Advance Software Engineering

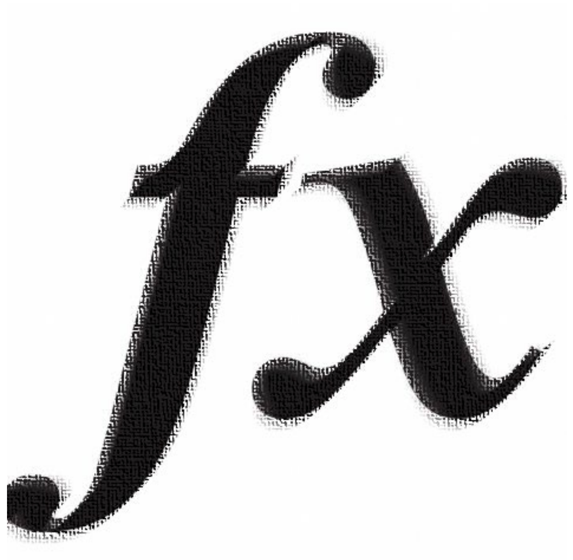
Clean Code Functions, Structures, and Exception

By:

Dr. Salwa Osama

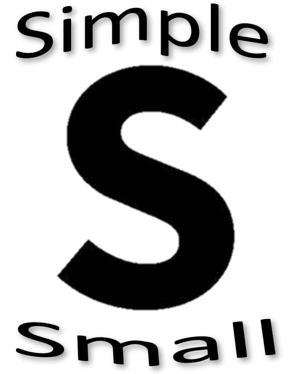
Functions

Clean Code



Keep simple, small functions with meaningful names

1



- The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.
- Remember that your code must be clean enough to be easily readable without spending too much time trying to guess what a function does.
- Smaller functions are easier to read and to understand.

Long Method

- A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- To reduce the length of a method body, use **Extract Method**.
-

Do One Thing

2

- Functions Should Do One Thing.
- Functions Should Do It Well.
- Functions Should Do It Only.
- If you follow this rule, it is guaranteed that they will be small.

Do One Thing

- The only thing that function does should be stated in its name.
- Sometimes it is hard to look at the function and see if it is doing multiple things or not
- One good way to check is to try to extract another function with a different name.
- If you can find it, that means it should be a different function.

Example – Long Method Before Refactoring

```
public static void addDataToFile(String path,Person person)  {
        FileInputStream readData = new FileInputStream(path);
        ObjectInputStream readStream = new ObjectInputStream(readData);
        ArrayList people = (ArrayList<Person>) readStream.readObject();
        readStream.close();
        people.add(person);
        FileOutputStream writeData = new FileOutputStream(path);
        ObjectOutputStream writeStream = new ObjectOutputStream(writeData);
        writeStream.writeObject(people);
        writeStream.flush();
        writeStream.close();
        System.out.println(people.toString());
   }
```

Extract methods from long method

```
private static void PrintPersonDataFile(ArrayList people ) {
        System.out.println(people.toString());
    }
    private static void writeDataTofile(String path, ArrayList people) {
        FileOutputStream writeData = new FileOutputStream(path);
        ObjectOutputStream writeStream = new ObjectOutputStream(writeData);
        writeStream.writeObject(people);
        writeStream.flush();
        writeStream.close();
    }
    private static ArrayList readDataFromFile(String path) {
        FileInputStream readData = new FileInputStream(path);
        ObjectInputStream readStream = new ObjectInputStream(readData);
        ArrayList  people = (ArrayList<Person>) readStream.readObject();
        readStream.close();
        return people;
    }
```

Example – Long Method After Refactoring

```
public static void addDataToFile(String path,Person person) {
        ArrayList people = readDataFromFile(path);
        people.add(person);
         writeDataTofile(path, people);
         PrintPersonData(people);
    }
```

Do One Thing

- For example, addDataToFile function has its role to add the person to data file not print or display any data or add the item to Arraylist.
- So, remove PrintPersonData(people) and refactor it

```
public static void addDataToFile(String path,Person person) {
        ArrayList people = readDataFromFile(path, person);
        people.add(person);
        writeDataTofile(path, people);
         PrintPersonData(people);
    }
```

Refactoring addDataToFile() Method

- Move the code section for adding a person to the people list to the new method addDataToList and also move the printing function out.
- Change the parameter from adding person only to actually responsibility to add a list of people to file

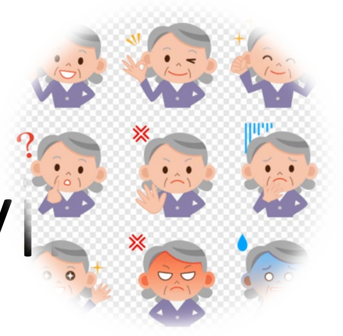
```
public static ArrayList addDataToList(Person person,ArrayList people){
        people.add(person);
        return people;
  }
public static void addDataToFile(String path,ArrayList people){
        writeDataTofile(path, people);
  }
```

Calling all Methods at Main(User Level)

```
public static void main(String[] args) {
        Person p3 = new Person("ahmed", "hesham", 1990);
        String filePath="peopledata.ser";
        ArrayList people = readDataFromFile(filePath);
        addDataToList(p3,people);
        addDataToFile(filePath,people);
        ArrayList afterAddpeople = readDataFromFile(filePath);
        PrintPersonDataFile(afterAddpeople);
    }
```

Remember

Use Intention Revealing Names - Improvi



```
public List<int[]> [redacted] {
   List<int[]> flaggedCells = new ArrayList<int[]>();
   for (int[] cell : gameBoard)
     if (cell[STATUS_VALUE] == FLAGGED)
       flaggedCells.add(cell);
   return flaggedCells;
}
```

What’s the name of the function?

Remember

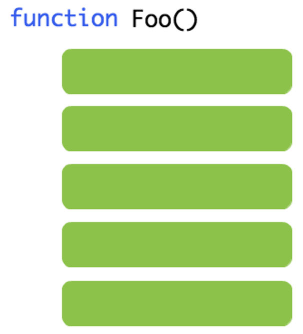
Use Intention Revealing Names - Example

We can go further and write a simple class for cells instead of using an array of ints.

It can include an intention-revealing function (call it isFlagged) to hide the magic numbers. It results in a new version of the function:

```
public List<Cell> getFlaggedCells() {
        List<Cell> flaggedCells = new ArrayList<Cell>();
        for (Cell cell : gameBoard) {
                if (cell.isFlagged())
                flaggedCells.add(cell);
        }
        return flaggedCells;
}
```

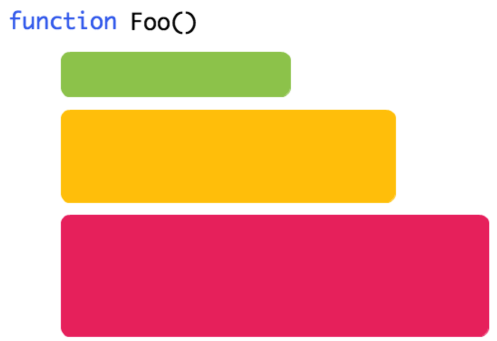
One Level of Abstraction per Function



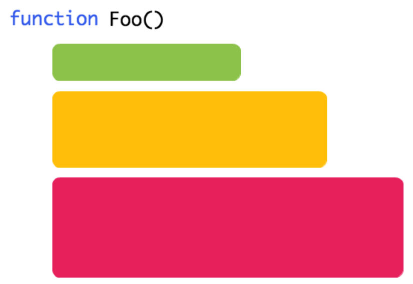
3

Concept in OOPS. hiding the “how” part and expose “what”

- Also called Single Level of Abstraction
- A method is a form of **abstraction**
- In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our function are all at the same level of abstraction.
- Mixing levels of abstraction within a function is always confusing.
- Readers may not be able to tell whether a particular expression is an essential concept or a detail.



Example



```
private static void printNumbersOneToThirtyFive(){

 printNumbersOneToTen();  //high level of abstraction
                                        /*low level of abstraction */
 for(int i=11; i < 21; i++) {
   System.out.println(i);
 }
 System.out.println(21);
 if(22 % 11 == 0) {
   System.out.println(22);
   for(int i=23; i < 31; i++) {
       System.out.println(i);
       if(i == 30) {
           for(int j=31; j < 36; j++) {
           System.out.println(j);}}}}  }
}
```

```
 private static void printNumbersOneToTen() {
   for(int i=1; i < 11; i++) {
     System.out.println(i); }}}
```

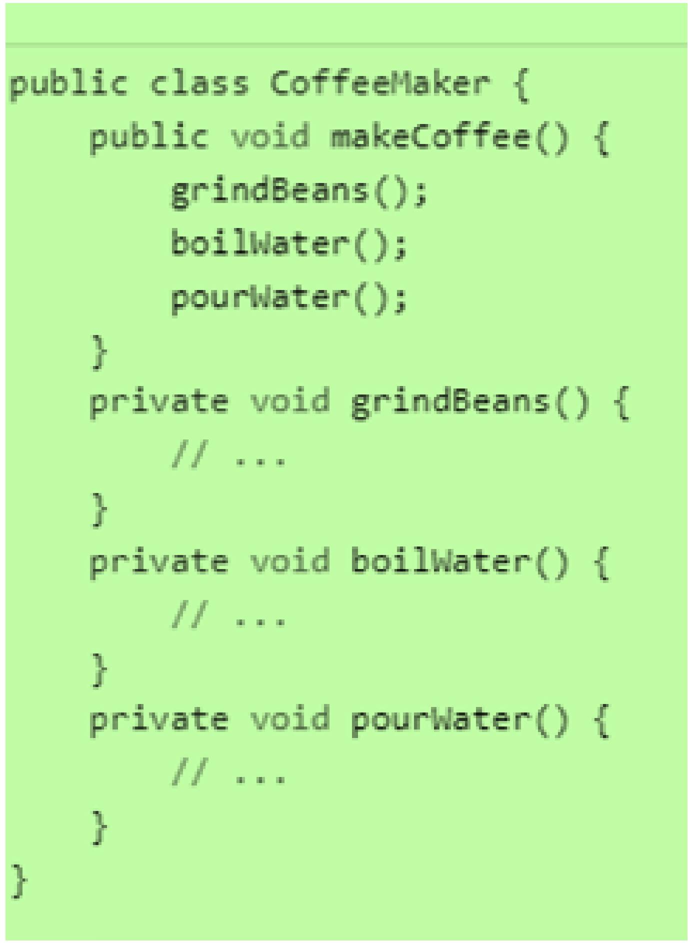
One Level of Abstraction per Function

- The method has same level of abstraction

```
public static void main(String[] args) {
  printNumbersOneToThirtyFive();
}

private static void printNumbersOneToThirtyFive() {
  printNumbersOneToTen();
  printNumbersElevenToTwenty();
  printNumberTwentyOne();
  printNumbersTwentyTwoToThirtyFive();
}
```

One Level of Abstraction per Function

- The method has same level of abstraction



```
public class CoffeeMaker {
    public void makeCoffee() {
        grindBeans();
        boilWater();
        pourWater();
    }
    private void grindBeans() {
        // ...
    }
    private void boilWater() {
        // ...
    }
    private void pourWater() {
        // ...
    }
}
```

4

The fewer function arguments

- Function arguments are troublesome and make it harder to understand the function. It is best to build zero-argument functions, then one-argument and two-argument functions.
- Try to avoid functions with 3 arguments, and functions with multiple arguments (3+) should not be used at all.
- For example, writeText(text) function is way easier to understand than writeText(outputStream, text).
- You can always get rid of the outputStream argument by defining it as a class field.

The fewer function arguments

- When a given function requires too many arguments, some of them might probably be placed in separate classes. For instance:

```
public void sendRequest(String ip, int port,String user, String password,String body);
```

```
public void sendRequest(Request request);
```

Flag Arguments

4

- Flag arguments are ugly.
- Passing a Boolean into a function is a truly terrible practice.
- It means the function does more than one thing.
- It does one thing if the flag is true and another if the flag is false!
- We should have split the function into two.

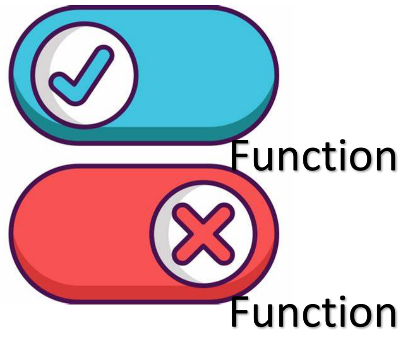
Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

Flag Arguments

```
Public void printAddress(Boolean long)
{
If(long)
//code for printing Long address
else
//code for pronting Short address
}
```

```
Public void printshortAddress()
{
//code for printing short address
}
Public void printLongAddress()
{
//code for pronting Long address
}
```

Keep small, Blocks and if-else



4

- The blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call.
- Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.
- This also implies that functions should not be large enough to hold nested structures.
- Therefore, the indent level of a function should not be greater than one or two.
- This, of course, makes the functions easier to read and understand.

Replace Conditional With Polymorphism

5

- If you have code that splits a flow or acts differently based on some condition with limited values, with constructs like if or switch statements, then that code naturally **violates Open Closed principle** because to add a new conditional flow you will have to modify that class.
- Such a conditional check, if not designed correctly, is likely to be duplicated at multiple places.
- This principle suggests to use polymorphism to replace the conditional.

Example Vehicle Class

```
public class Vehicle {
   private Type type;
   enum Type { BICYCLE,BIKE,CAR,BUS}
   enum ParkingSpotSize {SMALL,MEDIUM,LARGE,XL}
   public ParkingSpotSize spaceRequiredForParking() {
      switch (type) {
         case BICYCLE:
            return ParkingSpotSize.SMALL;
         case BIKE:
            return ParkingSpotSize.MEDIUM;
         case CAR:
            return ParkingSpotSize.LARGE;
         case BUS:
            return ParkingSpotSize.XL; }}
   public BigDecimal parkingCharges() {
      switch (type) {
         case BICYCLE:
            return BigDecimal.valueOf(10.00);
         case BIKE:
            return BigDecimal.valueOf(20.00);
         case CAR:
            return BigDecimal.valueOf(50.00);
         case BUS:
            return BigDecimal.valueOf(100.00);}}}
```

Example - Vehicle Class

- Many conditionals (Hard for readability).
- Violate the rule of open close principle
- We must change these places when a new vehicle type is to be added.
- We must change these charges when a new vehicle type is to be added.
- Violate the rule of must do one thing

Solution

- Create interface class Vehicle
- Adding new vehicle type involve adding new implementation.
- Separate each Vehicle type to separated implementation class

```
public interface Vehicle {
    ParkingSpotSize spaceRequiredForParking();
    BigDecimal parkingCharges();
    enum ParkingSpotSize {SMALL,MEDIUM,LARGE,XL
 }
}
```

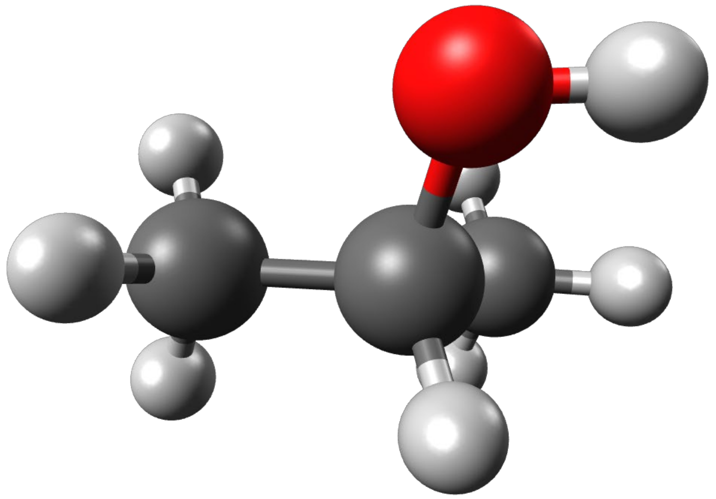
Solution

- Create implantation class for Car, Bike, Bus, Bicycle and for each new Vehicle type
- For example, the implementation class for car will be

```
public class Car implements Vehicle {
    @Override
    public ParkingSpotSize spaceRequiredForParking()
{
        return ParkingSpotSize.LARGE;    }
    @Override
    public BigDecimal parkingCharges() {
        return BigDecimal.valueOf(50.00);  }
}
```

Objects and Structure

CleanCode



Data Abstraction:

1

- **Abstraction** means hiding implementation.
- Hiding implementation isn't means making the variables private and using **getters** and **setters** to access these variables.
- Rather it means providing abstract interfaces (abstract ways) that allow its users to manipulate the data, without having to know its implementation.

Data Abstraction:

```java
public class Shape {
    private double width;
    private double height;

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }
}
```

This is not Abstraction

Data Abstraction:

- Also, the following code not represents **Abstraction**. But, it uses concrete terms to communicate the fuel level of a vehicle, because these methods sure are just **accessors** (**getters** or **get methods**) of variables.

```
public interface Vehicle {
    double getFuelTankCapacityInGallons();
    double getGallonsOfGasoline();
}
```

This is not Abstraction

Data Abstraction:

It looks like Rule 3 “One Level of Abstraction per Function” in function

- The following code represents **Abstraction**. Which it does the same purpose of the previous code but with the abstraction of percentage. So, in this abstract case you do not at all know anything about the form of the data (internal details).

```
public interface Vehicle {
    double getPercentFuelRemaining();
}
```

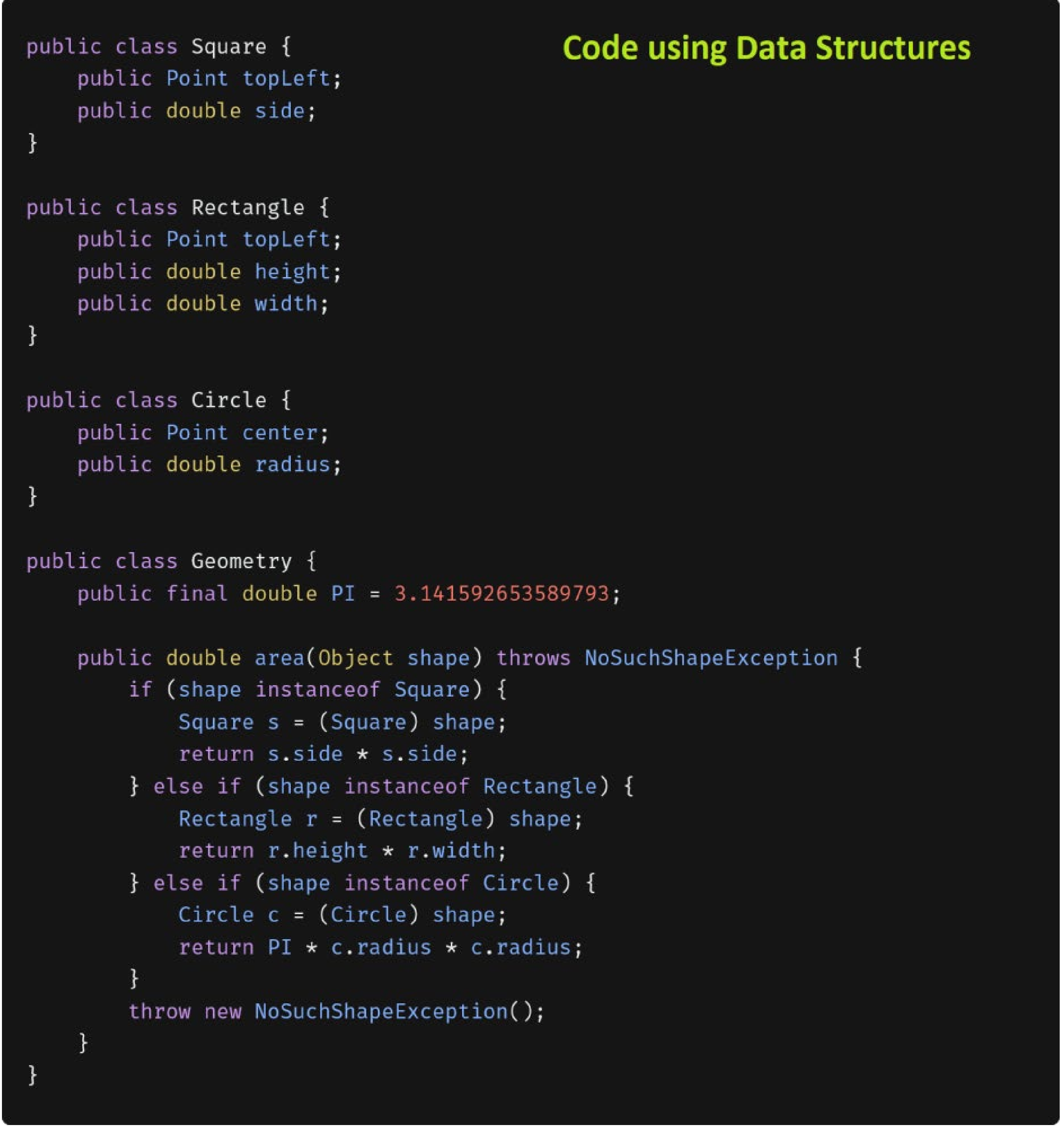
This is Abstraction

Because we do not want to expose (show) the details of our data. Rather we want to express our data in abstract terms (hiding details).

Data/Object Anti-Symmetry

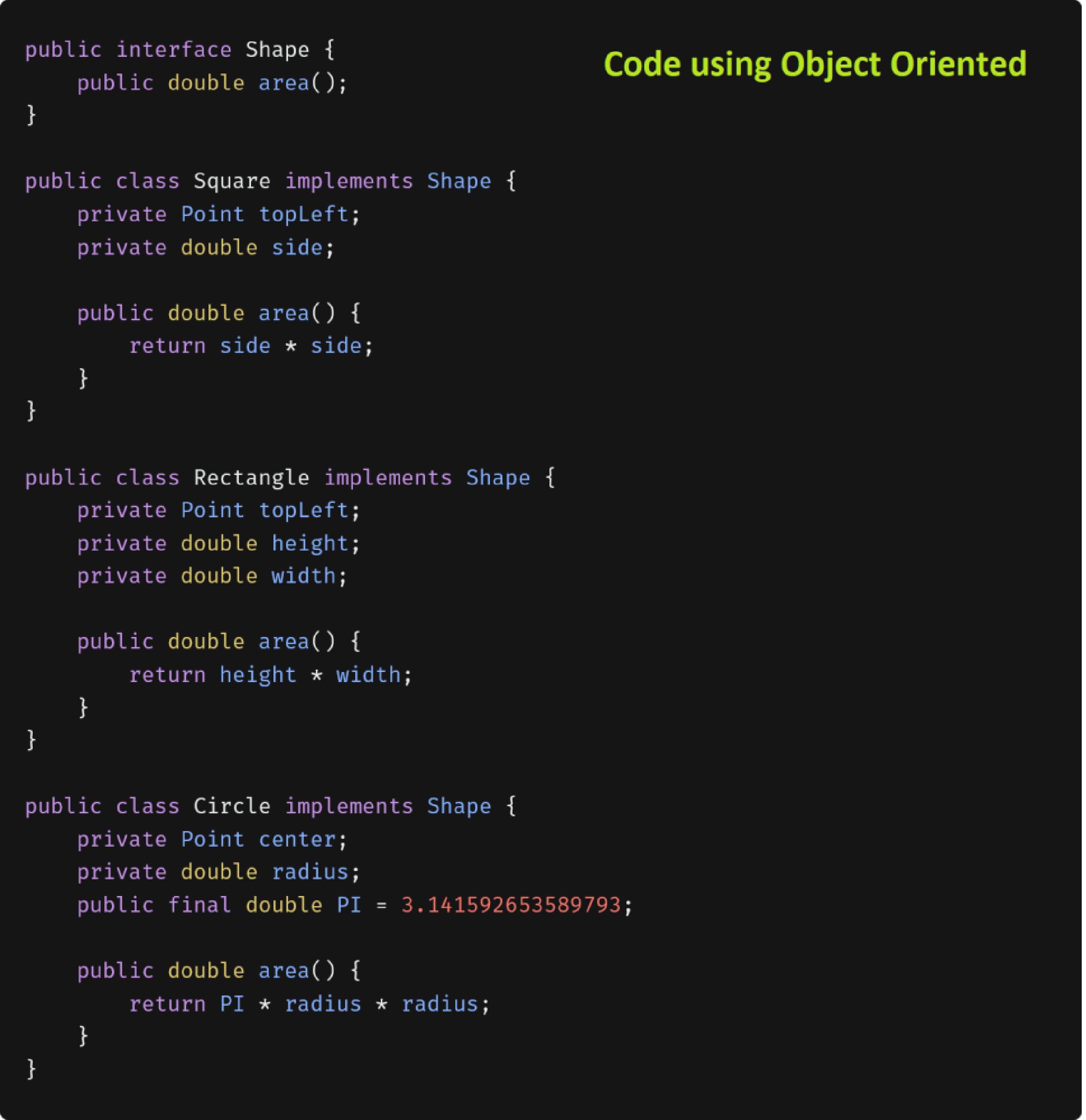
2

- **Objects**: hide their data (be private) and have functions to operate on that data.
- **Data Structures**: show their data (be public) and have no functions.
- Look at the following **Procedural Shape** Example *(***code using data structures**). The **Geometry** class operates on the three shape classes. The **shape classes** are simple data structures without any behavior (function). All the behavior is in the **Geometry** class.


Code using Data Structures

```java
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuchShapeException {
        if (shape instanceof Square) {
            Square s = (Square) shape;
            return s.side * s.side;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.height * r.width;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

- Consider what would happen if I add a **perimeter()** function to **Geometry.** The **shape classes** would be unaffected. On the other hand, if I add a new shape (new data structure), I must change all the functions in **Geometry** to deal with it.

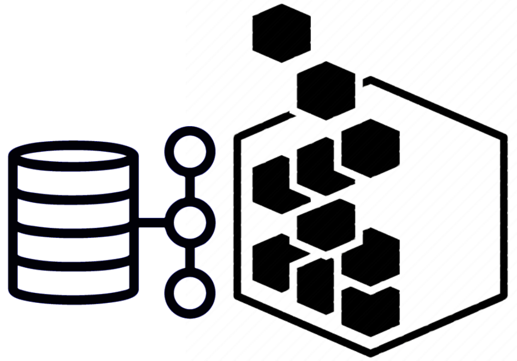


Code using Object Oriented

```java
public interface Shape {
    public double area();
}

public class Square implements Shape {
    private Point topLeft;
    private double side;

    public double area() {
        return side * side;
    }
}

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

    public double area() {
        return height * width;
    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

- Now, look at the following **Polymorphic Shapes** Example (**code using object oriented**). Here the **area()** function is polymorphic.
  No **Geometry** class is necessary. So if I add a new shape, none of the existing functions are affected, but if I add a new function all of the shapes must be changed.

Data/Object Anti-Symmetry



- Then, we note from the previous that the two concepts are opposites:
- ***Procedural code*** *(code using data structures),*
    - Makes it easy to add new functions without changing the existing data structures.
    - Makes it hard to add new data structures because all the functions must change.
- ***OO code*** (code using object oriented),
    - Makes it hard to add new functions because all the existing classes must change.
    - Makes it easy to add new classes without changing existing functions.
- So, the things that are hard for **OO** are easy for **Procedural**, and vice-versa.

Data/Object Anti-Symmetry

Q: when OO code is more appropriate?

A. *In any complex system, because we will having to add more data types rather than new functions.*

Q: when procedural code is more appropriate?

A. on the other hand, *when we'll want to add new functions more rather than data types.*

The Law of Demeter(LoD):

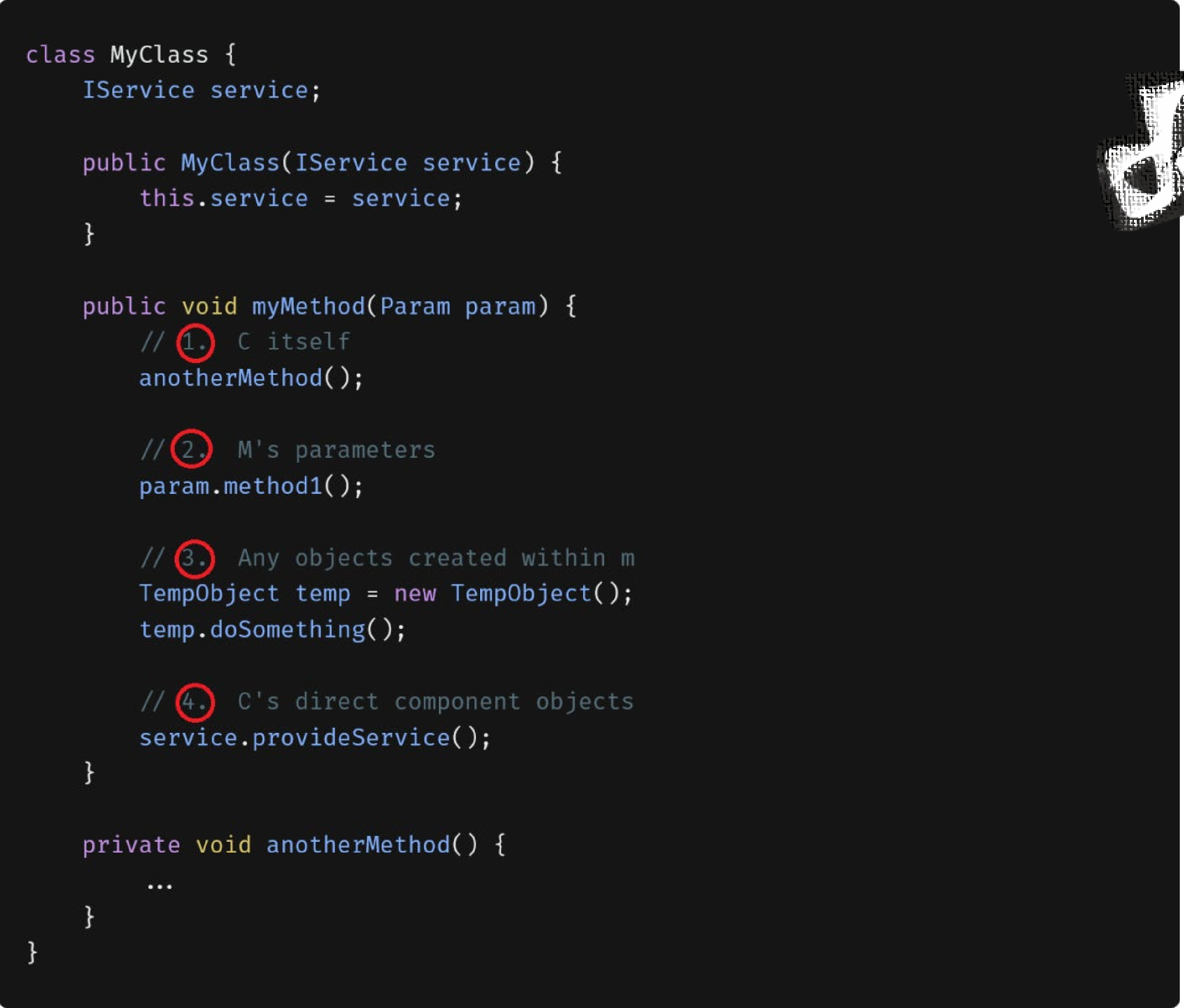
3

- The *Law of Demeter* or **principle of least knowledge** is a design guideline for developing software, says the module should not know about the inner details of the objects it manipulates. In other words, a software component or an object should not have the knowledge of the internal working of other objects or components. the LoD is a specific case of loose coupling.

The Law of Demeter(LoD):

- The Law of Demeter says that a method ***M*** of class ***C*** should only call the methods of these
  - **C** itself.
  - **M**'s parameters.
  - Any objects created within **M.**
  - **C**'s direct component objects.
  - A global variable, accessible by **C**, in the scope of **M.** (static fields in Java)

The Law of Demeter is often described this way:
“Only talk to your immediate friends.”
or, put another way:
“Don't talk to strangers.”



```java
class MyClass {
    IService service;

    public MyClass(IService service) {
        this.service = service;
    }

    public void myMethod(Param param) {
        // 1.  C itself
        anotherMethod();

        // 2.  M's parameters
        param.method1();

        // 3.  Any objects created within m
        TempObject temp = new TempObject();
        temp.doSomething();

        // 4.  C's direct component objects
        service.provideService();
    }

    private void anotherMethod() {
        ...
    }
}
```

tself.

- **M**'s parameters.
- Any objects created within **M.**
- **C**'s direct component objects.
- A global variable, accessible by **C**, in the scope of **M.** (static fields in Java)

The Law of Demeter(LoD):

- *This* is a kind of code that violates the **Law of Demeter**, for example:

```
public void showData(Car car) {
    printStreet(car.getOwner().getAddress().getStreet());
    ...
}
```

- **Why it violate the law?** The method received the parameter **car**, so all method calls on this object are allowed. But, calling any methods (in this case **getAddress()** and **getStreet()**) on the object returned by **getOwner()** is not allowed.

The Law of Demeter(LoD):

- The following “refactored” code is better, but, it still violates the Law:

```
Owner owner = car.getOwner();
Address ownerAddress = owner.getAddress();
Street ownerStreet = ownerAddress.getStreet();
```

- Because, these objects **owner**, **ownerAddress** and **ownerStreet** are still not covered by any of the previous five rules, therefore no methods should be called on them.

Error handling

Clean Code

```
Void Main()
{
    (HttpStatus status, string content) = DownloadContent("https://code4it.dev");
    if (status == HttpStatus.Ok)
    {
        // do something with the content
    }
    else if (status == HttpStatus.NotFound)
    {
        // do something else
    }
    // and so on
}

public (HttpStatus, string) DownloadContent(string url)
{
    // do something
}

// Define other methods and classes here

public enum HttpStatus
{
    Ok,
    NotFound,
    Unauthorized,
    GenericError
}
```

- In Uncle Bob's opinion, **we should always prefer exceptions over status codes when returning values.**

1

- In example, when downloading a string, returns both the status code and the real content of the operation.

```
void Main()
{
    try
    {
        string content =
DownloadContent("https://code4it.dev");
    }
    catch (NotFoundException nfe) {/*do something*/}
    catch (UnauthorizedException ue) {/*do something*/}
    catch (Exception e) {/*do something else*/}
    }
}

public string DownloadContent(string url)
{
    // do something
    return "something";
    // OR throw NotFoundException
    // OR throw UnauthorizedException
    // OR do something else
}
```

- When you use status codes, you have to manually check the result of the operation with a switch or an if-else. So, if the caller method forgets to check whether the operation was successful, you might incur in unexpected execution paths.
- Now, let's transform the code and use exceptions instead of status codes:

What are the pros and cons of using exceptions over status codes?

- PRO: the "happy path" is easier to read
- PRO: every time you forget to manage all the other cases, you will see a meaningful exception instead of ending up with a messy execution without a clue of what went wrong
- PRO: the execution and the error handling parts are strongly separated, so you can easily separate the two concerns
- CON: you are defining the execution path using exceptions instead of status (which is bad...)
- CON: every time you add a try-catch block, you are adding overhead on the code execution.

Exceptions over status codes

- The reverse is obviously valid for status codes.
- So, what to do? Well, exceptions should be used in exceptional cases, so if you are expecting a range of possible status that can be all managed in a reasonable way, go for enums. If you are expecting an "unexpected" path that you cannot manage directly, go for exceptions!

TDD can help you handling errors

2

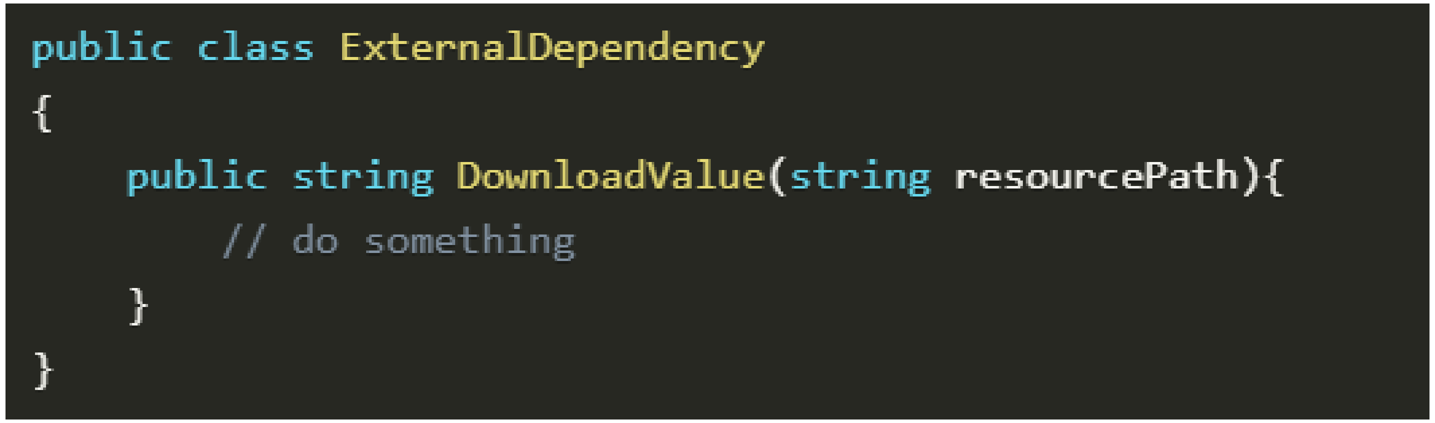
- Don't forget that error handling must be thoroughly tested. One of the best ways is to write your tests first: this will help you figuring out what kind of exceptions, if any, your method should throw, and which ones it should manage.

- Once you have written some tests for error handling, add a try-catch block and start thinking to the actual business logic: you now can be sure that you're covering errors with your tests.

Wrap external dependencies to manage their exceptions

3

- This method throws some custom exceptions, like ResourceNotFoundException, InvalidCredentialsExceptions and so on.

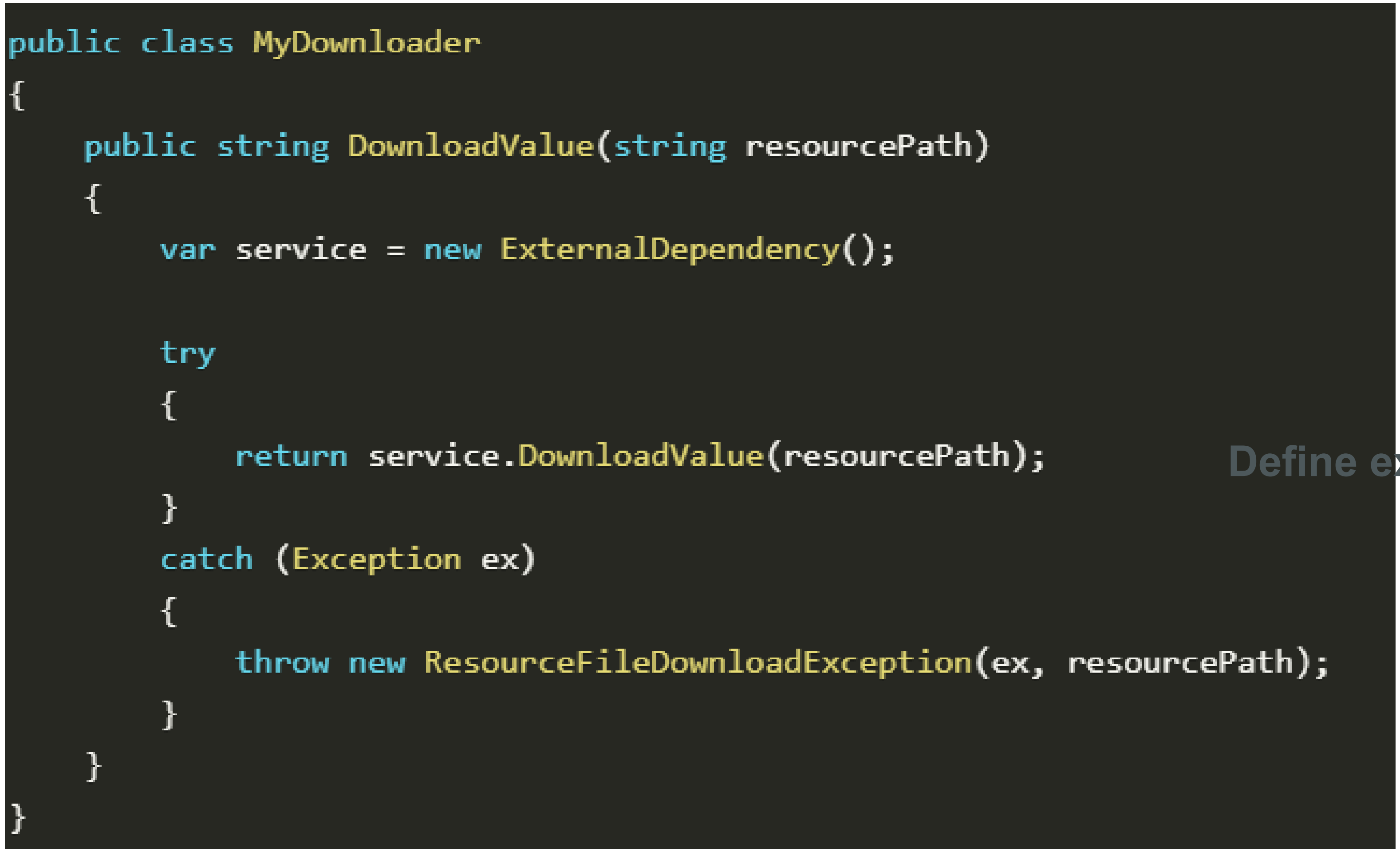


```
public class ExternalDependency
{
    public string DownloadValue(string resourcePath){
        // do something
    }
}
```

```
void Main()
{
    ExternalDependency service = CreateExternalService();

    try
    {
        var value = GetValueToBeDowloaded();
        service.DownloadValue(value);
    }
    catch (ResourceNotFoundException rnfex)
    {
        logger.Log("Unable to get resource");
        ManageDownloadFailure();
    }
    catch (InvalidCredentialsExceptions icex)
    {
        logger.Log("Unable to get resource");
        ManageDownloadFailure();
    }
    catch (Exception ex)
    {
        logger.Log("Unable to complete the operation");
        DoSomethingElse();
    }
}
```

- This seems reasonable, but what does it imply?
- First of all, we are repeating the same error handling in multiple catch blocks.
- Here I have only 2 custom exceptions, but think of complex libraries that can throw tens of exceptions.
- Also, what if the library adds a new Exception? In this case, you should update every client that calls the DownloadValue method.

wrap the External Dependency class:



```
public class MyDownloader
{
    public string DownloadValue(string resourcePath)
    {
        var service = new ExternalDependency();

        try
        {
            return service.DownloadValue(resourcePath);
        }
        catch (Exception ex)
        {
            throw new ResourceFileDownloadException(ex, resourcePath);
        }
    }
}
```

- Why didn't we add another types of exception?

Define exception types thinking of the clients

Define exception types thinking of the clients

- Why haven't I exposed multiple exceptions, but I chose to throw only a ResourceFileDownloadException? Because you should define your exceptions thinking of how they can be helpful to their caller classes.

- I could have thrown other custom exceptions that mimic the ones exposed by the library, but they would have not brought value to the overall system. In fact, the caller does not care that MyDownloader failed because the resource does not exist, but it cares only that an error occurred when downloading a resource. It doesn't even care that that exception was thrown by MyDownloader!

- So, when planning your exceptions, think of how they can be used by their clients rather than where they are thrown.

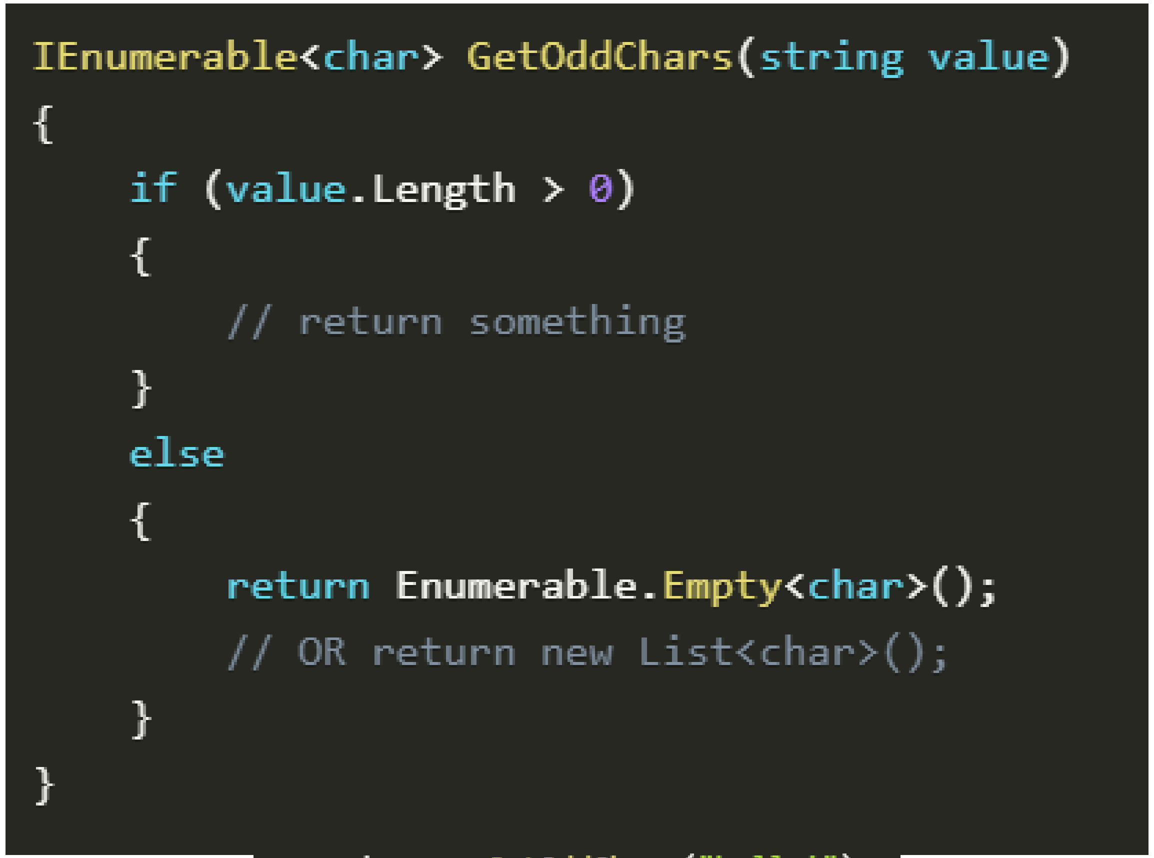
Fighting the devil: null reference

3

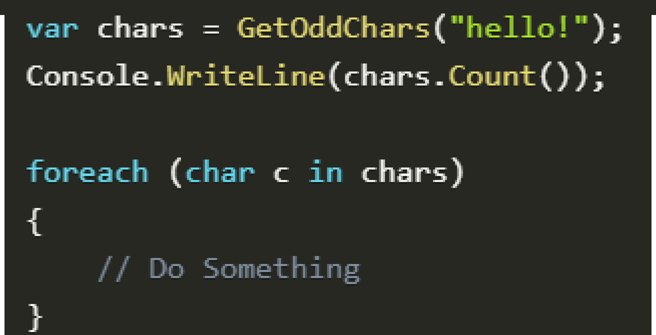
- Everyone fights with null values. If you refence a null value, you will break the whole program with some ugly messages, like cannot read property of ... in JavaScript, or with a NullReferenceException in C#.
- So, the best thing to do to avoid this kind of error is, obviously, to reduce the amount of possible null values in our code.
- We can deal with it in two ways:
  1. avoid returning null from a function
  2. avoid passing null values to functions

How to avoid returning null values

- Unless you don't have specific reasons to return null, so when that value is acceptable in your domain, try not to return null.
- For string values, you can simply return empty strings, if it is considered an acceptable value.
- For lists of values, you should return an empty list.



```
IEnumerable<char> GetOddChars(string value)
{
    if (value.Length > 0)
    {
        // return something
    }
    else
    {
        return Enumerable.Empty<char>();
        // OR return new List<char>();
    }
}
```



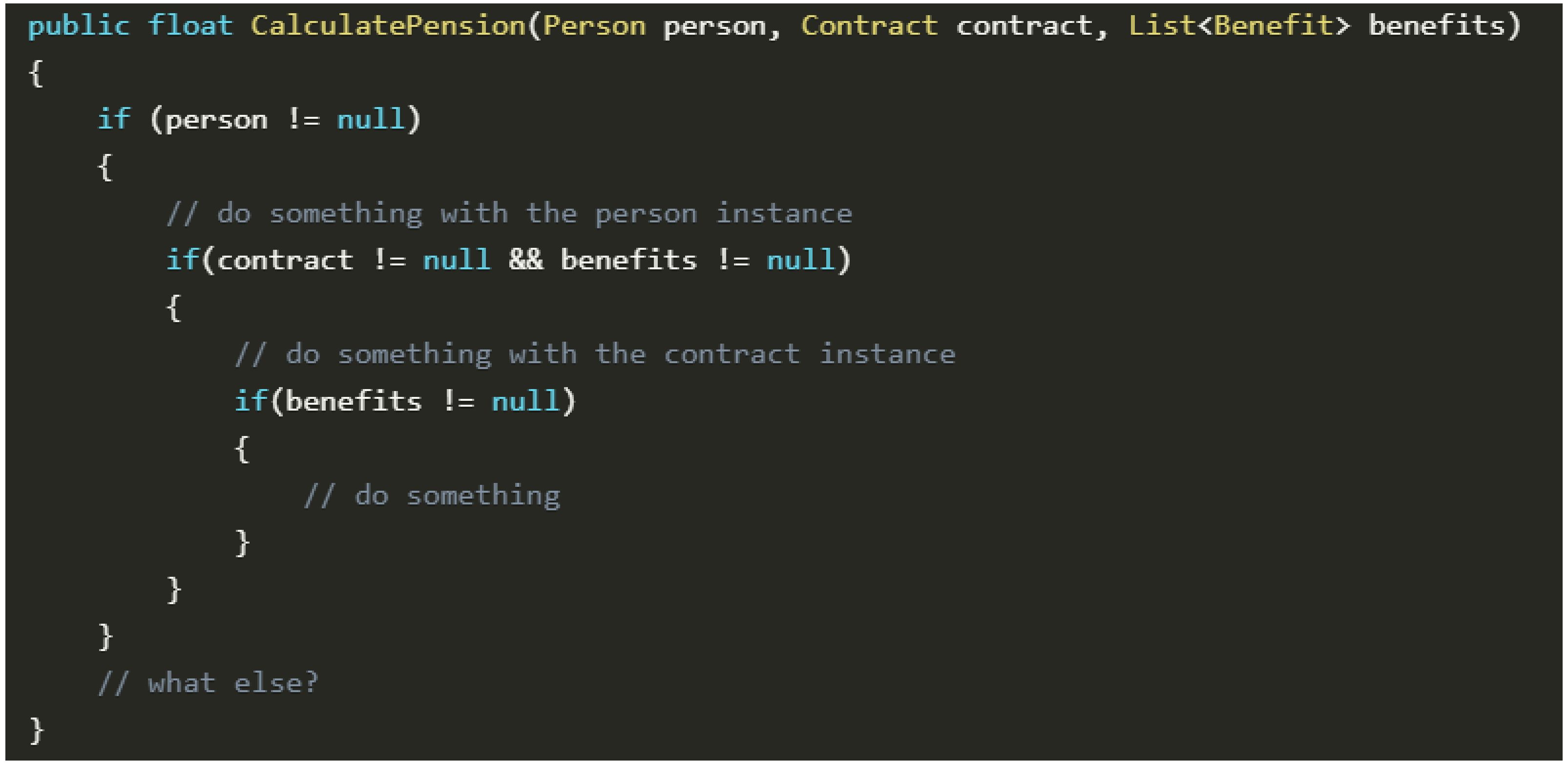
```
var chars = GetOddChars("hello!");
Console.WriteLine(chars.Count());

foreach (char c in chars)
{
    // Do Something
}
```

How to avoid returning null values

- What about objects? There are many approaches that you can take, like using the **Null Object pattern** which allows you to create an instance of an abstract class which does nothing at all, so that your code won't care if the operations it does are performed on an actual object or on a Null Object.
- **Null Object pattern:** In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available.

How to avoid passing null values to functions

- Well, since we've already avoided nulls from return values, we may expect that we will never pass them to our functions. Unfortunately, that's not true: what about you were using external libraries to get some values and then use them on your functions?



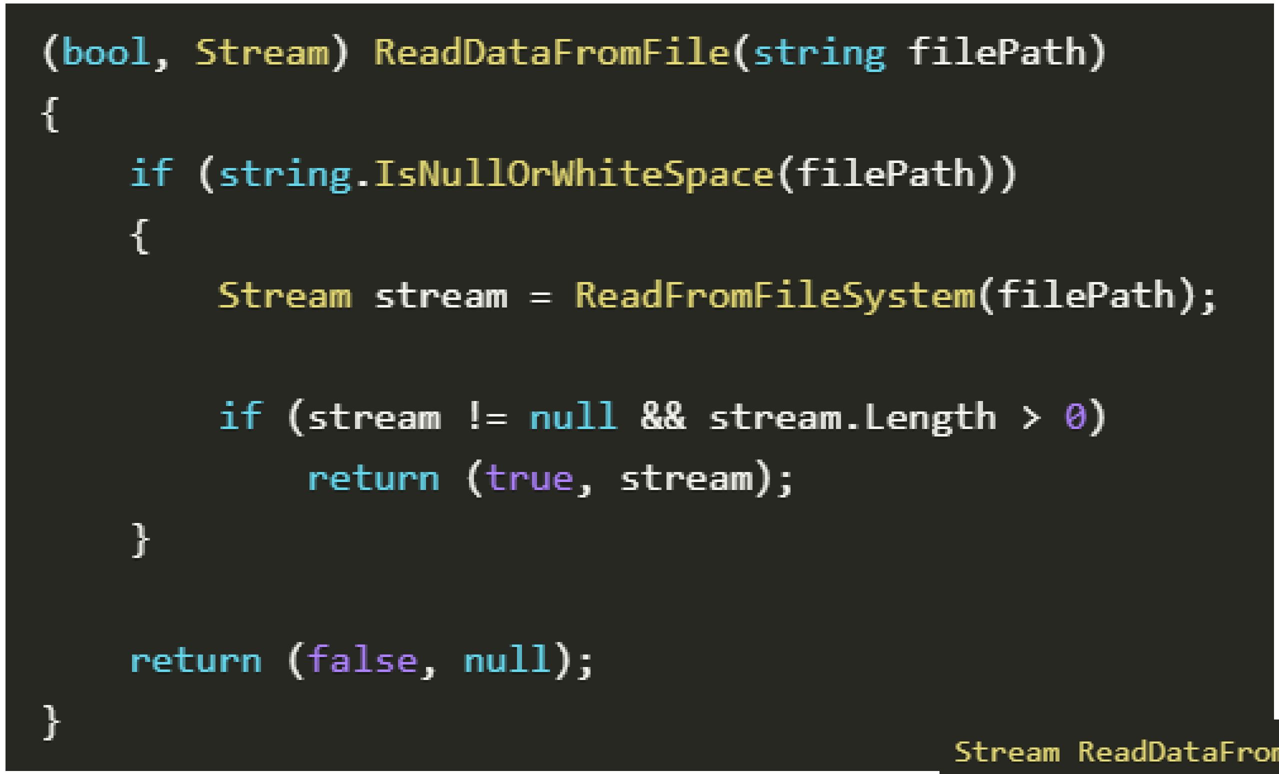
```
public float CalculatePension(Person person, Contract contract, List<Benefit> benefits)
{
    if (person != null)
    {
        // do something with the person instance
        if(contract != null && benefits != null)
        {
            // do something with the contract instance
            if(benefits != null)
            {
                // do something
            }
        }
    }
    // what else?
}
```

... and now see what happens when you repeat those checks for every method you write.
As we say, **prevention is better than the cure!**

Progressive refinements

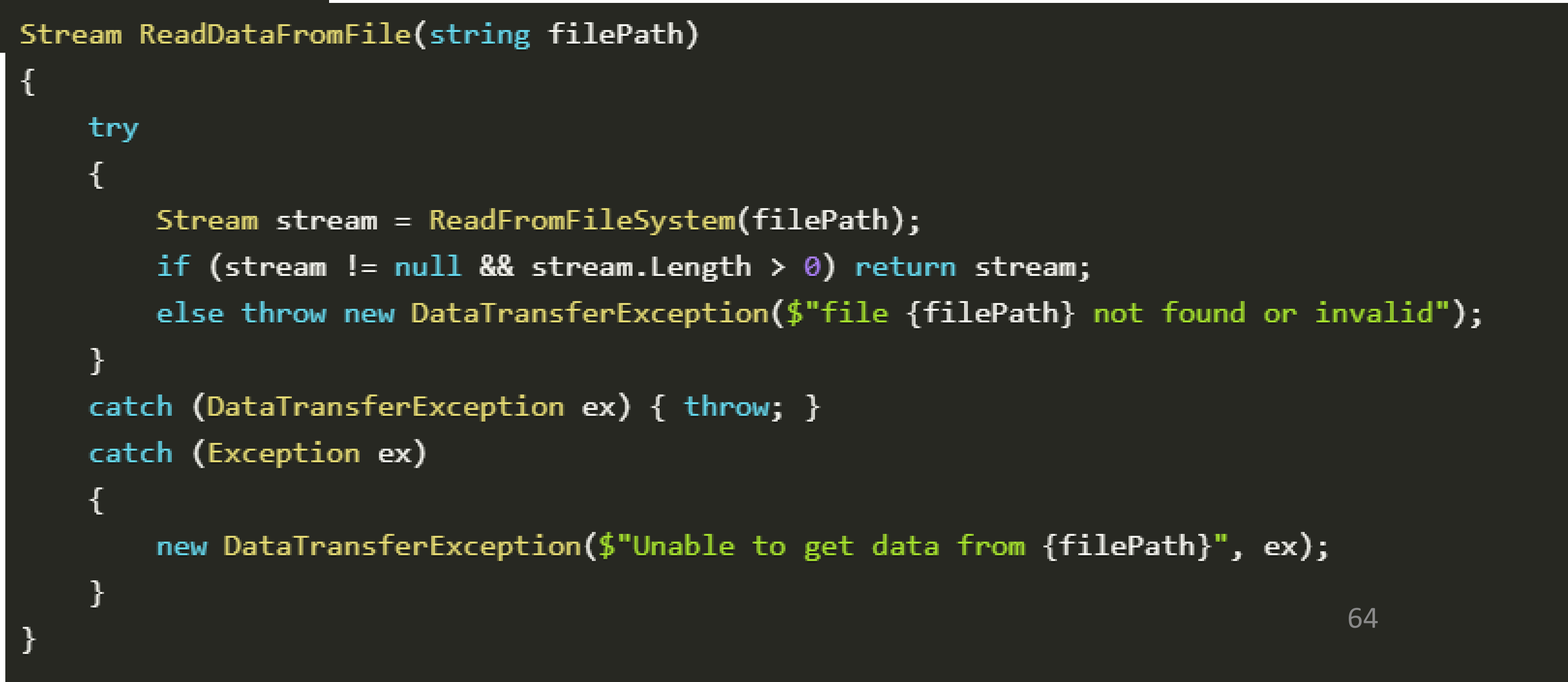
- It's time to apply those tips in a real(ish) scenario. Let's write a method that read data from the file system, parses its content, and sends it to a remote endpoint.

First step: read a stream from file system:



```
(bool, Stream) ReadDataFromFile(string filePath)
{
    if (string.IsNullOrWhiteSpace(filePath))
    {
        Stream stream = ReadFromFileSystem(filePath);

        if (stream != null && stream.Length > 0)
            return (true, stream);
    }

    return (false, null);
}
```

Better

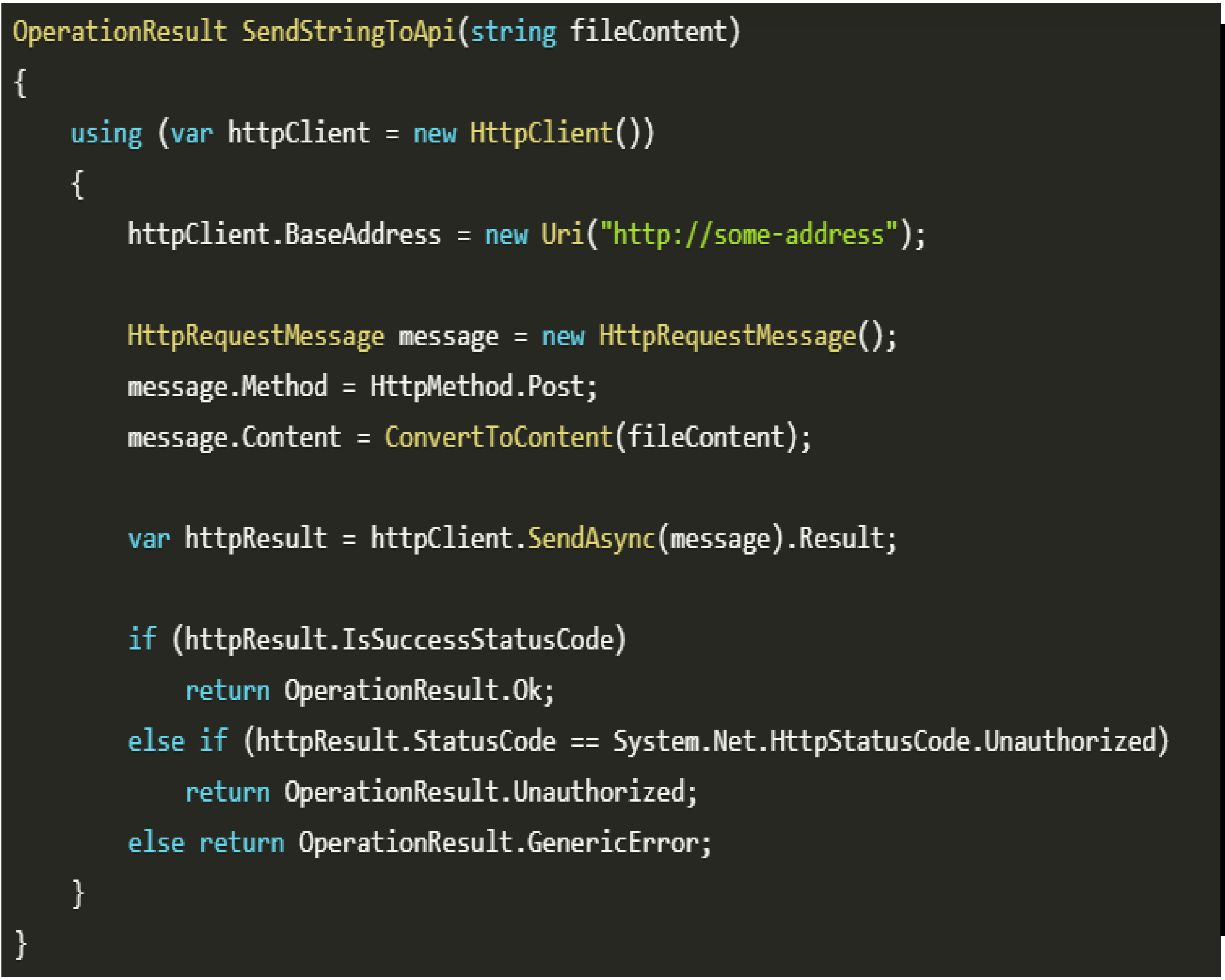


```
Stream ReadDataFromFile(string filePath)
{
    try
    {
        Stream stream = ReadFromFileSystem(filePath);
        if (stream != null && stream.Length > 0) return stream;
        else throw new DataTransferException($"file {filePath} not found or invalid");
    }
    catch (DataTransferException ex) { throw; }
    catch (Exception ex)
    {
        new DataTransferException($"Unable to get data from {filePath}", ex);
    }
}
```

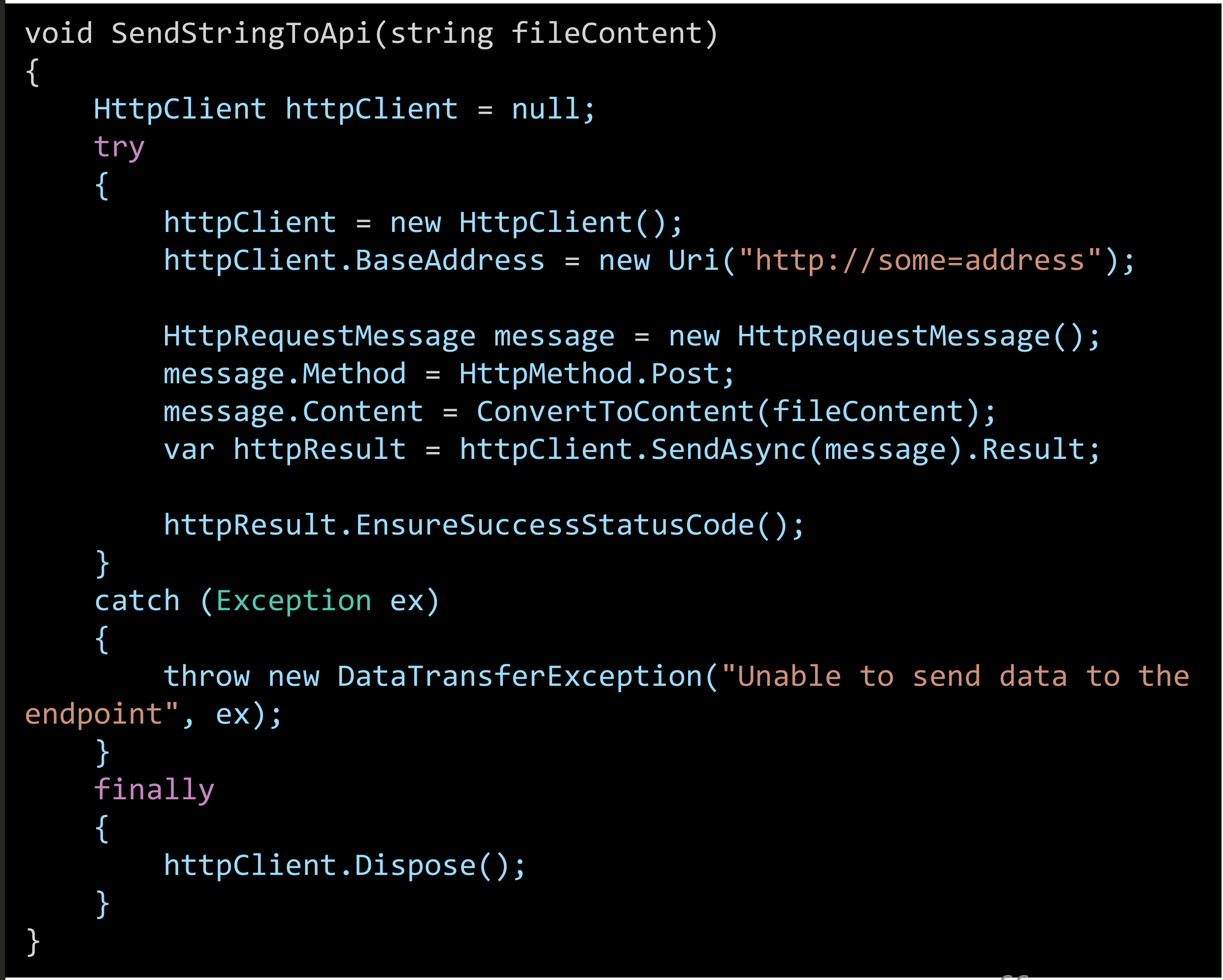
Second step: Convert that stream into plain text:

```
string ConvertStreamIntoString(Stream fileStream) {
return fileStream.ConvertToString(); }
```

Third step: Send the string to the remote endpoin



```
OperationResult SendStringToApi(string fileContent)
{
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri("http://some-address");

        HttpRequestMessage message = new HttpRequestMessage();
        message.Method = HttpMethod.Post;
        message.Content = ConvertToContent(fileContent);

        var httpResult = httpClient.SendAsync(message).Result;

        if (httpResult.IsSuccessStatusCode)
            return OperationResult.Ok;
        else if (httpResult.StatusCode == System.Net.HttpStatusCode.Unauthorized)
            return OperationResult.Unauthorized;
        else return OperationResult.GenericError;
    }
}
```

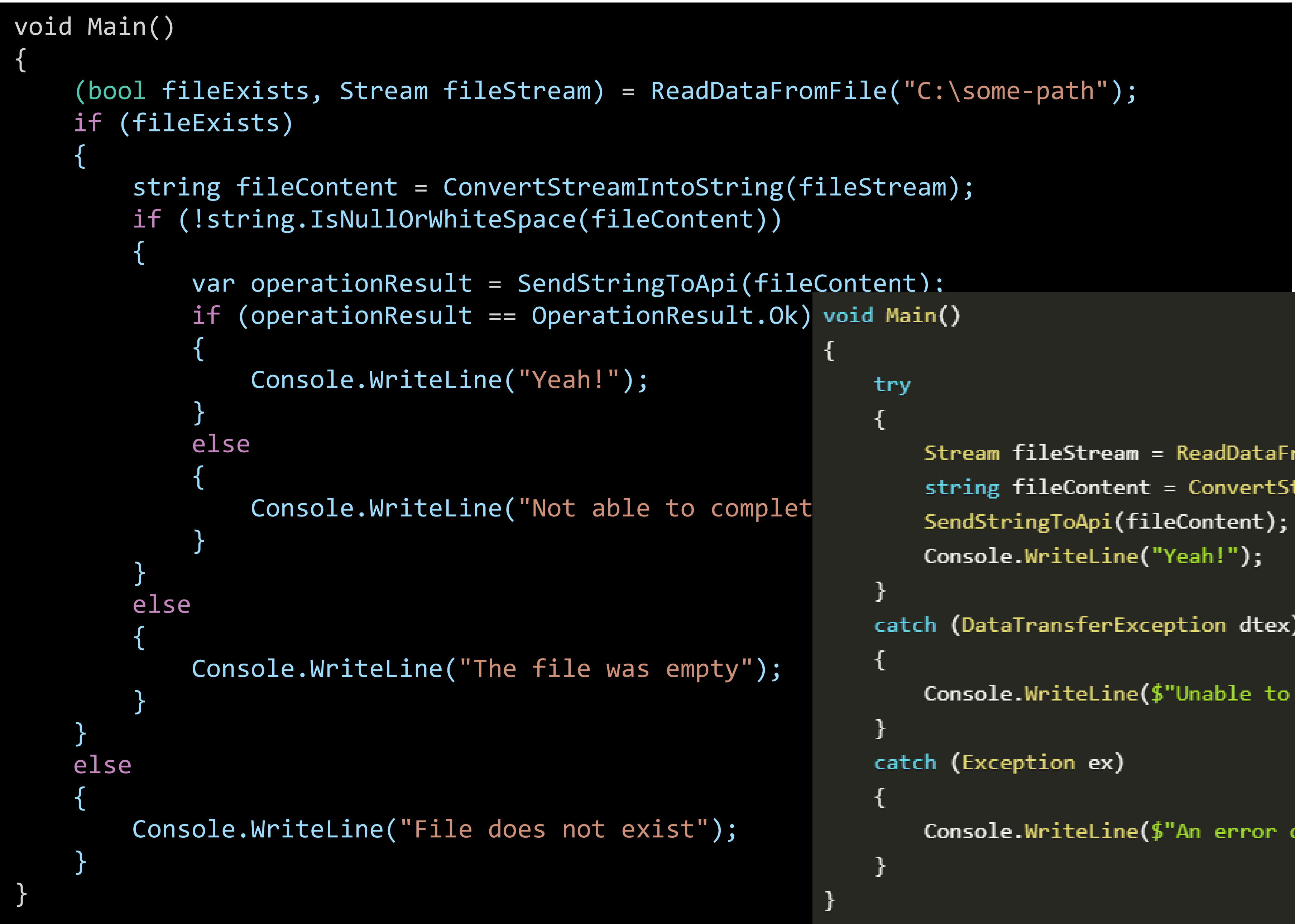


```
void SendStringToApi(string fileContent)
{
    HttpClient httpClient = null;
    try
    {
        httpClient = new HttpClient();
        httpClient.BaseAddress = new Uri("http://some=address");

        HttpRequestMessage message = new HttpRequestMessage();
        message.Method = HttpMethod.Post;
        message.Content = ConvertToContent(fileContent);
        var httpResult = httpClient.SendAsync(message).Result;

        httpResult.EnsureSuccessStatusCode();
    }
    catch (Exception ex)
    {
        throw new DataTransferException("Unable to send data to the
endpoint", ex);
    }
    finally
    {
        httpClient.Dispose();
    }
}
```

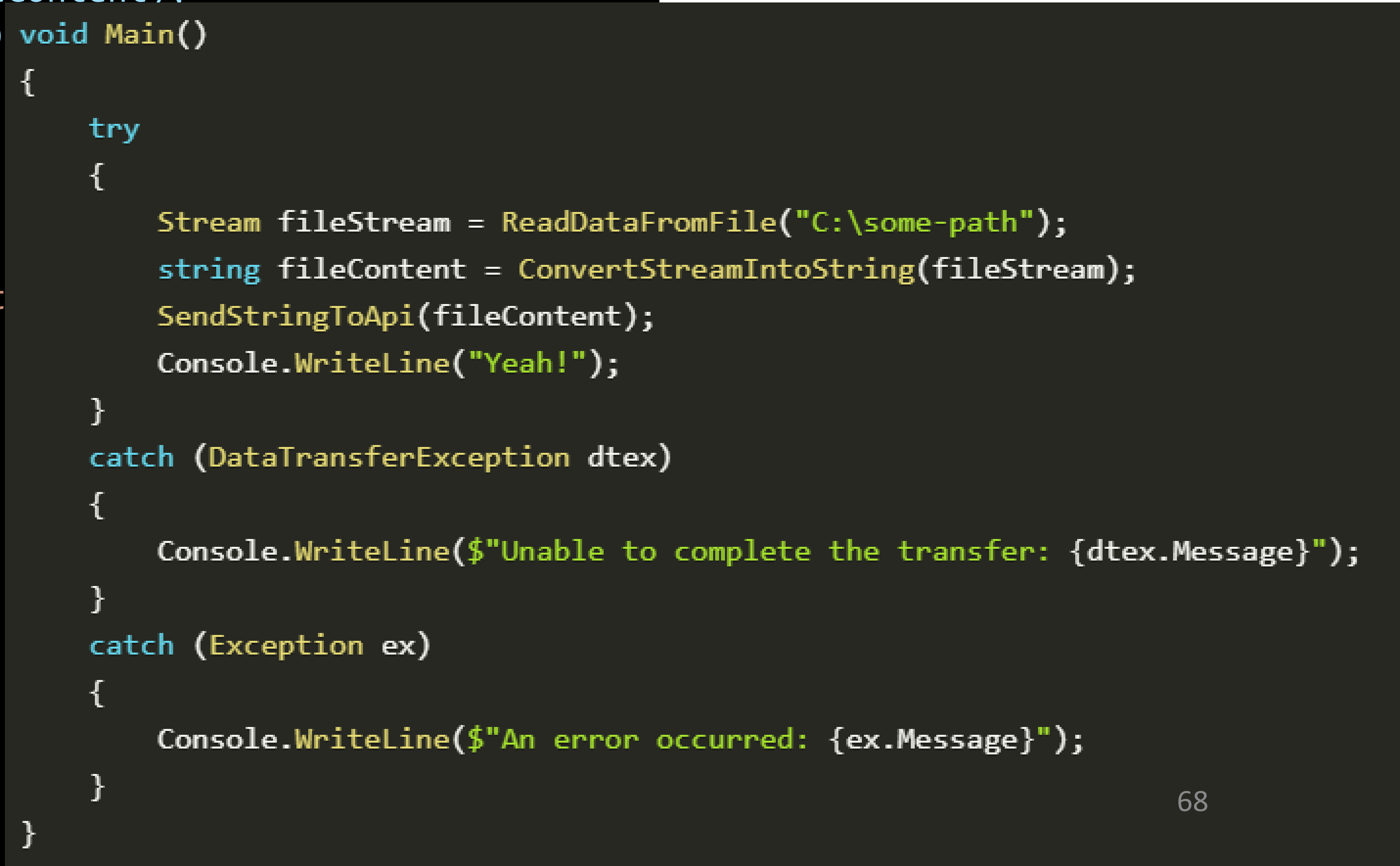
Finally: Main method

```
void Main()
{
    (bool fileExists, Stream fileStream) = ReadDataFromFile("C:\some-path");
    if (fileExists)
    {
        string fileContent = ConvertStreamIntoString(fileStream);
        if (!string.IsNullOrWhiteSpace(fileContent))
        {
            var operationResult = SendStringToApi(fileContent);
            if (operationResult == OperationResult.Ok)
            {
                Console.WriteLine("Yeah!");
            }
            else
            {
                Console.WriteLine("Not able to complete the operation");
            }
        }
        else
        {
            Console.WriteLine("The file was empty");
        }
    }
    else
    {
        Console.WriteLine("File does not exist");
    }
}
```

Finally: Main method



```
void Main()
{
    (bool fileExists, Stream fileStream) = ReadDataFromFile("C:\some-path");
    if (fileExists)
    {
        string fileContent = ConvertStreamIntoString(fileStream);
        if (!string.IsNullOrWhiteSpace(fileContent))
        {
            var operationResult = SendStringToApi(fileContent);
            if (operationResult == OperationResult.Ok)
            {
                Console.WriteLine("Yeah!");
            }
            else
            {
                Console.WriteLine("Not able to complet
            }
        }
        else
        {
            Console.WriteLine("The file was empty");
        }
    }
    else
    {
        Console.WriteLine("File does not exist");
    }
}
```



```
void Main()
{
    try
    {
        Stream fileStream = ReadDataFromFile("C:\some-path");
        string fileContent = ConvertStreamIntoString(fileStream);
        SendStringToApi(fileContent);
        Console.WriteLine("Yeah!");
    }
    catch (DataTransferException dtex)
    {
        Console.WriteLine($"Unable to complete the transfer: {dtex.Message}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"An error occurred: {ex.Message}");
    }
}
```

References

- Replace Conditional With Polymorphism - design-principles (injulkarnilesh.github.io)
- The Single Level of Abstraction Principle (SLAP) - danparkin.com
- Robert C. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship”.
- Refactoring: clean your code

THANKS
SEE U NEXT LECTURE