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

LẬP TRÌNH NÂNG CAO

HỌC PHẦN MỞ RỘNG (CO203E)

Tìm hiểu về Lập trình hướng đối tượng và Lập trình hàm và Hiện thực game C++ bằng cấu trúc OOP

GVHD: TS Trương Tuấn Anh
SV thực hiện: Kha Sang – 2010576

Tp. Hồ Chí Minh, Tháng 05/2022

Mục lục

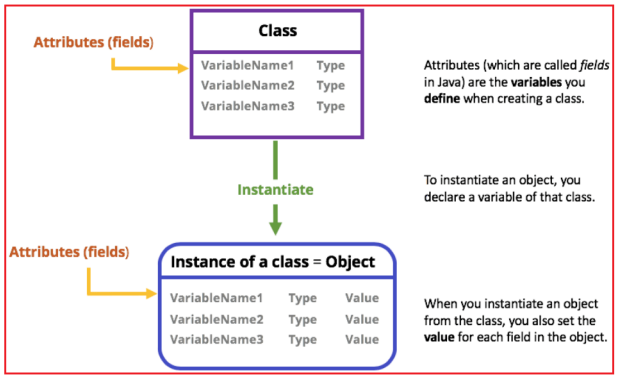
Danh sách hình vẽ

1 Cơ sở lý thuyết

1.1 Lập trình hướng đối tượng (Object Oriented Programming)

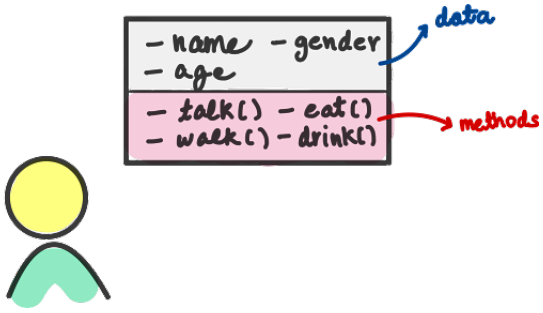
1.1.1 Các khái niệm cơ bản

- *Lập trình hướng đối tượng* (Object Oriented Programming, hay còn được gọi tắt là OOP) là một kỹ thuật lập trình dựa trên khái niệm các "đối tượng (object)".
- *Lớp và Đối tượng*: Một lớp là bản mẫu cho một đối tượng, có thể có nhiều đối tượng sinh ra từ một lớp. Ví dụ đối với class SinhVien, ta sẽ có các đối tượng như Nguyễn Văn A, Nguyễn Thị B,..



Hình 1: *Lớp và Đối tượng*

- *Thuộc tính và Phương thức*: Mỗi object sẽ chứa các thông tin riêng, cũng như là các thao tác trên đó. Trong các class, ta sẽ có sẵn các trường dữ liệu, nhưng giá trị của nó phải do bản thân object quyết định.
  - Dữ liệu được lưu trữ dưới dạng các trường (hay còn gọi là thuộc tính), chứa thông tin của bản thân đối tượng đó. Thông tin có thể là các loại biến số học, chuỗi kí tự, hay kể cả là một object.
  - Thao tác (phương thức) là những hàm tác động trực tiếp lên đối tượng đó, có thể truy xuất hoặc gán giá trị lên thuộc tính của đối tượng.
  - Ví dụ như class Employee sẽ có các thuộc tính như số ngày làm việc, mã nhân viên,... còn phương thức là tính tiền lương, lấy mã số nhân viên,...
  - Người ta sử dụng khái niệm "this" (hoặc là "self" trong Python) cùng với toán tử dot (.) để ám chỉ biến hoặc phương thức đó là của đối tượng hiện tại.



Hình 2: *Thuộc tính và Phương thức*

- *Constructors*: Giống như các loại biến khác, một đối tượng cũng phải có cách khởi tạo để lưu các giá trị. Tuy nhiên, do một object là tập hợp nhiều loại biến cơ sở (số học, chuỗi) ta cần phải lập trình để compiler hiểu các giá trị mà mình muốn gán vào.
  - Constructor là một phương thức đặc biệt ở các class phục vụ việc khởi gán các thuộc tính cho object, và sẽ được tự động gọi khi ấy để lấy các tham số gán vào các thuộc tính trong object.

- Tuy nhiên, số lượng tham số truyền vào là một vấn đề. Do đó, ở một số class, ta phải lập trình nhiều constructors với số lượng tham số khác nhau (constructor overloading), hay kiểu biến khác nhau để phục vụ các nhu cầu của user. Constructor không có tham số được gọi là default constructor.
- Có một loại Constructor khác, nó sẽ nhận tham số là một object cùng class. Tuy nhiên cần lưu ý rằng ta phải truyền tham chiếu (với C++ thì có kí hiệu & phía trước tên object) để tránh hành động gọi đệ quy vô tận.
- Ngoài ra, còn có khái niệm constructor và kế thừa, ta sẽ đề cập ở phần sau.

```
class MyClass
{
    int a;
    string b;

    // Default Constructor
    MyClass(){
        this->a = 42;
        this->b = "string";
    }

    // Parameterized constructor
    MyClass(int a, string b){
        this->a = a;
        this->b = b;
    }

    //Copy constructor
    MyClass(MyClass &c){
        this->a = c.a;
        strcpy(this->b,c.b);
    }
};
```

Listing 1: Constructor with C++ example

- *Destructor*: Trong khi Constructor là phương thức xây dựng object, Destructor là phương thức để xóa object đó, với tác dụng là tránh rò rỉ bộ nhớ (memory leak).
  - Destructor được tự động gọi bởi compiler khi đối tượng vượt ngoài tầm vực (out of scope).
  - Cần lưu ý rằng mặc dù có thể có rất nhiều constructors, destructor cho một class là duy nhất và ta không được truyền bất kì tham số nào vào trong phương thức hàm hủy.
  - Bên cạnh việc phải hiện thực "thủ công" như C++, một số ngôn ngữ như Java, Python có cơ chế Garbage Collection xóa object thừa một cách tự động.

```
class Table
{
    int *dishes;
public:
    Table(){
        dishes = new int[10];
    }
    Table(int numDishes){
        dishes = new int[numDishes];
    }
    ~Table(){
        delete[] dishes;
    }
};
```

Listing 2: Destructor with C++ example

- *Static*: bên cạnh những thuộc tính và và phương thức chỉ tồn tại trong từng object, từ khóa `static` dùng để chỉ các thuộc tính và phương thức tồn tại trong phạm vi một class. Nhờ vậy, ta có thể gọi phương thức `static` của một class mà không cần thiết phải tạo object và giúp tính toán một số tác vụ trên toàn class như tổng số object được tạo.

1.1.2 Các nguyên lý của Lập trình hướng đối tượng

1. **Tính bao đóng (Encapsulation)**

- *Khái niệm* Tính bao đóng là việc gói dữ liệu và thuộc tính của một đối tượng vào trong một đơn vị đơn lẻ. Nhờ vậy, các thuộc tính bên trong không bị tác động trực tiếp từ bên ngoài và được "bảo vệ".
- *Đặc điểm*
  - Ở đây ta sẽ giới thiệu khái niệm *access modifier*, hay còn gọi là phạm vi truy xuất. Đây là một đặc trưng của tính bao đóng giúp bảo vệ các thuộc tính bên trong đối tượng.
  - Phạm vi truy xuất với 3 từ khóa `private`, `public` và `protected` gắn vào trong các thuộc tính (đối với C++, ngoài ra đối với Java còn có thêm từ khóa `default`) quyết định tầm vực mà user có thể chỉnh sửa thuộc tính trong class.
  - Ở đây để ngắn gon, em chỉ giới thiệu hai từ khóa `private` và `public`. Cụ thể với `private`, tầm vực chỉ là trong class, các thao tác bên ngoài class sẽ không tác động được. Ngược lại đối với `public`, thuộc tính có thể được thay đổi dù cho nơi truy nhập ở bất kì đâu.
  - Tính bao đóng thể hiện ở chỗ `private`. Người dùng không thể trực tiếp thay đổi thuộc tính mà phải qua các hàm trung gian Getter và Setter được khai báo trong class (chế độ public), nơi mà ta có thể xử lý yêu cầu người dùng có hợp lệ hay không.
  - Từ khóa `protected` cũng giới hạn phạm vi truy xuất của class là các lớp con của nó. Em sẽ trình bày rõ hơn về khái niệm này trong phần sau.

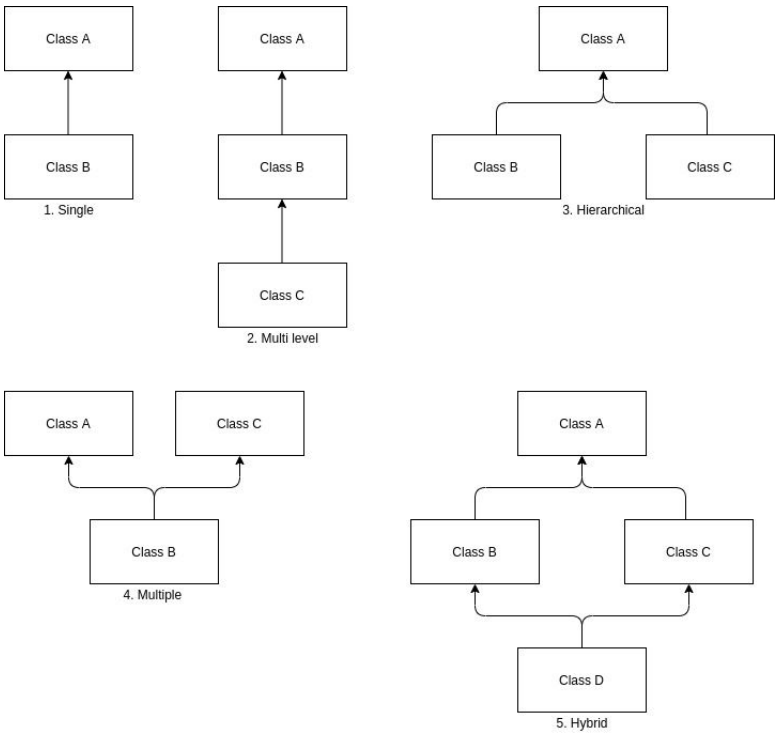
```
1  public class Person {
2    private String name; // private = restricted access
3    private int age;
4
5    // Getter
6    public String getName() {
7      return name;
8    }
9    public String getAge() {
10     return age;
11   }
12
13   // Setter
14   public void setName(String newName) {
15     this.name = newName;
16   }
17   public void setAge(int newAge) {
18     this.age = Age;
19   }
20 }
21
22 public class Main {
23   public static void main(String[] args) {
24     Person myObj = new Person();
25     myObj.name = "John";  // error
26     System.out.println(myObj.name); // error
27   }
28 }
```

Listing 3: Getter-Setter with Java example

- *Tác dụng*
  - Ta có thể làm cho một class "read-only" hay "write-only". Với "read-only" thì chỉ cần cung cấp các phương thức getter, còn "write-only" thì cấp phương thức setter.
  - *Kiểm soát dữ liệu*: người dùng không thể tùy tiện thay đổi thuộc tính trong class mà phải qua hàm setter để kiểm tra sự phù hợp của giá trị gán vào.
  - *Che dấu dữ liệu*: bên ngoài class không thể truy cập vào bên trong class trực tiếp nếu không được "cho phép".

2. Tính kế thừa (inheritance)

- *Khái niệm* Kế thừa là cơ chế mà một lớp con được thừa hưởng các thuộc tính và phương thức của lớp cha. Ví dụ, nếu tứ giác là lớp cha, các lớp con có thể là hình thang, hình chữ nhật, hình vuông,... do chúng đều có 4 cạnh và 4 đỉnh của lớp tứ giác.
- *Đặc điểm*
  - Tùy theo ngôn ngữ lập trình mà cách thức kế thừa khác nhau. Đối với C++, lớp con chỉ có thể truy nhập vào các thuộc tính và phương thức được gán nhãn `protected` và `public` từ lớp cha.
  - Kế thừa thể hiện mối quan hệ **IS-A** giữa các đối tượng, tức class con *là* là class cha. Ví dụ: học sinh "là" con người.
  - Có 5 loại kế thừa:
    (a) Đơn kế thừa (single): 1 class con kế thừa duy nhất từ một class cha.
    (b) Kế thừa đa cấp (multi-level): 1 class con kế thừa duy nhất từ một class cha, class cha đó lại kế thừa từ class khác.
    (c) kế thừa phân cấp (hierarchy): nhiều hơn một class con kế thừa từ class cha.
    (d) Đa kế thừa (multiple): class con được kế thừa từ nhiều hơn một class cha. Một số ngôn ngữ (ví dụ như Java) không hỗ trợ đa kế thừa.
    (e) Kế thừa phức hợp (hybrid): là tổ hợp của 4 loại kể trên.

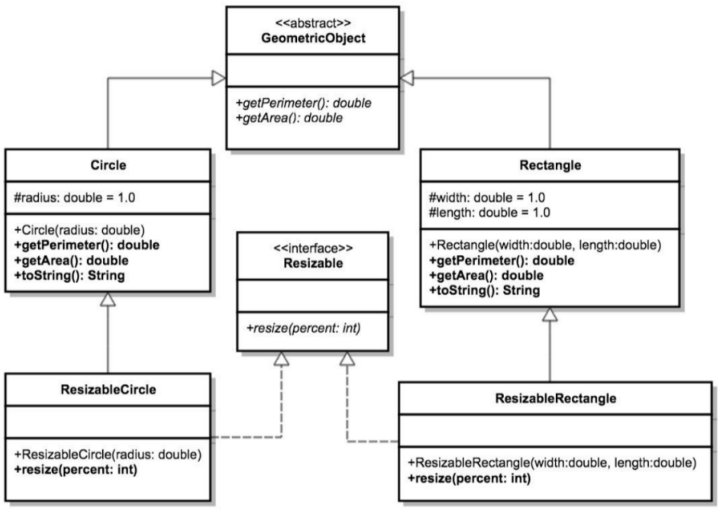


Hình 3: *Inheritance Types*

  - *Upcasting và Downcasting*: là việc chuyển đổi đối tượng kiểu class con và cha. Nếu chuyển từ class con sang cha (upcasting) sẽ không bảo toàn các thuộc tính, việc chuyển từ class cha sang class con sẽ nhận một số giá trị rác không được thừa kế từ class cha nhưng lại đảm bảo tính tổng quát hóa.
- *Tác dụng*
  - *Tái sử dụng code*: lớp con thừa hưởng các thuộc tính và phương thức của lớp cha nên ta không cần phải hiện thực lại.
  - *Dễ sửa lỗi*: chỉ cần sửa lỗi ở lớp cha thì các lớp con cũng được thay đổi theo.
  - *Hỗ trợ tính đa hình*: override các phương thức với mối quan hệ IS-A

3. Tính trừu tượng (Abstraction)

- *Khái niệm*
  - Trừu tượng là quá trình che dấu chi tiết hiện thực mà chỉ biểu diễn các chức năng đến người dùng, bởi vì họ thật sự chỉ cần tác dụng của phương thức chứ không quan tâm đến cách hiện thực phương thức ấy.
  - Nói cách khác, trừu tượng nghĩa là chỉ đưa ra các đặc điểm yêu cầu mà không cần các chi tiết về code.
- *Đặc điểm*: Trong OOP, ta có hai mức độ trừu tượng, trừu tượng một phần (abstract) và trừu tượng hoàn toàn (interface)
  - *Trừu tượng một phần*: Với mức độ này, dữ liệu và phương thức sẽ không thể truy cập từ bên ngoài class.
    - *Lớp trừu tượng*: là lớp có từ khóa `abstract` ở phía trước khai báo. Chúng ta không thể tạo ra các object từ lớp trừu tượng, tuy vậy nhưng nó vẫn có constructor và các phương thức `static`.
    - *Phương thức trừu tượng*: là phương thức có từ khóa `abstract` ở phía trước khai báo. Phương thức này không chứa code bên trong và chỉ chứa dấu chấm phẩy.
    - Nếu có ít nhất một phương thức trừu tượng bên trong class thì class đó cũng trừu tượng và lớp trừu tượng có thể không có phương thức trừu tượng.
    - Lớp kế thừa đầu tiên của lớp trừu tượng phải cung cấp phần code cho tất cả phương thức trừu tượng bên trong, nếu không thì lớp kế thừa cũng được xem là lớp trừu tượng.
  - *Trừu tượng toàn phần (interface)*: Với mức độ gồm 100% phương thức trừu tượng, ta tiến hành trừu tượng hóa từ design cho đến coding, chỉ tập trung hoàn toàn vào "what" chứ không phải "how". Nói cách khác, interface là bản mẫu thiết kế của một lớp (không có constructor và không thể tạo object trực tiếp từ interface) và người dùng phải hiện thực ít nhất tất cả các phương thức trong interface khi thiết kế lớp đó.



Hình 4: *Interface VS Abstract*

- *Tác dụng*
  - Trừu tượng giúp cho code chỉ tập trung vào các tính năng cốt lõi và bỏ qua những chi tiết nhỏ nhất, giúp chương trình dễ bảo trì và hạn chế lỗi.
  - Interface giúp một số ngôn ngữ OOP đơn kế thừa như Java giải quyết vấn đề đa kế thừa (có thể có nhiều interface cho một class).
  - Về mặt quan hệ, các class được tạo thành từ interface "độc lập" với nhau, có nghĩa là ngoài 2 class không hề liên quan có thể hiện thực trên cùng một interface cho một tính năng tương đồng nào đó.

4. **Tính đa hình (polymorphism)**

- *Khái niệm*: Đa hình là khả năng mà một đối tương có nhiều hành động khác nhau trên cùng một phương thức. Nói cách khác, một hành động cùng tên có thể được thực hiện khác nhau đối với ngữ cảnh khác nhau.
- *Đặc điểm*:
  - *Nạp chồng phương thức (Method overloading)*: là kĩ thuật thực hiện trên compile time, áp dụng cho một lớp có nhiều phương thức cùng tên nhưng lại khác về mặt tham số, với điều kiện duy nhất là danh sách tham số phải khác nhau đôi một.
  - *Ghi đè phương thức (Method overidding)*: là kĩ thuật thực hiện trên run time, áp dụng cho lớp con có phương thức cùng tên với lớp cha. Điều kiện là các phương thức phải có cùng danh sách tham số, kiểu trả về và phạm vi truy xuất của phương thức ghi đè phải rộng hơn phương thức "gốc".
  - Phương thức `static` được nạp chồng nhưng không thể bị ghi đè. Lý do là vì các phương thức đó gắn liền với class thay vì object trong compile time. Tương tự, constructor cũng chỉ có thể được overload.
  - Còn có một loại đa hình Compile-Time là *nạp chồng toán tử (operator overloading)*, với cách thức hoạt động cũng tương tự như nạp chồng phương thức. Thay vì thực hiện các phép toán số học hay indexing trên các kiểu dữ liệu cơ bản, nạp chồng toán tử giúp ta thực hiện các thao tác ấy với kiểu dữ liệu là tên của object, ví dụ như tính toán trên các class Phân Số, Số Phức,...

```java
class Hello{
    public void sayHi() {
        System.out.println("Coding in C++");
    }
    public void sayHi(String language) {
        System.out.println("Hello in "+language);
    }
}
public class Main {
    public static void main(String[] args) {
        Hello obj = new Hello();
        obj.sayHi(); // "Hello"
        obj.sayHi("Java"); // "Hello in Java"
    }
}
```

Listing 4: Method Overload with Java example

```java
public class HelloFather {
    public void sayHi() {
        System.out.println("Hello in Father");
    }
}
public class HelloChild extends HelloFather{
    public void sayHi() {
        System.out.println("Hello in Child");
    }
}
public class Main {
    public static void main(String[] args) {
        HelloFather obj = new HelloChild();
        obj.sayHi(); // "Hello in Child"
    }
}
```

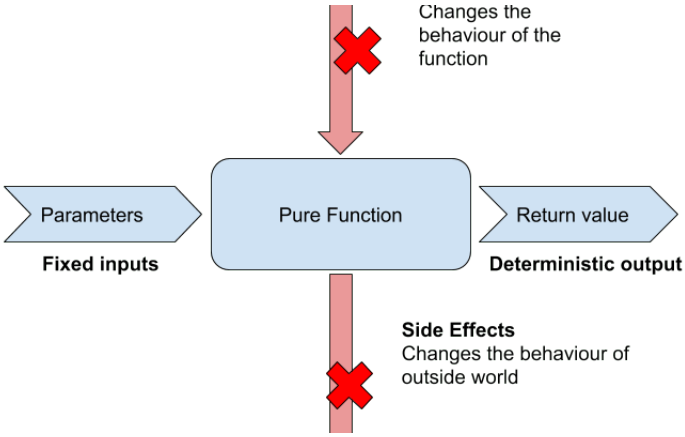
Listing 5: Method Overriding with Java example

- *Tác dụng*
  - *Compile-Time Polymorphism*: Tăng độ đa dạng của các phương thức overloading, nhờ vậy phù hợp với nhiều đầu vào khác nhau.
  - *Run-Time Polymorphism*: Với phương thức overriding của lớp con, ta có thể tìm được thao tác phù hợp ứng với kiểu object.

1.2 Lập trình hàm (Functional Programming)

1.2.1 Giới thiệu về Lập trình hàm

- *Lập trình hàm* (Functional Programming, hay còn được gọi tắt là FP) là một kỹ thuật lập trình mà chương trình được xây dựng dựa trên việc sử dụng và tạo các hàm, tức lấy các hàm là đơn vị thao tác cơ bản.
- *Pure function*: Lập trình hàm sử dụng Pure function, thường có tính chất như các hàm toán học, và trả về kết quả độc lập với các biến có tầm vực rộng hơn.
  - Side effect là hiện tượng một hàm số thay đổi trạng thái của chương trình (I/O, tác động biến static cục bộ hay global) và có thể ảnh hưởng đến đầu ra của hàm. Điều này làm tăng độ phức tạp của chương trình vì số trạng thái khá nhiều.
  - Referential transparency là khái niệm để chỉ output của một hàm dù có thay đổi thì vẫn không đổi kết cục của chương trình.
    * Pure function luôn trả về duy nhất một giá trị đầu ra nếu cùng một input và đảm bảo referential transparency.
    * Pure function không gây ra side effect.



Hình 5: *Pure Function*

```
1 const add = (a, b) => {
2   return a + b;
3 };
```

Listing 6: Pure function with Javascript example

- *Immutable data*: Những biến đã được khai báo một lần thì sẽ luôn không thay đổi giá trị đã được khởi gán.
  - Immutable là một trong các đặc trưng cơ bản nhất của Functional Programming. Bên cạnh đó, Mutable (có thể gán lại biến nhiều lần) là điều tối kỵ cần tránh khi lập trình FP.
  - Do lập trình hàm thiên về khái niệm các hàm số trong toán học, điều này giúp ta hạn chế một số vấn đề. Với lập trình đa luồng, điều này giúp 1 luồng có thể tác động lên đối tượng immutable mà không cần lo đến việc đối tượng bị ai đó chỉnh sửa, do đó bảo đảm tính an toàn.
  - Immutable còn tránh tình trạng shared-state functions (nhiều hàm phụ thuộc vào một state) và mutating-state functions (hàm phụ thuộc vào hàm mutable khác), vì chúng làm chương trình mất tính mô-đun (modular).
  - Ta có thể sử dụng kĩ thuật gọi đệ quy (recursion) để bảo đảm tính immutable.

```
// Mutable function
def iFactorial(n: Int): Int = {
  var i = n
  var f = 1
  while (i > 0) {
    f = f * i
    i = i - 1
  }
  return f
}

// Immutable function
def rFactorial(n: Int): Int = {
  if (n <= 0)
    return 1
  else
    return n * rFactorial(n - 1)
}
```

Listing 7: Mutable and Immutable with Scala example

- *Higher-order function*: tương tự với khái niệm hàm của toán học, higher-order function là một hàm nhận đầu vào là một hàm khác hoặc trả về kết quả là một hàm.
  - Trong Functional Programming, hầu như mọi functions đều là higher-order function, vì chúng đều có thể nhận vào và ném ra các functions.
  - High-order functions cho phép tính linh hoạt tốt hơn trong việc gọi hàm và trừu tượng hóa hành động.

```
// getItem return "by" function
const getItem = (by) => {
  return (arr) => {
    return by(arr);
  };
};
// maxNumber get max value in array
const maxNumber = (arr) => {
  return Math.max(...arr);
};

const getMaxNumber = getItem(maxNumber);

getMaxNumber([4, 6, 2, 3, 1, 8, 7, 5]);
// => 8
```

Listing 8: High-order functions with Javascript example

- *Composition*: Mỗi hàm đại diện cho một thao tác và có thể ghép một chuỗi có thứ tự các hàm cơ bản thành hàm phức tạp hơn.
  - Đây là khái niệm bắt nguồn từ hàm hợp của Toán học. Composition đơn thuần chỉ là sự lồng ghép các hàm với nhau, tạo thành một hàm lớn với chức năng phức tạp. Có hai kỹ thuật trong Composition là `compose` và `pipe`
  - *Compose*: Theo như tính chất pure function, FP tồn tại các hàm nhỏ và đơn giản. Vì mỗi hàm chỉ làm 1 việc, khi muốn thực hiện nhiều hành động lên input, ta chỉ cần kết hợp các hàm cần thiết với nhau.
  - *Pipe*: vận hành theo chiều ngược lại so với `compose`, và nhìn có vẻ thuận mắt hơn với người dùng.

```
import functools
# compose(a,b,c) = a(b(c(x)))
def compose(*functions):
    def compose2(f, g):
        return lambda x: f(g(x))
    return functools.reduce(compose2, functions, lambda x: x)
```

```
# pipe(a,b,c) = c(b(a(x)))
def pipe(*functions):
    def pipe2(f, g):
        return lambda x: f(g(x))
    return functools.reduceRight(compose2, functions, lambda x: x)
```

Listing 9: Functional composition with Python example

– *Curried function*: kỹ thuật chuyển một hàm số với nhiều tham số đầu vào thành một dãy các hàm số nhận duy nhất một tham số. Ví dụ với $x = f(a, b, c)$ thì $h = g(a), i = h(b), x = i(c)$

```
aaa  = curry (\ (x,y) -> 2*x+y)
{-
Input: aaa 2 3
Output: 7
-}
```

Listing 10: Curry function with Haskell example

1.2.2 Lập trình hàm và Lập trình hướng đối tượng

Lập trình hướng đối tượng hoạt động theo nguyên tắc Imperative, tức chỉ rõ các bước để máy tính thực hiện tuần tự (tập trung vào "how"). Tuy nhiên, với lập trình hàm, ta dùng kiểu tư duy Declarative, sử dụng và kết hợp các hàm đơn vị để thực hiện mục đích của mình (tập trung vào "what"). Do đó, FP sẽ không có các đặc trưng sau đây

- For/while: do ta đã có quy tắc đầu ra, đầu vào, vòng lặp là không cần thiết

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Listing 11: Sum of a list with Haskell example

- New/this: do khái niệm của FP thuần không có object, khái niệm new/ this cũng không còn.
- If/else: ta thường sử dụng các toán tử logic và toán tử ba ngôi (ternary) để hạn chế việc dùng `if/else` vốn khá phổ biến trong OOP.

Functional Programming	Object Oriented Programming
Hàm là đơn vị cơ bản	Đối tượng là đơn vị cơ bản
Pure functions không có side-effects.	Các phương thức có thể có side-effects.
Theo mô hình Declarative (câu hỏi "what")	Theo mô hình Imperative (câu hỏi "how")
Không thể tạo các đối tượng mutable với ngôn ngữ thuần FP	OOP có thể hỗ trợ cả đối tượng mutable and immutable, tùy vào ngôn ngữ lập trình.
Chỉ đưa ra đúng một output với input nhất định và biến sau khi gán không được thay giá trị. Do đó, FP dễ debug, test và có thể áp dụng cho lập trình đa luồng.	OOP hoạt động dựa trên quan hệ các object, mỗi object lại bao gồm thông tin và thao tác. Data được ưu tiên hơn các hàm gọi.

Hình 6: *Functional Programming vs Object Oriented Programming*

Thực tế cho thấy, cả hai kĩ thuật lập trình đều có những điểm mạnh và yếu riêng. Ngày nay, các lập trình viên đang rất ưa chuộng ngôn ngữ lập trình phục vụ cả hai phương pháp (ví dụ như Python và JavaScript) dựa trên yêu cầu và mục tiêu của dự án.

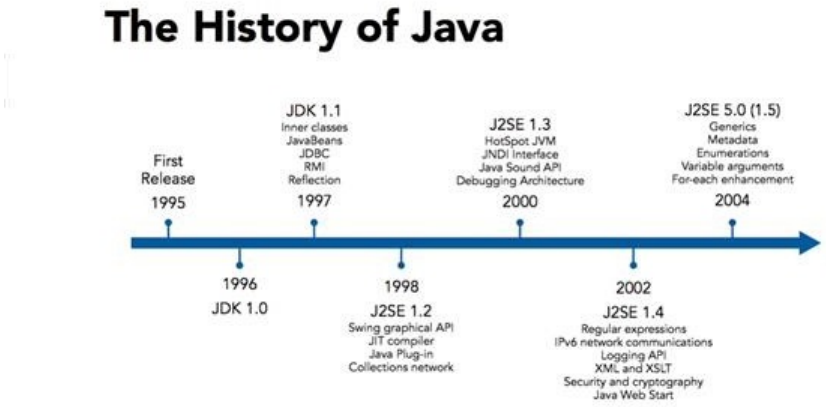
2 Lập trình Hướng đối tượng với Java và Ruby

2.1 Ngôn ngữ lập trình Java

Hình 7: *Logo Java*

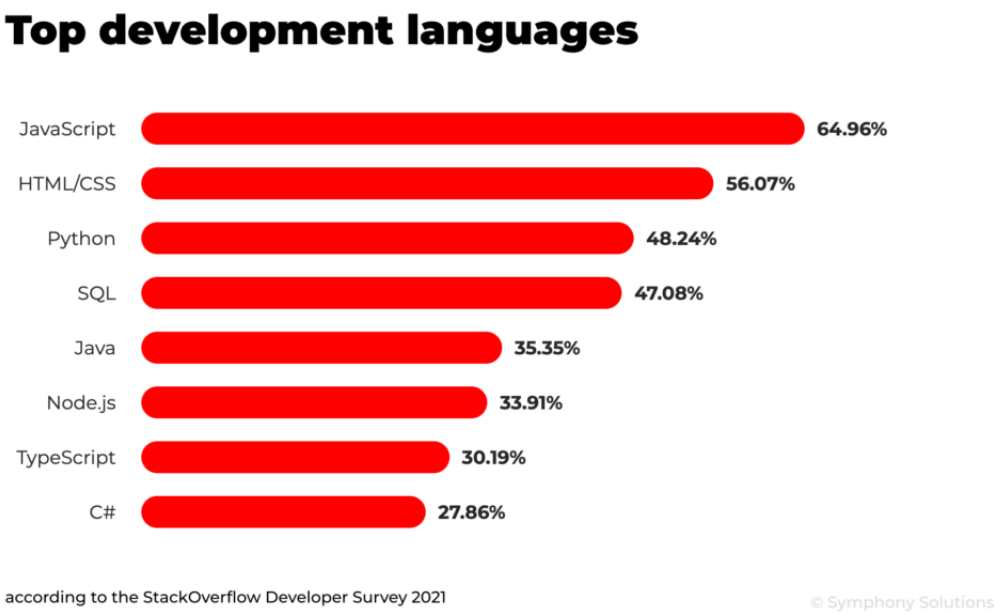
2.1.1 Lịch sử phát triển của Java

- *Nguyên nhân ra đời*: Java là một ngôn ngữ lập trình hướng đối tượng (OOP) được phát triển bởi *James Gosling* vào đầu thập niên 90. Vào thời điểm ấy, tác giả và đội ngũ "Green Team" của mình chỉ hướng đến việc phát triển một ngôn ngữ sử dụng cho các thiết bị số như tivi, radio và thay thế một ngôn ngữ đang rất thịnh hành lúc bấy giờ. Về sau, người ta nhận thấy rằng Java thích hợp hơn lập trình trên web, và từ đó độ phổ biến của ngôn ngữ này ngày càng tăng.



Hình 8: *Lịch sử của Java*

- *Lược sử tên gọi*: Ban đầu, Green Team gọi dự án phát triển ngôn ngữ lập trình này là "Greentalk" với định dạng có đuôi là ".gt" và mang tên "Oak". Tên gọi này bắt nguồn từ cây sồi (oak tree) được trồng trong vườn nhà ông Gosling, và là biểu tượng cho tính đoàn kết của các thành viên trong team. Sau đó, nhóm buộc phải đổi tên vì trùng với thương hiệu sản xuất phần cứng Oak Technologies, và cái tên Java ra đời kể từ đó. Cái tên "Java" cùng với biểu tượng được chọn bắt nguồn từ một loại hạt espresso, bất giác được tác giả nghĩ tới trong khi đang uống cà phê tại cuộc họp đặt tên.
- *Nguyên tắc*: Ngôn ngữ Java được tạo ra với nguyên tắc "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".
- *Ứng dụng*: Hiện tại, Java được sử dụng trong lập trình web, thiết bị di động, game,... và đang là một trong những ngôn ngữ lập trình phổ biến và mạnh mẽ nhất trong những năm vừa qua.

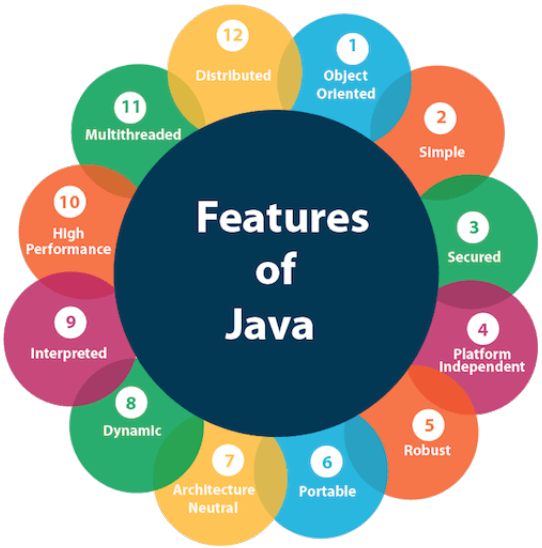


Hình 9: *Java nằm trong top 5 ngôn ngữ năm 2021. Nguồn: Stackoverflow*

2.1.2 Một số đặc điểm và thế mạnh nổi bật của Java

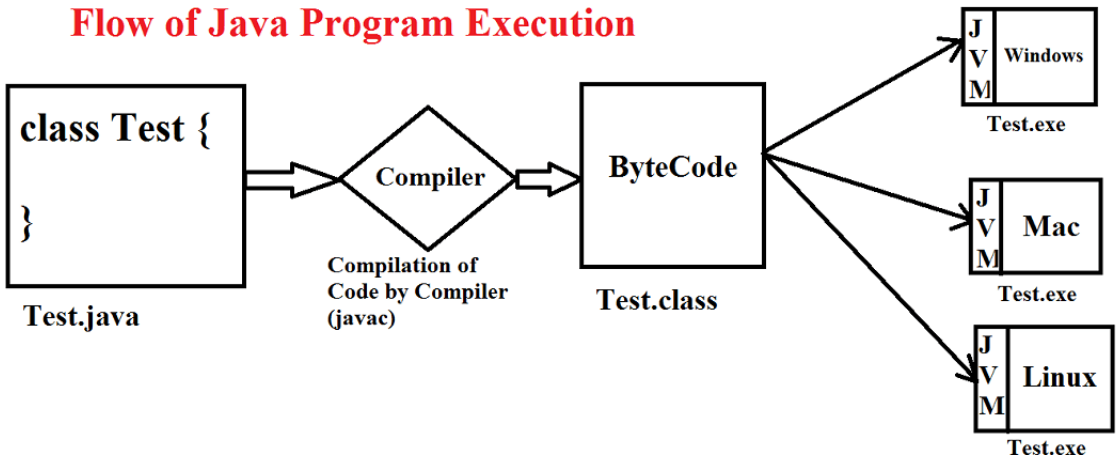
Không phải dễ dàng mà Java có thể chiếm một vị trí vững chắc trong cộng động lập trình viên trong gần ba thập kỉ qua. Ở đây, em sẽ trình bày một số đặc điểm nổi trội của Java:

1. **Simple**: Java là một ngôn ngữ khá dễ học với cấu trúc ngữ nghĩa đơn giản hơn ngôn ngữ tiền nhiệm như C và C++. Nhờ vậy mà lập trình viên có thể dễ code hơn mà không lo một số lỗi cú pháp ở việc quản lý bộ nhớ. Một ví dụ điển hình là khái niệm con trỏ và nạp chồng toán tử vốn là đặc sản của C++ đã được lược bỏ ở Java, khiến cho code đọc dễ hiểu và "sạch" hơn.
2. **Robust**: Như đã nói ở trên, Java không đi sâu vào việc quản lý bộ nhớ như C và C++. Lý do là vì tác giả đã tích hợp cơ chế *Garbage Collector*, giúp compiler có thể tự động "quét dọn" các ô nhớ thừa của object vượt khỏi scope. Ngoài ra, Java còn có cơ chế *Exception Handling* tự động xử lý một số ngoại lệ mà người dùng chưa lập trình sẵn.
3. **Portable**: Trước khi nói đến đặc điểm này ta cần phải biết cách Java compiler hoạt động: Source code sau khi đã hoàn chỉnh (không còn error) sẽ được biên dịch thành "bytecode", sau đó JVM (Java Virtual Machine) sẽ thực thi bytecode file (lưu dưới dạng đuôi *.class*) để chạy chương trình. Nhờ vậy, bytecode file sẽ được lưu trữ sẵn và chuẩn bị chạy với bất kì platform hỗ trợ.



Hình 10: *Các đặc trưng của Java*

4. **Platform-Independent**: Không giống như C và C++ với cơ chế biên dịch riêng cho từng platform, Java đảm bảo tính năng "viết một lần, chạy nhiều lần". Bytecode file sau khi được tạo có thể chạy ở bất kì máy nào có JRE (Java Runtime Environment) với bất kì hệ điều hành.



Hình 11: *Platform-Independent trong Java*

5. **Secured**: Java là ngôn ngữ được thiết kế để tối ưu hóa việc bảo mật. Java hỗ trợ một lượng lớn các API, công cụ bảo mật với độ tin cậy cao. Bên cạnh đó, chương trình Java luôn chạy trong JRE và gần như không tương tác với hệ điều hành bên ngoài, nhờ vậy giảm khả năng tiếp xúc với virus.

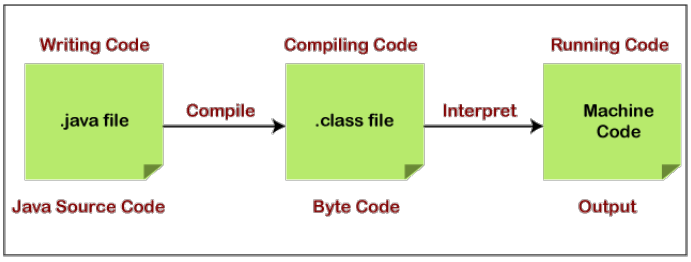
6. **High Performance**: Mặc dù là ngôn ngữ thông dịch và tốc độ cho kết quả không bằng với C/C++, Java với cơ chế Just-In-Time compiler (trình biên dịch runtime, nghĩa là biên dịch code trong khi chương trình đang thực thi) phần nào giảm thiểu sự chênh lệch đó.

7. **Multithreading**: Ta có thể lập trình đa luồng trên Java, tận dụng nguồn tài nguyên sẵn có để giúp chương trình chạy nhiều tác vụ một cách đồng thời.

8. **Architectural Neutral**: việc tổng hợp các bytecode của trình biên dịch không bị lệ thuộc bởi kiến trúc máy tính, do đó mà Java có thể dễ dàng được chạy trên nhiều loại máy tính có kiến trúc phần cứng lẫn phần mềm khác nhau.

9. **Object-Oriented**: Đây là một điểm hết sức quan trọng của Java. Trong ngôn ngữ này, hầu hết mọi thứ đều là đối tượng thuộc một class nào đó (ngoại trừ một số ít kiểu primitive), bên trong chứa các thuộc tính và phương thức. Java hỗ trợ đầy đủ từ khái niệm cơ bản đến các nguyên tắc của lập trình hướng đối tượng: Tính bao đóng, Tính thừa kế, Tính trừu tượng, Tính đa hình.

10. **Interpreted**: Trình thông dịch Java là một phần mềm dùng để thực thi JVM, đọc Java bytecode và chạy chương trình. Tuy nhiên, tốc độ xử lý của Java cao hơn các ngôn ngữ thông dịch khác (ví dụ như Python) bởi có JIT compiler giúp biên dịch bytecode sang mã máy. Nói cách khác, thay vì phải thông dịch từng dòng rồi chạy lần lượt, JIT biên dịch cả chương trình trong runtime.



Hình 12: *Quá trình chạy Java program.*

11. **Distributed**: Chương trình Java có thể được dễ dàng truyền từ máy này sang máy khác bằng Java bytecode. Do đó, Java khá phù hợp với lập trình trên internet, nơi mà các người dùng có thể download các chương trình Java về máy của mình để chạy mà không lo dính lỗi.

12. **Dynamic**: với tính chất thừa kế và trừu tượng của lập trình hướng đối tượng, ta có thể tái sử dụng những đoạn code đã được lập trình sẵn, cũng như sử dụng các thư viện hàm của Java.

2.1.3 Một số hạn chế của Java

1. **Hiệu năng thấp**: Java có tốc độ chậm hơn so với các ngôn ngữ biên dịch, còn Java thì hoạt động theo cơ chế biên dịch qua bytecode rồi thông dịch sang mã máy khi cần.

2. **Chi phí**: Bởi Java cần bộ xử lý và bộ nhớ mạnh, khi lập trình một ứng dụng phức tạp, người ta thường chuyển sang các ngôn ngữ khác tiện lợi hơn.

3. **Giao diện không bắt mắt**: Mặc dù có các GUIs và thư viện hỗ trợ như Swing, JavaFX, SWT, giao diện của Java nhìn khá lỗi thời và phải cần nghiên cứu lâu mới có thể thành thục một công cụ UI trong Java.

4. **Không có cơ chế backup**: Java chủ yếu làm việc trên kho lưu trữ và không quan tâm đến việc backup dữ liệu ở một nơi nào đó. Đây có thể xem là điểm yếu lớn nhất của Java.

5. **Tốn nhiều không gian bộ nhớ**: Do phải chạy trên máy ảo, Java sử dụng lượng bộ nhớ khá lớn so với C và C++. Trong quá trình Garbage Collection, hiệu năng của bộ nhớ sẽ bị ảnh hưởng.

6. **Câu lệnh phức tạp**: Tuy đã tinh giản và dễ hiểu hơn C/C++, Java còn khó hiểu và cấu trúc câu lệnh còn khá dài dòng so với một số ngôn ngữ lập trình được ưa chuộng hiện nay (ví dụ như tác vụ in trên console, trong khi Python chỉ cần lệnh `print()` thì ở Java ta phải gõ `System.out.println()`)

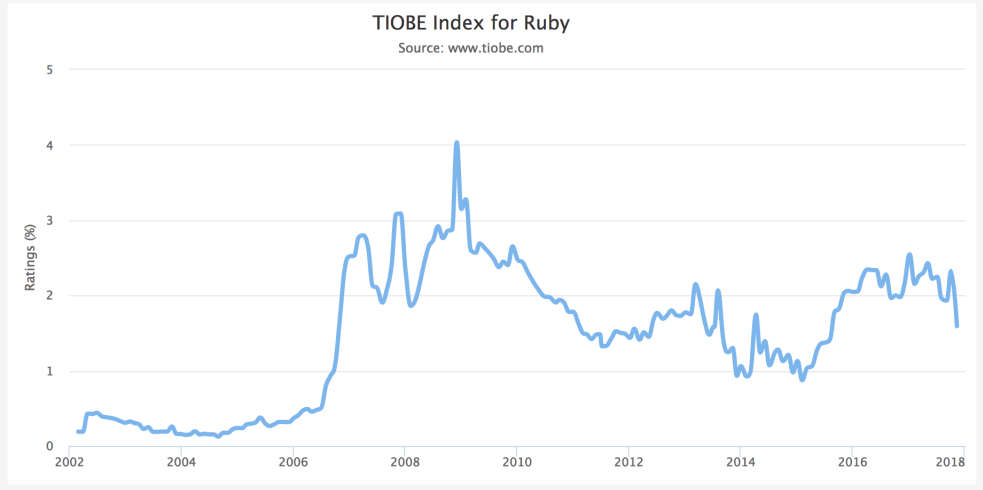
2.2 Ngôn ngữ lập trình Ruby

Ruby là một ngôn ngữ lập trình hướng đối tượng (OOP) được phát triển bởi *Yukihiro "Maz" Matsumoto* tại Nhật Bản vào giữa thập niên 90. Với chủ trương thiết kế một ngôn ngữ giúp lập trình viên có thể tăng hiệu suất công việc và sự hứng thú, Ruby nhấn mạnh tầm quan trọng của một phần mềm phải được tường mình dưới góc độ con người trước, sau đó mới là máy tính.

Hình 13: *Logo Ruby*

2.2.1 Lịch sử phát triển của Ruby

- *Nguyên nhân ra đời*: Matsutomo cho rằng Ruby ra đời vào năm 1993. Trước đó, ông và các cộng sự đã bàn về khả năng một ngôn ngữ thông dịch hướng đối tượng có thể nào được ra đời hay không, khi mà các ngôn ngữ lúc bấy giờ như Perl và Python chưa thật sự thuần "OOP". Với cương vị là tín đồ của lập trình hướng đối tượng trong suốt 15 năm, ông quyết định viết một ngôn ngữ vừa thuần OOP, vừa đơn giản cho người dùng.

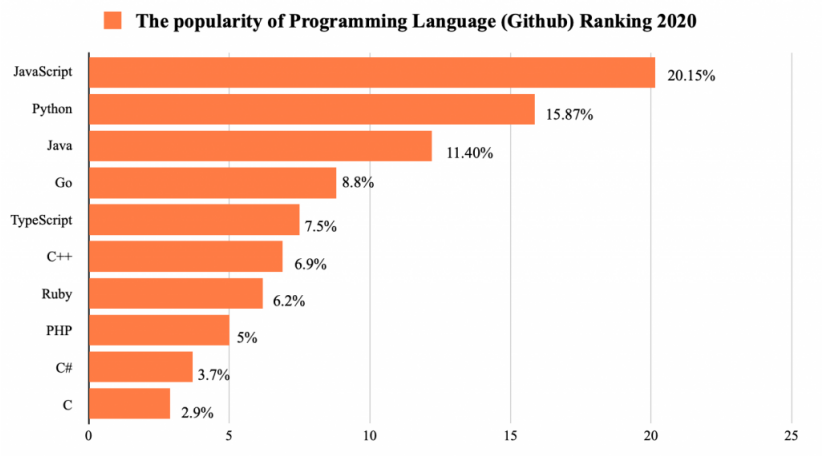


Hình 14: *Chỉ số phổ biến của Ruby theo các năm*

- *Lược sử tên gọi*: Cái tên "Ruby" bắt nguồn từ đoạn trò chuyện của Matsumoto và Keiju Ishitsuka vào đầu năm 1993, khi mà ông chưa viết dòng code nào. Ban đầu đã có hai cái tên được đặt ra: "Coral" và "Ruby", nhưng ông đã chọn cái thứ hai, đơn giản chỉ vì đó là birthstone (loại đá quý theo thang sinh) của đồng nghiệp ông.
- *Nguyên tắc*: Ngôn ngữ Java được tạo ra với nguyên tắc "conciseness, consistency, flexibility".

  - **Conciseness**: Ruby được thiết kế để tối thiểu công sức người dùng khi chuyển từ mã giả sang mã nguồn chạy được.
  - **Consistency**: Ruby chỉ sử dụng một nhóm quy tắc nhỏ để lập trình và trực quan với người dùng.
  - **Flexibility**: có nhiều cách code để đạt đến mục tiêu cuối, đảm bảo tính linh hoạt trong coding.

- *Ứng dụng*: Ruby dần dần chiếm được vị thế trong lĩnh vực phát triển các ứng dụng trên web. Ruby on Rails, một trong những framework nổi tiếng của Ruby, đã mang số lượng người dùng khổng lồ cho ngôn ngữ này. Hiện tại, Ruby có một cộng đồng người sử dụng rộng lớn trên toàn thế giới.



Hình 15: *Ruby nằm trong top 10 ngôn ngữ năm 2020. Nguồn: Github*

2.2.2 Một số đặc điểm và thế mạnh nổi bật của Ruby

1. **Open source**: Ruby hỗ trợ Ruby License, cung cấp mã nguồn của Ruby miễn phí trên GitHub. Đồng thời người dùng có thể đóng góp xây dựng các version sau.
2. **Interpreted**: Ruby là một ngôn ngữ thông dịch. Do đó, mã nguồn không cần comple vẫn có thể chạy ngay.
3. **Multi-platform**: Ruby có thể chạy được trên 3 hệ điều hành lớn là Windows, Linux và MacOS.
4. **Dynamic + duck typing**: Ruby là ngôn ngữ dynamic, tức là kiểu dữ liệu các biến chỉ được xét tại runtime. Các biến có thể thay đổi kiểu trong runtime, ví dụ nhưng biến a đang ở kiểu `string` có thể chuyển sang kiểu `integer` mà compiler không báo lỗi. Nhờ vậy, chương trình sẽ chạy nhanh hơn và việc coding cũng dễ dàng.

```ruby
class Cat
    def make_noise:
        puts "meow"
    end
end

class Dog
    def make_noise:
        puts "grrr"
    end
end

dog = Dog.new
cat = cat.new

animals = [dog,cat]

animals[0].make_noise # "grrr"
```

```
19 anumals[1].make_noise # "meow"
```

Listing 12: Duck Typing in Ruby

5. **Garbage Collector**: Cơ chế dọn bộ nhớ của object khi không cần thiết cùng với các giải thuật thông minh được Ruby sử dụng để tối ưu hóa vùng nhớ.
6. **Object-Oriented**: Ruby là một ngôn ngữ lập trình hướng đối tượng thuần túy, hỗ trợ các khái niệm của OOP như class, thừa kế, trừu tượng, đa hình và đóng gói. Các đặc trưng của OOP giúp việc bảo trì và phát triển code dễ dàng hơn.
7. **Embedding**: Ruby là một trong số các ngôn ngữ có thể nhúng vào HTML (eRuby), tương tự các ngôn ngữ script server-side như PHP, có ích trong triển khai mô hình MVC (Model-View-Controller).
8. **Applicable**: Ruby có thể dùng để lập trình CGI script (Common Gateway Interface, giao thức tương tác giữa web server và client), xây dựng website và các ứng dụng intranet.

```
  <table class="table">
    <thead>
      <tr>
        <td>#</td>
        <td>Title</td>
        <td>Description</td>
      </tr>
    </thead>
    <tbody>
      <% @groups.each do |group| %>
        <tr>
          <td>#</td>
          <td><%= link_to(group.title, group_path(group)) %></td>
          <td><%= group.description %></td>
          <td>
            <%= link_to("Edit", edit_group_path(group), class: "btn btn-sm btn-default") %>
            <%= link_to("Delete", group_path(group),    class: "btn btn-sm btn-default",
                        method: :delete, data: { confirm: "Are you sure?" } ) %>
          </td>
        </tr>
      <% end %>
    </tbody>
  </table>
</div>
```

Hình 16: *eRuby và HTML*

9. **Supportive**: Ruby hỗ trợ các công cụ GUI khác nhau như OpenGL, Tcl/Tk, GTK,... Bên cạnh đó, ngôn ngữ này cũng hỗ trợ tương tác kết nối với một số database như MySQL, DB2 và kiến trúc serverless như AWS Lambda.

Hình 17: *Xây dựng AWS Lambda bằng Ruby và Serverless Framework*

2.2.3 Một số hạn chế của Ruby

1. **Popularity**: Ruby vẫn là một ngôn ngữ chưa có độ phổ biến cao, dù vẫn nằm trong top 10. Do đó, Ruby hay Ruby on Rails thường không phải là lựa chọn đầu tiên của các lập trình viên. Thay vào đó, ta dễ tìm một người thạo ngôn ngữ Python hay Java hơn.

2. **Slow runtime and Scalability**: Đây là điểm yếu lớn nhất của Ruby. Vì là ngôn ngữ dùng trình thông dịch, Ruby chậm hơn các ngôn ngữ biên dịch như C/C++ và ngay cả Java. Do đó khi mở rộng đến lượng người dùng lớn hơn, vấn đề từ server hay database sẽ cản trở hiệu suất của chương trình. Còn xét nền tảng framework Ruby on Rails, nó chậm hơn các backend framework khác như NodeJS hay Go.

3. **Flexibility**: bản chất của Ruby là xây dựng ứng dụng web, vì vậy nếu sản phẩm được ứng dụng cho những tác vụ khác thì việc điều chỉnh có phần phức tạp.

4. **Development**: so với các ngôn ngữ lập trình khác, Ruby tung bản cập nhật ít và chậm hơn. Ví dụ như PHP, dù ra đời trễ hơn 2 năm, hiện có phiên bản 8.1.4 (17/03/2022) trong khi Ruby vẫn ở bản 3.1.1 (18/02/2022).

2.3 So sánh ngôn ngữ Java và Ruby

2.3.1 Điểm tương đồng của Java và Ruby

1. **Garbage Collection**: Khác với C/C++ phải quản lý bộ nhớ một cách thủ công, cả Java và Ruby (và một số ngôn ngữ cấp cao khác) đều hỗ trợ cơ chế dọn bộ nhớ tự động. Mặc dù mỗi ngôn ngữ có cơ chế xử lý vùng nhớ khác nhau, chúng đều bảo đảm an toàn cho vùng nhớ (không bao giờ xảy ra trường hợp xóa vùng nhớ có thể được sử dụng) cũng như tối ưu phần bộ nhớ trống.

2. **Multithreading**: Cả hai ngôn ngữ đều có các thư viện hỗ trợ lập trình đa luồng, xử lí song song các tác vụ. Trong Java, multithreading được thể hiện rõ bằng việc tận dụng tất cả nguồn tài nguyên CPU. Dù Ruby không tận dụng tốt như vậy, nó vẫn hỗ trợ chạy chương trình theo từng thread.

3. **Security**: Java và Ruby được hầu hết các lập trình viên công nhận là có tính bảo mật cao. Không có khái niệm con trỏ, người dùng bên ngoài không thể nào tùy tiện xâm nhập vùng nhớ được khai báo `private`. Ngoài ra, cả hai đều có các thư viện hỗ trợ việc bảo mật rất tốt.

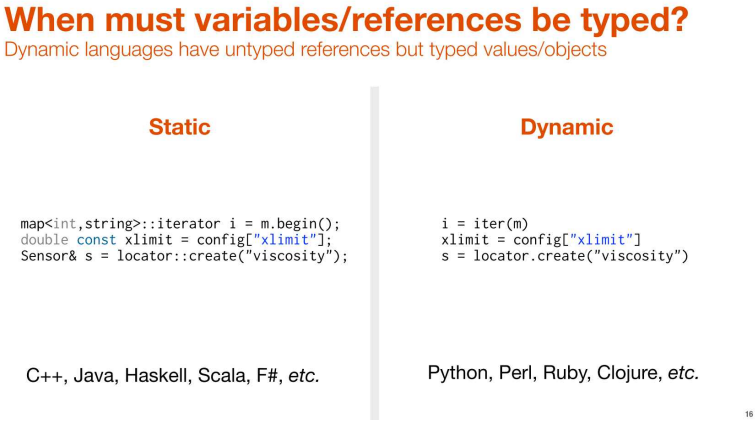
2.3.2 Điểm khác biệt của Java và Ruby

1. **Interpreted hay compiled**: Đối với thông dịch, ta thực thi code theo nguyên tắc "đọc dòng nào, chạy dòng đó", còn biên dịch thì cần phải dịch toàn bộ code sang mã máy trước khi chạy. Như đã nói, Ruby là một ngôn ngữ thông dịch. Còn Java, ngôn ngữ này vừa thực hiện thông dịch, vừa thực hiện biên dịch với cơ chế biên dịch JIT (Just-In-Time).

2. **Dynamic hay static typing**: Một sự khác biệt rõ ràng nữa là cách mà Java và Ruby quy định kiểu của một biến trong chương trình. Java sử dụng *static typing* (kiểu của biến đã được compile từ trước và không thể thay đổi trong tầm vực của nó trong runtime) còn Ruby sử dụng *dynamic typing* (không cần phải định nghĩa kiểu biến, và kiểu của một biến có thể thay đổi nhiều lần trong suốt chương trình). Trong khi Static Typing đảm bảo tính ràng buộc và sự tường minh trong code, Dynamic Typing cho phép người dùng linh hoạt thay đổi, nhờ đó tiết kiệm thời gian phát triển sản phẩm.

3. **Pure object-oriented hay hybrid object-oriented**: Ruby là một ngôn ngữ lập trình "thuần" hướng đối tượng bởi vì tất cả mọi thứ đều là đối tượng, bên trong chứ dữ liệu và phương thức. Còn Java thì không như vậy bởi có tám kiểu nguyên thể (primitive) cơ bản: byte, short, int, long, float, double, char, và boolean, còn lại thì đều là đối tượng và thừa kế từ lớp Object.

4. **Method và Operator overloading**: Đây là một đặc trưng quan trọng của OOP và chỉ rằng các thuộc tính cùng tên có thể có danh sách tham số khác nhau. Trong khi Java có phép nạp chồng phương thức dưới dạng generics, Ruby không được thừa hưởng điều này (vì không có kiểu biến tường minh cho Ruby). Còn về vấn đề nạp chồng toán tử, Ruby có hỗ trợ còn Java thì không.



Hình 18: *Static và Dynamic Typing*

2.3.3 Lựa chọn Java và Ruby

Java thường được sử dụng trong các sản phẩm mang tính doanh nghiệp, nơi mà yêu cầu lưu lượng thông tin người dùng lớn. Trong khi đó, Ruby thích hợp với phát triển ứng dụng web và cần tạo sản phẩm nhanh chóng đến tay người dùng.

Bên cạnh đó, có một số yếu tố đặc thù từng dự án mà ta cần phải cân nhắc để chọn ngôn ngữ lập trình phù hợp:

- **Java**:
  - Kiến trúc ứng dụng phức tạp.
  - Thường xuyên tương tác với các hệ thống backend, database,...
  - Không bị giới hạn về tài nguyên và thời gian.
- **Ruby**:
  - Đặt giá trị cung cấp cho khách hàng lên hàng đầu.
  - Chưa có một khái niệm cố định cho sản phẩm, thường xuyên phải chỉnh sửa ứng dụng và cần tạo bản mẫu nhanh trong thời gian nhất định.
  - Xây dựng ứng dụng vừa và nhỏ.

Tiêu chí	JAVA	RUBY
Interpreter/ Compiler	Ngôn ngữ lập trình biên dịch và thông dịch	Ngôn ngữ lập trình thông dịch
OOP	Tích hợp OOP và một số kiểu primitive	Thuần hướng đối tượng
Hiệu năng	Hiệu năng tốt cho tác vụ lớn	Hiệu năng tốt cho tác vụ vừa và nhỏ
Code syntax	Phức tạp nhưng các câu lệnh thông dụng	Đơn giản với một số cú pháp tắt riêng biệt
Kiểu biến	Static typing	Dynamic typing
Phạm vi truy xuất	Có phạm vi truy xuất cho thuộc tính và phương thức để hạn chế chỉnh sửa thông tin trực tiếp	Không có phạm vi truy xuất cho dữ liệu class (luôn private), còn các phương thức thì có
Framework nổi tiếng	Spring Boot, JSF, GWT,..	Ruby on Rails, Sinatra, Hanami,...
Ứng dụng	Uber, Airbnb, Netflix, Google,..	Github, Twitter, Shopify,…

Hình 19: *So sánh tính năng của Java và Ruby*

```java
//---------------JAVA--------------
// Encapsulation
public class Student{
    private double score;

    public double getScore(){
        return this.score;
    }
    public void setScore(double score){
        this.score = score;
    }
}

// Inheritance
public class Person{
    String name;
    Person(String name){
        this.name = name;
    }
}
public class Student extends Person{
    private double score;
    super(name);
    public double getScore(){
        return this.score;
    }
    public void setScore(double score){
        this.score = score;
    }
}

// Abstraction
public abstract class Shape{
    private String name;
    public abstract double getArea();
}

public class Square implements Shape{
    double a;
    public Square(double a){
        this.a = a;
    }
    @Override
    public double getArea(){
        return this.a * this.a;
    }
}

// Polymorphism
public class Animal{
    public void makeNoise(){
        System.out.println("Animallll??");
    }
}

public class Dog extends Animal{
    public void makeNoise(){
        System.out.println("Grrrrrr");
    }
}

public class Cat extends Animal{
    public void makeNoise(){
        System.out.println("Meoww");
    }
}
```

```ruby
#----------------RUBY-----------------
# Encapsulation
class Student
    def setScore=(score)
        @score = score
    end

    def getScore
        @score
    end
end


# Inheritance
class Person
    def initialize(name)
        @name = name
    end
end
class Student <Person
    def setScore=(score)
        @score = score
    end

    def getScore
        @score
    end

end


#Abstraction
class Shape
    def getArea
        raise NoMethodError
    end
end

class Square <Shape
    def initialize(a)
        @a = a
    end

    def getArea
        puts @a*@a
    end
end

# Polymorphism
class Animal
    def makeNoise
        puts "Animallll??"
    end
end

class Dog <Animal
    def makeNoise
        puts "Grrrrrr"
    end
end

class Cat <Animal
    def makeNoise
        puts "Meoww"
    end
end
```

3 Lập trình Hàm với Haskell và SML

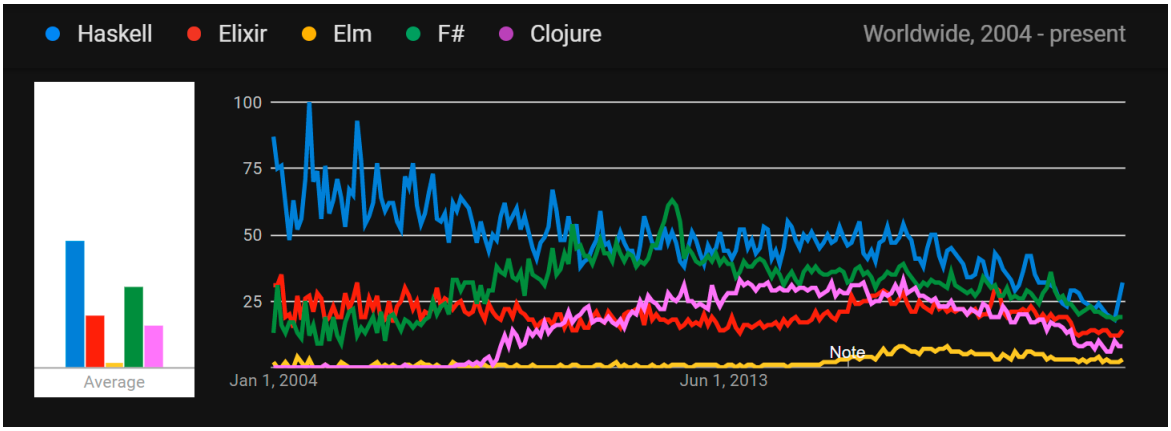
3.1 Ngôn ngữ lập trình Haskell

Haskell là một ngôn ngữ thuần lập trình hàm được sử dụng rộng rãi cho mục tiêu nghiên cứu (cả hướng toán học và lập trình), và hiện tại một số công nghiệp cũng đang sử dụng nó. Haskell lần đầu tiên được đề xuất bởi các nhà nghiên cứu đam mê giải tích lambda và lập trình hàm vào năm 1990, trong bản công bố "Haskell Language Report".

Hình 20: *Logo Haskell*

3.1.1 Lịch sử phát triển của Haskell

- *Nguyên nhân ra đời*: Ngôn ngữ lập trình hàm đầu tiên ra đời vào năm 1950 là LISP, với thiết kế kiểu hàm số nhưng còn khá phức tạp. Đến thập niên 80 của thế kỉ XX, các nhà khoa học bắt tay phát triển các ngôn ngữ lập trình hàm, ví dụ như Standard ML, Miranda,... tuy nhiên chúng không phải là các phần mềm mã nguồn mở. Do đó, một nhóm học thuật đã cùng nhau xây dựng một ngôn ngữ chung có thể đưa ra cộng đồng và thống nhất trong việc nghiên cứu. Vào năm 1990, Haskell ra đời và luôn được ủng hộ và cập nhật bởi cộng đồng học thuật cho đến ngày nay. Hiện nay, Glasgow Haskell Compiler (GHC) được sử dụng để chạy code Haskell.
- *Lược sử tên gọi*: Tên "Haskell" được thống nhất bởi hội đồng công bố bản báo cáo ngôn ngữ này, với nguồn gốc là tên của nhà toán học và logic học nổi tiếng người Hoa Kì *Haskell Curry*.
- *Nguyên tắc*: Haskell được xây dựng dựa trên cơ sở giải tích lambda hiện đại, với nguyên tắc xem tất cả thao tác tính toán là định giá của các hàm số toán học.
- *Ứng dụng*: Haskell hiện nay được dùng rộng rãi trong giảng dạy, nghiên cứu và công nghiệp. Haskell có thể được xem là cơ sở của các ngôn ngữ hybrid sau này như Python, Javascript,..



Hình 21: *Xu hướng phát triển Haskell từ 2004 đến nay, so với các ngôn ngữ FP khác. Nguồn: haskell.org*

3.1.2 Một số đặc điểm và thế mạnh nổi bật của Haskell

1. **Dễ hiểu**: Các hàm trong Haskell dễ hiểu do mang tính chất của hàm số trong toán học. Tuy nhiên, một số thao tác xây dựng hàm khá là khó, cần phải nắm vững nền tảng toán (ví dụ như đệ quy).

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )
```

Listing 13: Calculating factorial in Haskell

2. **Open-source**: Như đã trình bảy ở phần trên, Haskell được xây dựng để tất cả những ai có đam mê về lập trình hàm đều có thể đóng góp và sử dụng mà không cần tốn chi phí. Nhờ vậy, số lượng người dùng Haskell đông đảo hơn và tạo tiềm năng phát triển mạnh hơn.
3. **Quản lý bộ nhớ**: Không giống như C/C++ người dùng phải tự dọn dẹp vùng nhớ heap sau khi khai báo, Haskell có cơ chế Garbage Collector giúp phân bố tối ưu vùng nhớ.
4. **Tái sử dụng**:
   (a) Do các hàm trong Haskell mang tính chất Purity, chúng có thể được dùng lại cho tác vụ khác nhau với cùng mục đích mà không lo bị ảnh hưởng bởi side effect.
   (b) Ta còn có thể lồng ghép các hàm nhỏ lẻ thành hàm phức tạp hơn (từ first-order đến higher-order function).
   (c) Có thể tái sử dụng các hàm trong một module để áp dụng vào một module khác bằng việc `export` các định nghĩa hàm và `import` khi cần.
   (d) `Functor` và `Monad` là hai lớp (lớp dữ liệu chứ không phải lớp bên OOP) được tái sử dụng nhiều nhất.
   - Functor là một kiểu dữ liệu mà có thể định nghĩa được (có thể tồn tại) một hàm `fmap` sao cho nó nhận được một hàm và một Functor, và trả về một Functor khác. List, Map, Tree là ví dụ của Haskell Functor
   - Monad là một cấu trúc dễ diễn tả thứ tự các bước tính toán để xử lý side effect.

```
--Map--
myMap :: (a->b) -> [a] -> [b]
myMap f [] = []
myMap f (x:xs) = (f x):(myMap f xs)

--Filter--
filter1 :: (a->Bool) -> [a] -> [a]
filter1 p [] = []
filter1 p (x:xs)
    | p x = x: filter1 p xs
    | otherwise = filter1 p xs
```

Listing 14: Higher-Order Functions in Haskell

5. **Static Typing**:
   (a) Kiểu biến trong Haskell được tường minh ở compile-time, nhờ vậy compiler có thể phát hiện lỗi sai và giảm tính phức tạp khi mở rộng quy mô lập trình. Đồng thời, khi chương trình Haskell đã được biên dịch thành công thì nó không cần phải lo lắng về kiểu của các biến, do đó tăng hiệu năng ở run-time.
   (b) Hệ thống Typing của Haskell rất mạnh, nói cách khác, Haskell có thể bắt được hầu hết lỗi về kiểu biến và đảm bảo rằng trong run-time sẽ không có lỗi này xảy ra. Kiểu của một giá trị độc lập với cách sử dụng biến đó, tức là chúng ta không thể ép kiểu cho một biến kiểu A (đã định sẵn) sang kiểu B (được sử dụng).

```
main :: IO()
num :: Int
num = 'a'
main = return()
```

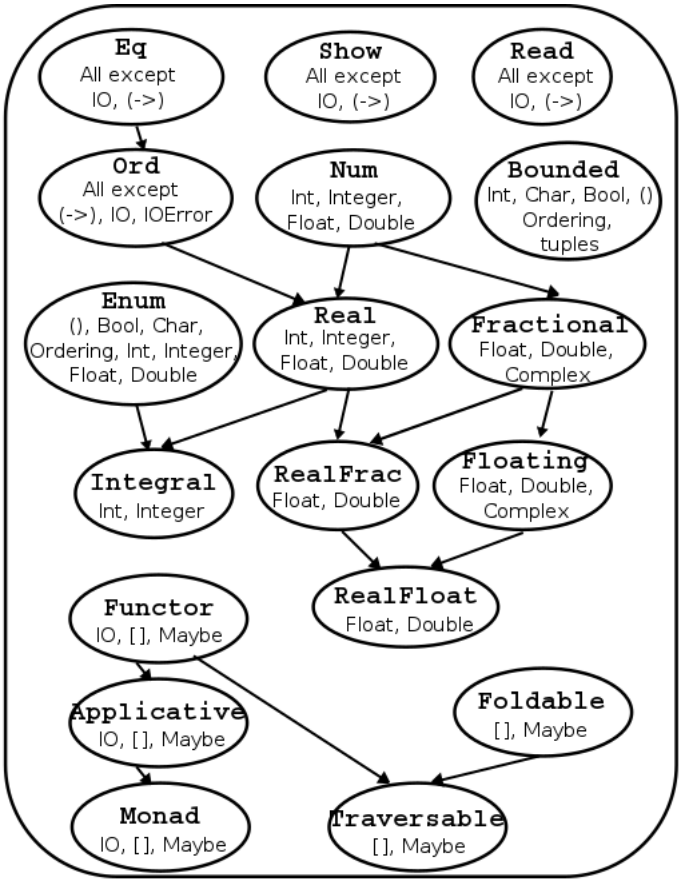
Listing 15: Gán kiểu động trong Haskell

(c) Các kiểu dữ liệu cơ bản: kí tự, boolean, danh sách (list) và dãy dữ liệu (tuple). Bên cạnh đó, người dùng có thể viết các Deriving Typeclasses.

```
data Temperature = C Float | F Float
    deriving (Show, Eq)

instance Eq Temperature where
(==) (C n) (C m) = n == m
(==) (F n) (F m) = n == m
(==) (C c) (F f) = (1.8*c+32) == f
(==) (F f) (C c) = (1.8*c+32) == f
(==) _ _ = False
```

Listing 16: Deriving Typeclasses in Haskell



Hình 22: *Kiểu biến trong Haskell*

6. **Lazy evaluation**: ở chế độ mặc định, Haskell dùng chiến thuật thực thi code mang tên *lazy evaluation*.

   (a) Cụ thể, một biểu thức sẽ không bao giờ được định giá cho đến khi nó buộc phải được dùng định giá một biểu thức lớn hơn khác.

   (b) Sau khi đã được định giá, kết quả sẽ được lưu trữ hoặc xóa bỏ nếu không còn code không còn chạy. Nhờ vậy, chỉ các thao tác thực sự cần thiết mới được thực thi, và giảm khối lượng tính toán.

   (c) Tuy nhiên, cần lưu ý rằng dung lượng bộ nhớ sử dụng không thể đoán trước được cho các biểu thức chưa được định giá. Ví dụ `1+5 :: Int` và `6 :: Int` đều chỉ giá trị là 6, lượng bộ nhớ sử dụng của hai biểu thức là khác nhau.

```
areaReg a b =
    a * b

```

```
Prelude> (x, y) = (2, 3)
Prelude> areaReg (x+1) (y+1)
5.0
```

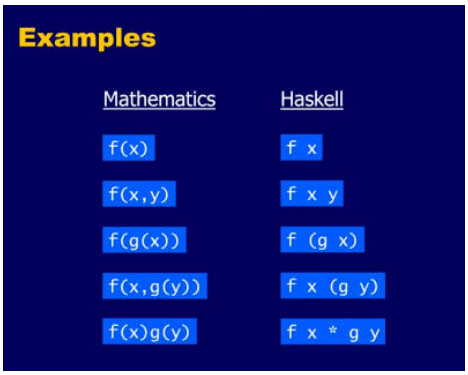
Listing 17: Lazy Evaluation in Haskell

Ở đoạn code trên, tham số (x, y) được định giá sau khi nó cần phải truyền vào hàm, cụ thể là sau khi toán tử * trong hàm được gọi.

7. **Thuần lập trình hàm**: tất cả những thao tác tính toán được quy về đơn vị là các hàm số trong toán học (first-order), nhờ đó mà sẽ không có side effect (một đầu vào chỉ có thể cho ra một kết quả mà không bị tác động bên ngoài). Nhờ vậy, người dùng dễ phát hiện lỗi sai trong code, cũng như tạo tính thống nhất cho hàm.

```
main :: IO()
num :: Int
num = 3
num = 4 -- Multiple Declaration of 'num'
main = return()
```

Listing 18: Biến chỉ được gán một lần trong Haskell



Hình 23: *Sự tương đồng các biểu thức toán học và hàm trong Haskell*

8. **Lập trình đa luồng**:

   (a) Việc đánh giá các pure functions không phụ thuộc vào thứ tự hoàn tất, nhờ vậy ta có thể tối ưu hiệu năng lập trình bằng xử lý đa lường mà cũng không lo mắc phải các vấn đề về concurrency.

   (b) Trên thực tế, có khá nhiều thư viện của Haskell tận dụng được điều này và lập trình các đoạn code của họ có khả năng tính năng song song hoặc tận dụng GPU.

   (c) Haskell sử dụng Haskell threads được thực thi trong runtime, và được chứng minh là hiệu quả hơn về không gian và thời gian so với các OS threads.

9. **Type Inference**:

   (a) Đây là một cách thực thi cho phương pháp *Implicit Typing*, tức thông tin về kiểu biến không được cung cấp trong mã nguồn. Với Type Inference, các kiểu biến được cấu trúc dựa trên cú pháp của giá trị của nó.

```
main :: IO()
num = 3 -- Haskell knows it is a number
char = 'a' -- Haskell knows it is a character
main = return()
```

Listing 19: Gán kiểu động trong Haskell

   (b) Haskell là ngôn ngữ có hỗ trợ tính đa hình cho các kiểu biến (cụ thể là Hinder-Milner (HM) type inference). Quy tắc trong HM bảo đảm rằng không có sai sót xảy ra trong việc suy diễn kiểu.

(c) Giả sử ta có một list các phần tử, thì Haskell không cần ta phải chỉ rõ kiểu từng biến `string` hay `int`,.. tương tự với khái niệm Generic trong Java. Điều này cho phép ta xây dựng các hàm còn phức tạp hơn, bởi nó có thể áp dụng cho nhiều kiểu biến khác nhau.

```
mymap f [] = []
mymap f (first:rest) = f first : mymap f rest

Prelude> :t mymap
mymap :: (t -> a) -> [t] -> [a]
```

Listing 20: Type Inference Example in Haskell

$$\frac{\tau \prec \Gamma(x)}{\Gamma \vdash_{\mathrm{HM}} x : \tau} \quad [\textsc{Var}]_{\mathrm{HM}}$$

$$\frac{\Gamma \vdash_{\mathrm{HM}} e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_{\mathrm{HM}} e_2 : \tau_1}{\Gamma \vdash_{\mathrm{HM}} e_1\ e_2 : \tau_2} \quad [\textsc{App}]_{\mathrm{HM}}$$

$$\frac{\Gamma \backslash x \cup \{x : \tau_1\} \vdash_{\mathrm{HM}} e : \tau_2}{\Gamma \vdash_{\mathrm{HM}} \lambda x \to e : \tau_1 \to \tau_2} \quad [\textsc{Abs}]_{\mathrm{HM}}$$

$$\frac{\Gamma \vdash_{\mathrm{HM}} e_1 : \tau_1 \quad \Gamma \backslash x \cup \{x : \mathit{generalize}(\Gamma, \tau_1)\} \vdash_{\mathrm{HM}} e_2 : \tau_2}{\Gamma \vdash_{\mathrm{HM}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad [\textsc{Let}]_{\mathrm{HM}}$$

Hình 24: *Luật suy diễn kiểu trong Haskell*

3.1.3 Một số hạn chế của Haskell

1. **Tính phức tạp**: cộng đồng xây dựng là các nhà toán học, do đó code với các kí hiệu quá trừu tượng đối với dân không chuyên hoặc nhập môn. Lập trình viên khi học Haskell phải có kiến thức vững về lập trình, ngay cả là lập trình hướng đối tượng.

2. **Thời gian**: Do kí hiệu quá phức tạp cùng với tài liệu học tập khan hiếm, việc tiếp thu và tường tận ngôn ngữ là một thách thức lớn. Đi với việc học thì việc hiện thực cũng lâu, vì người dùng phải nắm rõ các quy tắc toán học của một hàm trước khi code lại bằng Haskell.

3. **Tính rộng rãi**: Chưa có quá nhiều thư viện hỗ trợ Haskell hay các documentation hướng dẫn, và cộng đồng người dùng ngôn ngữ này ở thời điểm hiện tại là còn thấp so với các ngôn ngữ lập trình hướng đối tượng như C++/ Java. Do vậy, người dùng thường cảm thấy nản chí khi học ngôn ngữ này.

4. **Debug**: Cơ chế Lazy Evaluation đôi khi bỏ qua một số đoạn code, do đó việc debug với ngôn ngữ này khá khó khắn. Không có Stack traces cho Haskell, việc dò tìm lỗi sai càng khó khăn hơn.

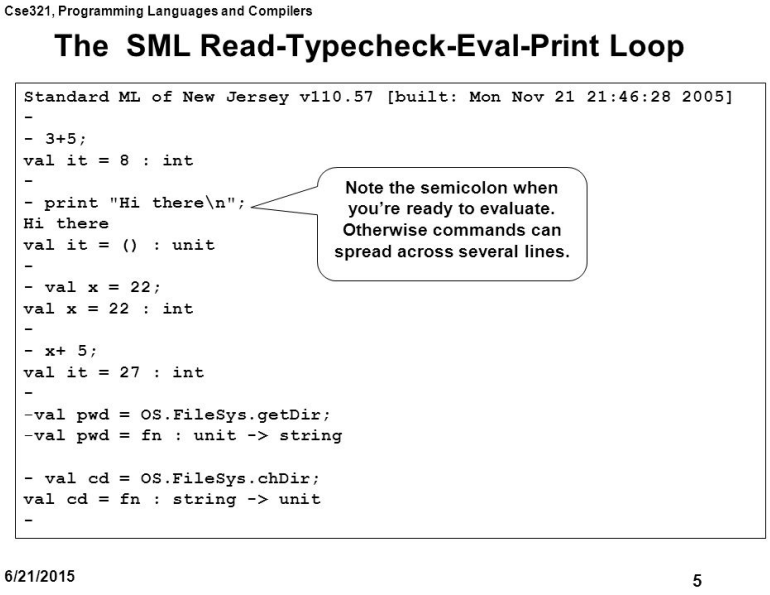
5. **Không có Side-Effect**: Bởi Haskell là ngôn ngữ thuần lập trình hàm, ta không thể dùng side-effect như I/O, biến static, global,.. Người dùng buộc phải sử dụng Monad, từ đó mất thêm thời gian để thiết kế.

3.2 Ngôn ngữ lập trình Standard ML

Hình 25: *Logo SML/NJ, một compiler và IDE của Standard ML*

3.2.1 Lịch sử phát triển của Standard ML

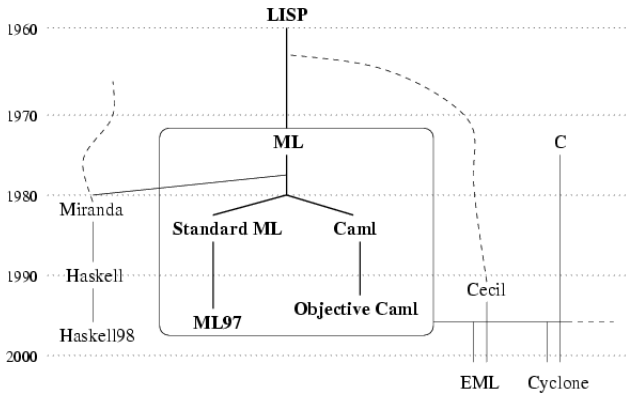
- *Nguyên nhân ra đời*: Standard ML là một trong các ngôn ngữ lập trình họ ML. Nguyên thể, ML bắt nguồn từ Meta Language của định lý LCF phát triển bởi Robin Milner và cộng sự vào thập niên 70, với mục đích tạo một ngôn ngữ script có logic chặt chẽ. Tuy nhiên, Standard ML (ra đời vào 1983) lại nổi trội hơn hẳn so với các anh em của mình, bởi tính lập trình hàm nghiêm ngặt và static-typing. Sau đó, vào năm 1986, từ một dự án xây dựng front-end của SML với mục đích nghiên cứu, SML/NJ (SML of New Jersey) được phát triển bởi Đại học Princeton và phòng thí nghiệm Bell ra đời là một IDE và compiler chính thức đầu tiên mã nguồn mở cho ngôn ngữ SML.



Hình 26: *Giao diện SML/NJ*

- *Lược sử tên gọi*: Do các ngôn ngữ trong họ ML khá đa dạng, Standard ML được đặt tên như vậy với mục đích tạo ra một ngôn ngữ làm tiêu chuẩn giữa các nhà nghiên cứu (hợp nhất hai trường phái lớn trong họ ML, HOPE và LCF/ML).
- *Nguyên tắc*: Standard ML nhấn mạnh tính an toàn trong lập trình, bao gồm tính chặt chẽ trong suy diễn kiểu và quản lý bộ nhớ. Đồng thời, SML có các thư viện và cơ chế compiler, I/0 phức tạp, nhằm tối ưu hiệu suất khi mở rộng quy mô ứng dụng.
- *Ứng dụng*: Standard ML đang được giảng dạy tại các trường đại học nổi tiếng trên thế với để tìm hiểu lập trình hàm. Đây cũng là một công cụ trong việc nghiên cứu chứng minh định lý và phân

tích chương trình. Một ví dụ thực tế là tại đại học Copenhagen đang sử dụng SML làm hệ thống quản lý sinh viên trực tuyến.



Hình 27: *Lịch sử của Standard ML so với các ngôn ngữ cùng thời*

3.2.2 Một số đặc điểm và thế mạnh nổi bật của Standard ML

1. **Mostly functional**: SML dựa trên mô hình đánh giá các hàm là đơn vị tương tác nhỏ nhất (first-order) và từ các hàm đơn lẻ ta có thể xây dựng một hàm phức tạp hơn (higher-order). Hầu như các biến và cấu trúc dữ liệu trong ngôn ngữ đều không thể thay đổi sau khi khởi gán. Ngoài ra, khác với ngôn ngữ thuần lập trình hàm như Haskell, SML cho phép side-effect xảy ra.
2. **Strict**: tất cả các tham số đều được định giá trước khi đi vào thân hàm, giúp ta phát hiện dễ phát hiện lỗi sai và kiểm soát sự tiêu tốn bộ nhớ của toàn bộ dữ liệu.
3. **Modular**: SML hỗ trợ các mô-đun và bản thiết kế (tương tự như khái niệm Class và Interface trong Java, ngoại trừ việc tất cả các thuộc tính đều có thể tự do truy cập từ bên ngoài), tạo ra các cấu trúc linh hoạt và phức tạp. Mỗi mô-đun có một hay nhiều bản thiết kế khác nhau. Ngoài ra, *functor* (một mô-đun nhận bản thiết kế của mô-đun khác là tham số) được kiểm tra trong compile-time, đảm bảo tính mô-đun hóa tốt hơn.

```
// Signature
signature DICTIONARY =
sig
    exception E;
    type (''key, 'value) dict
    val empty: (''key, 'value) dict
    val insert: (''key, 'value) dict -> (''key, 'value) dict -> (''key, 'value) dict
end

// Structure
structure Dict :> DICTIONARY =
struct
    type (''key, 'value) dict = (''key, 'value) list

    val empty = []

    fun insert (key, value) [] = [(key,value)]
      | insert (key, value) ((old_key, old_value)::dict) = if key = key2
        then (key,value) :: dict
        else (old_key, old_value) :: insert (key, value) dict
end
```

Listing 21: Structure and Interface in Standard ML

4. **Strong and Static Typing**: Mỗi câu lệnh trong SML được gán một kiểu và kiểm tra tính hợp lệ trong compile-time để đảm bảo rằng chỉ những thao tác thích hợp được cho phép. Quá trình này hỗ trợ việc dò lỗi dễ dàng và theo dõi sự cập nhật code tốt hơn.
5. **Complex and Abstract Types**: Bên cạnh các kiểu dữ liệu cơ bản như số nguyên, số thực, chuỗi, tuple, list,... ta có thể định nghĩa các đối tượng dữ liệu phức tạp không có sẵn và che giấu cách hiện thực bên trong.

6. **Type inference and consistency**: ML không yêu cầu lập trình viên ghi rõ các kiểu của biến và tham số hàm, vì compiler có thể được suy diễn kiểu. Người dùng chỉ cần khai báo `var` để thể hiện tên biến cần khai báo. Tuy nhiên cần lưu ý rằng, SML không hỗ trợ chuyển kiểu dữ liệu, ngay cả khi đó là phép tính cộng giữa số nguyên và số thực.

```
// Type Inference
fun f(a, b, c, d) =
 if a = b then c + 1 else
 if a > b then c else b + d;
// -> val f = fn : int * int * int * int -> int

// Type Consistency
3.4 + 5
// -> operator and operand don't agree (tycon mismatch): real*int
```

Listing 22: Type inference and consistency in Standard ML

7. **Polymorphism**: Mỗi hàm đều hoạt động theo cơ chế đa hình các kiểu, do đó nó có thể hoạt động với nhiều kiểu dữ liệu khác nhau nếu như các toán tử trong hàm hợp lệ với kiểu dữ liệu đó. Ví dụ, ta có thể sử dụng một hàm đảo ngược list mà không cần quan tâm đến kiểu dữ liệu. Ngoài ra, SML cũng hỗ trợ đa hình theo kiểu biến, chẳng hạn như một biến kiểu list có thể chứa list dạng số nguyên hay list dạng string.

```
fun f(a) = a;
// -> val f = fn : 'a -> 'a
f(10);
// -> val it = 10 : int
f(10.0);
// -> val it = 10.0 : real
f("hello");
// -> val it = "hello" : string
```

Listing 23: Polymorphism in Standard ML

8. **Exception Handler**: Standard ML có cơ chế xử lý ngoại lệ dễ dàng và đa dạng. Ta có thể ném ngoại lệ với bất kì giá trị nào, kể cả hàm.

```
case methodA(problem) of
      Success => Int.toString s
    | Failure => (case methodB(problem) of
                    Success s  => Int.toString s
                  | Failure    => "A and B failed to find"
                  | Impossible => "No existed solution")
    | Impossible => "No existed solution"
```

Listing 24: Exception Handler in Standard ML

3.2.3 Một số hạn chế của Standard ML

1. **Kiểu dữ liệu**: Như đã nói, tất cả kiểu dữ liệu trong SML không được chuyển đổi tự động, kể cả là từ kiểu số nguyên sang số thực. Điều này sẽ gây khó chịu và phức tạp cho người dùng vì ta cần phải nhiều thao tác hơn cho cùng một tác vụ so với các ngôn ngữ khác. Chưa kể, kiểu dữ liệu được nới lỏng hơn so với các ngôn ngữ như Haskell (Type inference), dễ gây ra lỗi trong run-time.

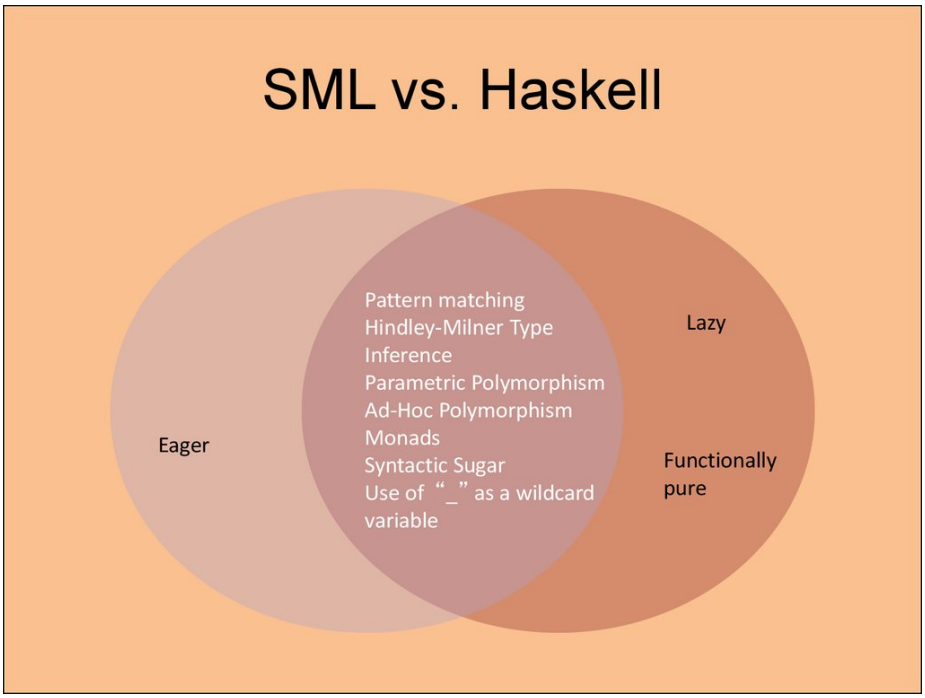
2. **Recompile**: Khác với C hay Java chỉ cần compile một lần, SML không hỗ trợ tốt quá trình biên dịch. Mỗi lần chạy SML khi có cập nhật ta hầu như phải tái biên dịch toàn bộ chương trình, do đó mất nhiều thời gian hơn.

3. **Support**: tài liệu hỗ trợ sử dụng Standard ML khá ít, số người sử dụng ngôn ngữ này cũng ở một con số khiêm tốn (chủ yếu là vì mục đích nghiên cứu). Ngoài ra, tính tương thích với một số hệ điều hành không cao, các thư viện bên thứ 3 hỗ trợ chưa thật sự mạnh mẽ, do đó hầu hết các tính năng phải lập trình từ nền tảng, mất thời gian và chi phí vận hành.

3.3 So sánh ngôn ngữ Haskell và Standard ML

3.3.1 Điểm tương đồng của Haskell và Standard ML

1. **Kiểu dữ liệu có tính đa hình và suy diễn kiểu**:
   - Haskell và SML đều có hệ thống suy diễn kiểu dựa trên định lý toán học Hilney-Milner, do đó khá hoàn chỉnh. Đây cũng là cơ sở để xây dựng các kiểu dữ liệu phức tạp hơn.
   - Một hệ quả sinh ra từ việc suy diễn kiểu tự động là tính linh động. Các hàm hay dữ liệu ở hai ngôn ngữ đều có tính đa hình, tức là một hàm hay kiểu dữ liệu tập hợp (list, tuple, dict,..) có thể sử dụng trên các kiểu dữ liệu cơ bản. Nhờ đó, ta có thể tái sử dụng một hàm cho nhiều loại dữ liệu khác nhau.
   - Tuy nhiên, một hạn chế nảy sinh là việc bảo trì code sẽ khó khăn. Người dùng gặp khó khăn khi debug vì không thể biết một cách tường mình kiểu dữ liệu của một biến, mà phải nhờ sự giúp đỡ của máy tính.
2. **Pattern Matching**: cả hai ngôn ngữ đều cho phép xây dựng hàm dựa trên một khuôn mẫu có sẵn, có thể là với kiểu dữ liệu xác định, hoặc là không (tính đa hình của datatype).
3. **Cú pháp mang tính chất toán học**: do đều được xây dựng là các ngôn ngữ lập trình hàm, các hàm trong Haskell và Standard ML đều khá giống với hàm số trong toán học, dù Haskell có phần nhỉnh hơn về độ tương tự so với SML (bù lại là các syntax phức tạp). Một ví dụ điển hình là cách cả hai ngôn ngữ xây dựng List Comprehension dựa trên thuần quy luật toán học.
4. **Syntactic Sugar và Wildcard**: Cả hai ngôn ngữ đều cố gắng tạo ra các cú pháp giúp cho người dùng dễ đọc hiểu, với thiên hướng là sự kết hợp của ngôn ngữ tự nhiên và toán học (cô đọng, xúc tích). Ngoài ra, wildcard giúp ta bỏ qua một số giá trị ít quan trọng và dễ debug hơn.



Hình 28: *Điểm tương đồng của Haskell và Standard ML*

3.3.2 Điểm khác biệt của Haskell và Standard ML

1. **Strict và Lazy evaluation**:

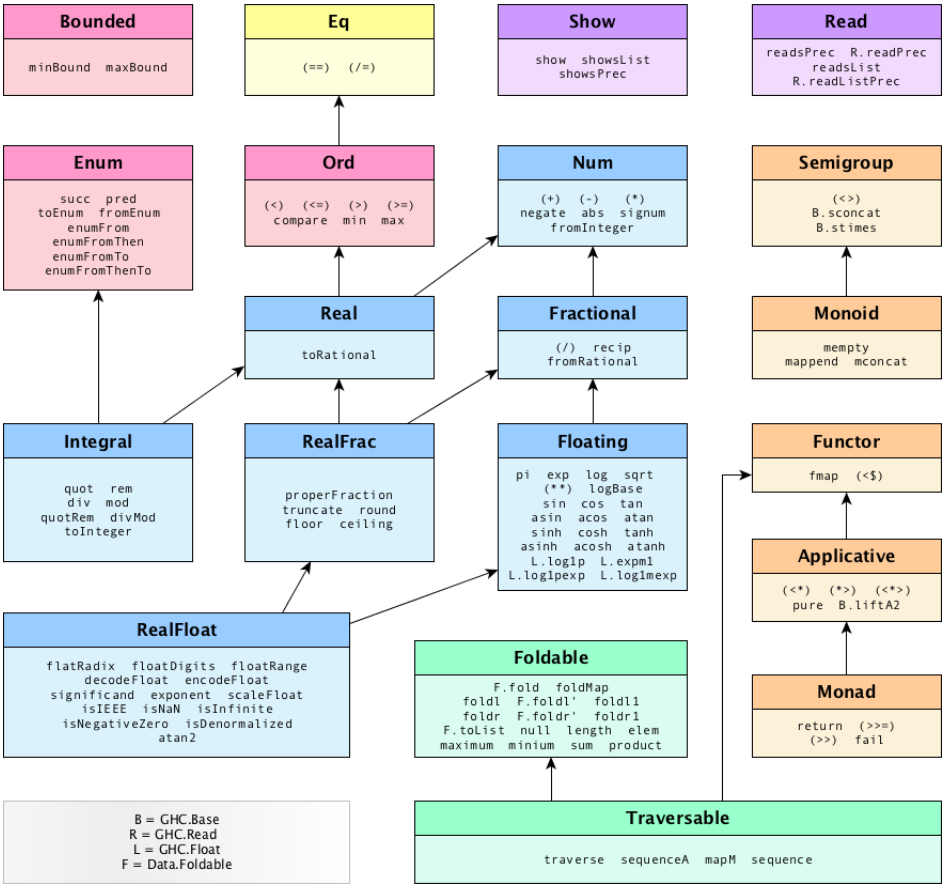
   - *Strict evaluation* là việc các tham số trong một hàm phải được định giá trước khi gọi, ngay cả khi hàm đó có được trong suốt quá trình thực thi hay không. Ngược lại, *Lazy evaluation* chỉ tính toán các giá trị cần thiết, khi mà hàm bắt buộc phải sử dụng tham số đó. Trong khi SML thuộc dạng Strict, Haskell thuộc dạng Lazy.
   - Lazy giúp chương trình tiết kiệm thời gian và tài nguyên tính toán các biểu thức không cần thiết, thay vào đó chỉ tập trung giải quyết vấn đề nhất định. Tuy nhiên việc này đôi khi sẽ khiến ta bỏ qua việc định giá những giá trị tiềm ẩn có thể làm tăng kích thước bộ nhớ (các giá trị không được định giá sẽ được lưu vào bộ nhớ). Không kiểm soát một cách hoàn toàn chương trình đôi khi dẫn tối việc khó sửa lỗi và crash máy trong run-time.
   - Strict thì dễ viết hơn, dễ hiểu và debug hơn do tất cả mọi thứ đều diễn ra một cách đơn định theo trật tự diễn ra trên source code, mặc dù phải tốn nhiều thời gian hơn để định giá hết tất cả các biến trong chương trình.

2. **Imperative**:

   - Đây là một đặc trưng của lập trình hàm, tập trung vào việc "biết" chương trình làm gì, thay vì diễn giải cách mà chương trình thực hiện. Nói cách khác, ta gói chương trình thành các phương thức, và chỉ quan tâm đầu ra và đầu vào. Các hàm sẽ tương tác bằng cách nối chuỗi tạo thành một hàm phức tạp hơn,
   - Haskell là ngôn ngữ thuần lập trình hàm, do đó bảo đảm không có side-effect (một hàm với đầu vào xác định chỉ cho một đầu ra duy nhất, hoàn toàn không bị ảnh hưởng bởi biến bên ngoài hay I/O) xảy ra. Với việc xóa dữ liệu hay nhập xuất, ngôn ngữ này sử dụng *monad*. Điểm mạnh là các hàm đều có thể tái sử dụng được, tuy nhiên chương trình sẽ phải phức tạp hơn và khó mở rộng quy mô để hạn chế các side-effect.
   - Đa số các câu lệnh trong SML là thuần lập trình hàm, nhưng không phải tất cả. Điều đó có nghĩa rằng trong một số trường hợp, Standard ML cho phép side-effect xảy ra. Nhờ vậy việc xử lý các side-effect của SML hiệu quả hơn Haskell.

3. **Cấu trúc kiểu dữ liệu**:

   - *Module trong Haskell* định nghĩa một tập hợp các giá trị, kiểu dữ liệu và mối quan hệ với các module khác. Nạp chồng hàm được diễn giải dưới khái niệm các *typeclasses*, tức là các kiểu dữ liệu có mối quan hệ với nhau. Ví dụ như ở hình bên dưới, Lớp dữ liệu `Real` có thể nạp chồng phép toán số học từ lớp dữ liệu `Num` như cộng trừ nhân chia,...
   - *Signature và Structure trong SML* xử lý cơ chế quản lý vùng dữ liệu cho kiểu dữ liệu phức tạp, bao gồm các kiểu dữ liệu cơ bản hay structure con bên trong. Signature hoạt động tượng tự như Interface trong Java, chỉ định tên các hoạt động trong kiểu dữ liệu đó, còn Structure thì sử dụng (một hay nhiều) Signature để mô tả cách thức hoạt động cụ thể. Do đó, cách tổ chức của Standard ML hạn chế lỗi xảy ra và có quy củ hơn Haskell.



Hình 29: *Typeclasses trong Haskell*

4. **Exception Handling**: Haskell sử dụng `monad` để xử lý ngoại lệ, trong khi đó Standard ML thực hiện cùng tác vụ trên ít phức tạp hơn vì nó cho phép side-effect.

```haskell
-- Haskell
isZero :: Int -> Either Text()
isZero n = if n == 0 then Right() else Left(error "is zero!")
```

```sml
// Standard ML
exception ZeroError;
fun isZero 0 = raise ZeroError
    | isZero n = n;
```

Listing 25: Exception Handling is Haskell and SML

	Haskell	Standard ML
Loại ngôn ngữ	Thuần lập trình hàm	Đa phần là lập trình hàm
Cơ chế evaluation	Lazy	Strict
Xử lý ngoại lệ	Phức tạp	Đơn giản
Hệ thống Module	Đơn giản	Có quy củ, chặt chẽ
Monads và Arrows	Phát triển hơn	Ít quan tâm nhưng vẫn có

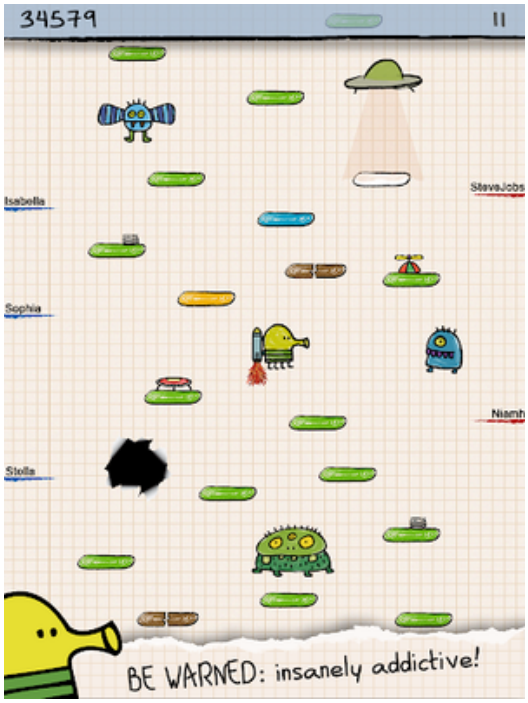
Hình 30: *So sánh các đặc điểm của Haskell và Standard ML*

4 Hiện thực một trò chơi đơn giản bằng C++ với Lập trình Hướng đối tượng

4.1 Giới thiệu trò chơi và Luật chơi

4.1.1 Giới thiệu trò chơi Doodle Jump

- **Doodle Jump** là một trò chơi dạng platform (người chơi di chuyển nhân vật vượt qua các thử thách) được phát triển và phát hành bởi tập đoàn Lima Sky đến từ Croatia cho nhiều hệ điều hành khác nhau trên điện thoại (Windows Phone, Android, iOS,..).
- Doodle Jump lần đầu tiên ra mắt cộng đồng vào năm 2009 cho hệ điều hành iOS và Android một năm sau đó. Khi vừa được phát hành, trò chơi đã nhận được nhiều lượt tải về bởi tính gây nghiện và "ức chế" cho người dùng rất cao. Theo số liệu thống kê, đến cuối năm 2011, trò chơi này đã bán hơn 10 triệu bản copy qua iTunes và Google Play, đồng thời đạt 15 triệu lượt tải về toàn cầu. Gần đây, vào năm 2020, Doodle Jump 2 ra đời với giao diện bắt mắt và nhiều tính năng mở rộng hơn.



Hình 31: *Giao diện trò chơi Doodle Jump*

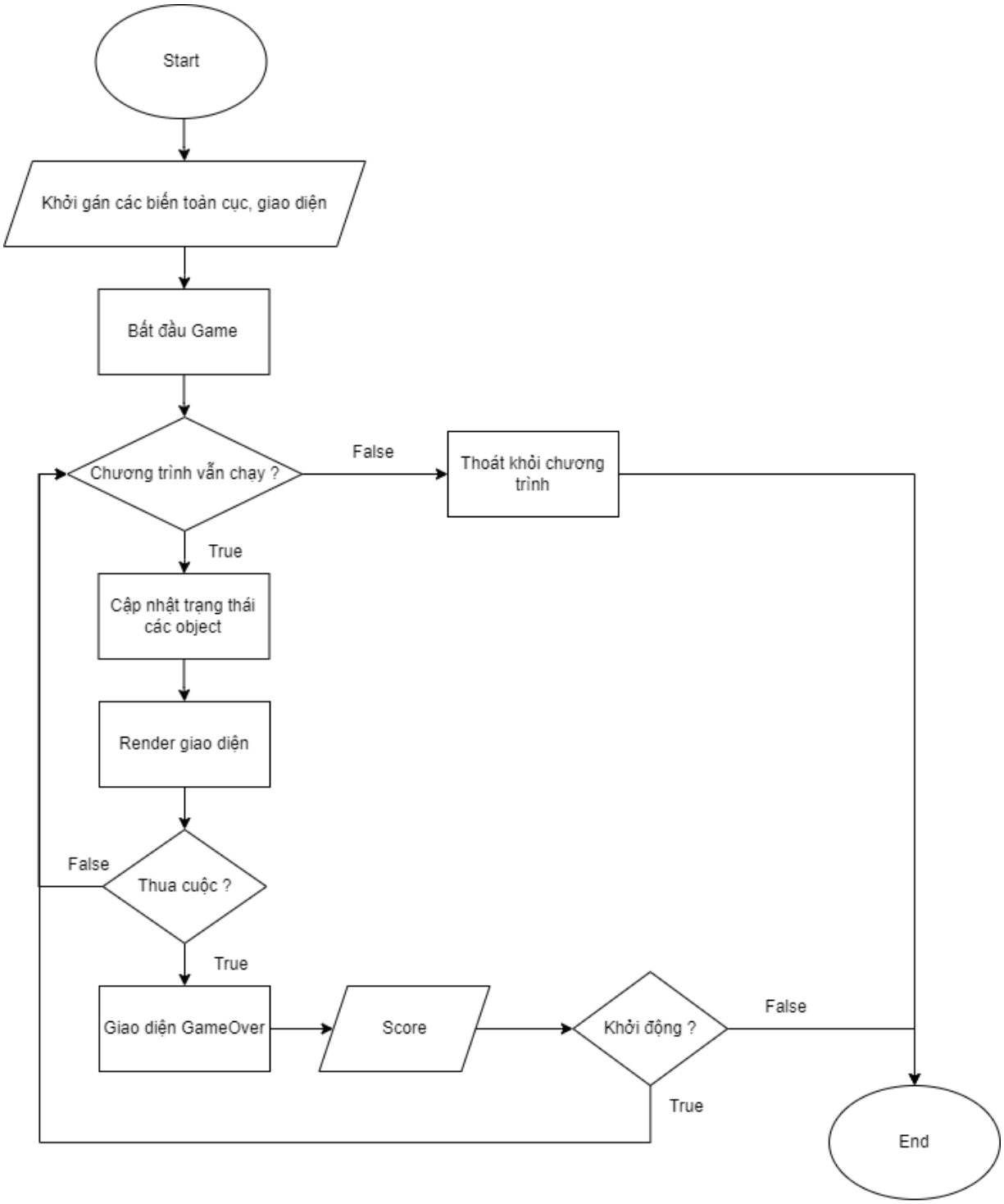
4.1.2 Luật chơi

- Luật chơi Doodle Jump khá đơn giản, đó là chỉ cần đưa The Doodler (nhân vật người dùng điều khiển) nhảy sang trái hoặc phải vào các bậc thang sao cho nó không rơi xuống đáy màn hình. Số bậc thang, hay độ cao của Doodler đạt được càng lớn thì số điểm được tích lũy càng cao. Bên trái và bên phải của màn hình liên kết với nhau, tức là khi Doodler khi đến mép phải thì nó sẽ xuất hiện ở mép trải và ngược lại.
- Các bậc thang platform cũng khá đa dạng, trải dài từ loại đứng yên và bất biến (màu xanh lá), loại di động và bất biến (màu xanh biển), loại đứng yên và chỉ có thể nhảy 1 lần (màu trắng), loại không nhảy được (màu nâu). Người chơi phải nhận dạng các platform khác nhau để có chiến thuật phù hợp.
- Bên cạnh đó sẽ có những chướng ngại vật như quái vật, UFO, hố đen... Khi chạm vào hoặc đứng gần những vật ấy thì xem như thua cuộc.

4.2 Giải thuật và Cấu trúc code

Ở đây, em chỉ hiện thực Doodle Jump đơn giản, tuy nhiên vẫn thể hiện đầy đủ bốn tính chất của Lập trình Hướng Đối tượng trong code. Cụ thể, em tạo game với 1 Doodler, hai loại Platform (loại màu xanh lá và màu trắng) và không có các chướng ngại vật.

4.2.1 Giải thuật hiện thực

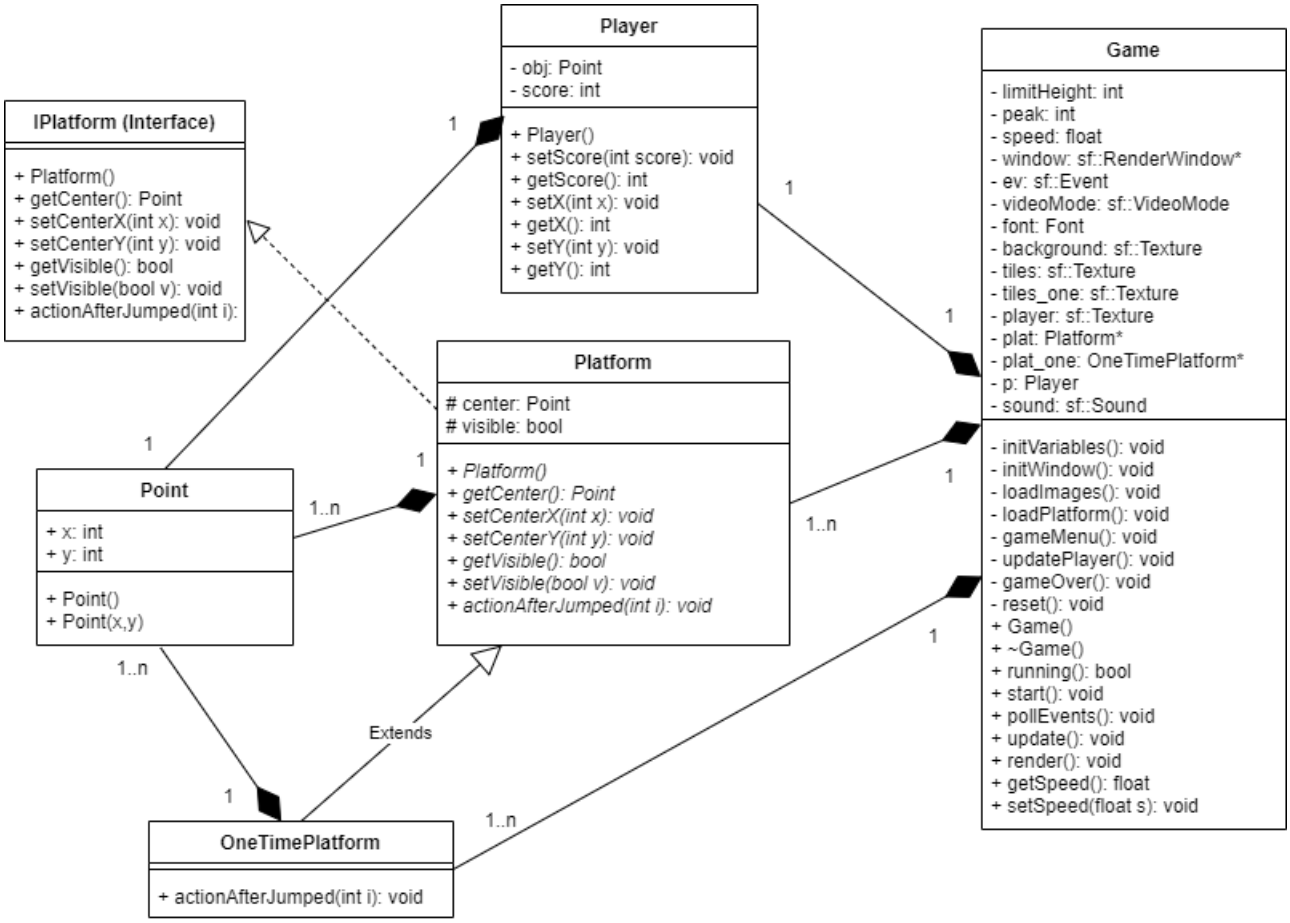


Hình 32: *Lưu đồ giải thuật trò chơi Doodle Jump*

Tóm tắt các bước chính hiện thực chương trình:

- *Bước 1*: Khởi gán các biến trong chương trình. Các biến này bao gồm cửa sổ giao diện, các object và số lượng, load các hình ảnh, audio, vị trí các object trong giao diện.
- *Bước 2*: Thực thi Game Loop chính của chương trình. Giai đoạn này bao gồm hai bước nhỏ:
  1. Cập nhật trạng thái các object: đợi thao tác I/O từ người dùng (cụ thể là nút Esc để thoát ngay lập tức) và cập nhật vị trí của các platform, Doodler và điểm số hiện tại (có cả kiểm tra thua cuộc để dừng game).
  2. Render giao diện: thực hiện đưa các object lên UI dựa vào tọa độ mới vừa có.
- *Bước 3*: Nếu ở bước 2, Doodler rơi ra ngoài màn hình thì xem như thua cuộc và xuất màn hình GameOver. Nếu chưa thua cuộc, tiếp tục thực hiện lại bước 2.
- *Bước 4*: Màn hình thua cuộc xuất hiện cùng với điểm số mà người chơi đạt được. Ta có thể ấn phím Enter để quay lại bước 2 chơi tiếp.

4.2.2 Cấu trúc code của trò chơi



Hình 33: *Class Diagram cho trò chơi Doodle Jump*

1. *Lớp Point:* chứa thông tin tọa độ 2D các điểm trên giao diện game.
   - Thuộc tính: tọa độ x, y.
   - Phương thức: default constructor và parameterized constructor đủ hai thuộc tính.
2. *Lớp Player:* chứa tọa độ của Doodler và điểm số người chơi và các hàm gán + truy xuất.
   - Thuộc tính: tọa độ và điểm của người chơi, đều đặt ở phạm vi `private`.
   - Phương thức: default constructor và các getter, setter hỗ trợ việc lấy các tọa độ và điểm số.
3. *Lớp IPlatform:* Interface thực hiện các class Platform
   - Thuộc tính: lớp thuần ảo (pure abstract class) nên không có thuộc tính.
   - Phương thức: default constructor và các getter, setter hỗ trợ việc xuất nhập tọa độ và sự xuất hiện của platform (ẩn/ hiện). Ngoài ra còn thêm 1 phương thức thay đổi sự xuất hiện của platform đó sau khi Doodler nhảy vào.
4. *Lớp Platform:* chứa tọa độ của từng bậc thang (platform) và các hàm gán + truy xuất
   - Thuộc tính: tọa độ và chế độ nhìn thấy của platform, đều đặt ở phạm vi `protected`.
   - Phương thức: default constructor và các getter, setter hỗ trợ việc xuất nhập tọa độ và sự xuất hiện của platform (ẩn/ hiện). Ngoài ra còn thêm 1 phương thức thay đổi sự xuất hiện của platform đó sau khi Doodler nhảy vào, cụ thể là Platform nếu có số thứ tự chia hết cho 5 (trong 1 mảng các Platform) sau khi tương tác sẽ bị ẩn.
5. *Lớp OneTimePlatform:*
   - Thuộc tính: thừa kế hoàn toàn từ lớp Platform, không có thuộc tính mới.
   - Phương thức: ghi đè (override) phương thức được đánh dấu `virtual`, cụ thể là hàm `actionAfterJumped(int i)` của lớp cha. Với OneTimePlatform, sau khi Doodler tương tác thì platform này sẽ bị ẩn.
6. *Lớp Game:*
   - Thuộc tính: các chỉ số cần thiết để thiết lập game
     (a) Giới hạn độ cao Doodler đạt tới để "đẩy" giao diện lên để sinh ra các platform mới và xóa platform gần phía dưới màn hình.
     (b) Tốc độ của Doodler cũng như tốc độ di chuyển các platform (không là một hằng số).
     (c) Dối tượng Doodler.
     (d) Mảng các đối tượng Platform và OneTimePlatform.
     (e) Các biến giao diện sử dụng trong *SFML*, một thư viện hỗ trợ game-making bằng C++ với namespace `sf`
   - Phương thức:
     (a) `Game()`: construct dùng để khởi gán các biến, load hình từ địa chỉ thư mục, xây dựng các platform với tọa độ tuân theo quy luật xác suất phân phối đều, bao gồm 4 hàm con với phạm vi `private` là `initVariables()`, `initWindow()`, `loadImages()`, `loadPlatform()`.
     (b) `running()`: hàm chạy gameloop của chương trình, trả về kết quả boolean chương trình còn chạy hay không.
     (c) `start()`: hàm bắt đầu gameloop nếu như thỏa điều kiện từ hàm phía trên, bao gồm 2 hàm con là `update()` và `render()`
     (d) `update()`: chạy cập nhật các đối tượng và nhận I/O từ người dùng, lần lượt là các tác vụ của phương thức `updatePlayer()` và `pollEvents()`.

(e) Hàm hỗ trợ getter và setter để thay đổi tốc độ chương trình khi có va chạm giữa Doodler và một trong các platform.

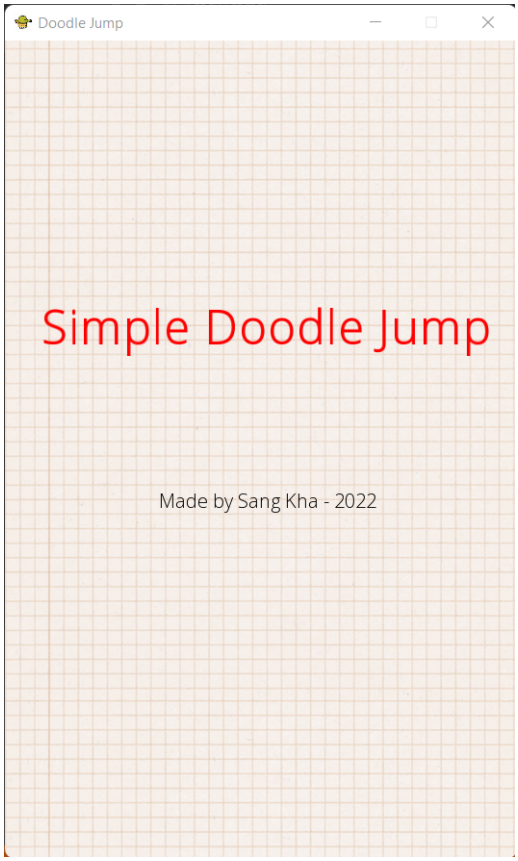
Các tính chất cũa lập trình hướng đối tượng được thể hiện trong Class Diagram:

1. **Tính đóng gói**: Các class đều có phạm vi truy xuất `private` hay `protected`, người dùng bên ngoài không thể trực tiếp chỉnh sửa mà phải thông qua các setter và getter.
2. **Tính trừu tượng**: hiện thực lớp thuần ảo IPlatform như là một Interface - bản thiết kế cho các platform.
3. **Tính kế thừa**: lớp OneTimePlatform kế thừa các thuộc tính và phương thức của lớp Platform.
4. **Tính đa hình**: ở hàm `actionAfterJumped(int i)`, lớp cơ sở Platform và lớp dẫn xuất OneTimePlatform có sự ghi đè (Run Time Polymorphism) hay constructor overloading ở lớp Point (Compile Time Polymorphism).

4.3 Demo trò chơi

Simple Doodle Jump được em hiện thực dựa trên game gốc Doodle Jump kèm với một số tính năng mới để tăng tính thú vị của trò chơi.

- *Giao diện Menu*: Đây là giao diện xuất hiện khi người dùng vừa mở trò chơi, với một số thông tin cơ bản. Người dùng có thể nhấn Enter để bắt đầu chơi.

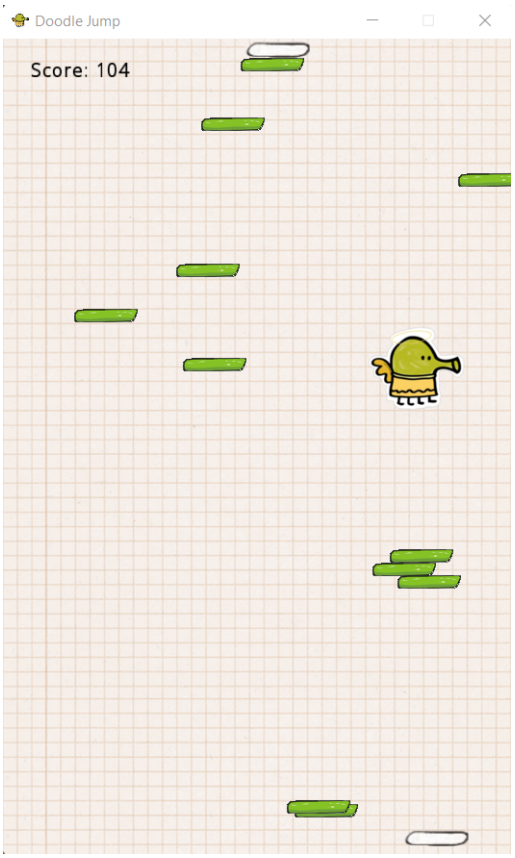


Hình 34: *Giao diện Menu cho trò chơi Doodle Jump*

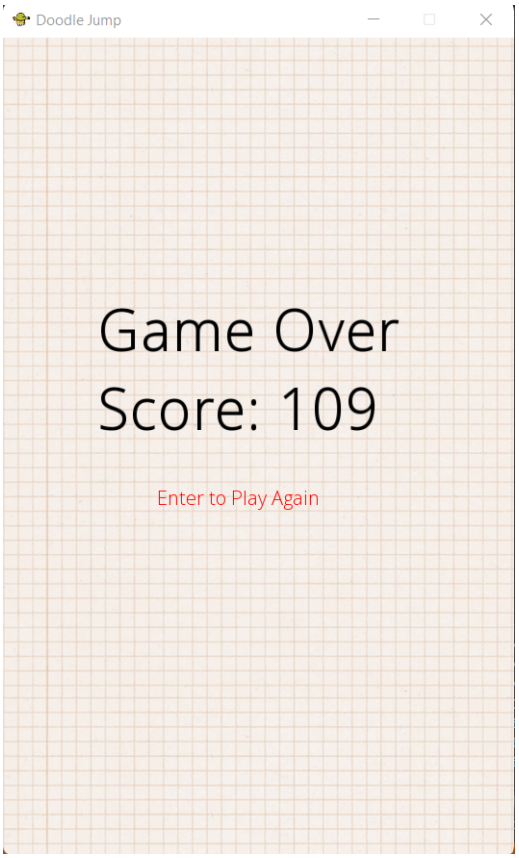
- *Giao diện chính*: Main Game của Simple Doodle Jump.
  - Trên màn hình sẽ có các platform, bao gồm hai loại: các bậc xanh lá (luôn tồn tại) và các bậc trắng (nhảy vào thì biến mất ngay).

  - Người chơi di chuyển Doodler sang trái hoặc phải lần lượt bằng phím A và phím D, với lưu ý rằng khi nó chạm vào mép trái thì sẽ xuất hiện ở mép phải và ngược lại. Ngoài ra, game còn hỗ trợ phím W (tăng gia tốc, Doodler sẽ bị lọt xuống bậc bên dưới để "lấy đà" nhảy cao hơn) và phím S (giảm gia tốc, nếu người dùng ấn phím khi Doodler đang đi lên thì nó sẽ "bay" lên với tốc độ chậm).
  - Ở góc trái thể hiện số điểm của người chơi, được quy đổi từ độ cao lớn nhất mà Doodler đạt được. Tuy nhiên, mỗi lần ấn phím W/S thì người chơi mất 1 điểm. Trò chơi kết thúc khi Doodler lọt khỏi đáy màn hình hoặc điểm người chơi bé hơn 0.

- *Giao diện GameOver*: màn hình tổng kết kết quả trò chơi với điểm số đạt được. Người chơi có thể ấn phím Enter để khởi động lại game (các platform sẽ được phân phối lại ngẫu nhiên, điểm reset về 0).
- Trong mọi trường hợp, người chơi đều có thể nhấn phím Esc hoặc dấu X trên giao diện để thoát khỏi game.



Hình 35: *Giao diện trong trò chơi Doodle Jump*



Hình 36: *Kết thúc trò chơi Doodle Jump*

Tài liệu

[1] Wikipedia, last edited April 5 2022 "Object-oriented programming". https://en.wikipedia.org/wiki/Object-oriented_programming

[2] Kyle Biniasz, Nov 12 2021 "What Are OOP Concepts in Java? How They Work and More". https://stackify.com/oops-concepts-in-java/

[3] Thanoshan MV, May 1 2020 "Object-Oriented Programming Principles in Java: OOP Concepts for Beginners" https://www.freecodecamp.org/news/java-object-oriented-programming-system-principles-oops-concepts-for-beginners/

[4] Mariana Berga, Rute Figueiredo, July 22, 2021 "Functional programming vs OOP: comparing paradigms" https://www.imaginarycloud.com/blog/functional-programming-vs-oop/#:~:text=As%20mentioned%2C%20functional%20programming%20relies,to%20achieve%20a%20certain%20task.

[5] Dong Nguyen, Aug 8 2019 "Functional Programming - Phần 1 - Con đường sáng" https://kipalog.com/posts/Functional-Programming---Phan-1---Con-duong-sang

[6] Ryan Thelin, Oct 15 2020 "What is functional programming? Explained in Python, JS, and Java" https://www.educative.io/blog/what-is-functional-programming-python-js-java

[7] Oracle, "Java Documentation" https://docs.oracle.com/en/java/

[8] James Britt, "Documentation for Ruby" https://ruby-doc.org/

[9] Equinix, "Documentation - Haskell.org" https://www.haskell.org/documentation/

[10] SML/NJ Fellowship, "Standard ML of New Jersey User's Guide" http://www.smlnj.org/doc/

[11] Hackerrank, Feb 22 2021, "Difference Between Java and Ruby" https://www.geeksforgeeks.org/difference-between-java-and-ruby

[12] Vahid Dejwakh, Nov 13 2020, "Encapsulation, Polymorphism, and Abstraction in Ruby" https://dev.to/dej/encapsulation-polymorphism-and-abstraction-in-ruby-41ia

[13] Wikipedia, last edited April 14 2022, "Doodle Jump" https://en.wikipedia.org/wiki/Doodle_Jump

[14] Documentation of SFML 2.5.1, https://www.sfml-dev.org/documentation/2.5.1/