

JAVASCRIPT DESIGN PATTERNS

LECTURE 1

By Mona Soliman

AGENDA

- What is Design Patterns
- Design Patterns Benefits
- Design Patterns Categories
- Constructor Pattern
- Singleton Pattern
- Factory Pattern
- Mixin Pattern
- Builder Pattern
- Dependency Injection Pattern

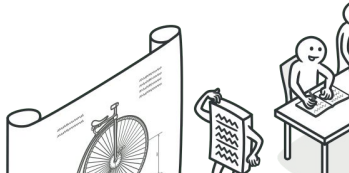
WHAT IS DESIGN PATTERNS ?

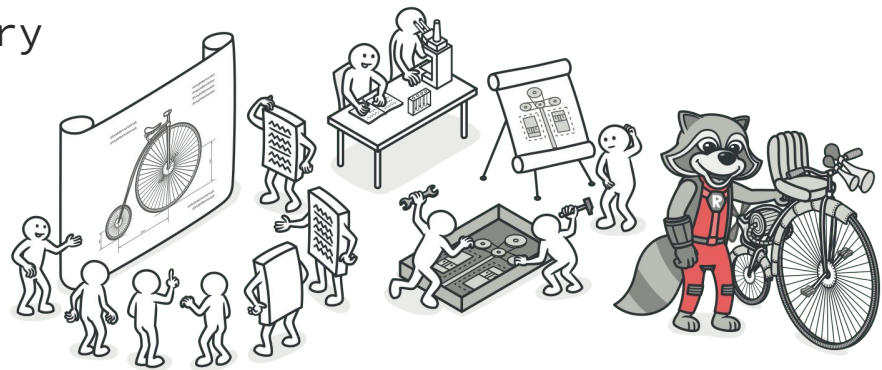
design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

A pattern is a reusable solution that can be applied to commonly occurring problems in software design

- Popularized by the Gang of four book (1994)
- Translated to many languages: C#, Java, C++, JS ...
- In our case - in writing JavaScript applications.

DESIGN PATTERNS BENEFITS

- 1- Make your life easier by not reinventing the wheel
 - 2- Improve your object-oriented skills
 - 3- Recognize patterns in libraries and languages
 - 4- Use the power of a shared vocabulary
 - 5- Patterns are proven solutions
- 



DESIGN PATTERNS CATEGORIES:

Creational Design Pattern

it provides the object or classes creation mechanism that enhance the flexibilities and reusability of the existing code.

Structural Design Patterns

responsible for assemble object and classes into a larger structure making sure that these structure should be flexible and efficient. They are very essential for enhancing readability and maintainability of the code.

Behavior Design Pattern

Behavior Design Patterns are responsible for how one class communicates with others.

DESIGN PATTERNS CATEGORIES:

Design patterns		
Creational	Structural	Behavioral
Abstract factory Builder Factory method Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Visitor Strategy Template method

DESIGN PATTERNS CATEGORIES:

- Creational :Create new things.
- Structure : Structure your code.
- Behavioral: behaviors your code.



CONSTRUCTOR PATTERN:

In classical object-oriented programming languages, a **constructor** is a special method used to initialize a newly created **object** once memory has been allocated for it.

```
1  class Car{
2      constructor(doors,engine,color){
3          this.doors=doors;
4          this.engine=engine;
5          this.color=color;
6      }
7  }
8
9  var car1=new Car(4,1600,"red")
10 console.log(car1);
11 class Suv extends Car{
12     constructor(doors,engine,color,wheels){
13         super(doors,engine,color)
14         this.wheels=wheels;
15     }
16 }
17
18 var mercedes=new Suv(4,1500,"green",4)
19 console.log(mercedes);
```


SINGLETON PATTERN:

The *Singleton* Pattern **limits the number of instances of a particular object to just one**. This single instance is called the singleton.

```
1  class Car{
2      constructor(doors,engine,color){
3          if(!Car.instance){
4              this.doors=doors;
5              this.engine=engine;
6              this.color=color;
7              Car.instance=this
8          }else{
9              return Car.instance
10         }
11     }
12 }
13
14
15 var car1=new Car(4,1600,"blue")
16 var car2=new Car(4,1500,"green")
17
18 console.log(car1);
19 console.log(car2);
```

FACTORY PATTERN:

The factory pattern is a creational design pattern that **provides a generic interface for creating objects**. In the factory pattern, we can specify the type of object being created and we do not need to explicitly require a constructor.

```
1  class ShapeFactory {
2      constructor(type) {
3          this.type = type
4      }
5
6      createShape() {
7          switch (this.type) {
8              case "Circle":
9                  return new Circle();
10             case "Rectangle":
11                 return new Rectangle();
12             case "Square":
13                 return new Square();
14             default:
15                 return "invalid type"
16             }
17         }
18     }
19
20     var shape1 = new ShapeFactory("Square")
21     var Obj = shape1.createShape()
22     Obj.draw()
```

MIXIN PATTERN:

Mixins are a great way **to mix functions and instances of classes after they have been created**. In other words you could use mixins to add interesting functions to the any class that you created earlier.

```
1  class Swimmer {
2      constructor(name, country) {
3          this.name = name;
4          this.country = country;
5      }
6  }
7
8
9  var swimProperties = {
10     setSwimProperties(direction, speed) {
11         this.direction = direction;
12         this.speed = speed;
13     },
14     getSwimProperties() {
15         console.log(`swimmer name is
16         ${this.name} from
17         ${this.country}
18         his speed is ${this.speed}
19         and his direction is ${this.direction}`);
20     }
21 }
22 };
23
24 Object.assign(Swimmer.prototype, swimProperties)
25 var swimmer1=new Swimmer("Ahmed","Egypt");
26 swimmer1.setSwimProperties("Up",600)
27 swimmer1.getSwimProperties()
```

BUILDER PATTERN:

Builder pattern is a **design pattern to provide a flexible solution for creating objects**. Builder pattern separates the construction of a complex object from its representation. Builder pattern builds a complex object using simple objects by providing a step by step approach. It belongs to the creational patterns.

```
1  class Course {
2      constructor(options) {
3          this.name = options.name;
4          this.sales = options.sales;
5          this.isFree = options.isFree;
6          this.isCampain = options.isCampain;
7          this.price = options.price;
8      }
9  }
10 }
11
12 class CourseBuilder{
13     constructor(name, sales=0, price=0) {
14         this.name = name;
15         this.sales = sales;
16         this.price = price;
17     }
18
19     makePaid(price){
20         this.isFree=false;
21         this.price=price;
22         return this
23     }
24
25     makeCampain(){
26         this.isCampain=true;
27         return this
28     }
29     build(){
30         return new Course(this)
31     }
32 }
33 // var course1=new Course("DP JS",5,false,true,1500);
34 // console.log(course1);
35
36 var course1=new CourseBuilder("DP JS",5).makeCampain().makePaid(1400).build()
37 console.log(course1);
```

DEPENDENCY INJECTION PATTERN:

Dependency injection is a **software design pattern** where an **object or function** makes **use of other objects or functions** (dependencies) without worrying about their underlying implementation details.

```
1 class Engine{
2     constructor(speed){
3         this.speed=speed
4     }
5 }
6
7 class Tiers{
8     constructor(name){
9         this.name=name
10    }
11 }
12
13 class Car{
14
15     constructor(engine,tiers){
16         this.engine=engine
17         this.tiers=tiers;
18     }
19 }
20
21 var car=new Car(new Engine(600),new Tiers("ay 7aga"));
22 console.log(car);
```

THANK YOU