Report

Module: CSP-101 Programming Paradigm

Name: Lanzheng Liu (Zayne)

HIC ID: 31423

Table of Contents

Task 1 - Procedural Programming

The syntax and semantics for procedural programming language (i.e. `C`) will be described in the comments.

```c
// Include statement.
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

// Macro definition.
#define Max_String_Length 30
#define Max_Record_List_Size 10

// `struct` definition.
struct CD
{
    int numberOfTracks;
    bool isAvailable;
    // String definition with fixed length.
    char title[Max_String_Length];
};

// Function definition.
int main(int argc, const char *argv[]){

    // Array of CD definition.
    struct CD cd_list[Max_Record_List_Size];

    // Variable definition and initialisation.
    struct CD cd1, cd2;
    int total_CDs = 0;

    strcpy(cd1.title, "title_CD1"); // Assign string value to variable.

    // Access struct data.
    cd1.numberOfTracks = 23;
    cd1.isAvailable = false;

    // Add item to array.
    dvd_list[0] = dvd1;
    dvd_list[1] = dvd2;

    // For loop.
    for (int i = 0; i<total_DVDs; i++) {
        printf("%s\n", dvd_list[i].title);
    }
    // Print function
    printf("%s\n", cd_list[i].title);

    return 0;
```

```
}
```

Task 2 - Object Oriented Programming

The syntax and semantics for OOP (object-oriented programming language) (i.e. `Java`) will be described in the comments.

```
// Package info (namespace)
package com.zayne.coursework;
// Import statement.
import java.util.ArrayList;

// Class definition.
public class Item {
    // Class member definition
    /** The title of the CD/DVD. */
    protected String title;

    // Property getter syntax
    public String getTitle() {
        return title;
    }

    // Property setter syntax
    public void setTitle(String title){
        this.title = title;
    }
    // Function declaration.
    public void Print(){

    }
}
```

```
package com.zayne.coursework;
import java.util.ArrayList;

public class Database {
   // Constructor function, is used to initialise data when creating the instanc
    public Database(){

    }
    // Class member & ArrayList definition.
    public static ArrayList<Item> itemList = new ArrayList<>(20) {
        // Overriding add method.
        @Override
        public boolean add(Item item) {
            // `if` statement.
```

```
            if (itemList.size() == 20) {
                System.out.println("CD list is full.");
                return false;
            }
            return super.add(item);
        }
    };

    public void addItem(Item item) {
        // Accessing class instance method.
        itemList.add(item);
    }

    public void ListAllItems() {
        // Print function.
        System.out.println("====== Items ======");

        // `foreach` loop.
        for (var item : itemList) {
            item.Print();
        }
    }
}
```

Task 3 - Advanced OOP: Inheritance and Polymorphism

Inheritance

The concept of inheritance in OOP is created to further describe the relationships between modeled 'real-world objects' (i.e. the relationship between different classes). Inheritance makes some shared code reusable (i.e. less code repeated), by describing it with a parent-child relationship. Parent class has the shared logic (i.e. common properties and methods), and all child classes can inherit the shared data from their parent class.

Before inheritance

Class CD

```
public class CD {
    protected String title;
    protected int playingTime;
    protected boolean isAvailable;
    protected String comments;
    private String artist;
    private int numOfTracks;
```

```
    // Getters
    public String getTitle() { return title; }
    public int getPlayingTime() { return playingTime; }
    public boolean getIsAvailable() { return isAvailable; }
    public String getComments() { return comments; }
    public String getArtist() { return artist; }
    public int getNumOfTracks() { return numOfTracks; }

    // Setters
    public void setArtist(String artist) { this.artist = artist; }
    public void setNumOfTracks(int numOfTracks) { this.numOfTracks = numOfTracks
    public void setTitle(String title) { this.title = title; }
    public void setPlayingTime(int playingTime) { this.playingTime = playingTime
    public void setIsAvailable(boolean available) { isAvailable = available; }
    public void setComments(String comments) { this.comments = comments; }
}
```

Class DVD

```
public class DVD {
    protected String title;
    protected int playingTime;
    protected boolean isAvailable;
    protected String comments;
    private String director;

    // Getters
    public String getTitle() { return title; }
    public int getPlayingTime() { return playingTime; }
    public boolean getIsAvailable() { return isAvailable; }
    public String getComments() { return comments; }
    public String getDirector() { return director; }

    // Setters
    public void setTitle(String title){ this.title = title; }
    public void setPlayingTime(int playingTime) { this.playingTime = playingTime
    public void setIsAvailable(boolean available) { isAvailable = available; }
    public void setComments(String comments) { this.comments = comments; }
    public void setDirector(String director) { this.director = director; }
}
```

There are many duplicated code in the example above as you might noticed. However with inheritance, we can avoid writing huge amount of repeated code by extract duplicated information into a parent class (i.e. super class).

After inheritance

Parent class `Item.java`

```
public class Item {
    protected String title;
    protected int playingTime;
    protected boolean isAvailable;
    protected String comments;

    // Getters
    public String getTitle() { return title; }
    public int getPlayingTime() { return playingTime; }
    public boolean getIsAvailable() { return isAvailable; }
    public String getComments() { return comments; }

    // Setters
    public void setTitle(String title){ this.title = title; }
    public void setPlayingTime(int playingTime) { this.playingTime = playingTime
    public void setIsAvailable(boolean available) { isAvailable = available; }
    public void setComments(String comments) { this.comments = comments; }
}
```

Child class `CD.java`

```
public class CD extends Item {
    private String artist;
    private int numOfTracks;

    // Getters
    public String getArtist() { return artist; }
    public int getNumOfTracks() { return numOfTracks; }

    // Setters
    public void setArtist(String artist) { this.artist = artist; }
    public void setNumOfTracks(int numOfTracks) { this.numOfTracks = numOfTracks
}
```

Child class `DVD.java`

```
public class DVD extends Item {
    private String director;

    // Getters
    public String getDirector() { return director; }

    // Setters
    public void setDirector(String director) { this.director = director; }
}
```

In the examples above, we extracted the common fields and methods into class `Item`, let `DVD` and `CD` inherit class `Item` and then, add only `DVD` and `CD` specific fields and methods into class `DVD` and `CD`. By doing so the amount of repeated code is heavily reduced.

Note: In the example the access modifier `protected` means protected methods or fields are only directly accessible to classes with in the same package, and in addition, protected members can be accessed via its child classes (i.e. sub-classes) out side of the current package (Oracle, 2020).

Here's a detailed access level table for access modifiers (Oracle, 2020).

Modifier	Class	Package	Sub-class	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier package-private	✓	✓		
`private`	✓			

Polymorphism

Polymorphism provides a way for a child class to define its own version of a method which the child class inherited from its parent class, this process is called method `overriding`.

`Item.java`

```
public class Item{
    public void Print(){

    }
}
```

`DVD.java`

```
public class DVD extends Item {
    // Overriding parent method `Print()`
    public void Print(){
        System.out.printf("Title: \t\t%s\n",getTitle());
        System.out.printf("Director: \t%s\n",getDirector());
        System.out.printf("Time: \t\t%s\n",getPlayingTime());
        System.out.printf("Available: \t%s\n",getIsAvailable());
        System.out.printf("Comments: \t%s\n",getComments());
        System.out.println("--------------------");
```

```
    }
}
```

`CD.java`

```
public class CD extends Item {
    // Overriding parent method `Print()`
    public void Print(){
        System.out.printf("Title: \t\t%s\n",getTitle());
        System.out.printf("Artist: \t%s\n",getArtist());
        System.out.printf("Tracks: \t%s\n",getNumOfTracks());
        System.out.printf("Time: \t\t%s\n",getPlayingTime());
        System.out.printf("Available: \t%s\n",getIsAvailable());
        System.out.printf("Comments: \t%s\n",getComments());
        System.out.println("--------------------");
    }
}
```

Apart from its usage with inheritance, polymorphism its self can alter the behaviour of any method by simply `overloading` the method with same method name, but different function signature(parameter names, parameter types and order of parameters).

Task 4

Procedural Programming vs. Object-Oriented Programming

Procedural programming paradigm is based on the concept of *"procedure call"*, which is essentially a series of steps or commands executing sequentially. Procedures (a.k.a. routines, subroutines, or functions) are the building blocks of procedural programming, with `if`, `while`, and `for` statement to implement *control flow* [citation needed].

Control flow can be categorized by their behaviour:

- Branching (i.e. to determine which part of the program should be executed based on certain conditions)
  - `if...else...`
  - `switch...case...`
- Iteration (i.e. to repeatedly execute identical operations)
  - `for` / `foreach`
  - ```
    // for loop in C.
    for (int i = 0; i<total_CDs; i++) {
    ```
    ```

```
        printf("%s\n", cd_list[i].title);
    }
```

There is `struct` in C to define structural data, but is not as powerful as `Class` provided by OOP.

```
struct CD
{
    int numberOfTracks;
    bool isAvailable;
    // Total time of playing.
    int playingTime;
    // String definition with fixed length.
    char artist[Max_String_Length];
    char title[Max_String_Length];
    char comments[Max_String_Length];
};
```

Whereas object-oriented programming paradigm is based on the concept of modeling real world `objects`, where each kind of objects is called a `class`. Classes encapsulate both `data` (i.e. fields, attributes or properties) and `behaviour` (i.e. methods) of certain object types.

```
public class Database {
    // Data.
    public static ArrayList<Item> itemsList = new ArrayList<>(20);

    // Behaviours - START
    public void addItem(Item item) { itemsList.add(item); }

    public void ListAllItems() {
        System.out.println("====== Items ======");
        // Foreach loop to iterate through `itemList`
        for (var item : itemsList) {
            item.Print();
        }
    }
    // Behaviours - END
}
```

Essentially, object-oriented programming empowers procedural programming with a much better way to describe and structure data while keeping the concepts of *Variables* and *Procedures*.

Also, in the code below, each instance of `Item` class has a different set of values for their non-static properties (in this case, the value of protected member `title`), the keyword `this` is used to refer to the current instance, in order to access instance specific data (Troelsen, 2012).

```java
public class Item {
    protected String title;

    // Getters
    public String getTitle() { return title; }

    // Setters
    public void setTitle(String title) { this.title = title; }
    public void Print() { }
}
```

The process of wrapping different kinds of data related to a certain type of object is called *Encapsulation*, which is the first pilar of OOP (Troelsen, 2012).

Conclusion

Encapsulation, Inheritance and Polymorphism as the foundation of OOP, provides us with the ability to describe and structure data with complex relationship with encapsulation, reducing the amount of code repeated by improving code re-usability with inheritance, and increasing the flexibility of class behaviours with polymorphism.

References

- Troelsen, A., 2012. "Understanding Inheritance and Polymorphism", Sixth edn, Apress, Berkeley, CA, pp. 213-251. Available at http://tinyurl.com/y288lopo.
- Oracle, 2020, Controlling Access to Members of a Class (The Java™ Tutorials > Learning the Java Language > Classes and Objects). Accessed on 8th August 2020. Available at https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html.