Library Management

1. Superclass `LibraryMedia`

Field implementation:

- Create three fields with the `protected` modifier so they are accessible in subclasses:
  - `String title` – stores the title of the library media
  - `int publicationYear` – stores the year of publication
  - `boolean available` – flag indicating whether the media is available for borrowing

Constructor implementation:

- The constructor should accept two parameters: `title` and `publicationYear`
- Inside the constructor, assign the values of the parameters to the class fields using the `this` keyword
- Additionally, set the `available` field to `true` (we assume that a newly added medium is available)

Method implementation:

- `borrow()`: This method should check whether the media is available. If it is, change the `available` status to `false` and display a message indicating it has been borrowed. If not, display a message that the media is already borrowed.
- `returnMedia()`: This method checks whether the media is borrowed (i.e., `available == false`). If it is, change the status to `available = true` and display a return confirmation message. If not, inform the user that the media wasn't borrowed.
- `displayInformation()`: This method displays basic information about the media: title, publication year, and availability. Use the ternary operator (`available ? "Available" : "Borrowed"`) to display the availability status.

2. Subclasses `Book` and `Movie`

Class `Book`

Field implementation:

- Add two private fields specific to books:
  - `String author` – stores the author's full name
  - `int numberOfPages` – stores the number of pages in the book

Constructor implementation:

- The constructor should accept four parameters: `title`, `publicationYear`, `author`, `numberOfPages`
- Call the superclass constructor using `super(title, publicationYear)`
- Initialize the book-specific fields: `this.author = author` and `this.numberOfPages = numberOfPages`

Method overriding:

- Override the `displayInformation()` method using the `@Override` annotation
- First, call the superclass version of this method using `super.displayInformation()`
- Then, add the display of information specific to books: author and number of pages

Implementation of a specific method:

- Create a method `checkNumberOfPages()` that evaluates whether the book is long
- Inside the method, check whether `numberOfPages > 500`. If so, display a message that it's a long book. Otherwise, say it's a standard book. In both cases, include the number of pages.

Class `Movie`

Field implementation:

- Add two private fields specific to movies:
  - `String director` – stores the director's full name
  - `int duration` – stores the movie duration in minutes

Constructor implementation:

- The constructor should accept four parameters: `title`, `publicationYear`, `director`, `duration`
- Call the superclass constructor using `super(title, publicationYear)`
- Initialize the movie-specific fields: `this.director = director` and `this.duration = duration`

Method overriding:

- Override the `displayInformation()` method using the `@Override` annotation
- First, call the superclass version of this method using `super.displayInformation()`
- Then, add the display of movie-specific information: director and duration

Implementation of a specific method:

- Create a method `checkDuration()` that evaluates whether the movie is long

- Inside the method, check whether `duration > 120` (2 hours). If so, display a message that it's a long movie. Otherwise, say it's a standard movie. In both cases, include the duration in minutes.

3. Demonstrating polymorphism in the `main` method

Implementation of the test class:

- Create a class `LibraryTest` with a `main` method

Creating objects:

- Create two `Book` objects with different data
- Create two `Movie` objects with different data

Implementing a polymorphic array:

- Create an array of type `LibraryMedia[]` with length 4
- Assign all the created objects to this array (two books and two movies)
- This is possible thanks to polymorphism – derived type objects (`Book`, `Movie`) can be treated as base type objects (`LibraryMedia`)

Polymorphic method calls:

- Create a `for-each` loop that iterates through all elements in the array
- Inside the loop, call the `displayInformation()` method for each element
- Thanks to polymorphism, although we call the same method on elements of type `LibraryMedia`, Java automatically selects the appropriate implementation depending on the actual object type (the overridden method in `Book` or `Movie`)

Demonstrating borrowing and returning operations:

- Show borrowing one object, trying to borrow it again (which should fail), and then returning it

Calling type-specific methods:

- Call type-specific methods: `checkNumberOfPages()` for a book and `checkDuration()` for a movie
- Show that to call these methods directly, you need to use variables of the specific type (you can't call them on a `LibraryMedia` variable)

Demonstrating casting:

- Show how you can access type-specific methods through casting
- Create a loop through the polymorphic array

- For each element, check its actual type using the `instanceof` operator
- Depending on the type, cast it to the appropriate type and call the type-specific method

Example project file structure:

1. Create a `LibraryMedia.java` file containing the definition of the base class
2. Create a `Book.java` file containing the definition of the first subclass
3. Create a `Movie.java` file containing the definition of the second subclass
4. Create a `LibraryTest.java` file containing the test class with the `main` method

Code example:

```
public class LibraryTest {
    public static void main(String[] args) {
        // Creating objects of different types
        Book book1 = new Book("The Witcher", 1990, "Andrzej Sapkowski",
320);
        Movie movie1 = new Movie("The Green Mile", 1999, "Frank Darabont",
189);

        // Demonstrating polymorphism - storing different types in a base
class array
        LibraryMedia[] mediaArray = new LibraryMedia[2];
        mediaArray[0] = book1;   // Book object stored as LibraryMedia
        mediaArray[1] = movie1;  // Movie object stored as LibraryMedia

        // Demonstrating polymorphism - calling methods on different types
        System.out.println("===== INFORMATION ABOUT ALL MEDIA =====");
        for (LibraryMedia media : mediaArray) {
            // Same method name, but the correct implementation is called
            // depending on the actual object type (polymorphism)
            media.displayInformation();
            System.out.println("--------------------");
        }

        // Demonstrating borrowing and returning operations
        System.out.println("\n===== BORROWING AND RETURNING OPERATIONS
=====");
        book1.borrow();  // Borrowing the book
        book1.borrow();  // Attempting to borrow again - should show that
it's already borrowed
        book1.returnMedia();  // Returning the book

        // Calling type-specific methods
        System.out.println("\n===== TYPE-SPECIFIC METHODS =====");
        book1.checkNumberOfPages();   // Method specific to Book
        movie1.checkDuration();       // Method specific to Movie
```

```
        // Demonstrating casting to call type-specific methods via base
class reference
        System.out.println("\n===== TYPE CASTING =====");
        for (LibraryMedia media : mediaArray) {
            if (media instanceof Book) {
                // Cast and call the method specific to Book
                Book b = (Book) media;
                b.checkNumberOfPages();

                // Alternatively, use one-liner cast:
                // ((Book) media).checkNumberOfPages();
            } else if (media instanceof Movie) {
                // Cast and call the method specific to Movie
                ((Movie) media).checkDuration();
            }
        }
    }
}
```