Class

1. Bài về con trỏ this

```
class Car {
  constructor(name) {
    this.name = name
  }

  print() {
    setTimeout(function () {
      // Làm sao để có thể truy cập đến thuộc tính name tại đây
    }, 1000)
  }
}
```

Trả lời

```
class Car {
  constructor(name) {
    this.name = name
  }

  print() {
    setTimeout(() => {
      console.log(this.name)
    }, 1000)
  }
}

new Car('BMW').print()
```

hoặc như thế này

```
class Car {
  constructor(name) {
    this.name = name
  }

  print() {
    setTimeout(
      function () {
        console.log(this.name)
      }.bind(this),
      1000
    )
```

```
  }
}

new Car('BMW').print()
```

2. Có class Rabbit kế thừa Animal. Đoạn code dưới đây không thể tạo đối tượng Rabbit, vì sao và sửa nó.

```
class Animal {
  constructor(name) {
    this.name = name
  }
}

class Rabbit extends Animal {
  constructor(name) {
    this.name = name
    this.created = Date.now()
  }
}

let rabbit = new Rabbit('White Rabbit') // Error: this is not defined
alert(rabbit.name)
```

Trả lời

```
class Animal {
  constructor(name) {
    this.name = name
  }
}

class Rabbit extends Animal {
  constructor(name) {
    super(name)
    this.created = Date.now()
  }
}

let rabbit = new Rabbit('White Rabbit')
alert(rabbit.name)
```

3. Class Clock với chức năng chính là in ra thời gian hiện tại mỗi một giây

```
class Clock {
  constructor({ template }) {
    this.template = template
  }
```

```
  render() {
    let date = new Date()

    let hours = date.getHours()
    if (hours < 10) hours = '0' + hours

    let mins = date.getMinutes()
    if (mins < 10) mins = '0' + mins

    let secs = date.getSeconds()
    if (secs < 10) secs = '0' + secs

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs)

    console.log(output)
  }

  stop() {
    clearInterval(this.timer)
  }

  start() {
    this.render()
    this.timer = setInterval(() => this.render(), 1000)
  }
}
const clock = new Clock({ template: 'h:m:s' })
clock.start()
```

Tạo một class ExtendedClock kế thừa Clock và thêm độ chính xác của tham số - số mili giây giữa các lần in. Mặc định là 1000 mili giây. Lưu ý không sửa đổi class Clock.

Trả lời

```
class ExtendedClock extends Clock {
  constructor({ template, precision = 1000 }) {
    super({ template })
    this.precision = precision
  }
  start() {
    this.render()
    this.timer = setInterval(() => this.render(), this.precision)
  }
}

const clock = new ExtendedClock({ template: 'h:m:s', precision: 500 })
clock.start()
```

4. Như chúng ta đã biết, tất cả các đối tượng thường kế thừa từ `Object.prototype` và có quyền truy cập vào các phương thức đối tượng "generic" như `hasOwnProperty`, v.v. Ví dụ:

```
class Rabbit {
  constructor(name) {
    this.name = name
  }
}

let rabbit = new Rabbit('Rab')

// hasOwnProperty method is from Object.prototype
alert(rabbit.hasOwnProperty('name')) // true
```

Nhưng nếu chúng ta đánh vần nó một cách rõ ràng như `class Rabbit extends Object`, thì kết quả sẽ khác với một class `Rabbit` đơn giản?

Có gì khác biệt?

Dưới đây là một ví dụ về code như vậy (nó không hoạt động – tại sao? Sửa nó?):

```
class Rabbit extends Object {
  constructor(name) {
    this.name = name
  }
}

let rabbit = new Rabbit('Rab')

alert(rabbit.hasOwnProperty('name')) // Error
```

Trả lời: Đầu tiên, hãy xem lý do tại sao code không hoạt động. Sẽ dễ dàng nhận biết hơn nếu chúng ta run code. Một class kế thừa thì nên có `super()` trong constructor. Nếu không `this` sẽ không được xác định. Đây là cách fix:

```
class Rabbit extends Object {
  constructor(name) {
    super() // need to call the parent constructor when inheriting
    this.name = name
  }
}

let rabbit = new Rabbit('Rab')

alert(rabbit.hasOwnProperty('name')) // true
```

Nhưng đó không phải là tất cả. Ngay cả khi fix được, vẫn còn một số sự khác nhau quan trọng về `class Rabbit extends Object` vs `class Rabbit`. Như chúng ta biết, cú pháp `extends` sẽ setup 2 prototype:

1. Giữa `prototype` của các constructor function (cho phương thức).
2. Giữa chính các constructor function (cho phương thức tĩnh).

Trong trường hợp của chúng ta, `class Rabbit extends Object` nghĩa là:

```
class Rabbit extends Object {}

alert(Rabbit.prototype.__proto__ === Object.prototype) // (1) true
alert(Rabbit.__proto__ === Object) // (2) true
```

Vậy nên `Rabbit` bây giờ có thể truy cập đến các phương thức tĩnh của `Object` thông qua `Rabbit`, như thế này:

```
class Rabbit extends Object {}

// normally we call Object.getOwnPropertyNames
alert(Rabbit.getOwnPropertyNames({ a: 1, b: 2 })) // a,b
```

Nhưng nếu chúng ta không có `extends Object`, thì `Rabbit.__proto__` sẽ không được gán bằng `Object` Đây là demo:

```
class Rabbit {}

alert(Rabbit.prototype.__proto__ === Object.prototype) // (1) true
alert(Rabbit.__proto__ === Object) // (2) false (!)
alert(Rabbit.__proto__ === Function.prototype) // mặc định cho bất cứ
function nào

// error, no such function in Rabbit
alert(Rabbit.getOwnPropertyNames({ a: 1, b: 2 })) // Error
```

Vậy `Rabbit` không thể truy cập đến các phương thức tĩnh của `Object` trong trường hợp này.

Nhân tiện, `Function.prototype` có các phương thức như `call`, `bind`...Chúng đều có ở cả 2 trường hợp bởi vì được tích hợp sẵn trong `Object` constructor, `Object.__proto__ === Function.prototype`.

5. Trong đoạn mã dưới đây, tại sao `instanceof` trả về `true`? Ta có thể dễ dàng thấy rằng `a` không được tạo bởi `B()`.

```
function A() {}
function B() {}

A.prototype = B.prototype = {}

let a = new A()

alert(a instanceof B) // true
```

Trả lời

Vâng, trông có vẻ lạ. Nhưng `instanceof` không quan tâm đến function, cái `instanceof` quan tâm là thuộc tính `prototype` của function. Và ở đây `a.__proto__ == B.prototype`, vì thế `instanceof` return true. Vậy nên về mặt logic của `instanceof`, `prototype` mới là thứ định nghĩa type, chứ không phải constructor function