**1. Singleton**

**Singleton** is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

class Singleton {  
 private static instance: Singleton;  
 private constructor() {}  
 static getInstance() {  
 if (!Singleton.instance) {  
 Singleton.instance = new Singleton();  
 }  
 return Singleton.instance;  
 }  
}  
let e = new Singleton(); // Error: constructor of 'Singleton' is private.  
let v = Singleton.getInstance();

**2. Prototype**

**Prototype** is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes.

// prototype object monster with attack method  
const monster = {  
 attack() {  
 return "attack!!!";  
 },  
};  
  
const ghost = Object.create(monster, { name: { value: "ghost" } });  
  
console.log(ghost); // { name: 'ghost' }  
console.log(ghost.attack()); // attack!!!

**3. Builder**

**Builder** is a creational design pattern, which allows constructing complex objects step by step.

class Burger {  
 bun: string;  
 meat: string;  
 cheese: string;  
  
 constructor() {}  
 addBun(bun) {  
 this.bun = bun;  
 return this;  
 }  
 addMeat(meat) {  
 this.meat = meat;  
 return this;  
 }  
 addCheese(cheese) {  
 this.cheese = cheese;  
 return this;  
 }  
}  
  
const tomBurger = new Burger();  
tomBurger.addBun("sesame").addMeat("beef").addCheese("cheddar");

**4. Facade**

**Facade** is a structural design **pattern** that provides a simplified (but limited) interface to a complex system of classes, library or framework.

class ElectricalSystem {  
 turnOn() {  
 console.log("ElectricalSystem turn on");  
 }  
 turnOff() {  
 console.log("ElectricalSystem turn off");  
 }  
}  
  
class WaterSystem {  
 turnOn() {  
 console.log("WaterSystem turn on");  
 }  
 turnOff() {  
 console.log("WaterSystem turn off");  
 }  
}  
  
class FacadeHome {  
 electricalSystem: ElectricalSystem;  
 waterSystem: WaterSystem;  
  
 constructor() {  
 this.electricalSystem = new ElectricalSystem();  
 this.waterSystem = new WaterSystem();  
 }  
  
 turnOn() {  
 this.electricalSystem.turnOn();  
 this.waterSystem.turnOn();  
 }  
  
 turnOff() {  
 this.electricalSystem.turnOff();  
 this.waterSystem.turnOff();  
 }  
}  
  
const facadeHome = new FacadeHome();  
facadeHome.turnOn();  
facadeHome.turnOff();

**5. Proxy**

**Proxy** is a structural design **pattern** that provides an object that acts as a substitute for a real service object used by a client.

class User {  
 constructor(public name: string, public age: number) {}  
}  
  
class UserProxy {  
 private user: User;  
  
 constructor(user: User) {  
 this.user = user;  
 }  
  
 get name() {  
 return this.user.name;  
 }  
  
 get age() {  
 return this.user.age;  
 }  
}  
  
const user = new User("Tom", 18);  
const userProxy = new UserProxy(user);  
console.log(userProxy.name); // Tom  
console.log(userProxy.age); // 18

**6. Iterator**

**Iterator** is a behavioural design **pattern** that allows sequential traversal through a complex data structure without exposing its internal details.

function\* range(start: number, end: number, step: number = 1) {  
 while (start < end) {  
 yield start;  
 start += step;  
 }  
}  
  
for (const i of range(0, 10, 2)) {  
 console.log(i);   
}  
// output: 0, 2, 4, 6, 8  
  
class IteratorClass {  
 private index: number = 0;  
 constructor(private items: any[]) {}  
  
 next() {  
 return this.items[this.index++];  
 }  
  
 hasNext() {  
 return this.index < this.items.length;  
 }  
}  
  
const iteratorObject = new IteratorClass([1, 2, 3, 4, 5]);  
while (iteratorObject.hasNext()) {  
 console.log(iteratorObject.next());  
}  
// output: 1, 2, 3, 4, 5

**7. Observer**

**Observer** is a behavioural design pattern that allows some objects to notify other objects about changes in their state.

class Subject {  
 private observers: Observer[] = [];  
  
 constructor() {}  
  
 subscribe(observer: Observer) {  
 this.observers.push(observer);  
 }  
  
 unsubscribe(observer: Observer) {  
 const index = this.observers.indexOf(observer);  
 this.observers.splice(index, 1);  
 }  
  
 notify() {  
 this.observers.forEach((observer) => observer.update());  
 }  
}  
  
class Observer {  
 constructor(private name: string) {}  
  
 update() {  
 console.log(`${this.name} has been notified`);  
 }  
}  
  
const subject = new Subject();  
const observer1 = new Observer("observer1");  
  
subject.subscribe(observer1);  
subject.notify(); // observer1 has been notified  
subject.unsubscribe(observer1);

**8. Mediator**

**Mediator** is a behavioural design pattern that reduces coupling between components of a program by making them communicate indirectly, through a special mediator object.

class Mediator {  
 private users: User[] = [];  
  
 constructor() {}  
  
 addUser(user: User) {  
 this.users.push(user);  
 }  
  
 sendMessage(user: User, message: string) {  
 this.users.forEach((u) => {  
 if (u !== user) {  
 u.receiveMessage(message);  
 }  
 });  
 }  
}  
  
class User {  
 constructor(private name: string, private mediator: Mediator) {  
 this.mediator.addUser(this);  
 }  
  
 sendMessage(message: string) {  
 this.mediator.sendMessage(this, message);  
 }  
  
 receiveMessage(message: string) {  
 console.log(`${this.name} received message: ${message}`);  
 }  
}  
  
const mediator = new Mediator();  
const user1 = new User("user1", mediator);  
const user2 = new User("user2", mediator);  
const user3 = new User("user3", mediator);  
  
user1.sendMessage("hello");  
// user2 received message: hello  
// user3 received message: hello  
user2.sendMessage("hi");  
// user1 received message: hi  
// user3 received message: hi  
user3.sendMessage("how are you?");  
// user1 received message: how are you?  
// user2 received message: how are you?

**9. State**

**State** is a behavioural design pattern that allows an object to change its behaviour when its internal state changes.

class Person {  
 private state: State;  
  
 constructor() {}  
  
 think() {  
 this.state.think();  
 }  
  
 setState(state: State) {  
 this.state = state;  
 }  
}  
  
interface State {  
 think(): void;  
}  
  
class FastState implements State {  
 think() {  
 console.log("think fast");  
 }  
}  
  
class SlowState implements State {  
 think() {  
 console.log("think slow");  
 }  
}  
  
const person = new Person();  
person.setState(new FastState());  
person.think(); // think fast  
person.setState(new SlowState());  
person.think(); // think slow

**10. Factory**

**Factory method** is a creational design pattern that solves the problem of creating product objects without specifying their concrete classes.

class BurgerFactory {  
 static createBurger(type: string) {  
 switch (type) {  
 case "cheese":  
 return new CheeseBurger();  
 case "veggie":  
 return new VeggieBurger();  
 default:  
 throw new Error("No such burger");  
 }  
 }  
}  
  
class CheeseBurger {  
 constructor() {  
 console.log("Cheese burger");  
 }  
}  
  
class VeggieBurger {  
 constructor() {  
 console.log("Veggie burger");  
 }  
}  
  
const cheeseBurger = BurgerFactory.createBurger("cheese");  
const veggieBurger = BurgerFactory.createBurger("veggie");