; // Amount between 1 and 5
Goods goods = Goods.values()[rand.nextInt(Goods.values().length)];
Trades.add(new Trade(gems, amount, goods));

}}
```

Task 5:

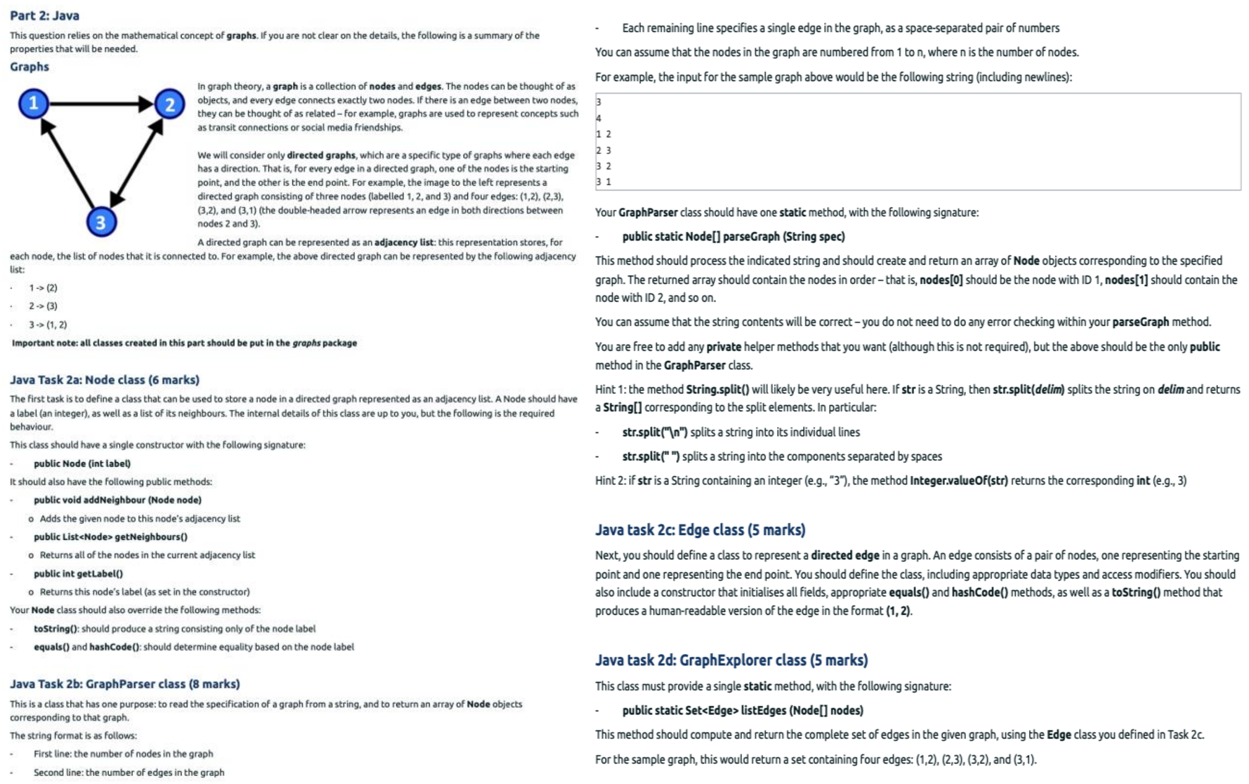
```
public void execute(Trader trader, Citizen citizen) {
if (!trader.getTrades().contains(this)) {
throw new IllegalArgumentException("Trade not offered by this trader.");
}

if (citizen.executeTrade(this)) {
trader.addRandomTrade(); // Add a new random trade after success
}
}
```

To test:

```
public class Main {
public static void main(String[] args) {
Citizen citizen = new Citizen(5); // Citizen starts with 5 gems
Trader trader = new Trader(); // Trader starts with one trade
Trade trade = trader.getTrades().get(0); // Get the first trade
System.out.println("Before Trade: " + citizen.getGems() + " gems, " +
citizen.getAmount(trade.getGoods()) + " " + trade.getGoods());
trade.execute(trader, citizen);
System.out.println("After Trade: " + citizen.getGems() + " gems, " +
citizen.getAmount(trade.getGoods()) + " " + trade.getGoods());
}
}
```

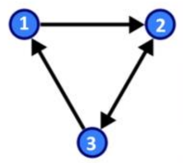
Lab Exam 2022:



Part 2: Java

This question relies on the mathematical concept of **graphs**. If you are not clear on the details, the following is a summary of the properties that will be needed.

Graphs



In graph theory, a **graph** is a collection of **nodes** and **edges**. The nodes can be thought of as objects, and every edge connects exactly two nodes. If there is an edge between two nodes, they can be thought of as related – for example, graphs are used to represent concepts such as transit connections or social media friendships.

We will consider only **directed graphs**, which are a specific type of graphs where each edge has a direction. That is, for every edge in a directed graph, one of the nodes is the starting point, and the other is the end point. For example, the image to the left represents a directed graph consisting of three nodes (labelled 1, 2, and 3) and four edges: (1,2), (2,3), (3,2), and (3,1) (the double-headed arrow represents an edge in both directions between nodes 2 and 3).

A directed graph can be represented as an **adjacency list**: this representation stores, for each node, the list of nodes that it is connected to. For example, the above directed graph can be represented by the following adjacency list:

- 1 -> (2)
- 2 -> (3)
- 3 -> (1, 2)

Important note: all classes created in this part should be put in the *graphs* package

Java Task 2a: Node class (6 marks)

The first task is to define a class that can be used to store a node in a directed graph represented as an adjacency list. A Node should have a label (an integer), as well as a list of its neighbours. The internal details of this class are up to you, but the following is the required behaviour.

This class should have a single constructor with the following signature:

- **public Node (int label)**

It should also have the following public methods:

- **public void addNeighbour (Node node)**
  - Adds the given node to this node's adjacency list
- **public List<Node> getNeighbours()**
  - Returns all of the nodes in the current adjacency list
- **public int getLabel()**
  - Returns this node's label (as set in the constructor)

Your **Node** class should also override the following methods:

- **toString()**: should produce a string consisting only of the node label
- **equals()** and **hashCode()**: should determine equality based on the node label

Java Task 2b: GraphParser class (8 marks)

This is a class that has one purpose: to read the specification of a graph from a string, and to return an array of **Node** objects corresponding to that graph.

The string format is as follows:

- First line: the number of nodes in the graph
- Second line: the number of edges in the graph
- Each remaining line specifies a single edge in the graph, as a space-separated pair of numbers

You can assume that the nodes in the graph are numbered from 1 to n, where n is the number of nodes.

For example, the input for the sample graph above would be the following string (including newlines):

```
3
4
1 2
2 3
3 2
3 1
```

Your **GraphParser** class should have one **static** method, with the following signature:

- **public static Node[] parseGraph (String spec)**

This method should process the indicated string and should create and return an array of **Node** objects corresponding to the specified graph. The returned array should contain the nodes in order – that is, **nodes[0]** should be the node with ID 1, **nodes[1]** should contain the node with ID 2, and so on.

You can assume that the string contents will be correct – you do not need to do any error checking within your **parseGraph** method.

You are free to add any **private** helper methods that you want (although this is not required), but the above should be the only **public** method in the **GraphParser** class.

Hint 1: the method **String.split()** will likely be very useful here. If **str** is a String, then **str.split(*delim*)** splits the string on ***delim*** and returns a **String[]** corresponding to the split elements. In particular:

- **str.split("\n")** splits a string into its individual lines
- **str.split(" ")** splits a string into the components separated by spaces

Hint 2: if **str** is a String containing an integer (e.g., "3"), the method **Integer.valueOf(str)** returns the corresponding **int** (e.g., 3)

Java task 2c: Edge class (5 marks)

Next, you should define a class to represent a **directed edge** in a graph. An edge consists of a pair of nodes, one representing the starting point and one representing the end point. You should define the class, including appropriate data types and access modifiers. You should also include a constructor that initialises all fields, appropriate **equals()** and **hashCode()** methods, as well as a **toString()** method that produces a human-readable version of the edge in the format **(1, 2)**.

Java task 2d: GraphExplorer class (5 marks)

This class must provide a single **static** method, with the following signature:

- **public static Set<Edge> listEdges (Node[] nodes)**

This method should compute and return the complete set of edges in the given graph, using the **Edge** class you defined in Task 2c.

For the sample graph, this would return a set containing four edges: (1,2), (2,3), (3,2), and (3,1).

Task 2a:

```java
package graphs;

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

public class Node {
private final int label;
private final List <Node> neighbours;

public Node(int label) {
this.label = label;
this.neighbours = new ArrayList<>();
}
public void addNeighbour (Node node)
{
if (!neighbours.contains(node)) { // Avoid duplicate neighbours
neighbours.add(node);
}
}

public int getLabel() {
return label;
}

public List<Node> getNeighbours() {
return new ArrayList<>(neighbours); // Return a copy to prevent external modification
}



@Override
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + label;
return result;
}

@Override
public boolean equals(Object obj) {
if (this == obj)
return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
Node other = (Node) obj;
if (label != other.label)
return false;
return true;
}

@Override
public String toString() {
return Integer.toString(label);
}

}
```

2b:

```
package graphs;

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

public class GraphParser {

// Static method to parse the graph specification from a string
public static Node[] parseGraph(String spec) {
```

```
String[] lines = spec.split("\n");

int numNodes = Integer.parseInt(lines[0].trim());
int numEdges = Integer.parseInt(lines[1].trim());

// Create an array of nodes, indexed from 1 to numNodes (ignoring index 0)
Node[] nodes = new Node[numNodes];
for (int i = 0; i < numNodes; i++) {
nodes[i] = new Node(i + 1); // Nodes are 1-indexed
}

// Parse the edges and add neighbors to the nodes
for (int i = 2; i < 2 + numEdges; i++) {
String[] edgeData = lines[i].split(" ");
int fromNode = Integer.parseInt(edgeData[0].trim());
int toNode = Integer.parseInt(edgeData[1].trim());
nodes[fromNode - 1].addNeighbour(nodes[toNode - 1]);
}

return nodes;
}
}
```

Task 2c:

```
package graphs;

import java.util.Objects;

public class Edge {
private Node start;
private Node end;

// Constructor to initialize the edge with start and end nodes
public Edge(Node start, Node end) {
this.start = start;
this.end = end;
}

// Override equals to compare edges based on start and end nodes
@Override
public boolean equals(Object obj) {
if (this == obj) return true;
if (obj == null || getClass() != obj.getClass()) return false;
Edge edge = (Edge) obj;
return start.equals(edge.start) && end.equals(edge.end);
}
// Override hashCode to ensure consistent hashing
@Override
public int hashCode() {
return Objects.hash(start, end);
}

// toString method to return a string representation of the edge
@Override
public String toString() {
return "(" + start.getLabel() + ", " + end.getLabel() + ")";
}
}
```

```
Task 2d:

package graphs;

import java.util.HashSet;
import java.util.Set;

public class GraphExplorer {

// Static method to list all edges in the graph
public static Set<Edge> listEdges(Node[] nodes) {
Set<Edge> edges = new HashSet<>();

// Loop through each node and its neighbors to create edges
for (Node node : nodes) {
for (Node neighbour : node.getNeighbours()) {
edges.add(new Edge(node, neighbour));
}
}

return edges;
}
}
```

Main File Example:

```
package graphs;

import java.util.Set;

public class Main {
    public static void main(String[] args) {
```

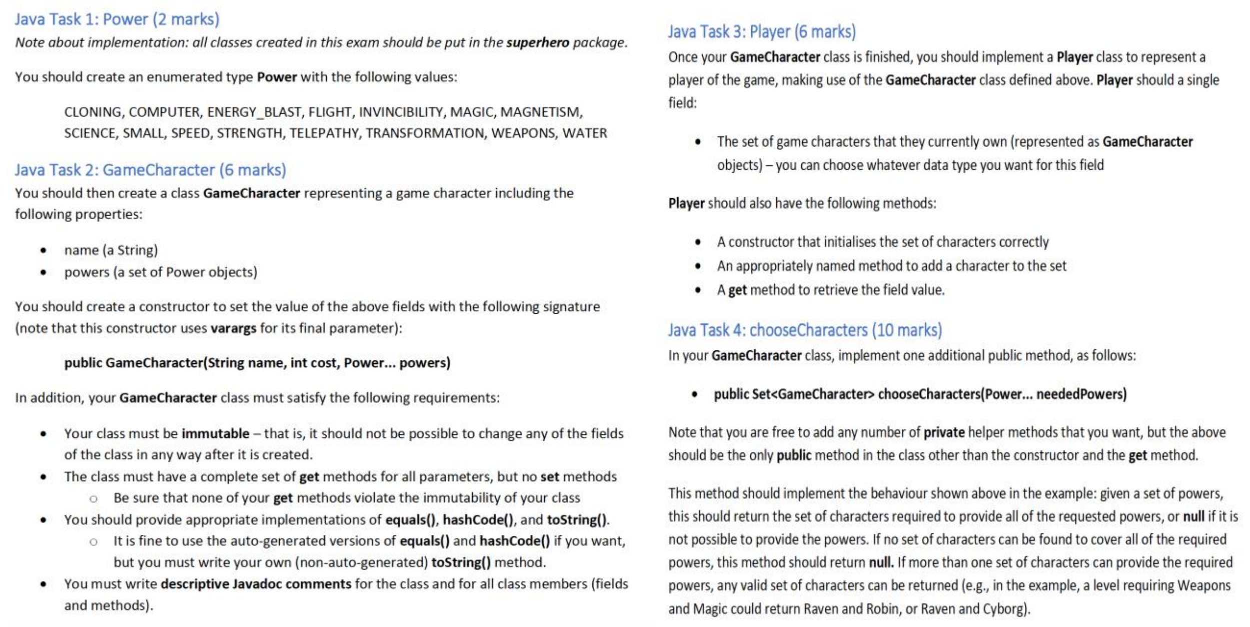
```
    // Sample graph specification
    String spec = "3\n4\n1 2\n2 3\n3 1\n3 2";

    // Parse the graph from the specification
    Node[] nodes = GraphParser.parseGraph(spec);

    // List all edges in the graph
    Set<Edge> edges = GraphExplorer.listEdges(nodes);

    // Print all edges
    for (Edge edge : edges) {
      System.out.println(edge);
    }
  }
}
```

Lab Exam 2019:



Java Task 1: Power (2 marks)

*Note about implementation: all classes created in this exam should be put in the **superhero** package.*

You should create an enumerated type **Power** with the following values:

CLONING, COMPUTER, ENERGY_BLAST, FLIGHT, INVINCIBILITY, MAGIC, MAGNETISM, SCIENCE, SMALL, SPEED, STRENGTH, TELEPATHY, TRANSFORMATION, WEAPONS, WATER

Java Task 2: GameCharacter (6 marks)

You should then create a class **GameCharacter** representing a game character including the following properties:

- name (a String)
- powers (a set of Power objects)

You should create a constructor to set the value of the above fields with the following signature (note that this constructor uses **varargs** for its final parameter):

public GameCharacter(String name, int cost, Power... powers)

In addition, your **GameCharacter** class must satisfy the following requirements:

- Your class must be **immutable** – that is, it should not be possible to change any of the fields of the class in any way after it is created.
- The class must have a complete set of **get** methods for all parameters, but no **set** methods
  - Be sure that none of your **get** methods violate the immutability of your class
- You should provide appropriate implementations of **equals()**, **hashCode()**, and **toString()**.
  - It is fine to use the auto-generated versions of **equals()** and **hashCode()** if you want, but you must write your own (non-auto-generated) **toString()** method.
- You must write **descriptive Javadoc comments** for the class and for all class members (fields and methods).

Java Task 3: Player (6 marks)

Once your **GameCharacter** class is finished, you should implement a **Player** class to represent a player of the game, making use of the **GameCharacter** class defined above. **Player** should a single field:

- The set of game characters that they currently own (represented as **GameCharacter** objects) – you can choose whatever data type you want for this field

Player should also have the following methods:

- A constructor that initialises the set of characters correctly
- An appropriately named method to add a character to the set
- A **get** method to retrieve the field value.

Java Task 4: chooseCharacters (10 marks)

In your **GameCharacter** class, implement one additional public method, as follows:

- **public Set<GameCharacter> chooseCharacters(Power... neededPowers)**

Note that you are free to add any number of **private** helper methods that you want, but the above should be the only **public** method in the class other than the constructor and the **get** method.

This method should implement the behaviour shown above in the example: given a set of powers, this should return the set of characters required to provide all of the requested powers, or **null** if it is not possible to provide the powers. If no set of characters can be found to cover all of the required powers, this method should return **null.** If more than one set of characters can provide the required powers, any valid set of characters can be returned (e.g., in the example, a level requiring Weapons and Magic could return Raven and Robin, or Raven and Cyborg).

Task 1

```
package superhero;

public enum Power {
CLONING, COMPUTER, ENERGY_BLAST, FLIGHT, INVINCIBILITY, MAGIC, MAGNETISM, SCIENCE, SMALL, SPEED,
STRENGTH, TELEPATHY, TRANSFORMATION, WEAPONS, WATER;
}
```

Task 2:

```
package superhero;

import java.util.Collections;
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

public class GameCharacter {
private final String name;
private final Set<Power> powers;

public GameCharacter(String name, Set<Power> powers) {
this.name = name;
this.powers = powers;
}

public String getName() {
return name;
}

public Set<Power> getPowers() {
return powers;
}

@Override
public int hashCode() {
int hash = 5;
hash = 29 * hash + Objects.hashCode(this.name);
hash = 29 * hash + Objects.hashCode(this.powers);
return hash;
}

@Override
public boolean equals(Object obj) {
if (this == obj) {
return true;
}
if (obj == null) {
return false;
}
if (getClass() != obj.getClass()) {
return false;
}
final GameCharacter other = (GameCharacter) obj;
if (!Objects.equals(this.name, other.name)) {
return false;
}
return this.powers == other.powers;
}

@Override
public String toString() {
return name + "has" + ", powers=" + powers;
}

public static Set<GameCharacter> chooseCharacters(Player player, Power... neededPowers) {
Set<Power> powersRequired = new HashSet<>();
Collections.addAll(powersRequired, neededPowers);

Set<GameCharacter> selectedCharacters = new HashSet<>();
for (GameCharacter character : player.getCharacters()) {
if (!Collections.disjoint(character.getPowers(), powersRequired)) {
selectedCharacters.add(character);
powersRequired.removeAll(character.getPowers());
}
if (powersRequired.isEmpty()) {
return selectedCharacters;
}
}
return null; // Not enough powers to cover the requirements
}


}
```

Task 3:

```
package superhero;

import java.util.HashSet;
import java.util.Set;

/**
* Represents a player in the game with a set of owned characters.
*/
public class Player {
private final Set<GameCharacter> characters;

/**
* Constructor to initialize the player's character set.
*/
public Player() {
this.characters = new HashSet<>();
}

/**
* Adds a character to the player's set of characters.
*
* @param character the character to add
*/
public void addCharacter(GameCharacter character) {
characters.add(character);
}

/**
* @return the set of characters owned by the player
*/
public Set<GameCharacter> getCharacters() {
return characters;
}
}
```

Task 4: Is in Task 2

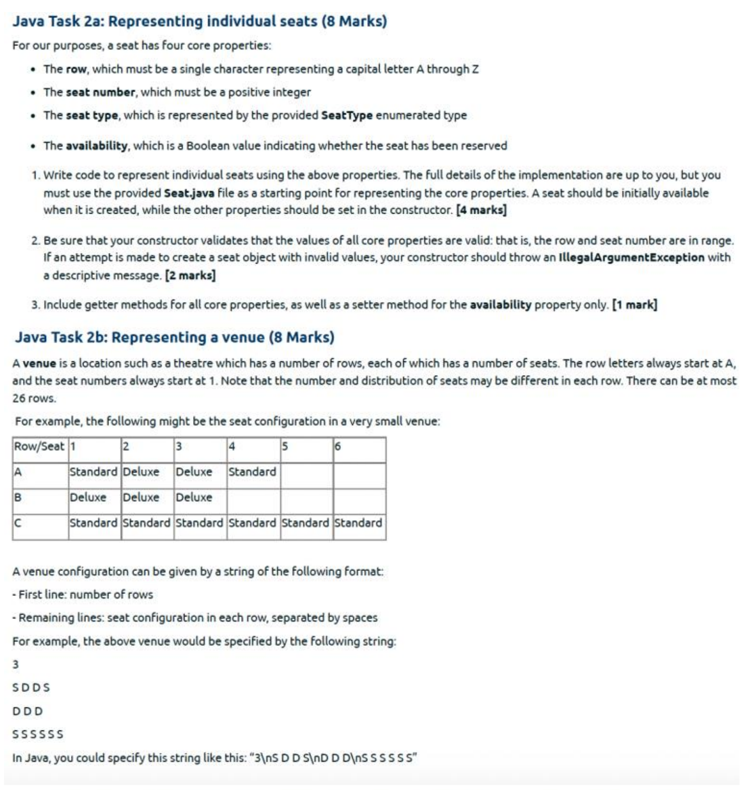
For Testing:

```
package superhero;

public class TestGame {
    public static void main(String[] args) {
        Player player = new Player();
        player.addCharacter(new GameCharacter("Robin", Power.WEAPONS));
        player.addCharacter(new GameCharacter("Starfire", Power.FLIGHT, Power.ENERGY_BLAST));
        player.addCharacter(new GameCharacter("Cyborg", Power.STRENGTH, Power.WEAPONS));
        player.addCharacter(new GameCharacter("Beast Boy", Power.TRANSFORMATION));
        player.addCharacter(new GameCharacter("Raven", Power.MAGIC));

        System.out.println(GameUtils.chooseCharacters(player, Power.WEAPONS, Power.STRENGTH)); // Cyborg
        System.out.println(GameUtils.chooseCharacters(player, Power.FLIGHT, Power.STRENGTH,
Power.TRANSFORMATION)); // Starfire, Cyborg, Beast Boy
        System.out.println(GameUtils.chooseCharacters(player, Power.TRANSFORMATION, Power.MAGIC, Power.SCIENCE));
// null
    }
}
```

Lab exam 2023:



Java Task 2a: Representing individual seats (8 Marks)

For our purposes, a seat has four core properties:

- The **row**, which must be a single character representing a capital letter A through Z
- The **seat number**, which must be a positive integer
- The **seat type**, which is represented by the provided **SeatType** enumerated type
- The **availability**, which is a Boolean value indicating whether the seat has been reserved

1. Write code to represent individual seats using the above properties. The full details of the implementation are up to you, but you must use the provided **Seat.java** file as a starting point for representing the core properties. A seat should be initially available when it is created, while the other properties should be set in the constructor. **[4 marks]**
2. Be sure that your constructor validates that the values of all core properties are valid: that is, the row and seat number are in range. If an attempt is made to create a seat object with invalid values, your constructor should throw an **IllegalArgumentException** with a descriptive message. **[2 marks]**
3. Include getter methods for all core properties, as well as a setter method for the **availability** property only. **[1 mark]**

Java Task 2b: Representing a venue (8 Marks)

A **venue** is a location such as a theatre which has a number of rows, each of which has a number of seats. The row letters always start at A, and the seat numbers always start at 1. Note that the number and distribution of seats may be different in each row. There can be at most 26 rows.

For example, the following might be the seat configuration in a very small venue:

Row/Seat	1	2	3	4	5	6
A	Standard	Deluxe	Deluxe	Standard		
B	Deluxe	Deluxe	Deluxe			
C	Standard	Standard	Standard	Standard	Standard	Standard

A venue configuration can be given by a string of the following format:

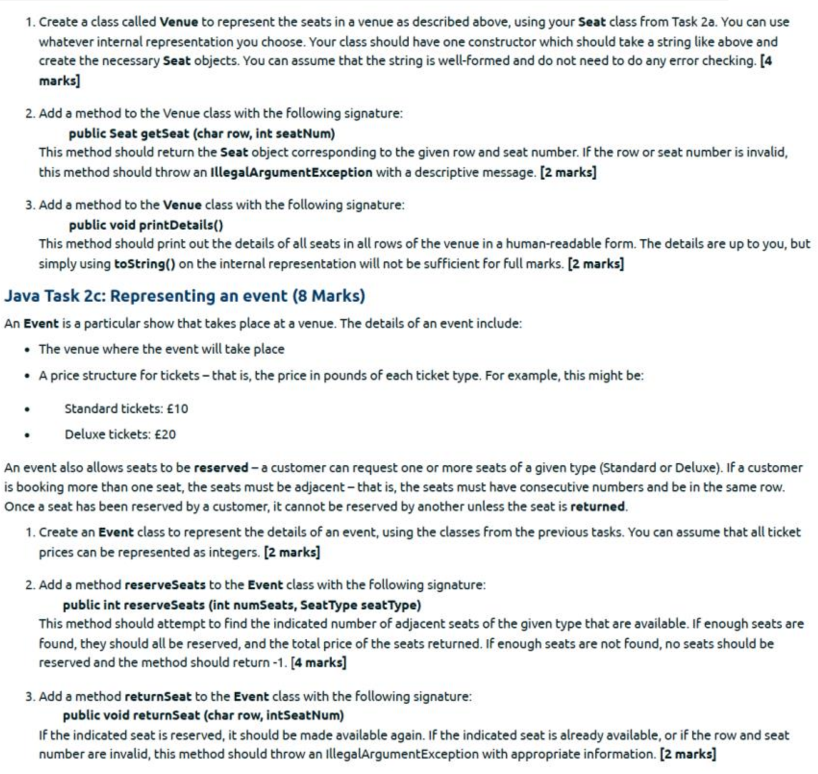
- First line: number of rows

- Remaining lines: seat configuration in each row, separated by spaces

For example, the above venue would be specified by the following string:

```
3
S D D S
D D D
S S S S S S
```

In Java, you could specify this string like this: "3\nS D D S\nD D D\nS S S S S S"



1. Create a class called **Venue** to represent the seats in a venue as described above, using your **Seat** class from Task 2a. You can use whatever internal representation you choose. Your class should have one constructor which should take a string like above and create the necessary **Seat** objects. You can assume that the string is well-formed and do not need to do any error checking. **[4 marks]**
2. Add a method to the Venue class with the following signature:
   public Seat getSeat (char row, int seatNum)
   This method should return the **Seat** object corresponding to the given row and seat number. If the row or seat number is invalid, this method should throw an **IllegalArgumentException** with a descriptive message. **[2 marks]**
3. Add a method to the **Venue** class with the following signature:
   public void printDetails()
   This method should print out the details of all seats in all rows of the venue in a human-readable form. The details are up to you, but simply using **toString()** on the internal representation will not be sufficient for full marks. **[2 marks]**

Java Task 2c: Representing an event (8 Marks)

An **Event** is a particular show that takes place at a venue. The details of an event include:

- The venue where the event will take place
- A price structure for tickets – that is, the price in pounds of each ticket type. For example, this might be:
- Standard tickets: £10
- Deluxe tickets: £20

An event also allows seats to be **reserved** – a customer can request one or more seats of a given type (Standard or Deluxe). If a customer is booking more than one seat, the seats must be adjacent – that is, the seats must have consecutive numbers and be in the same row. Once a seat has been reserved by a customer, it cannot be reserved by another unless the seat is **returned**.

1. Create an **Event** class to represent the details of an event, using the classes from the previous tasks. You can assume that all ticket prices can be represented as integers. **[2 marks]**
2. Add a method **reserveSeats** to the **Event** class with the following signature:
   public int reserveSeats (int numSeats, SeatType seatType)
   This method should attempt to find the indicated number of adjacent seats of the given type that are available. If enough seats are found, they should all be reserved, and the total price of the seats returned. If enough seats are not found, no seats should be reserved and the method should return -1. **[4 marks]**
3. Add a method **returnSeat** to the **Event** class with the following signature:
   public void returnSeat (char row, intSeatNum)
   If the indicated seat is reserved, it should be made available again. If the indicated seat is already available, or if the row and seat number are invalid, this method should throw an IllegalArgumentException with appropriate information. **[2 marks]**

Given:

```
package boxOffice;
```

```
public enum SeatType {
STANDARD, DELUXE;
}
```

Task 1:

```
package boxOffice;

public class Seat {

private final char row;
private final int seatNum;
private final SeatType seatType;
private boolean available;

public Seat(char row, int seatNum, SeatType seatType) {
if (row < 'A' || row > 'Z') {
throw new IllegalArgumentException("Row must be a capital letter A-Z.");
}
if (seatNum <= 0) {
throw new IllegalArgumentException("Seat number must be a positive integer.");
}
this.row = row;
this.seatNum = seatNum;
this.seatType = seatType;
this.available = true; // Initially available
}

public char getRow() {
return row;
}

public int getSeatNum() {
return seatNum;
}

public SeatType getSeatType() {
return seatType;
}

public boolean isAvailable() {
return available;
}

public void setAvailable(boolean available) {
this.available = available;
}
@Override
public String toString() {
return String.format("Seat[row=%c, seatNum=%d, type=%s, available=%b]", row, seatNum, seatType, available);
}
}
```

Task 2

```
package boxOffice;

public class Event {
private final Venue venue;
private final int standardPrice;
private final int deluxePrice;

public Event(Venue venue, int standardPrice, int deluxePrice) {
this.venue = venue;
this.standardPrice = standardPrice;
this.deluxePrice = deluxePrice;
}

public int reserveSeats(int numSeats, SeatType seatType) {
for (List<Seat> row : venue.getSeats()) {
int consecutive = 0;
for (Seat seat : row) {
if (seat.getSeatType() == seatType && seat.isAvailable()) {
consecutive++;
if (consecutive == numSeats) {
int totalPrice = numSeats * (seatType == SeatType.STANDARD ? standardPrice : deluxePrice);
for (int i = 0; i < numSeats; i++) {
row.get(row.indexOf(seat) - i).setAvailable(false);
}
return totalPrice;
}
} else {
consecutive = 0;
}
}
}
return -1;
}

public void returnSeat(char row, int seatNum) {
Seat seat = venue.getSeat(row, seatNum);
if (!seat.isAvailable()) {
seat.setAvailable(true);
} else {
```

```
throw new IllegalArgumentException("Seat is already available.");
}
}
}
```

Task 3:

```
package boxOffice;

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

public class Venue {
private final List<List<Seat>> seats;

public Venue(String configuration) {
seats = new ArrayList<>();
String[] lines = configuration.split("\\n");
int numRows = Integer.parseInt(lines[0]);

for (int i = 1; i <= numRows; i++) {
char row = (char) ('A' + i - 1);
String[] seatTypes = lines[i].split(" ");
List<Seat> rowSeats = new ArrayList<>();
for (int j = 0; j < seatTypes.length; j++) {
SeatType seatType = seatTypes[j].equals("S") ? SeatType.STANDARD : SeatType.DELUXE;
rowSeats.add(new Seat(row, j + 1, seatType));
}
seats.add(rowSeats);
}
}

public Seat getSeat(char row, int seatNum) {
if (row < 'A' || row >= 'A' + seats.size()) {
throw new IllegalArgumentException("Invalid row.");
}
int rowIndex = row - 'A';
List<Seat> rowSeats = seats.get(rowIndex);
if (seatNum < 1 || seatNum > rowSeats.size()) {
throw new IllegalArgumentException("Invalid seat number.");
}
return rowSeats.get(seatNum - 1);
}

public void printDetails() {
for (List<Seat> row : seats) {
for (Seat seat : row) {
```

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