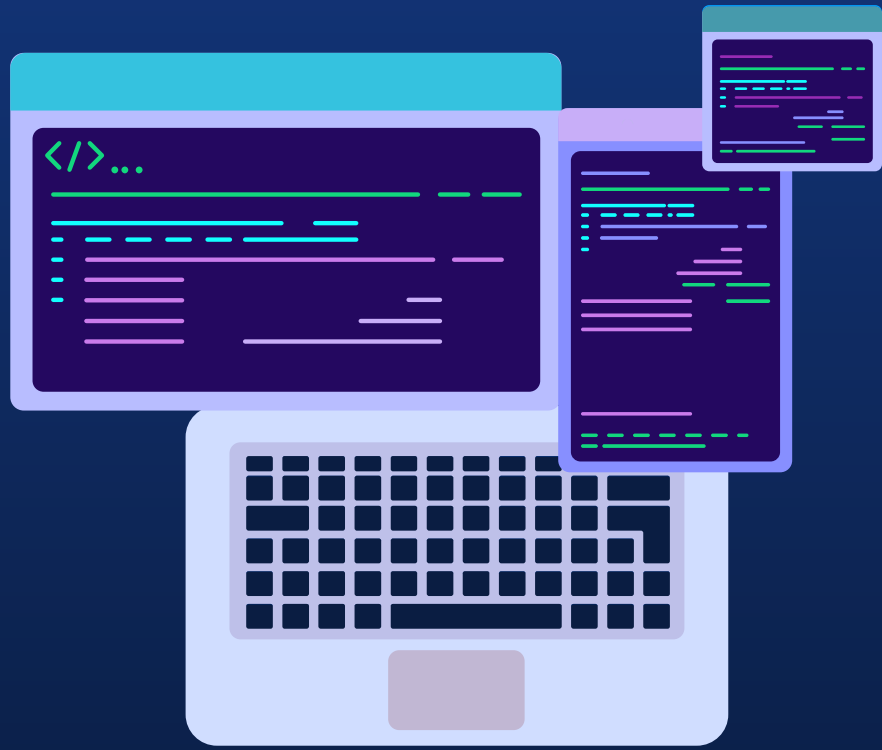
SOLID PRINCIPLES

In Object-oriented programming



About us

{

name: Gabriel Jonay Vera Estévez

email: alu0101398198@ull.edu.es

alternativeEmail: jonay.vera.32@ull.edu.es

}

{

name: Ginés Cruz Chávez

email: alu0101431079@ull.edu.es

alternativeEmail: gines.cruz.30@ull.edu.es

}

TABLE OF CONTENTS

01

Introduction to SOLID Principles

“The SOLID principles are a set of guidelines for software development that help us to create more maintainable, understandable, and flexible code. “

— Robert C. Martin

BENEFITS OF THE SOLID PRINCIPLES

- Increased maintainability and flexibility
- Better scalability and reusability
- Reduced coupling and improved cohesion
- Easier testing and debugging

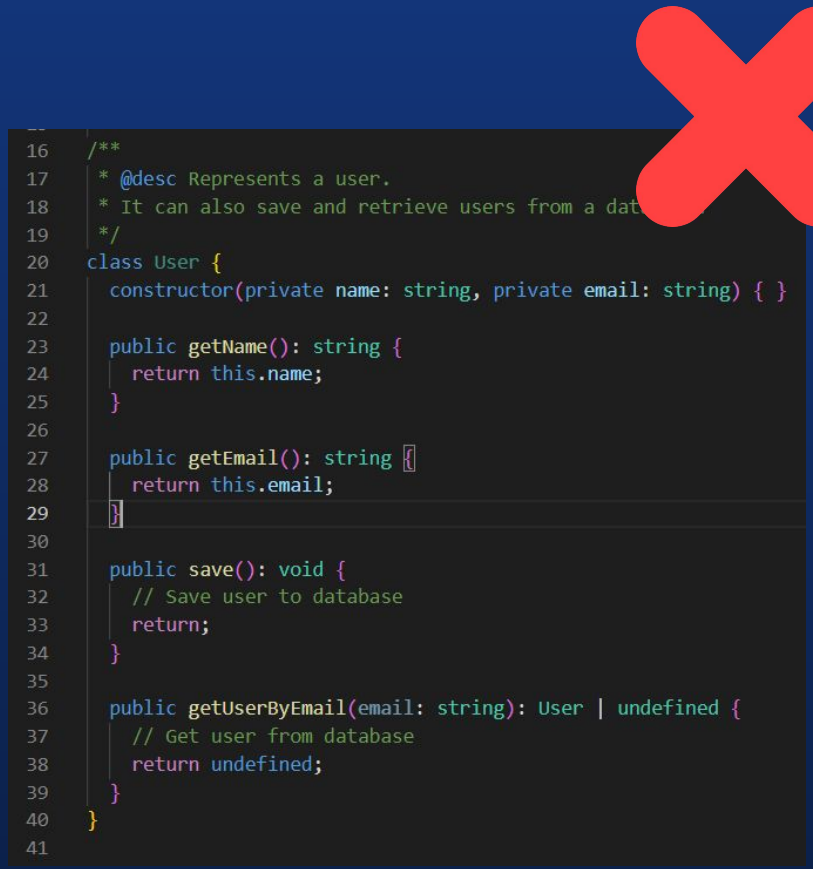
The code **WILL** change.

02

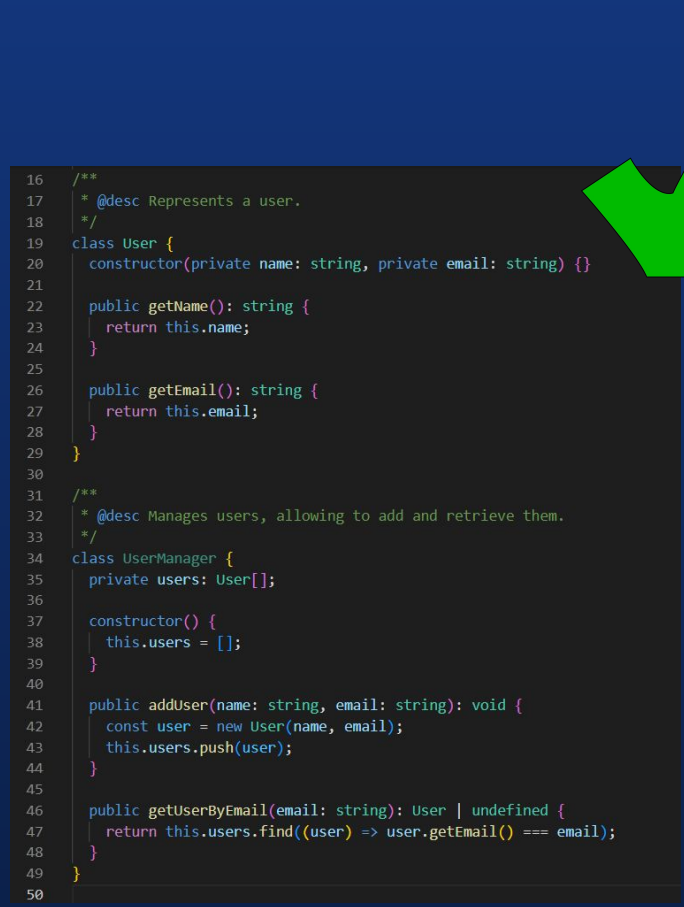
Single-responsibility principle

“A class should have one and only one reason to change, meaning that a class should have only one job.”

— **Robert C. Martin**

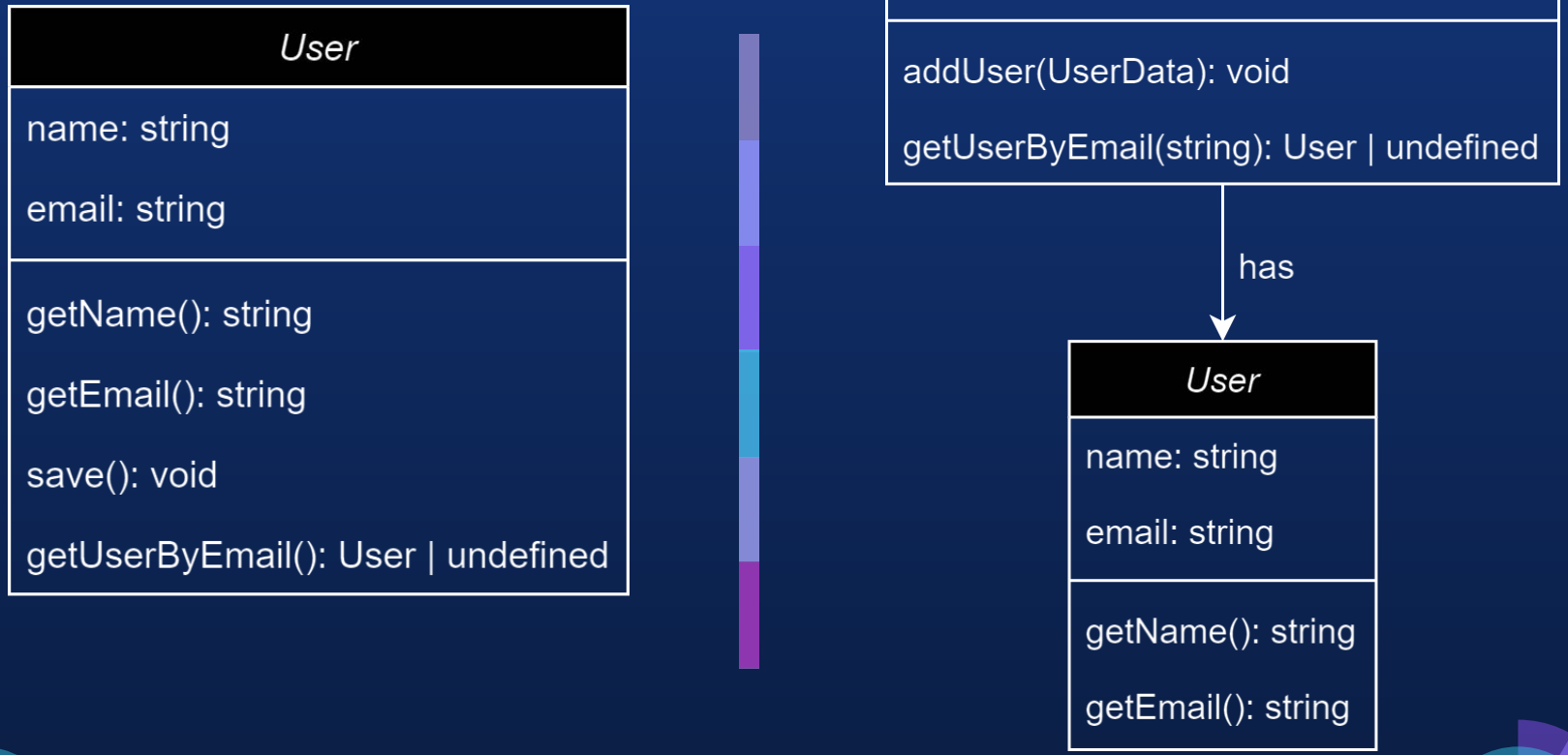


```ts
/**
 * @desc Represents a user.
 * It can also save and retrieve users from a dat
 */
class User {
  constructor(private name: string, private email: string) { }

  public getName(): string {
    return this.name;
  }

  public getEmail(): string {
    return this.email;
  }

  public save(): void {
    // Save user to database
    return;
  }

  public getUserByEmail(email: string): User | undefined {
    // Get user from database
    return undefined;
  }
}

```



```ts
/**
 * @desc Represents a user.
 */
class User {
  constructor(private name: string, private email: string) {}

  public getName(): string {
    return this.name;
  }

  public getEmail(): string {
    return this.email;
  }
}

/**
 * @desc Manages users, allowing to add and retrieve them.
 */
class UserManager {
  private users: User[];

  constructor() {
    this.users = [];
  }

  public addUser(name: string, email: string): void {
    const user = new User(name, email);
    this.users.push(user);
  }

  public getUserByEmail(email: string): User | undefined {
    return this.users.find((user) => user.getEmail() === email);
  }
}

```

[single-responsability]


User
name: string
email: string
getName(): string
getEmail(): string
save(): void
getUserByEmail(): User | undefined
UserManager
users: User[]
addUser(UserData): void
getUserByEmail(string): User | undefined
has
User
name: string
email: string
getName(): string
getEmail(): string

DOS AND DON'TS

DON'T

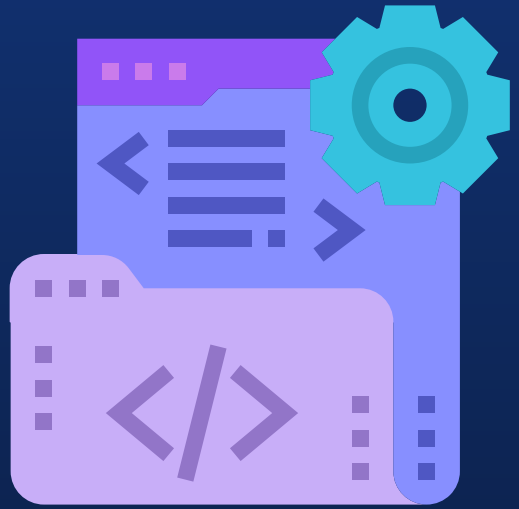
- Have multiple responsibilities in a single class.
- Mix concerns in functions or methods.
- Create methods that rely on internal state that isn't related to the method's purpose.

DO

- Separate concerns into different classes or modules.
- Focus on one responsibility per class.
- Use interfaces and abstract classes to define contracts

03

Open-closed principle



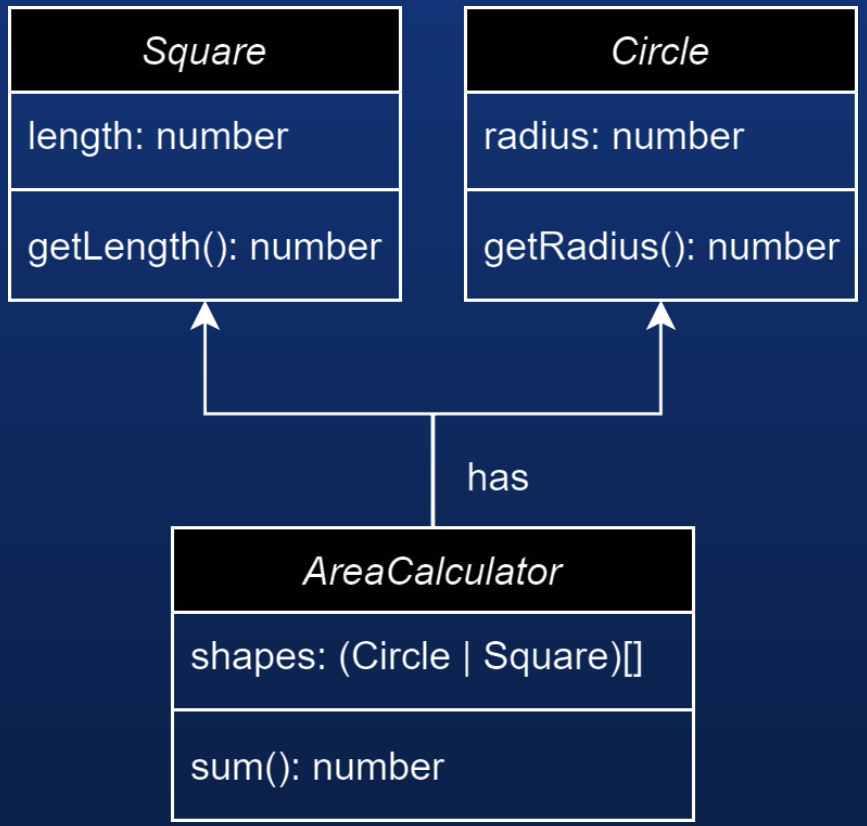
“Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the functionality of a software entity without modifying its existing code.”

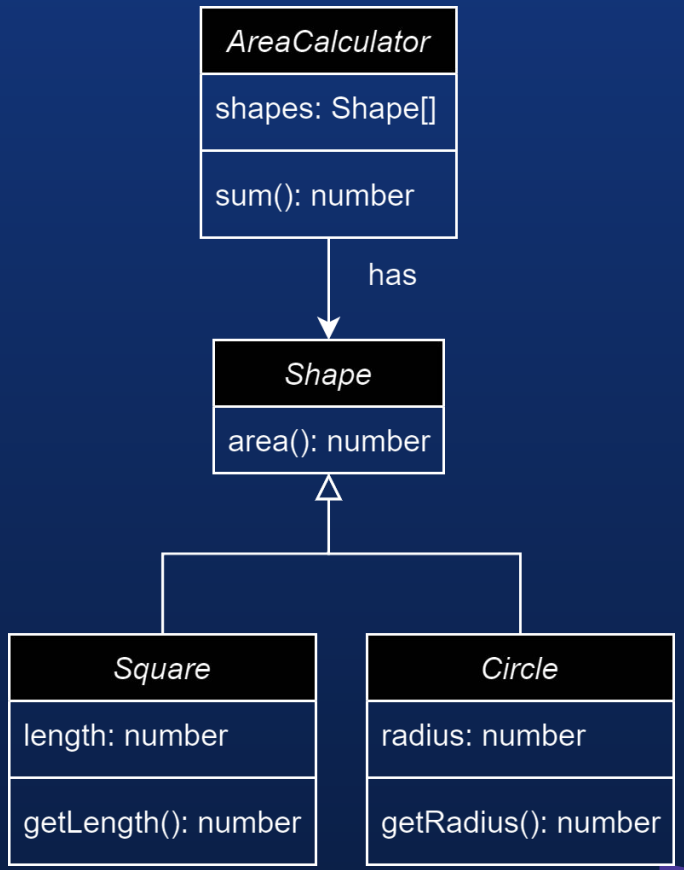
— Robert C. Martin

```ts
/**
 * @desc Represents a square
 */
class Square {
  constructor(private length: number) {}

  getLength(): number {
    return this.length;
  }
}

/**
 * @desc Represents a circle
 */
class Circle {
  constructor(private radius: number) {}

  getRadius(): number {
    return this.radius;
  }
}

/**
 * @desc Calculates the area of a set of shapes
 * @todo This class is not open for extension
 * @todo This class is not closed for modification
 */
class AreaCalculator {
  constructor(private shapes: (Circle | Square)[]) {}

  public sum(): number {
    let area: number = 0;

    for (let shape of this.shapes) {
      if (shape instanceof Square) {
        area += Math.pow(shape.getLength(), 2);
      } else if (shape instanceof Circle) {
        area += Math.PI * Math.pow(shape.getRadius(), 2);
      }
    }
    return area;
  }
}
```

```ts
/**
 * @desc Represents any shape
 */
interface Shape {
  area(): number;
}

/**
 * @desc A square is a shape with a side le
 * @implements Shape
 */
class Square implements Shape {
  constructor(private length: number) {}

  area(): number {
    return this.length * this.length;
  }
}

/**
 * @desc A circle is a shape with a radius.
 * @implements Shape
 */
class Circle implements Shape {
  constructor(private radius: number) {}

  area(): number {
    return Math.PI * this.radius * this.radius;
  }
}

/**
 * @desc Calculates the area of a set of shapes
 */
class AreaCalculator {
  constructor(private shapes: Shape[]) {}

  public sum(): number {
    let area: number = 0;
    for (let shape of this.shapes) {
      area += shape.area();
    }

    return area;
  }
}
```

[open-closed]


Square
length: number
getLength(): number
Circle
radius: number
getRadius(): number
has
AreaCalculator
shapes: (Circle | Square)[]
sum(): number


AreaCalculator
shapes: Shape[]
sum(): number
has
Shape
area(): number
Square
length: number
getLength(): number
Circle
radius: number
getRadius(): number

DOS AND DON'TS

DON'T

- Modify existing code.
- Tightly couple modules or components to each other.
- Use inheritance in ways that violate the Liskov Substitution Principle.

DO

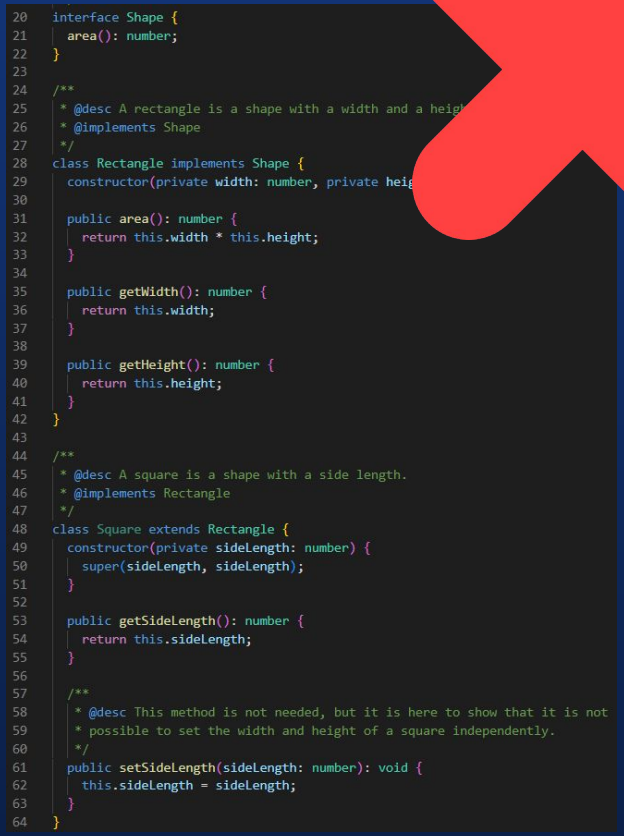
- Design modules and components that are open for extension
- Use inheritance, composition, and other techniques.
- Use interfaces and abstract classes to define the behavior of components

04

Liskov Substitution principle

“Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.”

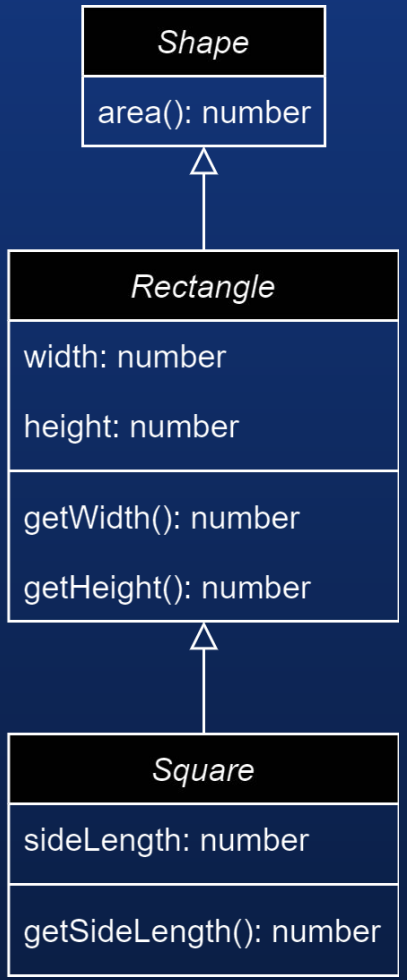
— **Robert C. Martin**

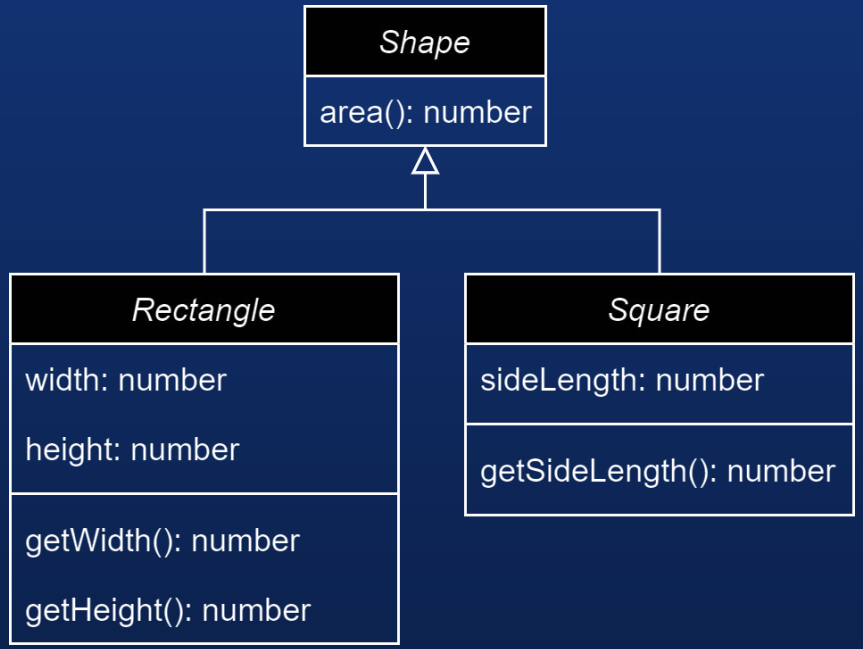


```ts
interface Shape {
  area(): number;
}

/**
 * @desc A rectangle is a shape with a width and a heig
 * @implements Shape
 */
class Rectangle implements Shape {
  constructor(private width: number, private heig

  public area(): number {
    return this.width * this.height;
  }

  public getWidth(): number {
    return this.width;
  }

  public getHeight(): number {
    return this.height;
  }
}

/**
 * @desc A square is a shape with a side length.
 * @implements Rectangle
 */
class Square extends Rectangle {
  constructor(private sideLength: number) {
    super(sideLength, sideLength);
  }

  public getSideLength(): number {
    return this.sideLength;
  }

  /**
   * @desc This method is not needed, but it is here to show that it is not
   * possible to set the width and height of a square independently.
   */
  public setSideLength(sideLength: number): void {
    this.sideLength = sideLength;
  }
}
```

```ts
/**
 * @desc Represents any shape that can have its area calculated.
 */
interface Shape {
  area(): number;
}

/**
 * @desc A rectangle is a shape with a width and a height.
 * @implements Shape
 */
class Rectangle implements Shape {
  constructor(private width: number, private height: number) {}

  public area(): number {
    return this.width * this.height;
  }

  public getWidth(): number {
    return this.width;
  }

  public getHeight(): number {
    return this.height;
  }
}

/**
 * @desc A square is a shape with a side length.
 * @implements Shape
 */
class Square implements Shape {
  constructor(private sideLength: number) {}

  public area(): number {
    return this.sideLength * this.sideLength;
  }

  public getSideLength(): number {
    return this.sideLength;
  }
}

function main() {
  const rectangle: Rectangle = new Rectangle(5, 6);
  const square: Square = new Square(5);

  console.log(rectangle.area()); // 30
  console.log(square.area()); // 25
}

if (require.main === module) {
  main();
}
```

[liskov-substitution]


Shape
area(): number
Rectangle
width: number
height: number
getWidth(): number
getHeight(): number
Square
sideLength: number
getSideLength(): number


Shape
area(): number
Rectangle
width: number
height: number
getWidth(): number
getHeight(): number
Square
sideLength: number
getSideLength(): number

DOS AND DON'TS

DON'T

- Change the behavior of a method in a subtype
- Throw exceptions that are not declared in the supertype
- Add new methods to a subtype that are not declared in the supertype

DO

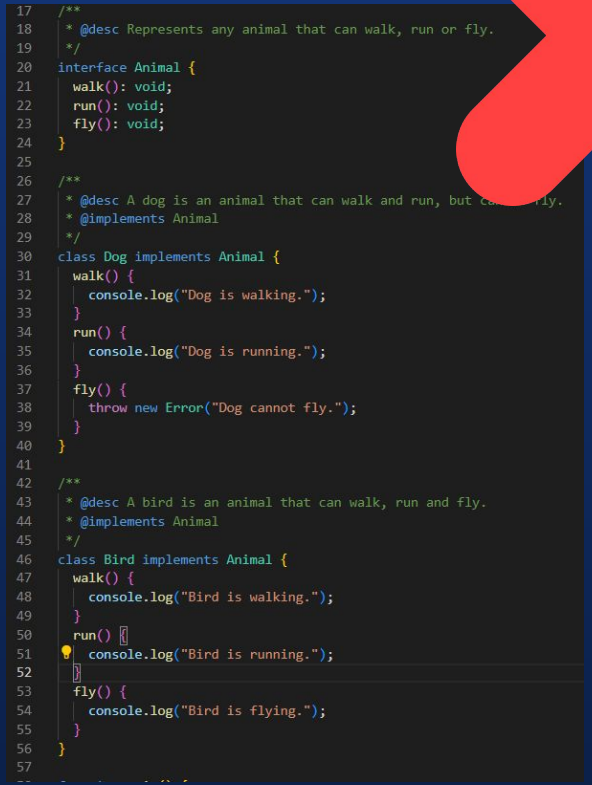
- Create subtypes that can be used interchangeably with their supertype
- Ensure that subclasses don't violate the contracts of their supertypes
- Create a clear and well-defined hierarchy of classes and interfaces

05

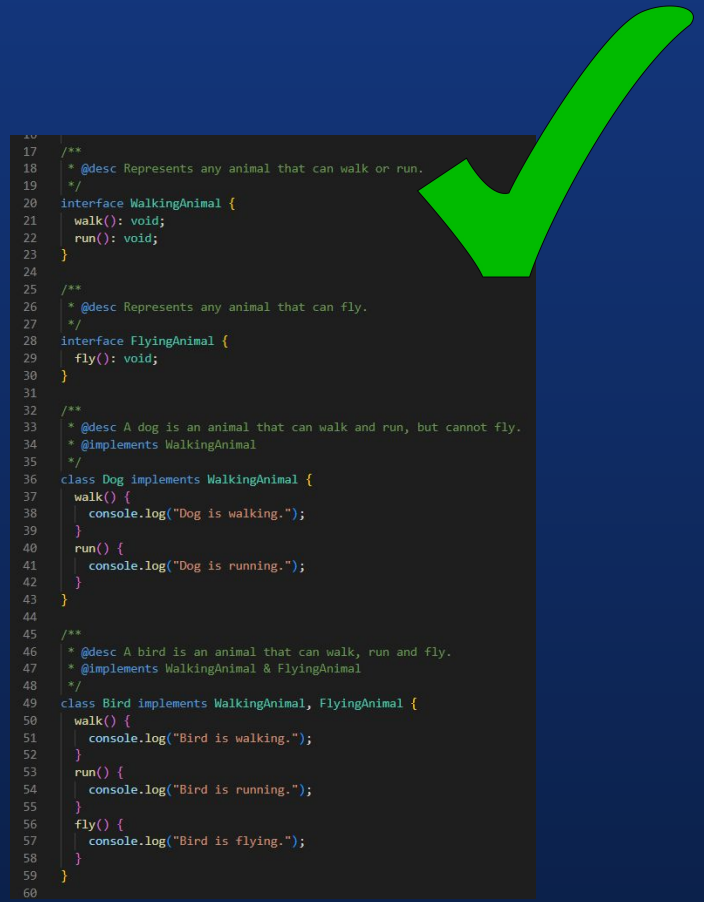
Interface segregation principle

“Clients should not be forced to depend on interfaces they do not use. This means that an interface should define only the methods that are relevant to the client that uses it.”

— **Robert C. Martin**

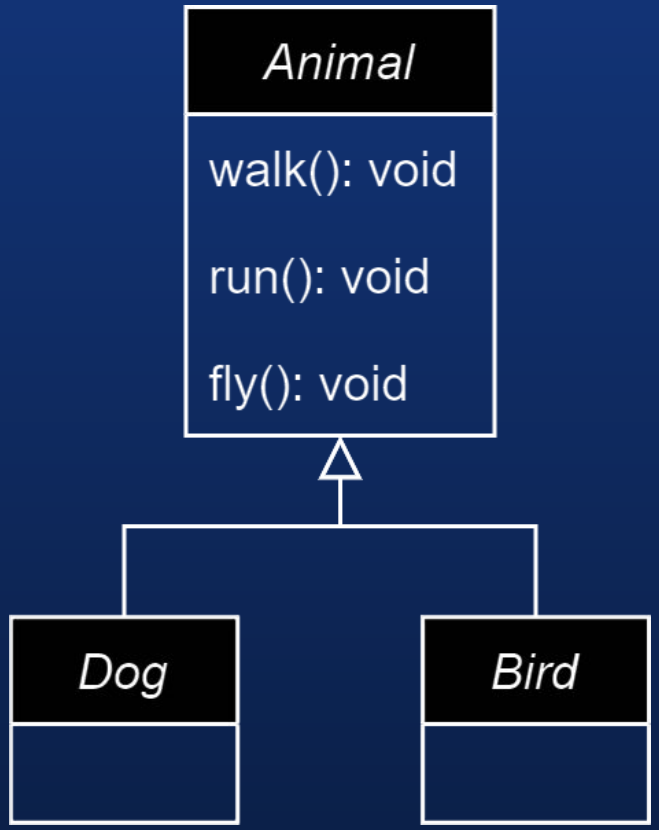


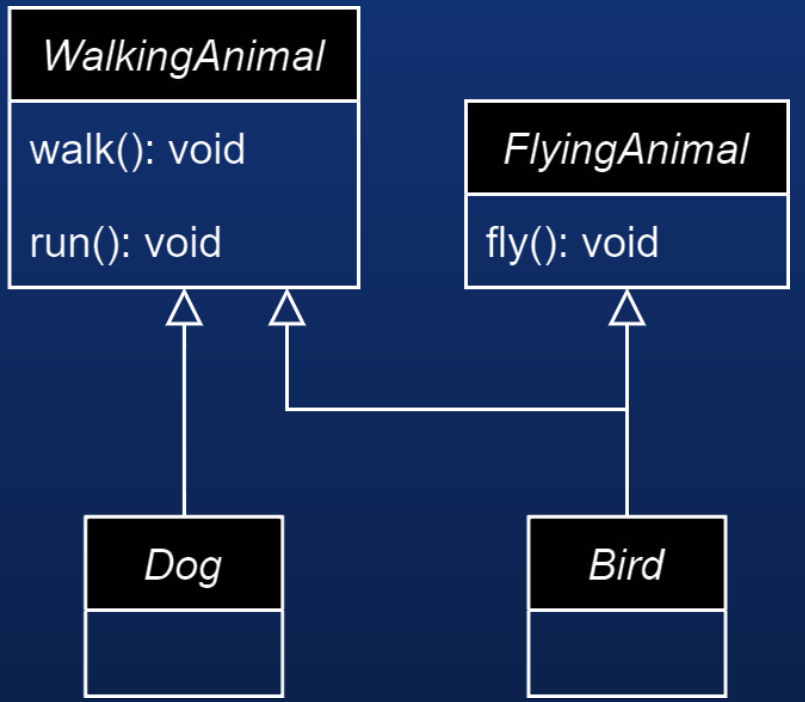
```
/**
 * @desc Represents any animal that can walk, run or fly.
 */
interface Animal {
  walk(): void;
  run(): void;
  fly(): void;
}

/**
 * @desc A dog is an animal that can walk and run, but ca… fly.
 * @implements Animal
 */
class Dog implements Animal {
  walk() {
    console.log("Dog is walking.");
  }
  run() {
    console.log("Dog is running.");
  }
  fly() {
    throw new Error("Dog cannot fly.");
  }
}

/**
 * @desc A bird is an animal that can walk, run and fly.
 * @implements Animal
 */
class Bird implements Animal {
  walk() {
    console.log("Bird is walking.");
  }
  run() {
    console.log("Bird is running.");
  }
  fly() {
    console.log("Bird is flying.");
  }
}
```



```
/**
 * @desc Represents any animal that can walk or run.
 */
interface WalkingAnimal {
  walk(): void;
  run(): void;
}

/**
 * @desc Represents any animal that can fly.
 */
interface FlyingAnimal {
  fly(): void;
}

/**
 * @desc A dog is an animal that can walk and run, but cannot fly.
 * @implements WalkingAnimal
 */
class Dog implements WalkingAnimal {
  walk() {
    console.log("Dog is walking.");
  }
  run() {
    console.log("Dog is running.");
  }
}

/**
 * @desc A bird is an animal that can walk, run and fly.
 * @implements WalkingAnimal & FlyingAnimal
 */
class Bird implements WalkingAnimal, FlyingAnimal {
  walk() {
    console.log("Bird is walking.");
  }
  run() {
    console.log("Bird is running.");
  }
  fly() {
    console.log("Bird is flying.");
  }
}
```

interface-segregation


Animal
walk(): void
run(): void
fly(): void
Dog
Bird


WalkingAnimal
walk(): void
run(): void
FlyingAnimal
fly(): void
Dog
Bird

DOS AND DON'TS

DON'T

- Define large, general-purpose interfaces that contain a large number of methods.
- Force a class to implement methods that it does not need or use.
- Create interfaces that are too specific or too tightly coupled to a particular implementation, making it difficult to swap out the implementation with another one.

DO

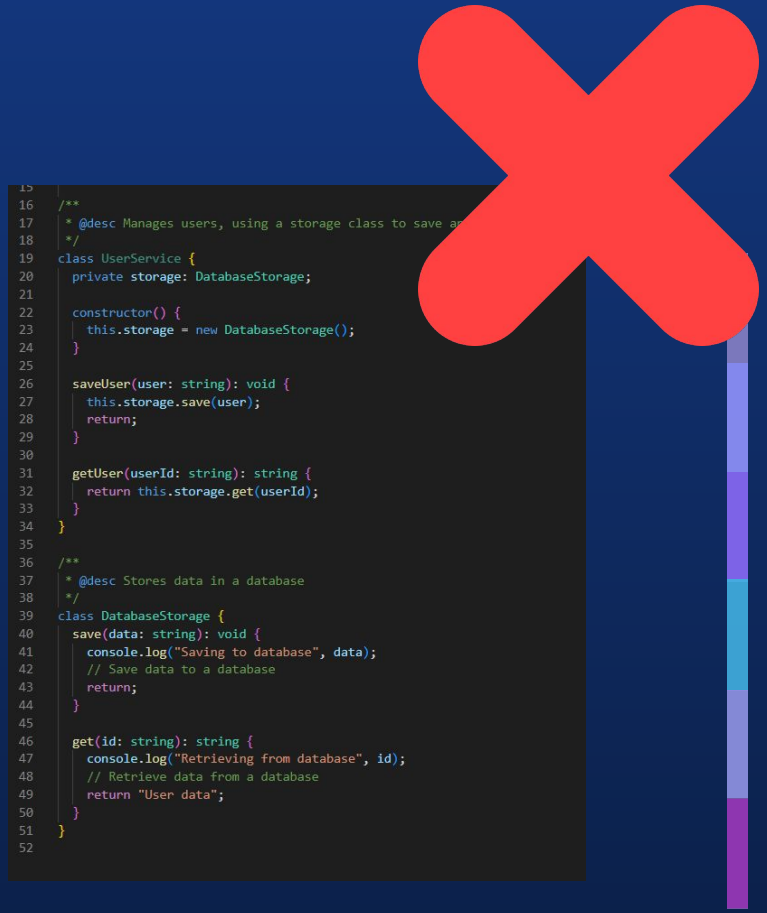
- Create interfaces that are specific to the needs of a particular class or module.
- Define small, cohesive interfaces that contain only the methods needed by the classes that implement them.
- Implement interfaces that contain more methods than needed only if you want to provide a common interface for multiple related classes.

06

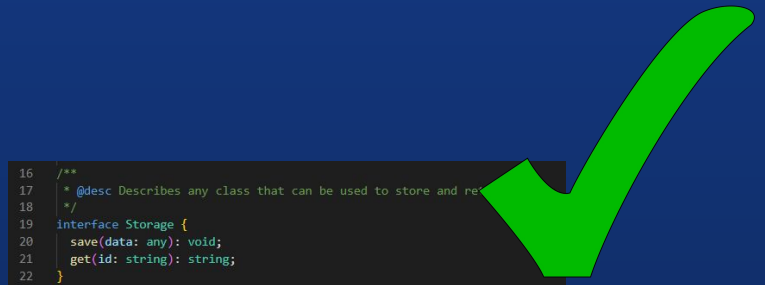
Dependency inversion principle

“Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.”

— **Robert C. Martin**

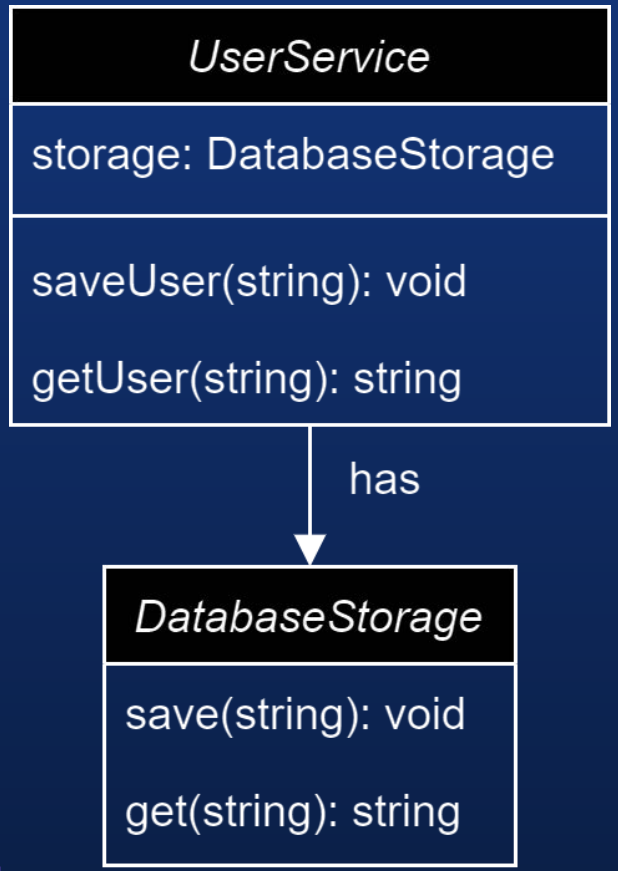


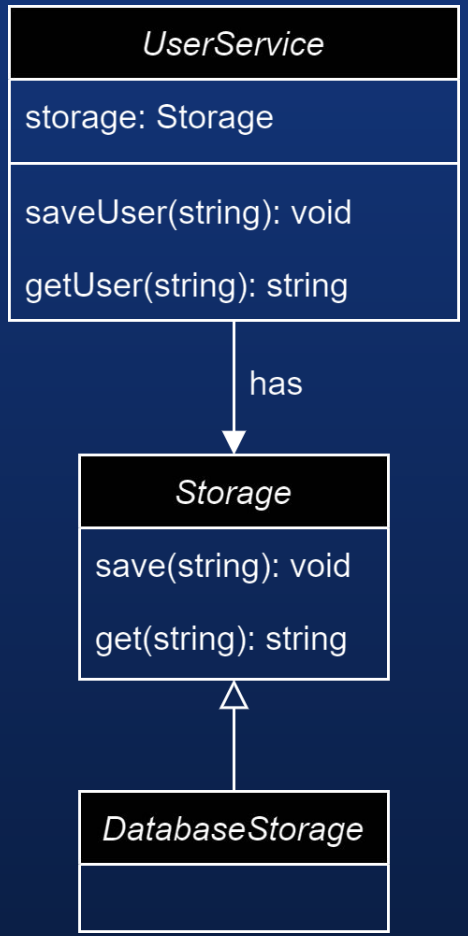
```
/**
 * @desc Manages users, using a storage class to save a
 */
class UserService {
  private storage: DatabaseStorage;

  constructor() {
    this.storage = new DatabaseStorage();
  }

  saveUser(user: string): void {
    this.storage.save(user);
    return;
  }

  getUser(userId: string): string {
    return this.storage.get(userId);
  }
}

/**
 * @desc Stores data in a database
 */
class DatabaseStorage {
  save(data: string): void {
    console.log("Saving to database", data);
    // Save data to a database
    return;
  }

  get(id: string): string {
    console.log("Retrieving from database", id);
    // Retrieve data from a database
    return "User data";
  }
}
```



```
/**
 * @desc Describes any class that can be used to store and re
 */
interface Storage {
  save(data: any): void;
  get(id: string): string;
}

/**
 * @desc Manages users, using a storage class to save and retrieve data
 */
class UserService {
  constructor(private storage: Storage) {}

  saveUser(user: string): void {
    this.storage.save(user);
    return;
  }

  getUser(userId: string): string {
    return this.storage.get(userId);
  }
}

/**
 * @desc Stores data in a database
 */
class DatabaseStorage implements Storage {
  save(data: string): void {
    console.log("Saving to database", data);
    // Save data to a database
    return;
  }

  get(id: string): string {
    console.log("Retrieving from database", id);
    // Retrieve data from a database
    return "User data";
  }
}
```

[dependency-inversion]


UserService
storage: DatabaseStorage
saveUser(string): void
getUser(string): string
has
DatabaseStorage
save(string): void
get(string): string


UserService
storage: Storage
saveUser(string): void
getUser(string): string
has
Storage
save(string): void
get(string): string
DatabaseStorage

DOS AND DON'TS

DON'T

- Hard code dependencies.
- Use a specific implementation.
- Create circular dependencies.

DO

- Define interfaces for any external dependencies that your classes rely on.
- Use dependency injection to inject dependencies into classes.
- Make sure that high-level modules do not depend on low-level modules, and that both depend on abstractions.

REFERENCES

- SOLID: The First 5 Principles of Object Oriented Design
- 2022-2023 - T1F - TS OOP
- Introduction to TypeScript
- TypeScript Tutorial
- TypeScript: Documentation - Classes

THANKS

Do you have
any questions?