Introduction to Java

Overview

In this lab, you'll create a new project and define a class that makes use of fundamental Java language features.

Source project directories

Student project directory: `Student/Student01_JavaIntro` (you'll create this)

Solution project directory: `Solutions/Solution01_JavaIntro`

Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Creating a project and adding a main class
2. Declaring and using variables
3. Working with operators
4. Additional suggestions

Exercise 1: Creating a project and adding a main class

Open Eclipse, and create a new Java project named `Student01_JavaIntro` in the `Student` directory.

In your project, create a new package named `student.javaintro` (this is the naming convention we use for packages on the Java part of this training course).

In the `student.javaintro` package, create a new Java class named `Main` (it doesn't actually matter what you call the class, but `Main` is as good as any ☺). In the `Main` class, write a simple `main()` method to act as the entry-point for your application.

Note the following points about Java vs. C/C++:

- In Java, all methods must belong to a class, even the `main()` method. This is different to C/C++, which allow global methods (i.e. outside of any classes).
- In Java, the `main()` method must take a `String[]` parameter (i.e. an array of strings passed in from the command line). This is different to C/C++, where `main()` is allowed to take no arguments.

Run the application as it stands, to make sure everything is OK so far.

Exercise 2: Declaring and using variables

In the `main()` method, add code to do the following:

- Declare some variables to hold information about a city (e.g. the name of the city, its population, its longitude and latitude, the average amount of rainfall in a year, the average number of hours sunshine in a year, etc). Choose appropriate names and types for all your variables.
- Display the details on the console.

Exercise 3: Working with operators

Explore the use of operators discussed in the chapter, including:

- Simple binary operators
- Unary operators (especially ++ and --, both the prefix and postfix versions)
- Assignment and compound assignment

Exercise 4 (If time permits): Additional suggestions

- At the end of the PowerPoint chapter there's a discussion of JAR files. Take a look at this section and have a go at creating and using JAR files in your application.

Defining and Using Classes

Overview

In this lab, you will write an application that defines and uses an `Employee` class. You will define methods and instance variables for the class, and create some instances of the class in the client code.

Source project directories

Student project directory: `Student/Student02_Classes`

Solution project directory: `Solutions/Solution02_Classes`

Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Defining a class
2. Creating objects
3. Overloading methods
4. Going further with object orientation

Exercise 1: Defining a class

Write an application that defines a simple `Employee` class and creates some instances.

Suggestions and requirements:

- The `Employee` class needs to hold the name and salary of the employee, and the date he/she joined the company.
- The class should have a constructor that initializes the employee's name and salary from passed-in values.
- The class should have another constructor that initializes the employee's name, and sets the salary to the minimum statutory salary (e.g. £7000 as a hard-coded figure). Make use of constructor chaining here.
- The class must honour the OO principle of encapsulation, so make sure the instance variables are `private`. Define `public` getter and setter methods if you need them.
- The class needs to allow an employee to have a pay raise, so define a `payRaise()` method that takes the amount of the pay raise and adds it to the employee's current salary.
- The class should also have a `toString()` method that returns a textual representation of the employee's info.

Exercise 2: Creating objects

Define a separate class, where you can create some `Employee` objects and invoke methods upon them. Exercise the methods fully, ensuring that they all work correctly.

Exercise 3: Overloading methods

In the `Employee` class, define a few overloaded versions of a `payBonus()` method.

- One version of the method can take no parameters and add a fixed percentage of the employee's salary (e.g. a 1% bonus).
- Another version of the method can take a `double` parameter that specifies the percentage of the bonus.
- Yet another version of the method can take three `double` parameters that specify the percentage of the bonus, along with a minimum and maximum salary (such that the bonus only applies if the employee's salary is within that range).

Exercise 4 (If time permits): Going further with object orientation

Define a suite of additional classes to represent related information about an employee. For example an employee has:

- A home address and a work address
- A home telephone number, a work telephone number, and a mobile telephone number
- A spouse (optionally)
- Personal information such as date of birth, place of birth, tax code, etc.

Inheritance

Extending Classes

Overview

In this lab, you will implement a simple library system. You will define an inheritance hierarchy to represent different types of items that can be borrowed from a library (books, DVDs, etc.).

Source project directories

Student project directory: `Student/Student03_Inheritance`

Solution project directory: `Solutions/Solution03_Inheritance`

Roadmap

There are 5 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Familiarization
2. Defining a base class
3. Defining subclasses
4. Writing the client application
5. Additional suggestions

Exercise 1: Familiarization

Open the student project, expand the `student.inheritance.extendingclasses` package, and take a look at the code in `Member.java`. This class represents a member in a library and has the following members:

- Instance variables containing the member's name and age, plus an integer indicating how many items the member has currently borrowed.
- A constructor, which initializes the instance variables.
- A `toString()` method, which returns a textual representation of the member's details.
- Methods named `borrowedItem()` and `returnedItem()`, which increment and decrement the "items borrowed" count respectively (you'll call these methods whenever an item is borrowed or returned by the member).

Exercise 2: Defining a base class

Define an `Item` class, which will be the base class for all the different types of item in the library. `Item` should be an `abstract` class - why?

The `Item` class should have the following instance variables:

- Title (a `String`, initialized in a constructor).
- Date borrowed (a `Date`, `null` initially to indicate it's not borrowed yet).
- Current borrower (a reference to a `Member` object, `null` initially).

The `Item` class should have the following instance methods:

- `isBorrowed()`
  Returns a `boolean` to indicate whether the item is currently borrowed (test the "current borrower" field to see if it's `null`).
- `canBeBorrowedBy()`
  Takes a `Member` object as a parameter, and returns a `boolean` to indicate whether this member is allowed to borrow this item. Returns `true` by default (subclasses might override this method, with different rules for whether a member can borrow specific types of item)
- `borrowItemBy()`
  Takes a `Member` object as a parameter, and returns a `boolean` to indicate whether the borrow succeeded. The implementation of this method should follow these guidelines:
  - The fundamental rule is that an item can't be borrowed if someone has already borrowed it!
  - There are additional constraints on who can borrow what, as defined by the `canBeBorrowedBy()` method, so invoke this method. If the method returns `true`, set the item's instance variables to record the borrower and the date borrowed. Also, invoke `borrowedItem()` on the `Member` object, to increment the number of items borrowed by this member.
  - Return `true` or `false`, to indicate whether the borrowing succeeded.
- `returnItem()`
  Takes no parameters, and returns `void`. The implementation of this method should follow these guidelines:
  - Call `returnedItem()` on the item's borrower, to decrement the number of items borrowed by that member.
  - Set the item's instance variables to record the fact that the item is no longer borrowed.
- `toString()`
  Returns a textual representation of the item's details, indicating whether the item is currently on loan (and to whom).

Exercise 3: Defining subclasses

Define `Book` and `DVD` classes, which inherit from the `Item` base class.

The `Book` class should have the following additional members, to extend the capabilities of the `Item` base class:

- Instance variables for the book's author, ISBN, and genre. Implement the genre as an enum (allowable options are `Children`, `Fiction`, `NonFiction`).
- Suitable constructor(s).
- An override for the `canBeBorrowedBy()` method. The policy for books is that any member can borrow fiction and non-fiction books, but only children (age <= 16) can borrow children's books.
- An override for `toString()`, to return a textual representation of a book (including all the basic information in the `Item` base class, of course).

The `DVD` class should have the following additional members, to extend the capabilities of the `Item` base class:

- Instance variables for the DVD playing time (in minutes) and classification. Implement the classification as an enum (allowable options are `Universal`, `Youth`, `Adult`).
- Suitable constructor(s).
- An override for the `canBeBorrowedBy()` method. "Universal" DVDs can be borrowed by anyone; "youth" DVDs can be borrowed by anyone 12 or over; and "adult" DVDs can be borrowed by anyone 18 or over.
- An override for `toString()`, to return a textual representation of a DVD (including all the basic information in the `Item` base class).

Exercise 4: Writing the client application

In `Main.java`, write a `main()` method to test the classes in your hierarchy.

Suggestions and requirements:

- First, create some `Member` objects with various names and ages.
- Then declare an `Item[]` array variable, which is capable of holding any "kind of item".
- Create some `Book` and `DVD` objects and place them in the array.
- Borrow some books and DVDs. Test the rules that govern whether a member is allowed to borrow a book.
- What happens if a member attempts to borrow an item that is already borrowed by someone else? Clearly this shouldn't be allowed, but does your application enforce this rule? Where would you write the code to make this test…?
- Write a loop to iterate through all the items and display each one. Verify that the correct `toString()` method is called on each item, thanks to polymorphism.

Exercise 5 (If time permits): Additional suggestions

In the `Item` class, define an `abstract` method named `dateDueBack()`. The method return type should be `Date`. The purpose of the method is to indicate when the item is due to be returned. This policy is different for each type of item, hence the reason for declaring it `abstract`.

In the `Book` class, override `dateDueBack()` so that it returns a date 21 days after the date the book was borrowed (you can use the `Calendar` class to do date arithmetic ☺).

In the `DVD` class, override `dateDueBack()` so that it returns a date 7 days after the date the DVD was borrowed.

Add some code in `main()` to test `dateDueBack()` policy.

Inheritance

Implementing Interfaces

(Optional Lab)

Overview

In this lab, you will add a "reservation service" to the library system that you worked on in the previous lab. The reservation service will allow members to reserve items that are currently borrowed by another member.

You will define a `Reservable` interface to identify the types of items that can be reserved (it's common for libraries to allow types of item to be reservable, but to disallow some other types of item from being reserved).

Source project directories

Student project directory: `Student/Student03_Inheritance`

Solution project directory: `Solutions/Solution03_Inheritance`

Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Defining an interface
2. Implementing the interface
3. Writing client code
4. Additional suggestions

Exercise 1: Defining an interface

In the student project, expand the `student.inheritance.implementinginterfaces` package. Take a quick look at the code we've provided as the starting point for this lab (this is the sample solution from the "Extending Classes" lab earlier).

Now define a new interface named `Reservable`, which represents "reservable capability". The `Reservable` interface should have the following methods:

- `isReserved()`
  Takes no parameters, and returns a `boolean` to indicate whether this item is currently reserved.
- `canBeReservedFor()`
  Takes a `Member` object as a parameter, and returns a `boolean` to indicate whether the member is allowed to reserve this item.
- `reserveItemFor()`
  Takes a `Member` object as a parameter, and returns a `boolean` to indicate whether the reservation succeeded.

Exercise 2: Implementing an interface

In the library system, not all books will be reservable. Popular books (such as those by bestselling authors) will be reservable because they are likely to be in high demand, but many books don't warrant this administrative overhead (e.g. would you bother reserving a book on Welsh Football Greats? I would, but most people probably wouldn't …. ☺).

With this in mind, define a new class named `ReservableBook`, to represent books which are reservable. `ReservableBook` must extend `Book` and implement the `Reservable` interface.

The `ReservableBook` class should have the following additional members, to extend the capabilities of the `Book` class:

- An instance variable indicating the member who has reserved the book (only one reservation is allowed at a time for books).
- Suitable constructor(s).
- An implementation of `isReserved()`, which returns a `boolean` to indicate whether there is a reserver already for this item.
- An implementation of `canBeReservedFor()`, which allows a reservation if:
    - The member passes the basic rules for borrowing a book (as specified in the `canBeBorrowedBy()` method).
- An implementation of `reserveItemFor()`, as follows:
    - If the book is currently borrowed by someone, is not currently reserved, and can be reserved by this member, reserve it and return `true`.
    - Otherwise, just return `false`.
- An override for the `returnItem()` method:
    - First, invoke superclass behaviour for this method, to perform the "normal" business rules for a returned item.
    - Then add some new code to test if the book is currently reserved by a member. If it is, immediately instigate a "borrow" operation for that member. In other words, as soon as a reserved book is returned, the reserver automatically becomes the new borrower. (Don't forget to set the "reserver" instance variable to `null` afterwards).

Exercise 3: Writing client code

Open `Main.java`, and uncomment the existing code. Then add some new code to test the reservation service.

Suggestions and requirements:

- Create a `Reservable[]` array, and populate it with all the *reservable* items (i.e. loop through the items and use `instanceof` to see if an item implements the `Reservable` interface. If it does, store a reference to the item in the `Reservable[]` array).
- Test whether you can reserve a `ReservableBook` that isn't yet borrowed. This should not be allowed.
- Test whether you can reserve a `ReservableBook` that has already been borrowed. This should succeed. Furthermore, when the book is eventually returned, it should be automatically borrowed by the member who reserved it.

Exercise 4 (If time permits): Additional suggestions

Implement reservation capabilities in the `DVD` class (all DVDs are reservable, because they're all very popular). A DVD can be reserved by up to 5 members at a time; use an array here.

Suggestions and requirements:

- Implement `isReserved()` to return a `boolean` to indicate whether there are any reservers for this item at the moment.
- Implement `canBeReservedFor()` to allow a reservation if:
  - The member passes the basic rules for borrowing a DVD (as specified in the `canBeBorrowedBy()` method).
- Implement `reserveItemFor()` as follows:
  - If the DVD is currently borrowed by someone, there are fewer than 5 reservations already, and the DVD can be reserved by this member, reserve it and return `true`. Be careful to ensure that reservations are stored in the order they are placed.
  - Otherwise, just return `false`.
- Override `returnItem()` as follows:
  - First, invoke superclass behaviour for this method.
  - Then add some new code to test if the DVD is currently reserved. If it is, immediately instigate a "borrow" operation for that first reserver. (Don't forget to remove the reserver from the list of reservers).
- Extend the `main()` code to test the reservation capabilities for DVDs.