

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC BÁCH KHOA  
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH



# BÁO CÁO ĐỀ TÀI MỞ RỘNG KSTN LẬP TRÌNH NÂNG CAO - HK212

---

SO SÁNH OOP JAVA & RUBY

SO SÁNH FP HASKELL & SML

SỬ DỤNG OOP C++ ĐỂ XÂY DỰNG GAME CARO

---

Giảng viên hướng dẫn: *TS. Trương Tuấn Anh*  
Sinh viên thực hiện: *Nguyễn Đức An - 2010102*  
Email: *an.nguyenduc1406@hcmut.edu.vn*

## Mục lục

<b>1</b>	<b>Mục đích nghiên cứu</b>	<b>2</b>
<b>2</b>	<b>Mục tiêu đề tài</b>	<b>2</b>
<b>3</b>	<b>So sánh Object Oriented Programming giữa Java và Ruby</b>	<b>2</b>
3.1	Giới thiệu một số tính chất cơ bản của Java và Ruby	2
3.2	Một số tính chất cơ bản	3
3.3	Biên dịch và thông dịch (Compiled and Interpreted)	4
3.4	Kiểu biến dữ liệu (Typed Variables)	4
3.5	Lớp (Class)	5
3.6	Đối tượng (Object)	6
3.7	Access Modifiers and Non-access Modifiers	7
3.8	Thuộc tính (attribute)	10
3.9	Phương thức (method)	11
3.10	Constructor	12
3.11	Garbage Collector	13
3.12	Encapsulation (tính đóng gói)	14
3.13	Inheritance (tính thừa kế)	15
3.14	Abstraction (tính trừu tượng)	16
3.15	Polymorphism (tính đa hình)	19
3.16	Interface và Implements	20
3.17	Xử lý ngoại lệ (Exceptions Handling)	22
<b>4</b>	<b>So sánh Functional Programming giữa Haskell và SML</b>	<b>23</b>
4.1	Giới thiệu một số tính chất cơ bản của Haskell và SML	23
4.2	Biên dịch chương trình	23
4.3	Một số tính chất cơ bản	24
4.4	Một số điểm giống nhau	25
4.5	Purely Functional	26
4.6	Lazy Evaluation and Strict	26
4.7	Generating List	27
4.8	List Comprehensive	28
4.9	Passing Argument	28
4.10	Type Classes and Modules	29
<b>5</b>	<b>Game Caro - Object Oriented Programming C++</b>	<b>30</b>
5.1	Giới thiệu về game Caro	30
5.2	Hướng dẫn chơi và luật chơi	30
5.3	Cấu trúc chương trình	31
5.4	Bản thiết kế của chương trình	32
5.4.1	Graphic Function	32
5.4.2	Interface Cell	32
5.4.3	Interface Board	33
5.4.4	Interface Game	34
5.4.5	Hàm main	35
5.5	Giao diện người dùng	36
5.5.1	Giao diện Menu	36
5.5.2	Giao diện bàn cờ	37
5.5.3	Giao diện luật chơi	37
5.5.4	Giao diện thông tin tác giả	38
5.6	Demo và kiểm thử chương trình	38
<b>6</b>	<b>Tài liệu tham khảo</b>	<b>39</b>

## 1 Mục đích nghiên cứu

**Object Oriented Programming** và **Functional Programming** là hai phong cách lập trình có vai trò quan trọng trong việc xây dựng, phát triển những chương trình, phần mềm, ứng dụng trong nhiều lĩnh vực khác nhau. Tuy nhiên, không thể nói là OOP hay FP, phong cách nào là tốt hơn vì mỗi phương pháp đều có những ưu điểm và nhược điểm riêng tùy thuộc và mỗi loại chương trình và mục đích sử dụng của người lập trình.

Về mặt ứng dụng, nếu bạn muốn thiết kế website hoặc desktop apps thì OOP là một lựa chọn tốt hơn vì nó đem lại thiết kế tốt, có thể dễ dàng bảo trì, phát triển thêm tính năng về sau này. Tuy nhiên, nếu bạn làm việc với những chương trình đòi hỏi nhiều phép tính toán, thuật toán, coi trọng tốc độ xử lý hơn việc phát triển lâu dài, để tránh tốn nhiều thời gian khi lập trình thì lúc này nên chọn FP.

Trong bài báo cáo này, em sẽ trình bày làm rõ hai phong cách lập trình trên thông qua việc so sánh OOP giữa hai ngôn ngữ đặc trưng là **Java** và **Ruby** và so sánh FP giữa hai ngôn ngữ đặc trưng là **Haskell** và **SML (Standard ML)**. Và phần 3, chúng ta sẽ vận dụng kiến thức OOP trong thực tế để xây dựng một game, ở đề tài này em chọn làm **Game Caro** bằng ngôn ngữ C++, thông qua game này chúng ta sẽ biết được lợi ích của OOP trong việc thiết kế phần mềm.

## 2 Mục tiêu đề tài

Trong đề tài, chúng ta sẽ cùng nghiên cứu về các vấn đề sau:

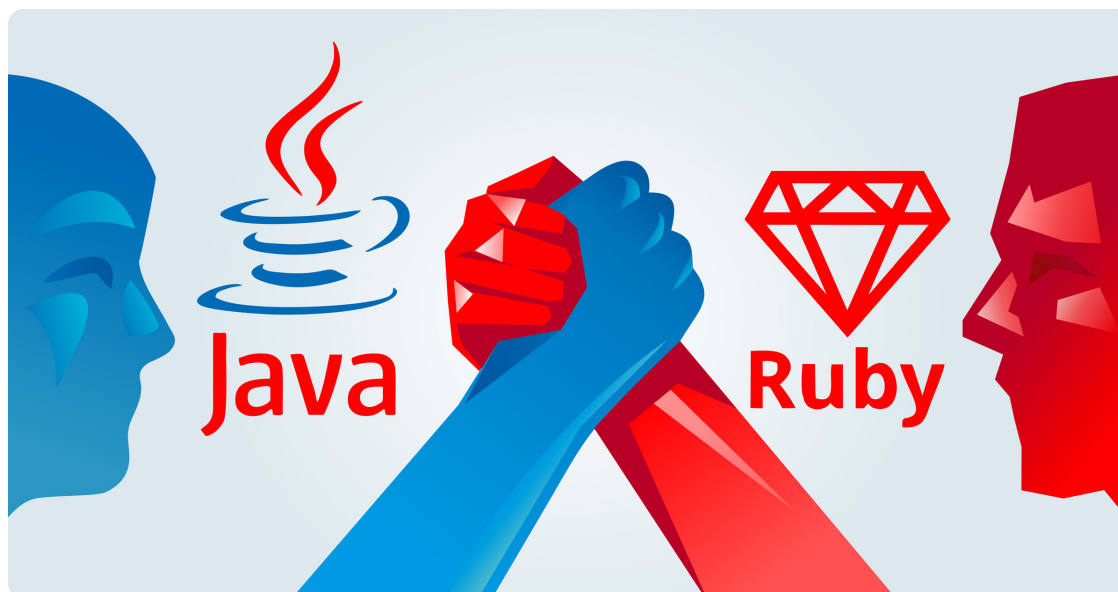
1. So sánh Object Oriented Programming giữa Java và Ruby.
2. So sánh Functional Programming giữa Haskell và SML.
3. Sử dụng phong cách lập trình OOP, xây dựng chương trình Game Caro bằng ngôn ngữ C++.

## 3 So sánh Object Oriented Programming giữa Java và Ruby

### 3.1 Giới thiệu một số tính chất cơ bản của Java và Ruby

**Java** là một ngôn ngữ high-level, open-source, và có tính chất hướng đối tượng (**object-oriented**). Java là ngôn ngữ cần cả hai giai đoạn **compiled** và **interpreted** để chạy chương trình, nghĩa là cả source code được biên dịch (compiled) trước, rồi khi chạy sẽ chạy một lượt phần mã code đã biên dịch của toàn bộ chương trình. Đối với Java, đầu tiên source code được compiled thành **byte code** rồi sau đó phần byte code sẽ được thực thi (interpreted) trên **JVM (Java Virtual Machine)**. Vào năm 1995, Java được ra đời bởi James Gosling tại Sun Microsystems.

**Ruby** là một ngôn ngữ high-level, open-source, và có tính chất hướng đối tượng, đặc biệt là **purely object-oriented**. Khác với Java, Ruby là chỉ cần một giai đoạn **interpreted** để chạy chương trình, nghĩa là khi chạy tới đâu biên dịch tới đó, khi chương trình xảy ra lỗi thì dừng ngay tại đó mà không cần biên dịch đoạn code phía sau. Trong giai đoạn giữa những năm 1990s, Ruby được ra đời bởi Yukihiro “Matz” Matsumoto tại Nhật Bản.



Hình 1: Java và Ruby

### 3.2 Một số tính chất cơ bản

Đặc điểm	Java	Ruby
Tính chất OOP	Hybrid OOP	Pure OOP
Build chương trình	Compiled và Interpreted	Interpreted
Kiểu biến dữ liệu	Static typing	Dynamic typing
Phương thức truy cập	public, protected, private, "package"	public, protected, private
Garbage Collection	mark-compact and mark-sweep	mark-sweep
Tốc độ	Phức tạp hơn, Chậm hơn	Đơn giản hơn, Nhanh hơn
Framework	Spring, JSF, GWT	Ruby on Rails, Sinatra, Hanami
Ứng dụng	Phần mềm nhiều truy cập, nhiều tính toán	Phần mềm ít truy cập, ít tính toán

Bảng so sánh một số tính chất cơ bản giữa Java và Ruby

Bên trên là một số tính chất cơ bản của Java và Ruby. Phần so sánh cụ thể giữa hai ngôn ngữ sẽ được làm rõ ở những phần sau.

### 3.3 Biên dịch và thông dịch (Compiled and Interpreted)

- Điểm khác nhau

- **Java: Compiled and Interpreted**

- Chương trình Java đòi hỏi phải được biên dịch (compile) thông qua trình biên dịch **javac** thành bytecode trước khi khởi chạy. Sau khi biên dịch, ta sẽ được file **filename.class**, file này được dùng để chạy chương trình

- **Ruby: Interpreted**

- Trong khi đó, Ruby là một ngôn ngữ interpreted, do đó được chạy trực tiếp mà không cần biên dịch để tạo bytecode

Ví dụ, ta có hai file **Main.java** và **main.rb**.

```
//Main.java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```
# main.rb
print("Hello World")
```

Để chạy chương trình, ta có thể chạy command line như sau:

- **Java**

```
> javac Main.java
> java Main
```

- **Ruby**

```
> ruby main.rb
```

- **Output**

```
Hello World
```

### 3.4 Kiểu biến dữ liệu (Typed Variables)

- Điểm giống nhau

- Cả Java và Ruby đều có kiểu biến **strongly typed**

- Điểm khác nhau

- **Java**

- Trong Java thì các biến số đều có kiểu cố định (**statically typed**), do đó chúng ta không thể thay đổi kiểu dữ liệu của một biến được định nghĩa trước đó.

#### ■ Ruby

- Tuy nhiên đối với Ruby thì biến có dạng động (**dynamically typed**) nghĩa là kiểu của biến số không được định nghĩa rõ ràng, khi gán lại value thì kiểu dữ liệu sẽ được tự duy diễn (**inferred type**) và thay đổi kiểu dữ liệu theo kiểu của giá trị được gán lần cuối cùng.

#### Ví dụ

##### • Java

```
String str = "Hello World"; // str is String type
String str = 1; // error incompatible types: int cannot be converted to String
```

##### • Ruby

```
str = "Hello World" # str is String type
str = 1             # Now str is int type
```

### 3.5 Lớp (Class)

#### • Điểm giống nhau

- Do cùng là ngôn ngữ hỗ trợ OOP, nên vai trò của class trong Java và Ruby tương tự nhau, đều là **bản thiết kế (blueprint)** để định nghĩa các thông tin cho đối tượng.
- Class cũng dùng để định nghĩa hai thành phần chính đó là **thuộc tính (attribute)** và **phương thức (method)** của đối tượng.

#### • Điểm khác nhau

##### ■ Java

- Trong Java, mọi thứ phải được định nghĩa trong class.
- Trong Java, class phải được định nghĩa cùng với các **access modifiers** như **public**, **private**, **protected** hoặc **non-access modifiers** như **final**, **abstract**,...
- Khi định nghĩa outer class ở tầm vực **public** thì tên của class (**class name**) bắt buộc phải **trùng với tên của file (file name)**.
- Mỗi class có thể có một hàm **main**, hàm **main** phải được định nghĩa **static**, nghĩa là hàm main có thể được truy cập bởi bất cứ đối tượng nào chứ không thuộc về riêng đối tượng đó. Hàm main dùng để chạy chương trình.
- Ngoài ra, một số class đặc biệt còn có thể được nghĩa như lớp trừu tượng được định nghĩa với từ khóa **abstract** (chúng ta sẽ tìm hiểu rõ hơn ở phần sau **3.13.Abstraction**), lớp thừa kế cuối cùng được định nghĩa với từ khóa **final** (chúng ta sẽ tìm hiểu rõ hơn ở phần sau **3.12.Inheritance**),...

##### ■ Ruby

- Trong Ruby, chương trình viết tới đâu chạy tới đó, mã lệnh không cần phải đặt trong class.

- Ruby sử dụng cặp từ khóa **class ... end** để định nghĩa 1 khối lớp hay phương thức.
- Tên của class (**class name**) có chữ cái đầu bắt buộc phải được in hoa (**uppercase**)
- Không cần định nghĩa **access modifiers** cùng với tên class hay method
- Không cần định nghĩa hàm **main** để chạy chương trình

### Ví dụ

- Java

```
public class Main {  
    // body  
    public static void main(String args[]) {  
        // main function  
    }  
}  
  
abstract class Animal {  
    // body  
}  
  
final class Circle {  
    // body  
}
```

- Ruby

```
class Main  
    # body  
end
```

## 3.6 Đối tượng (Object)

- Điểm giống nhau

- Cả hai ngôn ngữ đều có hỗ trợ việc lập trình với đối tượng.

- Điểm khác nhau

- Java

- Trong Java, chỉ có lớp (**class**) mới tồn tại khái niệm đối tượng.
- Một đối tượng được khởi tạo thông qua từ khóa **new** cùng với **Constructor** của lớp tương ứng.

- Ruby

- Trong Ruby, **mọi thứ** đều được xem như là một đối tượng. Các biến, giá trị cụ thể như số nguyên, số thực, chuỗi, cho tới các phương thức, các lớp, tất cả đều là đối tượng (**pure OOP**)
- Để khởi tạo đối tượng, ta có thể dùng phương thức **.new**, nó sẽ tự gọi đến constructor của class tương ứng (cụ thể là phương thức **initialize** của class đó)
- Chúng ta có thể dùng phương thức **.class** đối với **đối tượng** để lấy ra tên lớp của đối tượng và sử dụng phương thức **.ancestors** đối với **lớp** để kiểm tra xem lớp đó được kế thừa từ những lớp nào.

### Ví dụ

- Java

```
public class Main {  
    int x = 5;  
    public static void main(String[] args) {  
        Main myObj1 = new Main(); // Object 1  
        Main myObj2 = new Main(); // Object 2  
        System.out.println(myObj1.x);  
        System.out.println(myObj2.x);  
    }  
}
```

- Ruby

```
1.class  
> Integer  
  
Integer.ancestors  
> [Integer, Numeric, Comparable, Object, Kernel, BasicObject]  
  
Class.class  
> Class  
  
Class.ancestors  
> [Class, Module, Object, Kernel, BasicObject]  
  
# Create class Box  
class Box  
  
# class variable  
@@num_of_toy = 3  
  
end  
  
# Two Objects of Box class  
box1 = Box.new # Object 1  
box2 = Box.new # Object 2
```

## 3.7 Access Modifiers and Non-access Modifiers

- Điểm giống nhau

- Cả hai ngôn ngữ có 3 loại access modifiers phổ biến là public, protected, private.

- Điểm khác nhau

- Java

- Bên cạnh các access-modifiers ở trên, Java còn có thêm 1 loại access modifier nữa là **default**. Có nghĩa là khi không định nghĩa access modifier cho class thì Java sẽ tự định nghĩa kiểu default cho class theo kiểu **package-private**.



- Bên cạnh access-modifiers, Java còn có một số **non-access modifiers** như **final**, **abstract**,...
- Access modifiers và Non-access Modifiers có thể được định nghĩa cùng với class và method khi khởi tạo.
- Khi định nghĩa **outer class**, ta chỉ có thể định nghĩa với access modifier là **public** hoặc non-access modifiers như **abstract**, **final**. Còn các access modifiers như **private**, **protected**, **public** chỉ được định nghĩa cho các **inner class** (hoặc còn gọi là **nested class**) chứ không dùng để định nghĩa **outer class**
- Trong Java, nếu không định nghĩa thì phương thức truy cập mặc định (default access modifier) là **private**.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Hình 2: Access Modifiers in Java

## ■ Ruby

- Ruby không có **non-access modifiers**, nếu cần sử dụng người dùng phải tự định nghĩa.
- Không cần bắt buộc phải định nghĩa access modifiers cùng với class khi khởi tạo.
- Access modifiers trong Ruby dùng để định nghĩa các phương thức (**method**) chứ không dùng để định nghĩa class.
- Trong Java, nếu không định nghĩa thì phương thức truy cập mặc định (default access modifier) là **public**.

## Ví dụ

### • Java

```
class User {  
    // Outer class can define public, abstract, final  
    private class InnerUser {  
        // Inner class can define private, protected, public  
        public void print() {  
            System.out.println("Private inner class method called");  
        }  
    }  
    void print() {  
        System.out.println("Public outer class called ");  
        InnerUser innerUser = new InnerUser();  
        innerUser.print();  
    }  
}  
  
public class Main {  
    public static void main(String args[]) {  
        User user = new User();  
        user.print();  
    }  
}  
  
/* Output: Public outer class called  
           Private inner class method called  
*/
```

### • Ruby

```
class Employee  
    def initialize(exp)  
        @experience = exp  
    end  
  
    private  
        def setSalary(salary)  
            @salary = salary  
        end  
  
    public  
        def setData  
            if @experience == "Fresher"
```

```
        setSalary("1000 Rupees")
      elsif @experience == "Junior"
        setSalary("100000 Dollars")
      elsif @experience == "Senior"
        setSalary("1000000 Dollars")
      end
    end
  end
  def getSalary
    return @salary
  end
end
# Initialize Fresher Object
fresher = Employee.new("Fresher")
fresher.setData()
puts "Fresher Salary is #{fresher.getSalary()}" # Output Fresher Salary is 1000 Rupees
```

### 3.8 Thuộc tính (attribute)

- Điểm giống nhau

- Mỗi class đều được định nghĩa dựa trên thuộc tính và phương thức của đối tượng

- Điểm khác nhau

- Java

- Thuộc tính (**Attribute**) của đối tượng trong Java được định nghĩa trực tiếp, rõ ràng về kiểu dữ liệu, tên biến trong class.
- Có thể định nghĩa **access modifier** hoặc **non-access modifier** cho các attribute của class, mặc định là **private**
- Các tính chất của **attribute** của Java tương tự trong ngôn ngữ C++.

- Ruby

- Trong Ruby, thuộc tính (**Attributes**) được xem như các **instance variables** của đối tượng. Đây là một đặc trưng của Ruby so với các ngôn ngữ khác.
- **Instance Variable** bắt đầu với ký tự symbol @, được định nghĩa trong method **initialize**
- **Instance Variable** có tầm vực **private**, không thể truy cập trực tiếp bên ngoài class, đây là quy ước thiết kế theo tính chất **Encapsulation** của ngôn ngữ.
- Để **cấp quyền** truy cập các instance variable từ bên ngoài, ta phải sử dụng các **attr method** như **attr\_reader (only get)**, **attr\_writer (only set)**, **attr\_accessor (get and set)**.

#### Ví dụ

- Java

```
class Coffee {
  int price = 100; // Attribute
}
```

- Ruby

```
class Coffee
  def initialize
    @price = 100 # Attribute
  end
  attr_accessor :price # attr_accessor method
end

coffee = Coffee.new
puts coffee.price      # price = 100
coffee.price = 200
puts coffee.price      # price = 200
```

### 3.9 Phương thức (method)

- Điểm giống nhau

- Mỗi class đều được định nghĩa dựa trên thuộc tính và phương thức của đối tượng

- Điểm khác nhau

- Java

- Mỗi phương thức đều cần kiểu dữ liệu trả về cụ thể.
- Đối với các kiểu trả về khác void, bắt buộc cần từ khóa **return**

- Ruby

- Phương thức được định nghĩa bằng từ khóa **def**, khi trả về có thể tự suy diễn kiểu.
- Khi trả về, có thể không cần từ khóa **return**

#### Ví dụ

- Java

```
class User {
  String name = "Nguyen Duc An";
  public String getName() {
    return name;
  }
}

public class Main {
  public static void main(String[] args) {
    User user = new User();
    System.out.println(user.getName()); // Output: Nguyen Duc An
  }
}
```

- Ruby

```
class User
  def initialize
    @name = "Nguyen Duc An"
  end
end
```

```
end
def getName
  @name
end
end
user = User.new
puts user.getName() # Output : Nguyen Duc An
```

### 3.10 Constructor

- Điểm giống nhau

- Cả hai ngôn ngữ đều có hỗ trợ constructor dùng để khởi tạo đối tượng như cập nhật giá trị biến khởi tạo, cấp phát vùng nhớ cho đối tượng.
- Constructor được tự động gọi khi đối tượng được khởi tạo.

- Điểm khác nhau

- Java

- Java sẽ sử dụng lại tên của chính lớp đó để khai báo constructor
- Để khởi tạo đối tượng, Java sử dụng từ khóa **new**
- Trong Java, nhiều hơn một phương thức khởi tạo (constructor) có thể được khai báo ví dụ như Default Constructor, Parametered Constructor, Copy Constructor,...

- Ruby

- Ruby sử dụng phương thức **initialize** để khởi tạo đối tượng, vai trò của **initialize** tương tự như constructor
- Để khởi tạo đối tượng, Ruby sử dụng phương thức **.new**
- Trong Ruby, một lớp chỉ có thể có một phương thức khởi tạo. Nếu nhiều hơn một phương thức khởi tạo được khai báo, đối tượng sẽ sử dụng phương thức khởi tạo cuối cùng.

#### Ví dụ

- Java

```
class User {
  int id;
  public User() {
    id = 100;
  }
  public User(int id) {
    this.id = id;
  }
  public int getId() {
    return id;
  }
}
public class Main {
  public static void main(String[] args) {
```

```
User user1 = new User();           // Default Constructor
User user2 = new User(200);       // Parameterized Constructor
System.out.println(user1.getId()); // id = 100
System.out.println(user2.getId()); // id = 200
}
}
```

- Ruby

```
class User
  def initialize
    @id = 100
  end
  def initialize id
    @id = id
  end
  def getId
    return @id
  end
end

# user = User.new      # It will generate error
user = User.new 200    # Now we only have 1 constructor, it requires passed argument id
puts user.getId()      # Output : 200
```

### 3.11 Garbage Collector

**Garbage Collector** được định nghĩa như là một quá trình tự động giải phóng bộ nhớ sau khi sử dụng. Đây là một trong những đặc điểm nổi bật của các ngôn ngữ lập trình cấp cao như Java và Ruby, đảm bảo chương trình sẽ không xảy ra những lỗi "memory leak" hay là "out of memory" do sai sót khi giải phóng thủ công bằng toán tử delete trong C++.

- **Thuận lợi:** Quản lý vùng nhớ tốt, chính xác hơn so với việc giải phóng vùng nhớ thủ công
- **Bất lợi:** Do quá trình tìm kiếm, xác định những vùng nhớ không cần sử dụng nữa sẽ mất thời gian, tốc độ thực thi của Java và Ruby chậm hơn so với C++.

So sánh garbage collector giữa Java và Ruby

- Điểm giống nhau

- Trong Java, Ruby không có khái niệm con trỏ, không có toán tử delete. Phần data cần giải phóng trên Heap là dùng để khởi tạo các object.
- Cả hai ngôn ngữ đều có hỗ trợ Garbage Collector giải phóng bộ nhớ object sau khi sử dụng.

- Điểm khác nhau

- Java

- Trong Java, source code được dịch sang bytecode rồi chạy trên máy ảo Java hay viết tắt là JVM. Trong quá trình chạy chương trình, các đối tượng được khởi tạo ở vùng nhớ heap. Sau cùng, sẽ có một vài đối tượng mà chương trình không cần dùng đến. Các đối tượng này sẽ được garbage collector truy tìm và xóa bỏ để thu hồi lại dung lượng bộ nhớ.

- Có nhiều tiến trình garbage collection khác nhau nhưng phổ biến nhất là **Oracle HotSpot** theo cơ chế **mark-sweep-compact**.
- Đầu tiên ở bước "**mark**", các Object không được tham chiếu sẽ được đánh dấu sẵn sàng để được dọn rác. Tiếp theo là bước "**sweep**", trình thu gom rác sẽ tiến hành xóa các Object đó. Cuối cùng là bước "**compact**", vùng nhớ của các Object còn lại sẽ được nén lại và nằm liền kề nhau trong bộ nhớ Heap. Quá trình này sẽ giúp việc cấp phát bộ nhớ cho Object mới dễ dàng hơn

#### ■ Ruby

- Trong Ruby, mọi thứ đều là object, do đó Garbage Collector quản lý toàn bộ object được tạo ra (mọi thứ). Việc khởi tạo bộ nhớ sẽ mất thời gian, do đó Ruby khởi tạo trước vùng nhớ sẵn sàng cho cả ngàn đối tượng được cấp phát sau này. Cơ chế này được gọi là "**free list**".
- Tiến trình garbage collection ở Ruby chủ yếu theo cơ chế **mark-sweep**
- Ở bước "**mark**", Ruby duyệt qua tất cả các object và đánh dấu các object đang sử dụng bằng cách toggle cờ đánh dấu cho mỗi đối tượng theo cơ chế **bitmap** (đối tượng nào đang được sử dụng thì set bit cho đối tượng đó là 1, flag = true). Tiếp theo ở bước "**sweep**", Ruby sẽ xóa những Object không cần sử dụng nữa, nghĩa là các đối tượng có bit = 0 (flag = false)

### 3.12 Encapsulation (tính đóng gói)

- Các dữ liệu và phương thức có liên quan với nhau được **đóng gói** thành các lớp để tiện cho việc quản lý và sử dụng. Tức là mỗi lớp được xây dựng để thực hiện một nhóm chức năng đặc trưng của riêng lớp đó.
- Ngoài ra, đóng gói còn để che giấu một số thông tin và chi tiết cài đặt nội bộ để bên ngoài không thể nhìn thấy (**data hiding**)

- **Điểm giống nhau**

- Nhìn chung, tính bao đóng (encapsulation) của Java và Ruby là tương tự nhau.

#### Ví dụ

- **Java**

```
class Customer {
    int cust_id;
    String cust_name;
    String cust_addr;
    Customer(int id, String name, String addr) {
        this.cust_id = id;
        this.cust_name = name;
        this.cust_addr = addr;
    }
    public void display_infor() {
        System.out.println("Customer id: " + cust_id);
        System.out.println("Customer name: " + cust_name);
        System.out.println("Customer address: " + cust_addr);
    }
}
```

```
}  
}  
public class Main {  
    public static void main(String[] args) {  
        Customer customer = new Customer(1, "Nguyen Duc An", "Cao Lanh, Dong Thap");  
        customer.display_infor();  
    }  
}
```

- Ruby

```
class Customer  
    def initialize(id, name, addr)  
        @cust_id = id  
        @cust_name = name  
        @cust_addr = addr  
    end  
    # displaying result  
    def display_infor()  
        puts "Customer id: #{@cust_id}"  
        puts "Customer name: #{@cust_name}"  
        puts "Customer address: #{@cust_addr}"  
    end  
end  
# Create Objects  
customer = Customer.new("1", "Nguyen Duc An", "Cao Lanh, Dong Thap")  
# Call Methods  
customer.display_infor()
```

### 3.13 Inheritance (tính thừa kế)

- Điểm giống nhau

- Cả Java và Ruby đều có cung cấp tính thừa kế (**Inheritance**), cho phép một lớp có khả năng kế thừa từ một lớp khác
- Trong cả Ruby và Java, một lớp chỉ có thể kế thừa một lớp khác (**Single Inheritance**)

- Điểm khác nhau

- Java

- Trong Java, một lớp được kế thừa bằng cách sử dụng từ khóa **extends**
    - Java cung cấp đa thừa kế (**multiple inheritance**) với phương thức **interfaces** định nghĩa các phương thức trừu tượng (**abstract**), mà các phương thức đó được thực hiện bởi các lớp kế thừa từ interfaces đó.

- Ruby

- Trong Ruby, một lớp có thể kế thừa bằng cách sử dụng dấu <
    - Ruby cung cấp đa kế thừa với các mô-đun (**module**) và mixins.



- **Modules** có điểm tương đồng với class bởi vì chúng bao gồm những thuộc tính và phương thức. Tuy nhiên, **Modules** khác với class là modules không thể tạo instance và module không thể được phân lớp. Module được định nghĩa với từ khóa module. Khi một module được gọi trong một lớp khác bằng cách sử dụng từ khóa include, các biến, phương thức và các lớp trong module sẽ có sẵn cho lớp bao gồm nó.

#### Ví dụ

- Java

```
class SolarSystem {  
}  
class Earth extends SolarSystem {  
}  
class Mars extends SolarSystem {  
}  
public class Moon extends Earth {  
    public static void main(String args[]) {  
        SolarSystem s = new SolarSystem();  
        Earth e = new Earth();  
        Mars m = new Mars();  
        System.out.println(s instanceof SolarSystem); //true  
        System.out.println(e instanceof Earth); //true  
        System.out.println(m instanceof SolarSystem); //true  
    }  
}
```

- Ruby

```
module MyModule  
    def say_hello  
        puts "Hello, My name is Nguyen Duc An"  
    end  
end  
  
# Import module  
require MyModule  
  
class Hello  
    include MyModule  
end  
  
# Run code  
Hello.new.say_hello # Output: Hello, My name is Nguyen Duc An
```

### 3.14 Abstraction (tính trừu tượng)

**Tính trừu tượng** nghĩa là trừu tượng hóa phương thức của một class, đối tượng không cần biết một hàm được **implement** như thế nào nhưng vẫn có thể gọi phương thức đó ra để sử dụng (**interface**). Interface và Implementation được tách thành hai thành phần riêng biệt, bảo mật được mã nguồn bên phía người lập trình, cũng như dễ dàng cho các người dùng khi sử dụng.

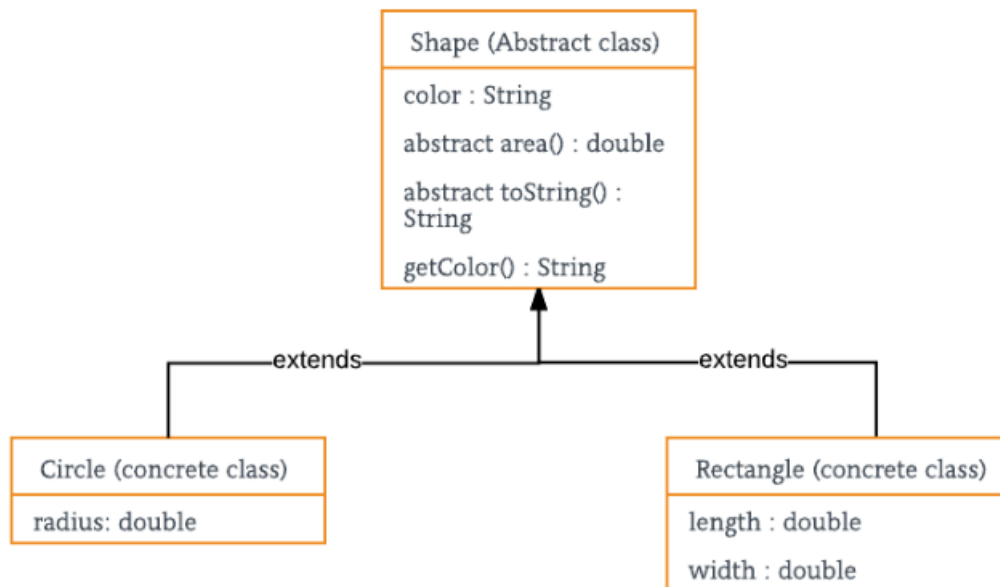
- Điểm giống nhau

- Cả hai ngôn ngữ đều có hỗ trợ tính trừu tượng (**abstraction** và **data hiding**)

- **Điểm khác nhau**

- **Java**

- Lớp trừu tượng (**Abstract Class**) là lớp được định nghĩa với từ khóa là **abstract**
    - Một phương thức trừu tượng (abstract method) là một phương thức được khai báo mà không có implement.
    - Một lớp trừu tượng có thể có hoặc không có abstract method.
    - Nếu phương thức trừu tượng (abstract method) được định nghĩa ở lớp cha thì nó **bắt buộc phải** được implement ở các lớp con thừa kế từ lớp đó.



Hình 3: Abstraction in Java

- **Ruby**

- Ruby không có từ khóa **abstract** như lớp trừu tượng (**abstract class** hay **abstract method**)
    - Để hiện thực tính trừu tượng của Ruby, chúng ta sẽ sử dụng các từ khóa **public** và **private**

**Ví dụ**

- **Java**

```

// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep() {
        System.out.println("Sleep: Zzz");
    }
}

// Subclass (inherit from Animal)
  
```

```
class Dog extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The dog says: woo woo");
    }
}

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

– Output:

```
The dog says: woo woo
Sleep: Zzz
```

- Ruby

```
# Ruby program to demonstrate Data Abstraction

class Geeks

    # defining publicMethod

    public

        def publicMethod
            puts "Call Public!"
            # calling privateMethod inside publicMethod
            privateMethod
        end

        # defining privateMethod

    private

        def privateMethod
            puts "Call Private!"
        end
    end

    # creating an object of class Geeks
    obj = Geeks.new

    # calling the public method of class Geeks
```

```
obj.publicMethod
```

– Output:

```
Call Public!  
Call Private!
```

### 3.15 Polymorphism (tính đa hình)

**Tính đa hình (Polymorphism)** có nghĩa là đa dạng, từ một dạng tổng quát có thể sinh ra nhiều hình thức biểu hiện khác nhau (Ví dụ lớp **Shape** thì các dạng đa hình của nó là **Circle, Square, Triangle**,...). Phép đa hình được sử dụng khi có nhiều lớp có liên quan với nhau cùng kế thừa một lớp cha, nhưng với cùng một phương thức (method) nhưng biểu hiện những hành vi khác nhau.

- **Điểm giống nhau**

- Tính đa hình của Java và Ruby tương đối giống nhau.

Ví dụ

- **Java**

```
class Animal {  
    public void animalSound() {  
        System.out.println("The animal makes a sound");  
    }  
}  
  
class Pig extends Animal {  
    public void animalSound() {  
        System.out.println("The pig says: wee wee");  
    }  
}  
  
class Dog extends Animal {  
    public void animalSound() {  
        System.out.println("The dog says: bow wow");  
    }  
}
```

- **Ruby**

```
# Ruby program of Polymorphism using inheritance  
class Vehicle  
    def tyreType  
        puts "Heavy Car"  
    end  
end  
  
# Using inheritance
```

```
class Car < Vehicle
  def tyreType
    puts "Small Car"
  end
end

# Using inheritance
class Truck < Vehicle
  def tyreType
    puts "Big Car"
  end
end
```

### 3.16 Interface và Implements

**Interface** là một lớp trừu tượng hoàn toàn (**completed abstract class**), dùng để định nghĩa các phương thức chứ không hiện thực các phương thức đó (**empty body**)

**Implements** dùng để hiện thực các phương thức trong Interface đã định nghĩa.

- **Điểm giống nhau**

- Trong Java, Ruby đều có cách để hiện thực interface và implement
- Việc phân chia interface và implement thành hai phần độc lập giúp quản lý chương trình tốt hơn. Đối với interface chúng ta cố gắng hạn chế số phương thức để người dùng dễ sử dụng, đối với implements chúng ta cố gắng hiện thực các phương thức độc lập, khi thay đổi cách hiện thực của một hàm không ảnh hưởng đến các hàm còn lại, do đó việc bảo trì nâng cấp code không ảnh hưởng đến trải nghiệm người dùng thông qua interface.

- **Điểm khác nhau**

- **Java**

- Trong Java, chúng ta có các từ khóa **interface** và **implement** để định nghĩa

- **Ruby**

- Trong Ruby, không có các từ khóa **interface** và **implement**
- Do đó, chúng ta định nghĩa interface và implement thông qua **module**

#### Ví dụ

- **Java**

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

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

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

class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

- Ruby

```
# module CSV
module CSV
    def to_csv
        raise "Not implemented"
    end

    def from_csv(line)
        raise "Not implemented"
    end
end
```

```
# class User include module CSV
class User
    include CSV

    def to_csv
        "#{@name},#{@age}"
    end

    def from_csv(line)
```

```
parts = line.split(",")
@name = parts[0]
@age = parts[1]
end
end
```

### 3.17 Xử lý ngoại lệ (Exceptions Handling)

- Điểm giống nhau

- Vì Java và Ruby đều là ngôn ngữ hướng đối tượng (OOP) nên cả 2 đều có cơ chế xử lý ngoại lệ (Exceptions Handling)
- Nhìn chung, xử lý ngoại lệ của Java và Ruby đều có chung các cặp keyword đi chung như sau: **begin->try, rescue->catch,ensure->finally**

- Điểm khác nhau

- Tất nhiên, điều khác nhau đầu tiên giữa hai ngôn ngữ là ở cú pháp (**syntax**) hai ngôn ngữ.
- Ngoài ra, xử lý ngoại lệ trong Ruby không dùng throws như Java. Trong Ruby có method catch và throw nhưng không được sử dụng để xử lý ngoại lệ

#### Ví dụ

- Java

```
try {
    // Firstly, we will check the condition is exception or not
} catch(Exception ex) {
    // If condition is exception, it will be catch here and throw
} finally {
    // In case of condition is exception or not, it still come to final block
}
```

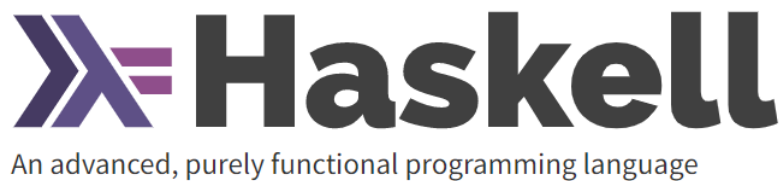
- Ruby

```
begin
    # Firstly, we will check the condition is exception or not
rescue
    # If condition is exception, it will be catch here and throw
ensure
    # In case of condition is exception or not, it still come to final block
end
```

## 4 So sánh Functional Programming giữa Haskell và SML

### 4.1 Giới thiệu một số tính chất cơ bản của Haskell và SML

**Haskell** là một ngôn ngữ thuần lập trình hàm (**purely-functional programming language**), có kiểu dữ liệu tĩnh (**statically typed**) và có tính chất **lazy evaluation**. Ngôn ngữ được đặt tên theo nhà toán học **Haskell Brooks Curry**, người sáng lập ra ngôn ngữ này. Haskell có cú pháp tương đối đơn giản, ngắn gọn, dễ hiểu do đó Haskell thường được sử dụng để lập trình cho các chương trình lớn, mã nguồn phức tạp để tiết kiệm thời gian lập trình, giúp source code ngắn gọn, dễ đọc, dễ hiểu hơn.



Hình 4: Haskell

**Standard ML (SML)** là một ngôn ngữ lập trình hàm (**functional programming language**), có tính chất **general-purpose, modular**. Đây là ngôn ngữ có kiểu dữ liệu tĩnh (**static type**) có type checking và type inference trong thời gian compile (compile-time). **SML** là một **ML language**, ML viết tắt cho Meta Language.

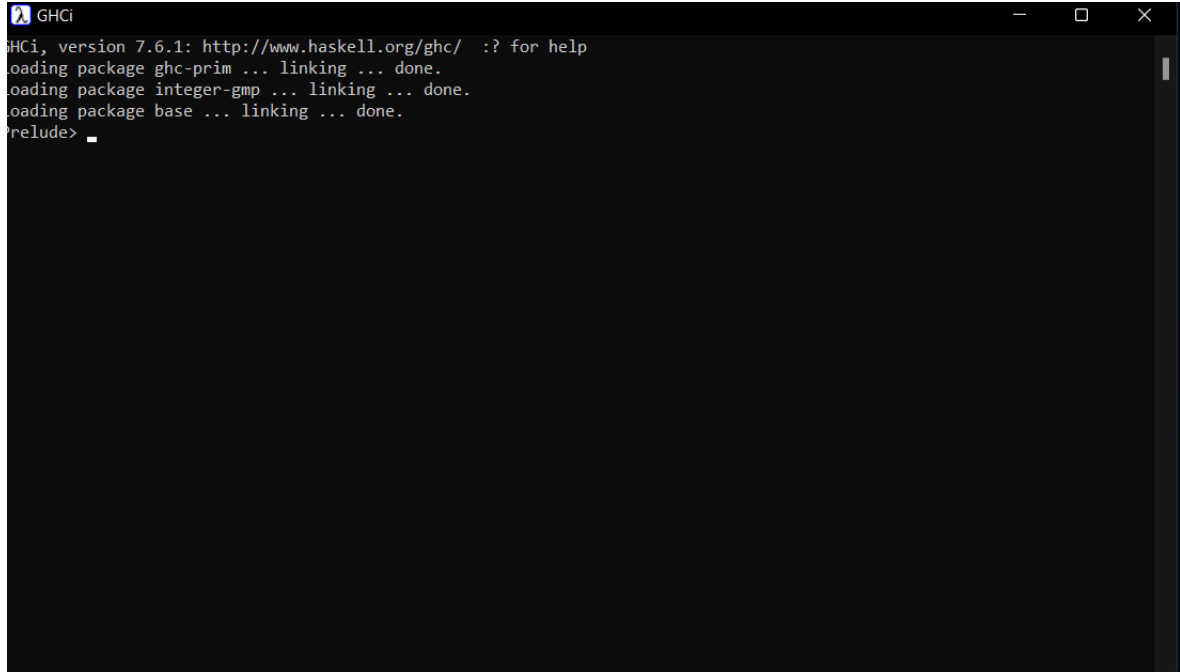


Hình 5: Standard ML of New Jersey

### 4.2 Biên dịch chương trình

- **Điểm giống nhau**
  - Cả hai đều là ngôn ngữ biên dịch (**compiled programming language**)
  - Chúng ta đều có thể viết source code trong script hoặc chạy trực tiếp trên terminal
- **Điểm khác nhau**
  - **Haskell**
    - Haskell sử dụng **GHCi** để compiled chương trình.



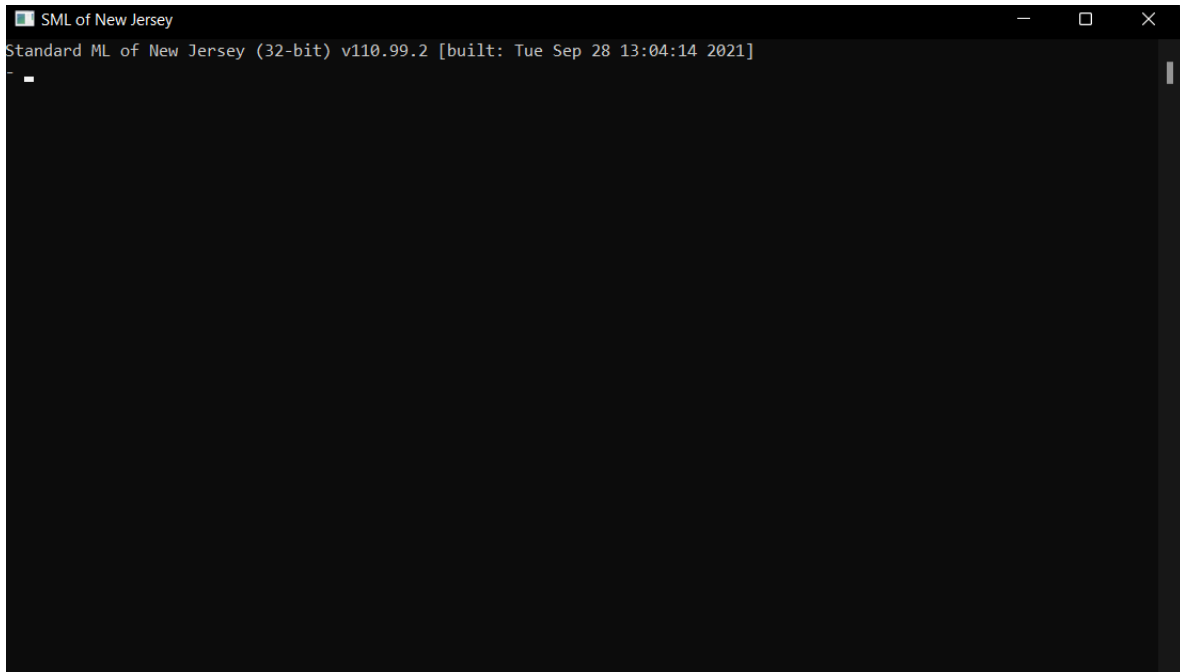


```
GHCi, version 7.6.1: http://www.haskell.org/ghc/ :? for help
loading package ghc-prim ... linking ... done.
loading package integer-gmp ... linking ... done.
loading package base ... linking ... done.
Prelude>
```

Hình 6: *GHCi*

#### ■ SML

- Haskell sử dụng **SML Of New Jersey** để compiled chương trình.



```
SML of New Jersey
Standard ML of New Jersey (32-bit) v110.99.2 [built: Tue Sep 28 13:04:14 2021]
-
```

Hình 7: *SML Of New Jersey*

### 4.3 Một số tính chất cơ bản

Đặc điểm	Haskell	Standard ML
Types	Int [a] (a,b)	int 'a list 'a * 'b
Type defined	x :: Int	x : Int
Pattern	x : xs	x :: xs
List append	xs ++ ys	xs @ ys
Composition of Functions	f . g	f o g
Lambda expression	$\lambda x \rightarrow x$	fn x -> x
Val	x = 1	val x = 1
Function Declaration	f x = x	fun f x = x
Function Implement	f :: Bool -> Int f b = if b then 1 else 0	fun f (b:bool) : int = if b then 1 else 0
Case	case xs of [] -> 0 h::_ -> h	case xs of [] => 0   h::_ => h
let	let x = 1 let y = 2 in (x+y, x-y)	let val x = 1 let val y = 2 in (x+y, x-y) end
Pair	type Pr a = (a,a)	type 'a pr = 'a * 'a

#### 4.4 Một số điểm giống nhau

- Cả hai đều có kiểu dữ liệu tĩnh (statically typed)
- Cả hai đều có kiểu dữ liệu trừu tượng (polymorphic types). Ví dụ kiểu dữ liệu là a đại diện cho bất cứ kiểu dữ liệu khác như: boolean, numeric, string,...
- Cả hai đều có **pattern matching** dùng để định nghĩa các bài toán đệ quy (**recursion**)
- Cả hai đều có hỗ trợ suy diễn kiểu dữ liệu dựa vào giá trị của biến, hay còn được gọi là **automatic type inference**

- Cả hai đều có "layered pattern" ("as" trong SML, "@" trong Haskell)
- Cả hai đều có những kiểu dữ liệu cơ bản (**basic types**) như integers, reals, booleans, strings

## 4.5 Purely Functional

### ■ Haskell

- Haskell là ngôn ngữ thuần lập trình hàm (**pure functionally programming**), không có **side effects** hoặc **exceptions**
- Một hàm thuần túy (pure) là hàm không phụ thuộc vào bất cứ điều gì khác ngoài các tham số (parameter) của nó, vì vậy khi được gọi trong các hàm khác nhau hoặc tại một thời điểm khác nhau với cùng parameter, nó sẽ trả về cùng kết quả. Ví dụ: nếu một hàm thay đổi các đối số của nó, đặt một biến ở ngoài hoặc trong (global hoặc local) hoặc thay đổi giá trị của biến global hoặc local khác với đầu vào của nó, thì hàm đó gọi là không thuần túy (not pure).
- Mọi hàm trong Haskell đều là một hàm theo nghĩa toán học (tức là pure - thuần túy), ở đây hàm nhận vào input là các parameter và trả về output chỉ dựa vào các parameter đó. Mỗi hàm chỉ làm duy nhất một nhiệm vụ của nó, các đoạn mã trong hàm chỉ tập trung làm nhiệm vụ đó, không xảy ra side effects. Ví dụ như một hàm nếu dùng để trả về một kiểu dữ liệu gì đó, thì nó không có nhiệm vụ in kết quả đó ra màn hình.

Ví dụ trong **Haskell** ta có 2 trường hợp sau

- Cách viết này là hợp lý

```
"Hello: " ++ "World!"
```

- Cách viết này sẽ xảy ra lỗi **type error**, vì lúc này **getline** chưa phải là parameter đầu vào hợp lệ

```
"Name: " ++ getline
```

### ■ SML

- SML hay ML nói chung chỉ là một ngôn ngữ lập trình hàm (**functional programming**), có tồn tại exceptions (control effects)
- Không phải tất cả các hàm trong SML đều có tính pure.

## 4.6 Lazy Evaluation and Strict

### ■ Haskell

- Haskell là ngôn ngữ có tính chất **lazy evaluation**
- **Lazy Evaluation** còn được gọi là **call-by-need**, nghĩa là không phải tất cả các tham số của hàm đều được kiểm tra hết, mà hàm chỉ kiểm tra những tham số thật sự **cần thiết** cho giá trị trả về của hàm, phần còn lại bỏ qua không xét.

### ■ SML

- Ngược lại với Haskell, SML là một ngôn ngữ có strict evaluation hay còn gọi là **eager evaluation**

- SML cũng như một số ngôn ngữ theo kiểu **strict**, nó sẽ kiểm tra điều kiện của tất cả tham số đầu vào trước, nếu không hợp lệ thì chương trình sẽ break, không xuất ra kết quả, mặc dù có thể phần không hợp lệ (invalid) không làm ảnh hưởng đến kết quả trả về của hàm.

#### Ví dụ

- Haskell

```
add :: Int -> Int -> Int
add x y = x + y
add (4,5) -- Result : 8
add (10, (289/0)) -- Result : 20
```

- SML

```
fun add(x:int, y:int):int = x + y
add (4,5) (* Result : 8 *)
add (10, (289/0)) (* Result: ?? *)
```

```
stdIn:1.2-1.12 Error: operator and operand do not agree [overload - bad instantiation]
operator domain: real * real
operand:         'Z[INT] * 'Y[INT]
in expression:
289 / 0
```

Hình 8: SML Problems

## 4.7 Generating List

- Haskell

- Haskell có thể tạo ra các **infinite list**, đây là một ứng dụng đặc biệt dựa vào tính chất **lazy evaluation** của Haskell

- SML

- SML không có tính chất trên, do đó để tạo ra các list khác nhau thì ta phải tự định nghĩa

#### Ví dụ

- Haskell

```
[1..] -- all positive integers [1,2,3,4,5,...]

[1, 3 .. ] -- all odd integers [1,3,5,7,...]
```

- SML

```
fun range (init: int, incr: int, final: int) =
  if incr < 0
  then if init < final then []
       else init :: range(init + incr, incr, final)
```

```
else if init > final then []  
    else init :: range(init + incr, incr, final);  
  
val r1 = range(1,1,10);  
val r2 = range(1,3,20);  
val r3 = range(10,~1,0);
```

#### Output

```
val range = fn : int * int * int -> int list  
val r1 = [1,2,3,4,5,6,7,8,9,10] : int list  
val r2 = [1,4,7,10,13,16,19] : int list  
val r3 = [10,9,8,7,6,5,4,3,2,1,0] : int list
```

## 4.8 List Comprehensive

### ■ Haskell

- Haskell có hỗ trợ **list comprehensive**, đây là một phương thức đặc biệt dùng để định nghĩa list, bao gồm cả chuỗi (String)

#### Ví dụ về List Comprehensive

```
> [x * x | x <- [1..5]]  
> [1,4,9,16,25]
```

### ■ SML

- SML không hỗ trợ **list comprehensive**.

## 4.9 Passing Argument

### ■ Haskell

- Các hàm trong Haskell mặc định là kiểu **currying**. Currying nghĩa là quá trình chuyển đổi một hàm nhận nhiều argument trong một bộ làm đối số của nó thành một hàm chỉ nhận một argument và trả về một hàm khác nhận các argument còn lại.

#### Ví dụ

Ta có hàm f nhận 3 đối số (argument) là a,b và c như sau

```
f :: a -> (b -> c)
```

Hàm f trên hoàn toàn có thể được viết lại như sau

```
f :: a -> b -> c
```

### ■ SML

- Các hàm trong SML hầu như sử dụng **tuple** hoặc **record** để biểu diễn cho các hàm nhận nhiều đối số (multi-argument).

- Ví dụ trường hợp sử dụng tuple truyền argument cho hàm

```
fun max(r1:real, r2:real):real =  
  if r1 < r2 then r2  
  else r1
```

- Ví dụ trường hợp sử dụng record truyền argument cho hàm

```
fun full_name{first:string,last:string,age:int,balance:real}:string =  
  first ^ " " ^ last (* ^ is the string concatenation operator *)
```

## 4.10 Type Classes and Modules

### ■ Haskell

- Haskell sử dụng **type classes** để hỗ trợ tính trừu tượng (**abstraction**) và **associated interfaces**.
- Ví dụ về chương trình tính diện tích hình tròn

```
data Area = Circle Float Float Float  
surface :: Area -> Float  
surface (Circle _ _ r) = pi * r * r  
main = print (surface $ Circle 10 20 10 )
```

### ■ SML

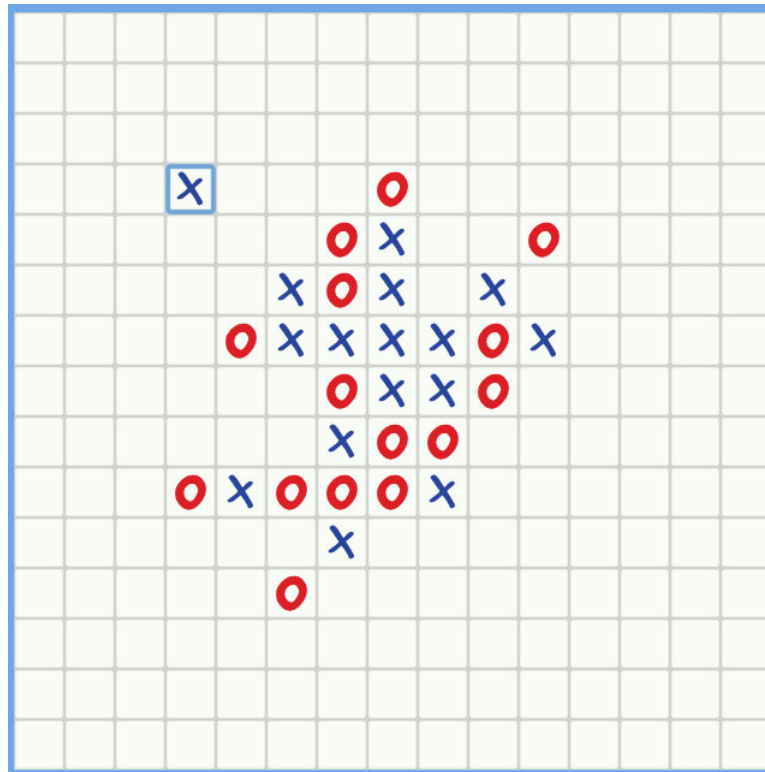
- SML sử dụng **modules** để hỗ trợ tính trừu tượng (**abstraction**) và **associated interfaces**.
- Ví dụ về chương trình Polymorphic functional stack

```
signature STACK =  
sig  
  exception E  
  type 'a retype (* <- INTERNAL REPRESENTATION *)  
  val new: 'a retype  
  val push: 'a -> 'a retype -> 'a retype  
  val pop: 'a retype -> 'a retype  
  val top: 'a retype -> 'a  
end ;
```

## 5 Game Caro - Object Oriented Programming C++

### 5.1 Giới thiệu về game Caro

- **Game Caro** là một trong những game phổ biến ở Việt Nam, còn được gọi là game **Tic-tac-toe** trên thế giới. Đây là một dạng board game, người chơi sẽ sử dụng chiến thuật của mình để đánh được 5 quân cờ cùng loại thẳng hàng liên tiếp trên cùng hàng ngang, hàng dọc, đường chéo trái hoặc đường chéo phải trước thì sẽ giành chiến thắng.



Hình 9: Game Caro

### 5.2 Hướng dẫn chơi và luật chơi

#### 1. Giới thiệu về chương trình game caro

- Game Caro là một trong những dạng game phổ biến ở Việt Nam, còn được gọi là game Tic-tac-toe trên thế giới.
- Đối với chương trình game này, chúng ta sẽ sử dụng bàn cờ có kích thước 12 x 12, hai quân cờ được sử dụng là "X" và "O".
- Game có hai chế độ là : chơi với máy và chơi 2 người.

#### 2. Hướng dẫn sử dụng phím để chơi cờ

- Theo quy ước, người chơi có thể sử dụng các phím mũi tên lên, xuống, trái phải (hoặc các phím W S A D) tương ứng để di chuyển các quân cờ theo chiều lên, xuống, trái, phải.
- Người chơi sử dụng phím space (hoặc enter) để đánh cờ.

#### 3. Luật chơi

- Theo quy ước, chúng ta sẽ sử dụng hai quân cờ "X" và "O" để tham gia đánh cờ, trong đó người chơi quân "X" sẽ được đi trước.
- Về điều kiện chiến thắng, người chơi nào đặt được 5 quân cờ giống nhau liên tiếp theo hàng ngang, hàng dọc, đường chéo trái, đường chéo phải trước thì sẽ giành chiến thắng. **Lưu ý**, chúng ta sẽ sử dụng luật chơi caro truyền thống, không xét đến quy ước chặn hai đầu.
- Trong trường hợp đi hết bàn cờ, cả hai bên đều chưa ai chiến thắng, thì được xem như Game Over cho cả hai bên.

#### 4. Đối với chế độ chơi với máy

- Đối với chế độ chơi với máy, người chơi sẽ đi quân "X" và máy tính sẽ đi quân "O".
- Người chơi sẽ đi trước, máy tính đi sau.

#### 5. Đối với chế độ chơi 2 người

- Đối với chế độ chơi 2 người, người đi trước sẽ đi quân "X" và người đi sau sẽ đi quân "O"
- Người chơi thứ 1 sử dụng các phím W S A D để di chuyển các quân cờ lên, xuống, trái, phải và sử dụng phím Space để đánh cờ, sử dụng quân "X".
- Người chơi thứ 2 sử dụng các phím mũi tên lên, xuống, trái, phải để di chuyển các quân cờ lên, xuống, trái, phải và sử dụng phím Enter để đánh cờ, sử dụng quân "O" Doi voi che do choi 2 nguoi, nguoi di truoc se di quan "X" va nguoi di sau se la quan "O".

#### 6. Bản quyền tác giả

- Đây là một phần của assignment mở rộng môn lập trình nâng cao, HK212 (CO203E) dành cho chương trình KSTN HCMUT. Tác giả chọn game caro cho assignment này.
- Game được thiết kế, hiện thực, bản quyền source code thuộc về tác giả Nguyễn Đức An (KSTN K20 KHMT HCMUT, MSSV: 2010102)
- Liên hệ tác giả qua email: an.nguyenduc1406@hcmut.edu.vn

### 5.3 Cấu trúc chương trình

Chương trình game được hiện thực theo cấu trúc OOP với ngôn ngữ C++, được chia thành nhiều file như sau. Source code chương trình được nộp chung với báo cáo BTL này.

- **Graphic.cpp**: bao gồm các hàm hỗ trợ đồ họa cho game trên cửa sổ terminal.
- **Cell.h**: định nghĩa interface cho đối tượng là ô của bàn cờ. Mỗi ô trong bàn cờ được định nghĩa bằng một class Cell.
- **Cell.cpp**: hiện thực (implement) các phương thức cho class Cell đã được định nghĩa trong file Cell.h.
- **Board.h**: định nghĩa interface cho đối tượng là bàn cờ. Bàn cờ được định nghĩa bằng một class Board.
- **Board.cpp**: hiện thực (implement) các phương thức cho class Board đã được định nghĩa trong file Board.h.
- **Game.h**: định nghĩa interface cho đối tượng là chương trình Game. Các phương thức được định nghĩa trong class Game bao gồm start game, các chế độ chơi, quit game,...



- **Game.cpp**: hiện thực (implement) các phương thức cho class Game đã được định nghĩa trong file Game.h.
- **AI.cpp**: đây là một file đặc biệt dùng để xây dựng một chương trình AI, đại diện cho computer player, dùng để hỗ trợ cho chế độ chơi với máy.
- **main.cpp**: Hàm main dùng để chạy chương trình game.

## 5.4 Bản thiết kế của chương trình

### 5.4.1 Graphic Function

Các hàm graphic sử dụng trong game là `resizeWindow`, `textColor` và `gotoXY`.

```
#include <Windows.h>

// function resizeWindow use to resize window of terminal.
void resizeWindow(int width, int height) {
    HWND console = GetConsoleWindow();
    RECT r;
    GetWindowRect(console, &r);
    MoveWindow(console, r.left, r.top, width, height, TRUE);
}

// function textColor use to change color of text.
void textColor(int color) {
    HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hStdout, color);
}

// function gotoXY use to move pointer to coordinator (x,y)
void gotoXY(int x, int y) {
    HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD position;
    position.X = x;
    position.Y = y;
    SetConsoleCursorPosition(hStdout, position);
}
```

### 5.4.2 Interface Cell

Mỗi ô được thiết kế bằng một class Cell, class Cell bao gồm:

- **Attribute**: tọa độ x,y, giá trị của ô ("X", "O" hoặc " ").
- **Method**: các phương thức get và set các thuộc tính.

```
#ifndef CELL_H
#define CELL_H
#include "Graphic.cpp"
#include <iostream>
#include <string.h>
using namespace std;
```

```
class Cell {
private:
    int x; // Coordinate x, y
    int y;
    string value; // cell contains value "X", "O", " "
public:
    // Constructor
    Cell();
    Cell(int x, int y, string value);
    // Set method
    void setX(int x);
    void setY(int y);
    void setValue(string value);
    // Get method
    int getX();
    int getY();
    string getValue();
    // Destructor
    ~Cell();
};
#endif
```

### 5.4.3 Interface Board

Bàn cờ của chương trình được đại diện bằng class Board, class Board bao gồm:

- **Attribute:** số dòng (row), số cột (col), độ rộng của dòng (rowWidth), độ rộng của cột (colWidth), một mảng hai chiều Cell\*\* board, nghĩa là với một đối tượng Board bao gồm nhiều phần tử là Cell (một mảng gồm nhiều ô).
- **Method:** các phương thức get và set các thuộc tính, thiết kế bàn cờ, xây dựng các phương thức di chuyển trên bàn cờ, cũng như kiểm tra người chiến thắng dựa trên quy ước luật chơi (rules).

```
#ifndef BOARD_H
#define BOARD_H
#include "Cell.cpp"
#include <iostream>
using namespace std;
class Board {
private:
    int row;
    int col;
    int rowWidth;
    int colWidth;
    int x_init = 34, y_init = 5;
    Cell** board; // Board
public:
    Board();
    Board(int row, int col, int rowWidth, int colWidth);
```

```
// Get method
int getRow();
int getCol();
int getRowWidth();
int getColWidth();
int getXInit();
int getYInit();
Cell** getBoard();
// Change cell value
void setCellValue(int i, int j, string value);
void setBoard(int row, int col, int rowWidth, int colWidth);
string getCellValue(int i, int j);
// Draw board
void drawTopBoard();
void blockBoard(int index);
void drawBottomBoard();
void drawBoard();
// Rule of the game
bool isSafe(int i, int j);
bool checkRow(int i, int j);
bool checkCol(int i, int j);
bool checkLeftDiagonal(int i, int j);
bool checkRightDiagonal(int i, int j);
string getWinner();
// Check fullboard
bool isFullBoard();
// Intialize board
void resetBoard();
// Print board
void printBoard();
// Build AI Player (Play with computer). Source code implement AI Bot in AI.cpp
Cell findBlankCell();
Cell getAISolution();
~Board();
};
#endif
```

#### 5.4.4 Interface Game

Giải thuật về các tính năng tương tác người dùng cũng như vận hành của game được đại diện bằng class Game, class Game bao gồm

- **Attribute:** bàn cờ (caroBoard)
- **Method:** các tính năng như loadGame, startGame, newGame, quitGame, PlayWithComputer, PlayWithPlayer, getInfoStudent, getRule

```
#include "Board.cpp"
#include <iostream>
#include <fstream>
```

```
#include <string>
using namespace std;

class Game {
private:
    Board caroBoard;
public:
    Game();
    // Load game
    void loadGame();
    // Start game
    void startGame();
    // Game Mode
    void PlayWithComputer();
    void PlayWithPlayer();
    // Get authorInfo
    void getStudentInfo();
    // Get Rule
    void getRule();
    ~Game();
};
```

#### 5.4.5 Hàm main

Hàm main là nơi để chạy chương trình

```
#include "Game.cpp"
using namespace std;
int main() {
    Game caro;
    caro.startGame();
    return 0;
}
```

## 5.5 Giao diện người dùng

### 5.5.1 Giao diện Menu



Hình 10: Menu Game

Một số lựa chọn của menu:

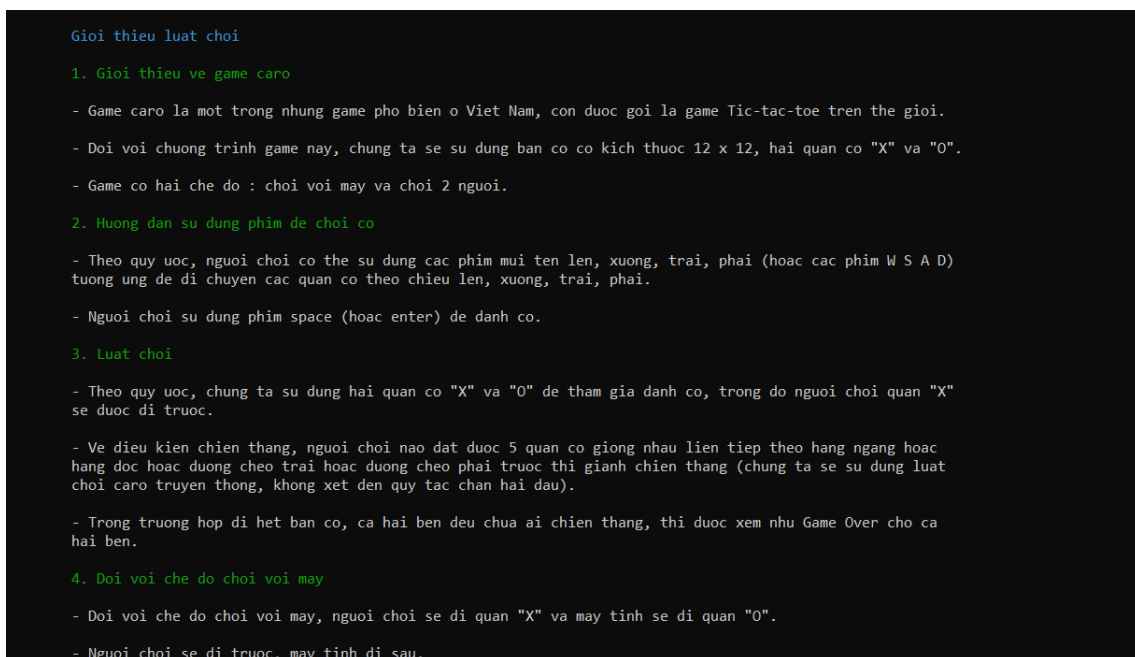
1. **Chơi với máy** : Tính năng chơi với máy, player chơi với một con AI bot do tác giả lập trình.
2. **Chơi 2 người**: Tính năng chơi 2 người, do 2 player tự chơi với nhau.
3. **Luật chơi**: Luật chơi game caro.
4. **Thông tin sinh viên**: Thông tin tác giả.
5. **Thoát ra**: Thoát khỏi chương trình.

## 5.5.2 Giao diện bàn cờ



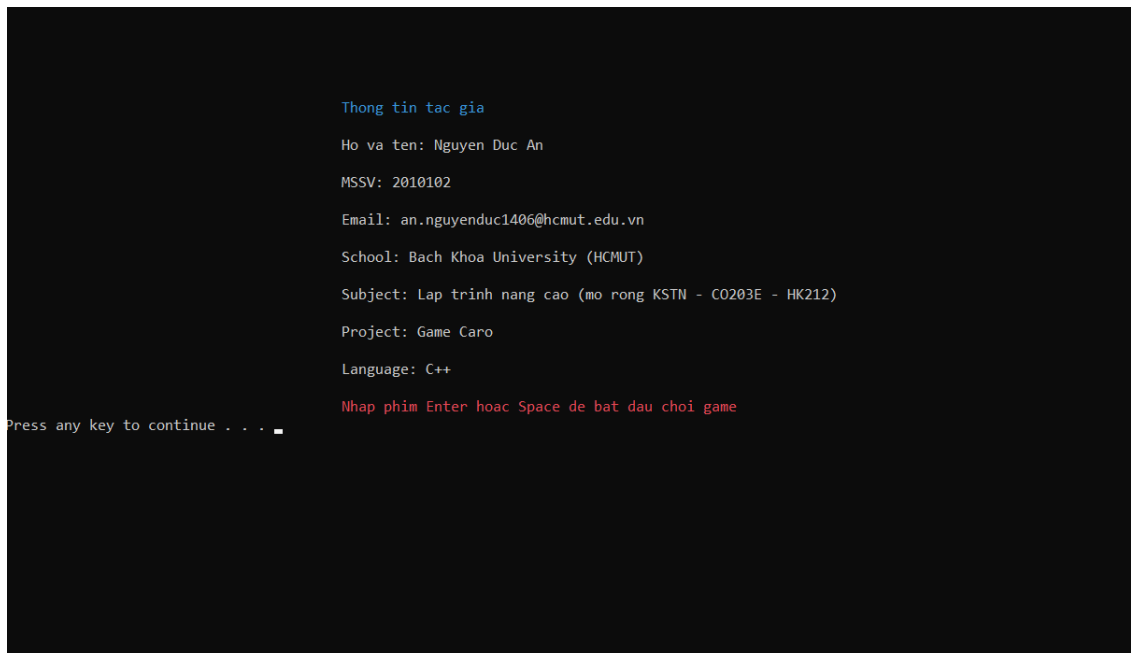
Hình 11: Game Mode

## 5.5.3 Giao diện luật chơi



Hình 12: Game Rule

#### 5.5.4 Giao diện thông tin tác giả



Hình 13: Thông tin tác giả

#### 5.6 Demo và kiểm thử chương trình

- Link demo: <https://youtu.be/zEB-Qb2CvZc>

## 6 Tài liệu tham khảo

### Tài liệu

- [1] Packt Publishing, "Oracle Blockchain Quick Start Guide: A practical approach to implementing blockchain in your enterprise", September 6, 2019.
- [2] Marconi Foundation, "Marconi Protocol", May 2, 2018.
- [3] Going the distance, "A blog about blockchains and smart contracts development", truy cập từ: <https://jeiwan.net/>
- [4] Bizfly Cloud, "Web 3.0 là gì? Tìm hiểu chi tiết về Web 3.0 - Kỷ nguyên mới của Internet", truy cập từ: <https://www.creative-tim.com/blog/educational-tech/web-1-0-vs-web-2-0-vs-web-3-0-what-are-the-differences/>