VNUHCM – UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Course: Object-Oriented Programming Method

A Comparison Between C++ And Java

Project Instructor: Master. Nguyen Minh Huy
Group Members:

Student ID	Full name
21120010	Nguyen Xuan Hieu (Group Leader)
21120056	Nguyen Dang Tuong Duy
21120136	Nguyen Tien Thanh
21120462	Do Khai Hung

Contents

Preface

C++ and Java are two of the most popular object-oriented programming languages today. These two languages are known for their powerful, versatile and especially optimized object-oriented programming.
C++ and Java have many similarities and differences in terms of object-oriented programming. That is the reason why comparing these two languages on the basis of object-oriented programming is a topic of interest.
We will go through all the aspects which are related to Object-oriented Programming for better understand of these two languages.

1 Overview of C++ and Java

1.1 The origin of C++

1.1.1 History

- C++ is the object-oriented programming language that was officially released in 1983.
- In the early days, the idea about creating a programming language that provides Simula (one of the earliest object-oriented programming languages to come) facilities for program organization together with C's efficiency and flexibility for system programming, was conceived by Bjarne Stroustrup for long.
- Beside Simula and C, C++ was also inspired by some other languages such as, ALGOL 68, Ada, CLU, ML, etc.

1.1.2 About Object-oriented Programming

- Just like it's previous name, C with class, C++ was born to become a powerful object-oriented programming language.
- C++ supports almost all properties that define an object-oriented program, including class, derived class, multiple types of access control, polymorphism with virtual functions, etc.

The section **1.1** refers to: [Str93]

1.2 The origin of Java

1.2.1 History

- Java is an object-oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s.
- The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities.
- Java is a programming language that made to be object-oriented and platform independent.

1.2.2 About Object-oriented Programming

- The main goal in the creation of Java is that it should use the object-oriented programming methodology.

- Everything in Java is a part of some classes. And objects are derived from another object class.
- Just like C++, Java provides classes, class inheritances, polymorphism, **inner classes**, Interface - a completely abstract class, etc.
  All these facilities make Java a powerful and flexible object-oriented programming language.

The section **1.2** refers to: [Tut]

2 Basic data types and built-in classes

2.1 Numeric Data Type

C++	Java	Meaning
`char`	`byte`	1-byte integer, from -128 to 127.
`short`	`short`	2-byte integer, from -32768 to 32767.
`int`	Invalid	Depend on system, 16-bit or 32-bit.
`char`	Invalid	ASCII characters, 1-byte in length.
`wchar_t`	`char`	Unicode characters, from U+0000 to U+FFFF.
`int64_t`	`long`	64-bit integer, from -2^{63} to $2^{63}-1$.
`bool`	`boolean`	Logical, `true (1)` or `false (0)`.
`float`	`float`	Floating point number, 32-bit.
`double`	`double`	Floating point number, 64-bit.

- The `char` data type in Java is built to be the core part of this language itself, whereas `wchar_t` in C++ isn't.
- Java has built-in classes that support base data types. These classes are wrapper classes, such as `Integer`, `Byte`, `Double`, `Float`, ...
  This allows easy manipulation of objects of base data types.
  For example:

```
    Integer x = 5;
    System.out.println(x.toString()); // "5"
```

2.2 String Data Type

- C++ has two string data types: C-string (string in C) and `std::string` or `std::wstring` in STL library of C++. However, C++ does not have a built-in string data type with full operators.
- In constrast, Java has a String class with full operators ($+$, $==$, $!=$, $>$,...) built as a base class that can even store Unicode characters.

```
    //C++ using character array (doesn't have operators)
    char a[] = "abc";

    //Java
    String a = "abc";
```

2.3 Dynamic Array

- Dynamic array is not built-in in C++. To use it, developers must use pointer and allocate the array themselves directly.
  Or instead, one can use `std::vector` in the STL library.
  But we can all see that dynamic arrays using pointers will be long and complex if we allocate/deallocate arrays of multiple dimensions.
  For example:

```
    // Allocate memory for a 3-D dynamic array of 3x4x5 size
    int m = 3, n = 4, k = 5;

    int*** arr = new int** [m];
    for (int i = 0; i < m; i++) {
        arr[i] = new int* [n];
        for (int j = 0; j < n; j++)
            arr[i][j] = new int[k];
    }

    // Deallocate the dynamic array
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++)
            delete[] arr[i][j];
        delete[] arr[i];
    }
    delete[] arr;
```

- Compared to C++, Java has a built-in dynamic array that works just like `std::vector` in C++ but has almost similar syntax like a normal array. We also don't need to worry about memory deallocation as Java will do it for us.
  For example:

```
    int m = 3, n = 4, k = 5;

    int [][][] arr = new int[m][n][k];
    // Do something...
```

The section **2** refers to: [TTK20a]

3 Object's Attributes and Methods

3.1 Constructors

In general, constructors of Java and C++ are quite similar. However, there are some differences:

- Constructor that calls another constructor:
  Java supports this feature and gives it the name "Constructor chaining".

```
class Fraction{
    private int m_den;
    private int m_num;

    public Fraction(): this(0,1) {}
    public Fraction(int den, int num) {
        m_den = den;
        m_num = num;
    }
}
```

Meanwhile, C++ also supports this feature and calls it ”Delegating constructor”.

```
class Fraction {
private:
    int m_num;
    int m_den;
public:
    Fraction() {
        m_num = 0;
        m_den = 1;
    }

    Fraction(int num, int den) : Fraction() {
        m_num = num;
        m_den = den;
    }
};
```

- Methods called in constructor:
  In C++, methods called in constructor of base class are executed from base class.

```
class A {
public:
    A() {
        this->print();
    }

    void print(){
        cout << "Initializing in A";
    }
};

class B: public A {
    public:
    B() {}
    void print(){
        cout << "Initializing in B";
    }
};
void main() {
    B object ; // Initializing in A
}
```

But in Java, they are executed from the nearest derived class.

```java
public class A {
    public A() {
        this.print();
    }
    public void print() {
        System.out.println("Initializing in A");
    }
}

public class B extends A {
    public B() {}
    public void print() {
        System.out.println("Initializing in B");
    }
}

public class Main{
    public static void main(String[] args) {
       B object = new B(); // Initializing in B
    }
}
```

- In Java, if there are any errors occur in constructors, the destructor will be automatically called, but in C++ this doesn't happen.
- Java doesn't support default copy constructor.

3.2 Destructors

- Developers are not allowed to modify destructor in Java because it has already had garbage collector which automatically releases unused memory.
- Destructor in C++ is allowed to be implemented by developers but it comes with some disadvantages:
  - Forget to free up occupied memory.

```cpp
void main() {
    int *p = new int;
    *p = 3;
    // Memory leaks
}
```

  - Release resources again.

```cpp
void main() {
    int *p = new int;
    *p = 3;
    delete p;

    // Do something...

    delete p; // Error
}
```

– Release memory still in use.

```cpp
void main() {
    int *p = new int;
    *p = 3;
    delete p;

    *p = 4; // Error
}
```

- But garbage collectors in Java does have drawback too, they can bring some runtime overhead that is out of the programmer's control. This could lead to performance problems for large applications that scale large numbers of threads or processors, or sockets that consume a large amount of memory. Therefore, most game developers should opt for C++ or another language for the smoothest possible real-time animation. [Edu22]

3.3 Operator overloading

- Operator overloading is not supported in Java.
- Objects in Java are passed to operators by reference, not by value.

3.4 Parameters

- Unlike C++, Java doesn't support `&` keyword for passing reference parameters.
- Parameters in Java are classified into two types: passing by value with primitive data type (`float`, `int`,. . .) and passing by reference with object of class.

The section **3** refers to: [TTK20b]

4 Encapsulation

Data encapsulation is a must-have property in every object-oriented program.
This property is the same for C++ and Java, as every object-oriented program should be encapsulated.
It is said that attributes of an object should not be exposed publicly, instead all access from outside the object should be via accessor methods (getters and setters). This makes a class become a black box and we call it the **Black box** rule. [Huy22] [Fow17]
For example:

- In C++:

```cpp
class Fraction {
private:
    int m_num;
    int m_den;
public:
    void setFrac(int num, int den) { // setter method
        m_num = num;
        m_den = den;
    }

    int getNum() { // getter methods
        return m_num;
    }

    int getDen() {
        return m_den;
    }
};
```

- In Java:

```java
public class Fraction {
    private int m_num;
    private int m_den;

    public void setFrac(int num, int den) {
        this.m_num = num;
        this.m_den = den;
    }

    public int getNum() {
        return this.m_num;
    }

    public double getDen() {
        return this.m_Den;
    }
}
```

But ones said that getters and setters also violate the **Black box** rule and that is when a principle called **Tell, Don't Ask** appears.
The name speaks for itself, **Tell, Don't Ask** reminds us that rather than asking an object for data and acting on that data, we should instead tell an object what to do. This encourages to move behavior into an object to go with the data. [Huy22] [Fow13]
For example:

- In C++:

```cpp
class Fraction {
private:
    int m_num;
    int m_den;
public:
    // To calculate the fraction value,
    // instead of asking the object for m_num and m_den,
    // we tell the object to calculate the value for us.

    double getValue() {
        return (1.0 * m_num) / (1.0 * m_den);
    }
};
```

- In Java:

```java
public class Fraction {
    private int m_num;
    private int m_den;

    public double getValue() {
        return Double.valueOf(m_num) / Double.valueOf(m_den);
    }
}
```

5 Inheritance

In both C++ and Java, the purpose of inheritance is the same. In both languages, inheritance is used to reuse code and/or create a 'IS-A' relationship. [Jav21a] [Jav21b]

- **Parent's members access:**
  Grandparent class members are not directly accessible in Java. We use `super` keyword to access parent class.
  In C++, you can access to parent class by using "`<parent class name>::`".

- **Inheritance specifiers:** For inheritance, Java uses the `extends` keywords. Unlike C++, Java lacks inheritance specifiers such as `public`, `protected` or `private`.

- **Protected keyword:** In Java, the protected member access specifier has a slightly different meaning as in C++.
  Even if class B does not inherit from class A, protected members of a class "A" are accessible in other classes "B" in the same package in Java (they both have to be in the same package).

- **Example:**
  In C++:

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

```

```cpp
class GrandParent {
public:
    void print() {
        cout << "GrandParent\n";
    }
};

class Parent : public GrandParent {
public:
    void print() {
        cout << "Parent\n";
    }
};

class Child : public Parent {
public:
    void print() {
        GrandParent::print();
        cout << "Child\n";
    }
};

int main() {
    Child child;
    child.print();
    // Output:
    //  GrandParent
    //  Child

    return 0;
}
```

In Java:

```java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        // Trying to access Grandparent's Print()
        super.super.Print(); // ERROR
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}

// In Java, we can access grandparent's members
// only through the parent class.
// So the above code can be rewritten like this:

class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();

        // Output:
        //  Grandparent's Print()
        //  Parent's Print()
        //  Child's Print()
    }
}
```

6 Abstraction

6.1 Virtual functions

Although Java doesn't have `virtual` keyword, by default, all the instance methods in Java are considered as the virtual function except `final`, `static`, and `private` methods so that these methods can be used to achieve polymorphism. [Jav21c]
Two programs below prints the same output:

- In Java:

```java
public class A {
    public void print() {
        System.out.println("Print function in A");
    }
}
public class B extends A {
    public void print() {
        System.out.println("Print function in B");
    }
}
public class Main{
    public static void main(String[] args) {
        A object = new B();
        object.print(); // Print function in B
    }
}
```

- In C++:

```cpp
class A {
public:
    virtual void print() {
        cout << "Print function in A";
    }
};

class B :public A {
public:
    void print() {
        cout << "Print function in B";
    }
};

void main() {
    A* object = new B;
    object->print(); // Print function in B
}
```

6.2 Abstract classes

- In C++, abstract classes are defined by containing one or more pure virtual methods:

```cpp
class A {
public:
    virtual void print() = 0; // Pure virtual method
    // So A is an abstract class
};
```

- In Java, The `abstract` keyword is a non-access modifier, used for classes and methods. To define an abstract class, you must have `abstract` keyword before class and abstract methods can only be used in an abstract class: [W3S22a]

```java
abstract class A {
    abstract void print();
}
```

- Of course in C++ and Java, abstract class can not be used to create objects, and they may contain a mix of methods declared with or without an implementation.

6.3 Interface

In Java, there is also a special thing that is a completely abstract class called an `interface`. Interface contains only abstract methods (no body). [W3S22b]

```java
interface Animal {
    public void makeSound(); // interface method  (does not have a body)
    public void eat();
}
class Pig implements Animal {
    public void makeSound() {
        // The body of makeSound() is provided here
        System.out.println("ut ut");
    }
    public void eat() {
        System.out.println("eatting");
    }
}
public class Main{
    public static void main(String[] args){
        Animal animal= new Animal();
        // Error, can not create object from an interface.

        Pig myPig = new Pig();  // Valid, create a Pig object.
        myPig.makeSound(); // "ut ut"
        myPig.eat(); // "eatting"
    }
}
```

7 Polymorphism

- Static polymorphism (Compile time polymorphism): Function overloading is supported in both Java and C++. But Java does not support operator overloading.
- Dynamic polymorphism (Runtime polymorphism):
  - In Java, within an inheritance hierarchy, a subclass can override a method of its superclass. If you instantiate the subclass, the Java Virtual Machine (JVM) will always call the overridden method, even if you cast the subclass to its superclass: [Wu09]

```java
public class A {
    void whereami() {
        System.out.println("You're in A");
    }
}
public class B extends A {
    void whereami() {
        System.out.println("You're in B");
    }
}
public static void main(String[] args) {
    A a = new A();
    a.whereami(); // A
    B b = new B();
    b.whereami(); // B
    A c = new B(); // upcasting.
    c.whereami(); // B
}
```

  - In C++, to achieve dynamic polymorphism, we have to use `virtual` keyword for methods in base class.

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

class A {
public:
    virtual void whereami() {
        cout << "You're in A" << endl;
    }
};
class B : public A {
public:
    void whereami() {
        cout << "You're in B" << endl;
    }
};

int main(int argc, char** argv) {
    A a;
    B b;
    a.whereami(); // A
    b.whereami(); // B
    A* c = new B();
    c->whereami(); // B
    return 0;
}
```

8 Improvement of Java over C++ in Object-oriented Programming

There are four reasons to learn Java as your first true Object-oriented programming language, rather than C++:

1. Java is a pure-object oriented language, which is one in which EVERYTHING is an object.
2. Java was designed to make common C++ programming errors harder or impossible.
3. Java was designed as a simplified version of C++, and it really does remove a number of C++'s complications.
4. C++ is still evolving rapidly, largely because it became necessary to introduce semi-automated memory management in the form of smart pointers (but not that smart pointer then don't use it). Did that last phrase confuse you? Yeah, that's the point.

References

[Str93] B. Stroustrup. "A History of C++: 1979-1991". In: *HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages*. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, p. 1. ISBN: 0897915704. URL: `https://www.stroustrup.com/hopl2.pdf`.

[Wu09] C. Thomas Wu. "An introduction to object-oriented programming with Java". In: 5th ed. McGraw-Hill, 2009. Chap. 13.

[Fow13] Martin Fowler. *TellDontAsk*. 2013. URL: `https://martinfowler.com/bliki/TellDontAsk.html`.

[Fow17] Martin Fowler. *SelfEncapsulation*. 2017. URL: `https://martinfowler.com/bliki/SelfEncapsulation.html`.

[TTK20a] Tran Dan Thu, Dinh Ba Tien, and Nguyen Tan Tran Minh Khang. "Object-oriented Programming". In: Science and Technics Publishing House, 2020. Chap. 2, pp. 57–73.

[TTK20b] Tran Dan Thu, Dinh Ba Tien, and Nguyen Tan Tran Minh Khang. "Object-oriented Programming". In: Science and Technics Publishing House, 2020. Chap. 3, pp. 85–108.

[Jav21a] JavaTpoint. *Inheritance in C++ vs Java*. 2021. URL: `https://www.javatpoint.com/inheritance-in-cpp-vs-java`.

[Jav21b] JavaTpoint. *Inheritance in Java*. 2021. URL: `https://www.javatpoint.com/inheritance-in-java`.

[Jav21c] JavaTpoint. *Virtual Function in Java*. 2021. URL: `https://www.javatpoint.com/virtual-function-in-java`.

[Edu22] Educative. *What are the pros and cons of Garbage Collector in Java?* 2022. URL: `https://www.educative.io/answers/what-are-the-pros-and-cons-of-garbage-collector-in-java`.

[Huy22] Master. Nguyen Minh Huy. *OOP-05-The Big Three and Encapsulation*. 2022, pp. 16–20.

[W3S22a] W3Schools. *Java Abstraction*. 2022. URL: `https://www.w3schools.com/java/java_abstract.asp`.

[W3S22b] W3Schools. *Java Interface*. 2022. URL: `https://www.w3schools.com/java/java_interface.asp`.

[Tut] Free Java Guide & Tutorials. *History of Java programming language*. URL: `http://www.freejavaguide.com/history.html`.