VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED PROGRAMMING (CO2039) - CC01

Advanced Programming
Final Report

Advisor: Trương Tuấn Anh
Student: Nguyễn Đình Sáng - 1952955

HO CHI MINH CITY, AUGUST 2021

Contents

1 Question 1

1.1 Topic

List and explain the key advantages of Polymorphism in C++ (also provide C++ example code to demonstrate your explanation).

1.2 What is Polymorphism in C++?

Polymorphism refers to the fact that something exists in multiple forms. Polymorphism, in simple terms, is the ability of a message to be displayed in multiple formats. A real-life example of polymorphism is when two people have distinct characteristics at the same time. At the same time, he is a father, a husband, and a worker. As a result, the same person behaves differently in different situations.

1.3 List and explain the key advantages of Polymorphism in C++

1.3.1 Compile-time Polymorphism

1.3.1.a Method overloading

When many functions with the same name but different parameters exist, they are said to be overloaded. Changes in the amount of arguments or the kind of arguments can cause functions to become overloaded.

methodOverloading.cpp > ...

```cpp
// C++ program to illustrate the concept of Method overloading
#include <iostream>
using namespace std;
class Hero {
 public:
  int add(int x, int y) { return x + y; }
  int add(int x, int y, int z) { return x + y + z; }
};

int main() {
    Hero hero;
    cout << hero.add(10,20);
    cout << '\n';
    cout << hero.add(10,20,30);
}
/**Output:
 * 30
 * 60
 */
```

C++ program to illustrate the concept of Method overloading

In the above example, a single function named **add** acts differently in three different situations which is the property of polymorphism. This thing is really helpful when we have many functions has the same names and do the same things with different the number of parameters.

1.3.1.b Operator overloading

Operator overloading is possible in C++. For example, we can use the string class's operator('+') to concatenate two strings. This is the addition operator, and its job is to add two operands.So, when the operator '+' is used between integer and string operands, it adds them together,and when it is used between string operands, it concatenates them.

Let's see an example of operator overloading in C++

operatorOverloading.cpp > ...

```cpp
// CPP program to illustrate
// Operator Overloading
#include <iostream>
using namespace std;

class Addition {
 private:
  int num1, num2;

 public:
  Addition(int r = 0, int i = 0) {
    num1 = r;
    num2 = i;
  }

  // This is automatically called when '+' is used with
  // between two Add addObjects
  Addition operator+(Addition const &addObj) {
    Addition result;
    result.num1 = num1 + addObj.num1;
    result.num2 = num2 + addObj.num2;
    return result;
  }
  void print() {
    cout << "[Object 2 information]: num1 = " << num1 << ", num2 = " << num2;
  }
};

int main() {
  Addition obj1(10, 15), obj2(20, 45);
  Addition obj3 = obj1 + obj2;  // An example call to "operator+"
  obj3.print();
}
/**Output
 * [Object 2 information]: num1 = 30, num2 = 60
 */
```

C++ program to illustrate Operator Overloading

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(**num1** and **num2**) but here the operator is made to perform addition of elements in 2 objects which are **obj1** and **obj2**.

1.3.2 Run-time Polymorphism

1.3.2.a Method overriding

Run-time polymorphism can be achieved by method overriding, it occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

methodOverriding.cpp > ...

```cpp
#include <iostream>
using namespace std;

class Add {
 public:
  virtual void print() {
    int a = 20, b = 30;
    cout << " base class Action is:" << a + b << endl;
  }
  void show() { cout << "show base class" << endl; }
};

class Sub : public Add {
 public:
  void print()  // print () is already virtual function in derived class, we
                // could also declared as virtual void print () explicitly
  {
    int x = 20, y = 10;
    cout << " derived class Action:" << x - y << endl;
  }

  void show() { cout << "show derived class" << endl; }
};

// main function
int main() {
  Add *aptr;
  Sub s;
  aptr = &s;

  // virtual function, binded at runtime (Runtime polymorphism)
  aptr->print();

  // Non-virtual function, binded at compile time
  aptr->show();

  return 0;
}
```

C++ program to illustrate Method Overriding

2 Question 2

2.1 Topic

Compare object-oriented programming in C++ and Java. Draw your own conclusions.

2.2 Background

2.2.1 Paradigms

C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.

Java is a general-purpose, class-based, object-oriented programming language designed for having lesser implementation dependencies.

2.2.2 Platform dependency

C++ is platform dependent. When a program is written and compiled in C/C++ language, the code is directly converted into machine readable language .i.e. executable code. This code is generated as .exe file. This generated .exe file can run only on specific operating system. i.e. when the program is compiled in windows OS .exe file can run only in windows OS and not on Unix OS.

Java is platform independent. When the Java program runs in a particular machine it is sent to java compiler, which converts this code into intermediate code called bytecode. This bytecode is sent to Java virtual machine (JVM) which resides in the RAM of any operating system. JVM recognizes the platform it is on and converts the bytecodes into native machine code. Hence java is called platform independent language.

2.2.3 What is Object-oriented programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software.

The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.

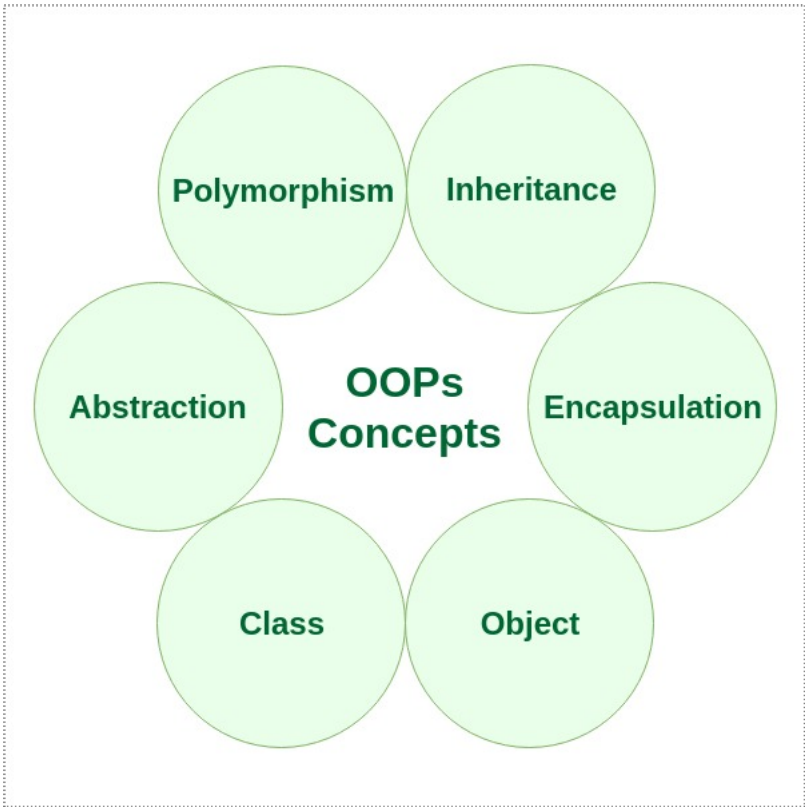
2.3 OOP: Similarities and Differences between Java and C++

2.3.1 Similarities

2.3.1.a Core concepts

Both C++ and Java support **6 core concepts of OOP**: class, object, abstraction, encapsulation, inheritance, and polymorphism. *(Example codes are provided in Differences part).*

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions



Characteristics in Object Oriented Programmings (Author: GeeksforGeeks)

or methods).

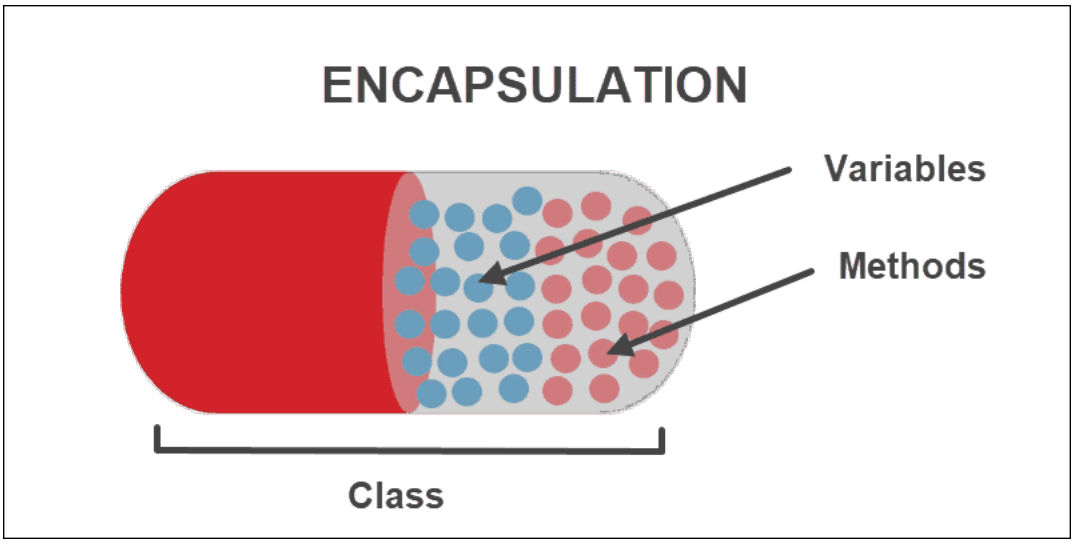
The user-defined objects are created using the class keyword. The class is a blueprint that defines a nature of a future object. An instance is a specific object created from a particular class. Classes are used to create and manage new objects and support inheritance—a key ingredient in object-oriented programming and a mechanism of reusing code.

Object is an instance of a class. An object in OOPS is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. For example color name, table, bag, barking. When you send a message to an object, you are asking the object to invoke or execute one of its methods as defined in the class.

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Encapsulation is one of the fundamentals of OOP (object-oriented programming). It refers to the bundling of data with the methods that operate on that data. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties'

direct access to them. Publicly accessible methods are generally provided in the class (so-called getters and setters) to access the values, and other client classes call these methods to retrieve and modify the values within the object.



Encapsulation (Author: phoenixNAP)

Inheritance is one of the most important aspects of Object Oriented Programming (OOP). The key to understanding Inheritance is that it provides code re-usability. In place of writing the same code, again and again, we can simply inherit the properties of one class into the other.

Polymorphism in OOPs is inseparable and an essential concept of every object-oriented programming language. An object or reference basically can take multiple forms in different instances. As the word suggests, 'poly' means 'many' and 'morph' points at 'forms'; thus, polymorphism as a whole would mean 'a property of having many forms.

2.3.1.b Static and dynamic binding

Binding generally refers to mapping of one thing to another. In the context of compiled languages, binding is the link between a function call and the function definition. When a function is called in C++, the program control binds to the memory address where that function is defined.

There are 2 types of binding in C++ and Java: Static (or early binding) and dynamic (or late) binding.

Static binding happens at the compile-time. Static binding happens when all information needed to call a function is available at the compile-time. Static binding can be achieved using the normal function calls, function overloading, and operator overloading.

```cpp
// C++ program to illustrate the concept of static binding
#include <iostream>
using namespace std;

class Hero {
 public:
  int attack(int strength, int blood) { return strength * blood;}
  int attack(int strength, int blood, int heal) { return strength + blood + heal; }
};

int main() {
  Hero hulk;
  cout << " Strength stat: " << hulk.attack(10, 20) << endl;
  cout << " Strenght stat: " << hulk.attack(10, 20, 30) << endl;

  return 0;
}
/**Output:
 * Strength stat: 200
 * Strenght stat: 60
 */
```

C++ program to illustrate the concept of Static Binding

Consider the above code, where the **attack()** function is overloaded to accept two and three integer arguments. Even though there are two functions with the same name inside the **Hero** class, the function call **attack()** binds to the correct function depending on the parameters passed to those functions. This binding is done statically during compile time.

```java
package com.example;
// Java program to illustrate the concept of static binding
public class Hero {
    private void attack(){
        System.out.println("The hero is attacking...");
    }
    Run | Debug
    public static void main(String[] args) {
        Hero hero = new Hero();
        hero.attack();
    }
}
/**Output:
 * The hero is attacking...
 */
```

Java program to illustrate the concept of Static Binding

Dynamic binding happens at the run-time, the functions are not resolved until run-time. Dynamic binding happens when the compiler cannot determine all information needed for a function call at compile-time, it can be achieved using vitual functions in C++. The major advantage of dynamic binding is that is is flexible since a single function can handle different types of objects at run-time. This significantly reduces the size of the code base and also makes the source code more readable.

c++ > C++ dynamicBinding.cpp > ...

```cpp
// C++ program to illustrate the concept of dynamic binding
#include <iostream>
using namespace std;

class Hero {
 public:
  // Virtual function
  virtual void f() { cout << "Base class [Hero] is called.\n"; }
};

class Hulk : public Hero {
 public:
  void f() { cout << "The derived class [Hulk] is called.\n"; }
};

int main() {
  Hero base;
  Hulk derived;

  Hero *basePtr = &base;
  basePtr->f();

  basePtr = &derived;
  basePtr->f();

  return 0;
}
/**Output:
 * Base class [Hero] is called.
 * The derived class [Hulk] is called.
 */
```

C++ program to illustrate the concept of Dynamic Binding

Consider the above program, **Hero** is the base class and has a derived class **Hulk**, **Hero** contains a virtual function **f()** which is overridden by **Hulk::f()**, which **f()** called depends on type of object pointed to **basePtr**. This process can only be available at run-time, and hence **f()** is subject to dynamic binding.

We can see another example bellow in Java, when we assign **hero = hulk** in line 18, if this line does not exist the output will be:

The hero is attacking...
Hulk is attacking...

hero is dynamically assigned to hulk so the output changed to 2 lines of "**Hulk is attacking...**"

src > main > java > com > example > DynamicBinding.java > ...

```java
package com.example;
// Java program to illustrate the concept of static binding

class Hero{
    public void attack(){
        System.out.println("The hero is attacking...");
    }
}
class Hulk extends Hero{
    public void attack(){
        System.out.println("Hulk is attacking...");
    }
}
class DynamicBinding {
    Run | Debug
    public static void main(String[] args) {
        Hero hero = new Hero();
        Hero hulk = new Hulk();
        hero = hulk;
        hero.attack();
        hulk.attack();
    }
}
/**Output:
 * Hulk is attacking...
 * Hulk is attacking...
 */
```

Java program to illustrate the concept of Dynamic Binding

2.3.1.c Method overloading

Method overloading is the process of overloading the method that has the same name but different parameters. C++ and Java provide this method of overloading features. Method overloading allows users to use the same name to another method, but the parameters passed to the methods should be different. The return type of methods can be the same or different.

C++ methodOverloading.cpp > ...

```cpp
// C++ program to illustrate the concept of Method overloading
#include <iostream>
using namespace std;
class Hero {
 public:
  int add(int x, int y) { return x + y; }
  int add(int x, int y, int z) { return x + y + z; }
};

int main() {
    Hero hero;
    cout << hero.add(10,20);
    cout << '\n';
    cout << hero.add(10,20,30);
}
/**Output:
 * 30
 * 60
 */
```

C++ program to illustrate the concept of Method Overloading

src > main > java > com > example > MethodOverloading.java > ...

```java
package com.example;
//Java program to illustrate the concept of Method overloading
class MethodOverloading{
    public void print(){System.out.println("There is no parameter");}
    public void print(int x, int y){
        System.out.println("Product: " + (x*y));
    }
    public void print(double x, double y){
        System.out.println("Sum: " + (x+y));
    }
    Run | Debug
    public static void main(String[] args){
        MethodOverloading obj = new MethodOverloading();
        obj.print();
        obj.print(2,4);
        obj.print(4.5,7.8);
    }
}
/**Output
 * There is no parameter
 * Product: 8
 * Sum: 12.3
 */
```

Java program to illustrate the concept of Method Overloading

2.3.2 Differences

2.3.2.a Class and Object

See class and object definitions at: 2.3.1.a

With C++, in addition to Classes, we can you Structures, and Unions in OOP.

class.cpp > Avenger

```cpp
// C++ program to illustrate the concept of the Class
#include <iostream>
using namespace std;
// C++ supports unions
union Evil {
  int strength;
  int bornYear;

 public:
  void showInfo() {
    cout << "[Evil information] "
         << "strength: " << strength << " bornYear: " << bornYear << endl;
  }
};
```

Union in C++

```cpp
// C++ supports Structures
struct Avenger {
  // public as default
  int age;
  string name;

 public:
  Avenger(int age, string name) : age(age), name(name) {}
  int getHeight() { return age; }
  string getName() { return name; }
  void setHeight(int age) { this->age = age; }
  void setName(int name) { this->name = name; }

  void showInfo() {
    cout << "[Avenger information] "
         << "age: " << age << " name: " << name << endl;
  }
};
```

Structure in C++

```cpp
class Hero {
  // private as default
  int height;
  int weight;

 public:
  Hero(int height, int weight) : height(height), weight(weight) {}
  int getHeight() { return height; }
  int getWeight() { return weight; }
  void setHeight(int height) { this->height = height; }
  void setWeight(int weight) { this->weight = weight; }

  void showInfo() {
    cout << "[Hero information] "
         << "height: " << height << " weight: " << weight << endl;
  }
};

int main() {
  Hero hero(100, 200);
  Avenger avenger(100, "Iron man");
  Evil evil;
  evil.strength = 300;
  evil.bornYear = 2000;

  // Show information of the objects: hero, avenger, and evil
  hero.showInfo();
  avenger.showInfo();
  evil.showInfo();
}
/**Output
 * [Hero information] height: 100 weight: 200
 * [Avenger information] age: 100 name: Iron man
 * [Evil information] strength: 2000 bornYear: 2000
 */
```

Class in C++

Java supports only Class.

```java
src > main > java > com > example > JavaClass.java > ...
package com.example;
// Class in Java
public class JavaClass {
    Run | Debug
    public static void main(String[] args) {
        Warrior warrior = new Warrior();
        System.out.println(
                "[Hulk information] strength: "
                + warrior.getStrength() + " blood: " + warrior.getBlood());

    }
}

class Warrior {
    private int strength = 100;
    private int blood = 200;

    public int getStrength() {
        return strength;
    }

    public int getBlood() {
        return blood;
    }

    public void setStrength(int strength) {
        this.strength = strength;
    }

    public void setBlood(int blood) {
        this.blood = blood;
    }
}
/**Output
 * [Hulk information] strength: 100 blood: 200
 */
```

Class in Java

How C++ and Java allocate memory? When you declare a variable with the type of a class in C++, storage for an object of that class is allocated, and the constructor function of that class is called to initialize that instance of the class. You're actually defining a pointer to a class object in Java; no storage is allocated for the class object, and the constructor method isn't called until you use "new".

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

```cpp
Hero H;   // H is a Hero; the Hero constructor function is called to initialize H.
Hero *h;  // p is a pointer to a Hero;
          // no Hero object exists yet, no constructor function has been called
h = new Hero;  // now storage for a Hero has been allocated
               // and the constructor function has been called
H.doSth();     // call H's doSth function
h->doSth();    // call the doSth function of the Hero pointed to by h
```

Declaring objects in C++

```
Hero J;          // J is a pointer to a Hero; no Hero object exists yet
J = new Hero();  // now storage for a Hero has been allocated
                 // and the constructor function has been called;
                 // we must use parentheses even when you are not
                 // passing any arguments to the constructor function
J.doSth();       // no -> operator in Java -- just use .
```

Declaring objects in Java

One more special things about object in Java is **Every object is an Object**.
Any class that doesn't have an extends clause implicitly inherits **Object**. Thus you never have to code a class like this:

```
public class Product extends Object...
```

If a subclass has an extends clause that specifies a super class other than **Object**, the class still inherits **Object**. That's because the inheritance hierarchy eventually gets to a super class that doesn't have an extends clause, and that super class inherits Object and passes it down to all its sub classes. Suppose you have these classes:

```
public class Manager extends SalariedEmployee...
public class SalariedEmployee extends Employee...
public class Employee extends Person...
public class Person...
```

Here the **Manager** class inherits the **Object** class indirectly because it inherits **SalariedEmployee**, which inherits **Employee**, which inherits **Person**, which inherits **Object**.

In Java, creating a class that doesn't inherit **Object** is not possible.

2.3.2.b Abstraction

See abstraction definition at: 2.3.1.a

Benefits of data abstraction:

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

abstraction.cpp > ...

```cpp
#include <iostream>
using namespace std;

class Warrior {
 private:
  int attack;
  int defend;

 public:
  // A pure virtual function
  virtual void fight() = 0;
  // method to set values of
  // private members
  void setInfo(int attack, int defend) {
    this->attack = attack;
    this->defend = defend;
  }
  void display() {
    cout << "attack = " << attack << endl;
    cout << "defend = " << defend << endl;
  }
};
class Hero : public Warrior {
 public:
  void fight() { cout << "Fighting..."; }
};
int main() {
  Hero Hulk;
  Hulk.setInfo(100, 200);
  Hulk.display();
  Hulk.fight();
  return 0;
}
/** Output
 * attack = 100
 * defend = 200
 * Fighting...
 */
```

C++ prgram to illustrate the concept of Abstraction

In the above C++ program: You can see in this program we are not allowed to access the variables **attack** and **defend** directly, however one can call the function **setInfo()** to set the values in **attack** and **defend** and the function **display()** to display the values of **attack** and **defend**.

We also have pure virtual functions (or abstract functions) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class. As in above example C++:

```cpp
virtual void fight() = 0;

```

The function **fight()** in class **Warrior** is overridden by **fight()** in class **Hero**.

Java abstraction

Abstract classes or **interfaces** can be used to accomplish abstraction in Java.
For classes and methods, the abstract keyword is a non-access modifier:

- **Abstract class**: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method**: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword **abstract** is used to make a class abstract.

src > main > java > com > example > Animal.java > ...

```java
package com.example;

// Abstract class
abstract class Animal {
    // Constructor
    Animal() {
        System.out.println("Base class Animal is called.");
    }
    // Abstract method (does not have a body)
    public abstract void say();
    // Final medthod can not be overridden
    final void sayHello() {
        System.out.println("Hello");
    }
    // Regular method
    public void sleep() {
        System.out.println("Zzz");
    }
}
// Abstract class without any abstract methos
abstract class Mammal {
    void say() {
        System.out.println("Mammal is called.");
    }
}
// Subclass (inherit from Animal)
class Dog extends Animal {
    public void say() {
        // The body of say() is provided here
        System.out.println("The Dog says: gaw gaw gaw");
    }
}
class Main {
    Run | Debug
    public static void main(String[] args) {
        Dog myDog = new Dog(); // Create a Dog object
        myDog.say();
        myDog.sleep();
    }
}
```

Java program to illustrate the concept of Abstraction

Like C++, in Java, an instance of an abstract class cannot be created, we can have references to abstract class type though.
Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created. For example, **myDog** in line 24 in the above program.
In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited. The class **Mamal** in the above code is an example.

Abstract classes can also have final methods (methods that cannot be overridden). For example, final method **sayHello()** in line 12.

Java interface
Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body)

src > main > java > com > example > Animal.java > ...

```java
package com.example;

// Interface
interface Animal {
    public void say(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}

// Dog "implements" the Animal interface
class Dog implements Animal {
  public void say() {
    // The body of say() is provided here
    System.out.println("The Dog says: gaw gaw");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  Run | Debug
  public static void main(String[] args) {
    Dog myDog = new Dog();  // Create a Dog object
    myDog.say();
    myDog.sleep();
  }
}
```

Java program to illustrate the concept of Abstraction with Interface

2.3.2.c Inheritance

Inheritance serves the same purpose in C++ and Java. Both languages use inheritance to reuse code and/or create a "is-a" relationship. The differences will be seen in the following examples. Inheritance is supported in Java and C++.

- In Java, all classes either directly or indirectly inherit from the Object class. As a result, in Java, there is always a single inheritance tree of classes, with the Object class as the root. When you create a class in Java, it derives from the Object class by default. However, in C++, there is a forest of classes, and whenever we build a class that does not inherit from another, we are creating a new tree in the forest.

```java
src > main > java > com > example > Inheritance.java > ...
package com.example;

public class Inheritance {
    // members of Inheritance
    Run | Debug
    public static void main(String[] args) {
        Inheritance s = new Inheritance();
        System.out.println("s is instanceof Object: " + (s instanceof Object));
    }
}
/**Output
 * s is instanceof Object: true
 */
```

- In Java, members of the grandparent class are not directly accessible because it violates encapsulation. We can access parent class with **super**.

```java
src > main > java > com > example > Inheritance.java > ...
package com.example;

public class Inheritance {
    // members of Inheritance
    public void print(){System.out.println("Parent class called.");}
    Run | Debug
    public static void main(String[] args) {
        Child child = new Child();
        child.print();
    }
}

class Child extends Inheritance {
    public void print(){
        // Call print from parent class
        super.print();
    }
}
/**Output:
 * Parent class called.
 */
```

- In Java, the **protected** member access specifier has a somewhat different meaning. Protected members of a class **A** in Java can be accessed by another class **B** in the same package, even though **B** does not inherit from **A**. (they both have to be in the same package).

```java
src > main > java > com > example > B.java > ...
package com.example;

// Filename B.java
class A {
    protected int x = 10, y = 20;
}

class B {
    Run | Debug
    public static void main(String args[]) {
        A a = new A();
        System.out.println("[a information]: x = " + a.x + " y = " + a.y);
    }
}
/**Output
 * [a information]: x = 10 y = 20
 */
```

- For **inheritance**, Java utilizes the **extends** keyword. Unlike C++, Java lacks inheritance specifiers such as **public**, **protected**, and **private**. As a result, in Java, we can't change the protection level of base class members; if a data member is public or protected in the base class, it will remain public or protected in the derived class.

  In C++ we can change protection level of parent class with additional access modifier when using inheritance like this:

```cpp
class Game : protected Warrior{};
class Avenger: private Warrior{};
class Hero : public Warrior {};
class Revenge : private Hero{};
// And more
```

  Unlike C++, in Java, we don't have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.

- In Java, methods are virtual by default. In C++, we explicitly use **virtual** keyword.
- **Multiple inheritance** is not supported in Java, unlike C++; a class cannot inherit from more than one class. A class, on the other hand, can **implement** many **interfaces**.

```cpp
multipleInheritance.cpp > A
#include<iostream>

class A{
    // members of class
};
class B{
    // members of class B
};
// We can multiple inherit class in C++
// But this is not supported in Java
class C : public A, protected B{
    // members of class C
};
```

Java also supports multiple interface:

src > main > java > com > example > Animal.java > ...

```java
package com.example;

// Interface
interface Animal {
    public void say(); // interface method (does not have a body)

    public void sleep(); // interface method (does not have a body)
}
interface Mammal {
    public void sayHello();
}

// Dog "implements" the Animal interface
class Dog implements Animal, Mammal {
    public void say() {
        // The body of say() is provided here
        System.out.println("The Dog says: gaw gaw");
    }
    public void sayHello() {
        // The body of sayHello() is provided here
        System.out.println("Hello");
    }
    public void sleep() {
        // The body of sleep() is provided here
        System.out.println("Zzz");
    }
}

class Main {
    Run | Debug
    public static void main(String[] args) {
        Dog myDog = new Dog(); // Create a Dog object
        myDog.say();
        myDog.sleep();
        myDog.sayHello(); // print Hello
    }
}
```

Java program to illustrate the concept of Inheritance with Interface

- The default constructor of the parent class is automatically called in C++, but we must use the Initializer list if we wish to call a parameterized constructor of a parent class. In Java, the parent class's default constructor is called automatically, just like in C++, but if we wish to use a parameterized constructor, we must use super to call the parent constructor.

In C++

pointersAndReferences.cpp > main()

```cpp
/**
 * Pointers and reference to the base class of derived objects
 */
#include <iostream>
#include <string>

class Base {
 protected:
  int m_value{};

 public:
  Base(int value) : m_value{value} {}

  std::string getName() const { return "Base"; }
  int getValue() const { return m_value; }
};

class Derived : public Base {
 public:
 // We cal parameterized constructor of the base class here
  Derived(int value) : Base{value} {}

  std::string getName() const { return "Derived"; }
  int getValueDoubled() const { return m_value * 2; }
};
```

Call a parameterized constructor of parent class in C++

In Java

src > main > java > com > example > Base.java > ...

```java
package com.example;

class Base {
    private int b;

    Base(int x) {
        this.b = x;
        System.out.println("Base constructor called");
    }
}

class Derived extends Base {
    private int d;

    Derived(int x, int y) {
        // Calling parent class parameterized constructor
        // Call to parent constructor must be the first line in a Derived class
        super(x);
        d = y;
        System.out.println("Derived constructor called");
    }
}
```

Call a parameterized constructor of parent class in Java

2.3.2.d Polymorphism

Compile-time polymorphism

The main difference between Java and C++ in polymorphism is **operator overloading**.

· **Operator overloading**: this concept is described at 1.3.1.b

· **Method overloading**: this concept is described at 2.3.1.c

Run-time polymorphism

· **Method overriding** (function overriding)

Like C++, Java also supports Method overrding and it occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

src > main > java > com > example > Polumorphism.java > ...

```java
package com.example;

// Java program for Method overriding
class Parent {

    void Print() {
        System.out.println("This is parent");
    }
}

class subclass1 extends Parent {

    void Print() {
        System.out.println("This is subclass1");
    }
}

class subclass2 extends Parent {

    void Print() {
        System.out.println("This is subclass2");
    }
}

class Polymorphism {
    Run | Debug
    public static void main(String[] args) {

        Parent a;

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Java program to illustrate method overriding

In the above code the function **Print()** in Parent is overriden by **Printt()** in **subclass1** and **subclass2**.

2.4 Conclusion

OOPS ideas are supported by both C++ and Java programming languages. C++ allows for runtime flexibility and can create complex type hierarchies. C++ is based on C and has features that are backward compatible with C. It's a low-level programming language with some high-level features thrown in for good measure.

Memory management in C++ is a manual procedure that requires the programmer's attention, which can lead to memory leaks and segmentation faults. Java features a built-in garbage collector that maintains track of allocated memory for objects and releases it when they're no longer needed.

Java is a strongly typed programming language with a variety of primitives and object types. It allows primitives to be transformed into their respective object types, such as an integer object using an Integer class object, and so on.

Polymorphism is implicit in Java, although it can be limited by prohibiting explicit method overriding. Java provides automatic Polymorphism and can restrict it by prohibiting explicit method overriding. Both C++ and Java have access specifiers that restrict the scope of attributes and methods within the class using private, within the package using protected and outside the class and package using public.

2.5 A simple mobile app to illustrate the use of OOP in Java

In this Final Report, I am very interested in the uses of OOPs, hence I decided to build a simple app to have better understanding in OOPs with Java.

The source code: simple-app source

Overview: The app has 2 screens (the screenshot below):

- The first screen: there are a TextView and a button to navigate to the second screen.
- The second screen: there are a EditText for recieving inputs from users and a button, the information from the EditText will be sent to the TextView in the first screen when user click the button.

```
1   package com.sang.connect2activities;
2
3   import ...
11  // This class implements the first screen
12  public class Activity1 extends AppCompatActivity {
13      TextView textView;
14
15      @Override
16      protected void onCreate(@Nullable Bundle savedInstanceState) {
17          super.onCreate(savedInstanceState);
18          setContentView(R.layout.layout_activity1);
19
20          textView = findViewById(R.id.textView1);
21          Button button = findViewById(R.id.button1);
22          String s = textView.getText().toString();
```

```java
package com.sang.connect2activities;

import ...

// This class implements the second screen
public class Activity2 extends AppCompatActivity {
    public static int return21 = 21; // Request code

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout_activity2);

        Intent intent = getIntent();
        String dataFromActivity1 = intent.getStringExtra( name: "dataFromActivity1");

        EditText editText = findViewById(R.id.edittext1);
        editText.setText(dataFromActivity1);
```

The main parts of this app we consider are two files: **Activity1,java** and **Activity2.java**. In the both file, we can see that the **Inheritance** in class **Activity1**, it extends the class named **AppCompatActivity**.

If we explore more about the **AppComparActivity** we can see the class just extends **FragmentActivity** but implements multiple interface **AppCompatCallback, TaskStackBuilder.SupportParentable** and so on. This things recall the topic we previously discussed that is Java does not support multiple inheritance but we can inherit multiple interface.

```java
public class AppCompatActivity extends FragmentActivity implements AppCompatCallback,
        TaskStackBuilder.SupportParentable, ActionBarDrawerToggle.DelegateProvider {

    private AppCompatDelegate mDelegate;
    private Resources mResources;

    Default constructor for AppCompatActivity. All Activities must have a default constructor for API 27 and
    lower devices or when using the default android.app.AppComponentFactory.
    public AppCompatActivity() { super(); }
```

The content of **Polymorphism** is shown when we create an instance of **TextView** and **Button** which is **textView** and **button**, respectively.

```java
public class Activity1 extends AppCompatActivity {
    TextView textView;
    Button button;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout_activity1);

        textView = findViewById(R.id.textView1);
        button = findViewById(R.id.button1);
        String s = textView.getText().toString();
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent();
                intent.setClass( packageContext: Activity1.this, Activity2.class);
                String s = textView.getText().toString();
                intent.putExtra( name: "dataFromActivity1", s);
```

Furthermore, in the code we see **super**, the super keyword in Java is a reference variable that is used to refer parent class objects. The **super()** in Java is a reference variable that is used to refer the class **AppCompatActivity** constructor.

The bellow image show the app when it runs on a mobile phone.

3 Question 3: Kotlin language approach

3.1 Topic

Together with OOP, our Advanced Programming course also covers the topic of functional programming, in particular the Haskell programming language. You have learned how to define

functions and have them invoked to implement a computer program in Haskell. Let compare the two programming paradigms and present their pros and cons and draw your own conclusions.

3.2 What is Functional programming?

Functions are the building blocks of code organization and can be found in all higher-order programming languages. In general, functional programming entails maximizing the use of functions to create clean, maintainable software. Functional programming is a set of coding methodologies that is often referred to as a programming paradigm.

Functional programming is sometimes defined in opposition to object-oriented programming (OOP) and procedural programming. That is misleading as these approaches are not mutually exclusive and most systems tend to use all three.

Functional programming has obvious advantages in some situations, is widely utilized in many languages and frameworks, and is a key feature of current software developments. It's a helpful and powerful tool that every developer should have in their conceptual and syntactic toolset.

For example, if you call function getSum() it calculates the sum of two inputs and returns the sum. Given the same inputs for x and y, we will always get the same output for sum.

3.3 The concepts of Funtional programming (FP) and OOP

While OOP has the concept of classes and methods, and complex ways of dealing with these structures, functional programming tries to simplify all of this into one basic concept: **functions**.

Functional programming is a good paradigm to employ when behavior is involved, just like OOP is a fantastic technique for modeling scenarios when several entities (real or abstract) are involved.

Imagine having to work on a data-flow solution that needs to capture data, clean it up, translate it (i.e., change its format), and then save it. These are all transformations that you can implement and represent as functions without having to worry about entities, their relationships, states, or behaviors.

```
function takePhoto(photo) {
  //...take photo
  return photo;
}
function editPhot(orginalPhoto) {
  //...edit the photo
  return editedPhoto;
}
function savePhoto(editedPhto) {
   //...save the photo
  return newPhoto;
}
function sharePhoto(photo) {
   //...share the photo
   return postedPhoto;
}
```

These are all simple functions that take an input, perform some actions around it, and then return something else. There are no classes, objects, inheritance, or any other extra abstraction

needed here. Just functions. Now notice how simple their signatures are and that, given we're talking FP here, we can do things like composing them:

```
sharePhoto(savePhoto(editPhoto(takePhoto(photo))))
```

3.3.1 Pure function, Imutability in FP

The ideal in functional programming is what is known as pure functions. A pure function is one whose output is solely determined by the input parameters and whose execution has no side effects, i.e. has no external impact other than the return value.

```haskell
.vscode > main.hs
import Data.List
import System.IO
-- Function to add two integers
addMe :: Int -> Int -> Int
addMe x y = x + y
--Function to add two pair of integers
addTuples :: (Int, Int) -> (Int, Int) -> (Int, Int)
addTuples (x, y) (x1, y1) = (x + x1, y + y1)
-- Function to calculate the factorial of an integer
factorial :: Int -> Int
factorial 0 = 1
factorial  n = n * factorial(n-1)
-- Function to check two string are the same or not
areStringEq :: String -> String -> Bool
areStringEq [] [] = True
areStringEq (x:xs) (y:ys) = x == y && areStringEq xs ys
areStringEq _ _ = False
-- Function to determine input is odd or even
isOdd :: Int -> Bool
isOdd n
    | mod n 2 == 0 = False
    | otherwise = True
```

Haskell program to illustrate the concept of Pure Function, Immutability

The above functions takes elements returns a new result correspondingly and do not modify anything from inputs. There is no need to re-create a separate data structure. Immutability protects us from all the nasty **side effects** of having two data structures operate on the same data. Other data structures work in the same way. For instance, the **areStringEq** recieves 2 strings and return a Boolean value.

Let's consider an example of Pure function and Immutability in Kotlin language.

```kotlin
fun getSum(a:Int,b:Int):Int{
    return a+b
}
fun isValidAmount(amount: Double, max: Double): Boolean = amount<max
fun main() {
    // val type is not mutable in Kotlin
    val number = 10;
    val imutableList = listOf(1, 2, 3,4,5,6,7,8,9);
    val getSum(10,20)
    val isValidAmount(20,10)
    println(imutableList)
}
```

Kotlin program to illustrate the concept of Pure Function, Immutability

3.3.2 Higher-order functions in FP

They are functions that takes another function as an argument.

```haskell
-- Function receives a list and output a list with each element multipled by 4
times4 :: Int -> Int
times4 x = x * 4
listTimes4 = map times4 [1,3,4]
mulBy4 :: [Int] -> [Int]
mulBy4 [] = []
-- Recursion
mulBy4 (x:xs) = times4 x : mulBy4 xs
-- Passing function into a function
doMul :: (Int -> Int) -> Int
doMul func = func 5
num5Times4 = doMul times4
-- Return a function f
addFunc :: Int ->(Int -> Int)
addFunc x y = x + y
add4 = addFunc 4
result = add4 8
fourPlusList = map add4 [1,2,3,4,5,6,7,8,9]
```

Haskell program to illustrate the concept of Higher-order Function

We can see 4 demo functions above takes another functions as an argument to do the tasks.

Another example in Kotlin about higher-order functions

```kotlin
.vscode > kotlin > demo > src > main > java > com > example > PureFunction.kt > ...
fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {  // 1
    return operation(x, y)                                          // 2
}

fun sum(x: Int, y: Int) = x + y                                     // 3

fun main() {
    val sumResult = calculate(4, 5, ::sum)                          // 4
    val mulResult = calculate(4, 5) { a, b -> a * b }               // 5
    println("sumResult $sumResult, mulResult $mulResult")
}
```

Kotlin program to illustrate the concept of Higher-order Function

The contents of OOPs is described at 2.3

3.3.3 Comparison

OOP Pros:
· Objects and methods are easy to read and comprehend.
· OOP utilizes an imperative style, in which code reads like a straight-forward set of instructions as a computer would read it.

OOP Cons:
· Shareable state ften used in OOP. As a result of so many objects and methods existing in the same state and being accessed in a totally unknown order, "race situations" can occur.

FP Pros:
· When you use pure functions, you get dependable functions with no side effects that do and return exactly what you expect.
· FP employs a declarative style that emphasizes what should be done rather than how it should be done. This focuses on performance and optimization while yet allowing you to refactor without having to entirely rewrite your code.

FP Cons:
· Functional programming, like one of OOP's strengths, may be difficult to read at times. When compared to the object-oriented style, functions can become exceedingly lengthy and difficult to follow. The Person object and createPerson function we discussed before provided a simple example of this.

	Functional programmings	**OOPs**
Definition	FP emphasizes on evaluation of functions	OOPs based on concept of objects
Data	FP use immutable data	OOPs use mutable data
Parallel programming	Support	Not support
Model	Declarative programming model	Imperative programming model
Execution of statements	Any order	Particular order
Iteration	Using recursion for iterative data	Using loops
Basic element	Variables and Functions	Objects and Methods
The uses	Using for few things with more operations	Many things with few operations

3.4 Conclusion

When we have a fixed set of actions on objects and our code changes mostly by adding new things, object-oriented languages are ideal. This can be done by creating new classes that implement existing methods while leaving the existing classes alone.

When we have a fixed collection of objects and our code advances mostly by adding new operations on existing things, functional languages are ideal. This is done by adding new functions

that compute with existing data types while leaving the existing functions alone.

To put it another way, when we're working across multiple borders, OOP is a great way to package everything up and keep it safe from unauthorized access. Functional programming, on the other hand, works effectively when complexity is kept to a minimum.

Functional programming thrives in front end spaces because back ends are often giving objects for front ends to process. The client doesn't care about maintaining object states. It's already given to them, probably in the form of a JSON object. You don't really need to play inception by putting an object into an object.

Object-oriented thinking works well in the back end because most of the time, you're required to construct something to give to the next boundary. It needs to be packaged up, wrapped in ribbon before posting it away into the unknown.
Both functional and object-oriented programming employ different approaches to data storage and manipulation. Data cannot be kept in objects in functional programming; instead, it can only be altered by writing functions. Data is saved in objects in object-oriented programming. Object-oriented programming is extensively used and successful among programmers.

Maintaining objects while raising inheritance levels is extremely difficult in object-oriented programming. It also violates the encapsulation principle and isn't fully modular. To execute functions in functional programming, a new object is constantly required, and the programs use a lot of memory to run.

To sum up, programmers or developers must always choose the programming language concept that makes their development productive and simple.

4 References

Books

1. Object Oriented Programming In C++ 4th Edition, RobertLafore.

2. Interactive Object-Oriented Programming in Java 2th Edition, Vaskaran Sarcar .

3. Object oriented programming oop with Java, R.G. (Dick) Baldwin R.L. Martinez, PhD.

Other resources

1. GeeksforGeeks 03 Jul 2020, Polymorphism in C++, accessed 09 August 2021, https://www.geeksforgeeks.org/polymorphism-in-c/.

2. GeeksforGeeks 28 Apr 2020, Object Oriented Programming in C++, accessed 10 August 2021, https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/.

3. Software Testing Help 05 August 2020, C++ Vs Java: Top 30 Differences Between C++ And Java With Examples, accessed 10 August 2021, https://www.softwaretestinghelp.com/cpp-vs-java/.

4. Guru99, What is Abstraction in OOPs? Java Abstract Class & Method, accessed 10 August 2021, https://www.guru99.com/java-data-abstraction.html.

5. Matthew Tyson from InfoWorld 01 Apr 2021, What is functional programming? A practical guide, accessed 12 August 2021, https://www.infoworld.com/article/3613715/what-is-functional-programming-a-practical-guide.html.

6. Sharjeel Haider from Medium 16 Oct 2019, Functional Programming in Kotlin (Part 1)—Higher-Order Functions, accessed 12 August 2021, https://medium.com/mindorks/functional-programming-in-kotlin-part-1-higher-order-functions-c60e2e4634b3.

7. Shaistha pathima from Medium 10 Jul 2019, Functional Programming VS Object Oriented Programming (OOP) Which is better. . . .?, accessed 13 August 2021, https://medium.com/@shaistha24/functional-programming-vs-object-oriented-programming-oop-which-is-better-82172e53a526.

8. Darrick Mckirnan from Medium 12 May 2017, Object Oriented Programming (OOP) & Functional Programming —What are they & the Pros and Cons, accessed 13 August 2021, https://medium.com/@darrickmckirnan/object-oriented-programming-oop-functional-programming-what-are-they-the-pros-and-cons-11f98a971e38.