# Classes, Objects & Data Containers

This course does not focus on object-oriented programming, yet objects - and also classes - are of course an important aspect of programming in general, even if you don't follow a purely object-orient style.

Some concepts discussed in this document (and the respective course section) DO come from the object-oriented world and are focused on object-oriented programming.

Nonetheless, I included all concepts that help with writing classes and working with objects in general (even when using a mixed programming style) and/ or which also are interesting when not focusing on OOP.

When it comes to working with classes and objects and writing clean classes and objects, there are a **couple of rules and concepts you should be aware of**.

1. We can **differentiate between objects and data containers / data structures**
2. **Consider using Polymorphism**
3. **Classes should be small**
4. **Classes should have a high cohesion**
5. **Respect the "Law of Demeter"**
6. **Write SOLID classes**

## Objects vs Data Containers / Data Structures

**Classes are blueprints for objects**.

There also are programming languages where you can create objects without (actively) using classes - for example JavaScript. And there are languages where you MUST use classes for pretty much everything (e.g. Java).

Either way, objects allow you to **group related data** (properties) and **functionalities** (methods) **together**. And objects typically expose a public API of methods which can be used anywhere in your code to interact with these objects.

For example:

```
customer.message(someMessage);
```

Even though we're technically always dealing with objects, we can differentiate between "**real objects**" (i.e. used as objects with a public API) and mere **data containers** (or data structures, though that term is also used in different ways).

A **data container is really just that - an object which holds a bunch of data**.

Here's an example:

```
class UserData {
  public name: string;
  public age: number;
}

const userData = new UserData();
userData.name = 'Max';
userData.age = 31;
```

This class has no methods and both properties are exposed `public`ly.

An **object on the other hand hides it's data from the public and instead exposes a public API** in the form of methods:

```
class User {
  private name: string;
  private age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi! I'm ${this.name} and I'm ${this.age} years old.`);
  }
}

const userData = new UserData('Max', 31);
userData.greet();
```

**Both are absolutely valid types of objects** – it's just important to use the right kind of object for the right job. And you should **avoid mixing the types.**

Why does this matter?

When writing clean code, we **shouldn't try to access some internals of an object**. This bares the danger of the object changing these internals and our code breaking because of that.

In addition, it makes code harder to read, if we have to interpret the meaning and values of properties of other classes.

Calling something like `greet()` on the other hand is straightforward.

Of course, when working with a data container in a place where a data container is needed, this is absolutely fine:

```
function validateInput(credentials) {
  if (!isEmail(credentials.email) || !isPassword(credentials.password))   {
    // ...
  }
}
```

In this example, it's clear that `credentials` just holds some data which we can extract and work with.

## Polymorphism

Polymorphism is a fancy term but in the end it just means that you re-use code (e.g. call the same method) **multiple times** but that the code will do **different things**, depending on the object type.

It's a powerful concept which can be utilized to avoid code duplication.

Here's an example that does not use polymorphism:

```
class Delivery {
  private purchase: Purchase;

  constructor(purchase: Purchase) {
    this.purchase = purchase;
  }

  deliverProduct() {
    if (this.purchase.deliveryType === 'express') {
      Logistics.issueExpressDelivery(this.purchase.product);
    } else if (this.purchase.deliveryType === 'insured') {
      Logistics.issueInsuredDelivery(this.purchase.product);
    } else {
      Logistics.issueStandardDelivery(this.purchase.product);
    }
  }

  trackProduct() {
    if (this.purchase.deliveryType === 'express') {
      Logistics.trackExpressDelivery(this.purchase.product);
    } else if (this.purchase.deliveryType === 'insured') {
      Logistics.trackInsuredDelivery(this.purchase.product);
    } else {
      Logistics.trackStandardDelivery(this.purchase.product);
    }
    }
  }
```

The problem with this code is that we have the same `if` checks with different logic inside of the `if` statements. So it's some code duplication but not everything is duplicated.

And we need these different cases because a product can of course be sent and tracked (and maybe we could even come up with additional methods).

Here's how this issue could be solved in a polymorphic way:

```
class Delivery {
  protected purchase: Purchase;

  constructor(purchase: Purchase) {
    this.purchase = purchase;
  }
}

class ExpressDelivery extends Delivery {
  deliverProduct() {
    Logistics.issueExpressDelivery(this.purchase.product);
  }

  trackProduct() {
    Logistics.trackExpressDelivery(this.purchase.product);
  }
}

class InsuredDelivery extends Delivery {
  deliverProduct() {
    Logistics.issueInsuredDelivery(this.purchase.product);
  }

  trackProduct() {
    Logistics.trackInsuredDelivery(this.purchase.product);
  }
}

class StandardDelivery extends Delivery {
  deliverProduct() {
    Logistics.issueStandardDelivery(this.purchase.product);
  }

  trackProduct() {
    Logistics.trackStandardDelivery(this.purchase.product);
  }
}
```

Above, there are three specialized classes (`ExpressDelivery` etc.) which all inherit from the `Delivery` base class to share common functionalities.

All these specialized classes implement the same `deliverProduct()` and `trackProduct()` methods - though the exact same code in these methods differs.

**That's the idea behind polymorphism!**

We can now write a factory function or class method that creates instances of these classes when needed. It's the factory function which holds the `if` check from above.

Hence we only need the check once instead of once per method.

```
function createDelivery(purchase) {
  if (purchase.deliveryType === 'express') {
    return new ExpressDelivery(purchase);
  } else if (purchase.deliveryType.deliveryType === 'insured') {
    return new InsuredDelivery(purchase);
  } else {
    return new StandardDelivery(purchase);
  }
}
```

You can now always run code like this and the exact result will depend on the `deliveryType` used for creating the objects:

```
const expressDelivery = createDelivery({deliveryType: 'express', ...});
expressDelivery.deliverProduct();
```

## Classes Should Be Small

Just like functions, classses should be **small**. Also just like functions, classes should kind of "do one thing", though here "one thing" is not a single task but instead a **single object all tasks should be related to**.

A `User` class might for example have a `login()` method. It might have a couple of other methods that do user-typical things but a `refund()` method would be strange.

`refund()` sounds more like a payment-related method so it would make more sense in a `Payment` class for example.

Hence a `User` class which also handles payments would be too big - even if it would only be made up of a couple of lines of code.

The **size of a class is therefore defined by its number of responsibilities**. And clean classes should **only have one responsibility** (*also see "Single Responsibility Principle", further down below*).

The responsibility of a `User` class should be to handle user-related tasks, for example `login()`, `logout()` etc.

## Classes & Cohesion

**Clean classes have a high cohesion.**

But what is "cohesion"?

**Cohesion basically describes the amount by which methods of a class use class properties.**

If a class has 2 properties and 3 methods and every method uses both properties, this class would have the **highest possible cohesion**.

If the same class would look such that no method uses the properties, the class would have **no cohesion** - the properties would be worthless for the methods.

**Low cohesion is a clear sign for a class that should maybe just be a data container** / data structure instead of an object with a public API (since that public API, the methods, doesn't interact with the internal properties).

In reality, you will rarely achieve maximum cohesion but **high cohesion should definitely be your goal**.

Once you note that cohesion decreases, it might be time to **split a class into smaller, more focused classes**.

Hence, if you care about cohesion (and you should!), you will automatically end up with smaller classes - which is good.

## The Law Of Demeter

When working with objects, the following code is considered to be bad / not clean:

```
this.customer.lastPurchase.date;
```

This code violates the **Law of Demeter** which states that you shouldn't access the internals of another object through another object.

This is also called the "**principle of least knowledge**".

**Code in a method** may only access direct internals (properties and methods) of:

- the object it **belongs to**
- objects that are **stored in properties of that object**
- objects which are **received as method parameters**

- objects which are **created in the method**

In this case, `lastPurchase` is a property (an attribute) of the `customer` object, which in turn is a property of the class the executing method belongs to. It's a different object, maybe based on some `Purchase` class. And that class then has a `date` property.

*Side-note: Here I directly access properties - it's the same if you would be using getters (`customer.getLastPurchase().getDate()`).*

Accessing this `date` property through `lastPurchase` violates the Law of Demeter.

The problem with code like this is that you can end up with very long statements, full of "chaining" (`.something.somethingElse.more()` is called "chaining").

This creates strong dependencies and whenever the `Purchase` class would change (e.g. `date` is renamed to `paymentDate`) all code snippets that look like above need to be adjusted.

The above code would better be implemented like this:

```
this.customer.getLastPurchaseDate();
```

Now the extra dependency on `lastPurchase.date` was removed.

**Even this solution is not optimal though** - it would be better to "**tell, not ask**". That means, it depends on what you actually needed to do with that date.

Let's say you needed the date to send the purchased products to the customer.

```
const date = this.customer.lastPurchase.date;
this.warehouse.deliverPurchasesByDate(this.customer, date);
```

Instead of getting the date and then checking for the products, the following code would've been a better way of avoiding the "Law of Demeter" violation:

```
this.warehouse.deliverPurchase(customer.lastPurchase);
```

As you can see, it's not just about moving some code around - sometimes a different solution (working with different objects) is the cleanest way of getting something done.

It's also important to note that the "Law of Demeter" really only applies to **property / attribute chaining**.

The following code would be fine (as long as these **methods are real methods and not just getters** wrapped around properties):

```
this.customer.sendMessage(message).retry(2);
```

Why would this be okay?

Because now we **utilize the public interfaces** of various classes instead of "abusing" their internal data structure.

If you are only working with a couple of **data containers**, you would **not be violating the "Law of Demeter"** when chaining their various properties together. Because the entire idea behind data containers **IS** that they expose their properties publicly - they got nothing else besides that!

## SOLID Classes

When it comes to creating clean classes, there are a couple of helpful rules, laws and concepts (e.g. "Law of Demeter", see above). But especially the SOLID principles are often cited.

Below, all five principles are explained and put into the context of writing clean code.

### S: Single-Responsibility Principle (SRP)

> *A class should only have a single responsibility - it should only change for this one responsibility.*

The SRP simply states that a class should be focused on one core responsibility. Only if that responsibility requires changes to the class code, such changes are acceptable.

If a class needs to change because of different responsibilities, it's **too big and should be split** into multiple smaller classes.

The SRP is an important principle when it comes to writing clean code, since it **enforces smaller, and therefore more readable, classes.**

Here's an example for a class that violates the SRP:

```
class ReportDocument {
  generateReport(data: any) { ... }

  createPDF(report: any) { ... }
}
```

The actual code in the methods is missing, because it's not needed - it's the structure and API of the class which matters here.

This `ReportDocument` class violates the SRP because it needs to change because of at least two reasons:

- If anything related to how a report is generated changes
- If anything related to how a report is printed as a PDF changes

It should be fair to assume that in most applications, these two features are probably not directly connected. The people deciding on how the PDF should look like are not necessarily the same people who decide how the report should be generated.

This class would therefore better be split - e.g. into `Report` and `PDFPrinter` classes.

The SRP is an **important principle when it comes to writing clean code**. Because it typically leads to smaller and more focused classes - which is in line with what we want for our classes from a clean code perspective!

## O: Open-Closed Principle (OCP)

> *A class should be open for extension but closed for modification.*

Whilst the above sentence might sound cryptic, this is actually a straightforward principle.

Once you decided how a class should look like (which public API / methods it has), you should **"close" the class for modification** - i.e. the code doesn't change anymore.

Of course, that's not really true though. You still **should continue working** on the class and especially fix errors.

But you should **not** edit the class all the time, just to handle certain new features or - worse - variants of features.

Here's an example of a class that violates the OCP:

```
class Printer {
  printPDF(data: any) {
    // ...
  }

  printWebDocument(data: any) {
    // ...
  }

  printPage(data: any) {
    // ...
  }

  verifyData(data: any) {
    // ...
  }
}
```

This `Printer` class has different methods for printing a web document (e.g. generating a HTML file), a "normal" page and a PDF.

Whenever we add a new type of document (e.g. a Word or Excel document), we need to add a new method - probably with a decent junk of code duplication.

This problem might sound familiar - we solved it with "Polymorphism" earlier!

And indeed, that's how we can fix the issue here as well (and follow the OCP).

This `Printer` class is currently **not closed** because we need to come back to it and edit it whenever a new type of document should be printed.

To follow the OCP, we should close it and instead extend it whenever we want to add a new type of document.

```
interface Printer {
  print(data: any);
}

class PrinterImplementation {
  verifyData(data: any) {}
}

class WebPrinter extends PrinterImplementation implements Printer {
  print(data: any) {
    // print web document
  }
}

class PDFPrinter extends PrinterImplementation implements Printer {
  print(data: any) {
    // print PDF document
  }
}

class PagePrinter extends PrinterImplementation implements Printer {
  print(data: any) {
    // print real page
  }
}
```

This snippet shows how we could fix the issue. Now, the base class `PrinterImplementation` (named like this because we have an interface named `Printer`) is closed. It doesn't need to change just because we want to support a new type of document.

**Instead, we extend it** - we add new subclasses which are based on `PrinterImplementation` (which adds common functionality like verification) and implement `Printer` (which forces all implementing classes to have a `print()` method).

The OCP is an **important principle when it comes to writing clean code**. Because it typically leads to smaller and more focused classes - which is in line with what we want for our classes from a clean code perspective!

## L: Liskov Substitution Principle (LSP)

> *Objects in a program should be replaceable with instances of their subtypes.*

The above sentence should be quite clear. Here's an example of a baseclass being replaced with a subclass of it:

```
class Bird {
  fly() {
    console.log('Fyling...');
  }
}

class Eagle extends Bird {
  dive() {
    console.log('Diving...');
  }
}

const bird = new Bird();
bird.fly();

// We can also replace Bird() with Eagle()
const eagle = new Eagle();
eagle.fly();
```

In this snippet, the `Bird` class is the superclass to the `Eagle` class and indeed, we can substitute the `Bird` which we construct with an `Eagle` without changing our code.

Now, let's add a new subclass which actually violates the principle:

```
class Penguin extends Bird {
  // Problem: Can't fly!
}
```

We can't use `Penguin` as drop-in replacement for `Bird`, because penguins can't fly!

So actually, as soon as we have a case like this, we **violated the LSP**.

A better way of modelling our data then would be to do it like this:

```
class Bird {}

class FlyingBird extends Bird {
  fly() {
    console.log('Fyling...');
  }
}

class Eagle extends FlyingBird {
  dive() {
    console.log('Diving...');
  }
}

const eagle = new Eagle();
eagle.fly();
eagle.dive();

class Penguin extends Bird {
  // Good! Birds don't need to fly!
}
```

Now we added a new `FlyingBird` class which is based on `Bird`. And we can now choose whether we have other birds which are "just birds" (like `Penguin`) or "flying birds" (like `Eagle`).

Now we follow the LSP.

Hence, it's fair to say that the LSP in the end has the goal of **forcing us to model our data correctly** and think carefully about our models and the entities we work with in our code.

The LSP is definitely an important and popular principle, but it doesn't have that big of an effect on our code if we look at it from a clean code perspective.

Yes, it could lead to smaller classes but it's really mostly focused on forcing us to model our data correctly.

## I: Interface Seggragation Principle (ISP)

> *Many client-specific interfaces are better than one general-purpose interface.*

Reading sentences like the one above can always be confusing, but the ISP is actually also quite straightforward.

When working with classes and in an OOP way, you often also work with "Interfaces". Not all programming languages support interfaces but many do.

Interfaces are basically contracts which force implementing classes to implement certain behaviors (methods and properties).

Here's an example of code that violates the ISP:

```
interface Database {
  storeData(data: any);
  connect(uri: string);
}

class SQLDatabase implements Database {
  connect(uri: string) {
    // connecting...
  }

  storeData(data: any) {
    // Storing data...
  }
}

class InMemoryDatabase implements Database {
  connect(uri: string) {
    // Needs a connect method (because of interface)
    // but hasn't anything to connect to :(
  }

  storeData(data: any) {
    // Storing data...
  }
}
```

In this example, we have a `Database` interface and then we got a couple of database classes for different kinds of databases.

The problem with that code is, that the `Database` interface is too generic - "too general purpose".

It forces the `InMemoryDatabase` to add a `connect()` method, even though this type of database has nothing to connect to (i.e. there is no extra database server or anything like that).

Hence it would be better to work with multiple, more specialized interfaces instead of the general purpose one. And that's exactly what the ISP says!

```
interface Database {
  storeData(data: any);
}

interface RemoteDatabase {
  connect(uri: string);
}

class SQLDatabase implements Database, RemoteDatabase {
  connect(uri: string) {
    // connecting...
  }

  storeData(data: any) {
    // Storing data...
  }
}

class InMemoryDatabase implements Database {
  storeData(data: any) {
    // Storing data...
  }
}
```

Now, with this code, we got **two interfaces** intead of one and every class can pick the interfaces that make sense for it. Hence our `InMemoryDatabase` is able to just implement `Database`, which forces it to have a `storeData()` method and ignore the `RemoteDatabase` interface, which would force it to add a `connect()` method.

The ISP is definitely an important and popular principle, but it doesn't have that big of an effect on our code if we look at it from a clean code perspective.

## D: Dependecy Inversion Principle (DIP)

> *One should depend upon abstractions, not concretions.*

This last principle is a principle which you will often already follow, if you follow the other SOLID principles.

It basically is all about not being too specific in your code.

Sounds strange?

Here's an example of code which we could improve (this code assumes that we have database classes as shown for the ISP available):

```
class App {
  private database: Database | RemoteDatabase;
  constructor(database: Database | RemoteDatabase) {
    if (database.connect) {
      database.connect('my-url');
    }
    this.database = database;
  }

  saveSettings() {
    this.database.storeData('Some data');
  }
}


const sqlDatabase = new SQLDatabase();
const app = new App(sqlDatabase);
```

In this snippet, we use the databases we created for the ISP. In the `constructor()` of our `App` class, we check whether we're getting a `RemoteDatabase` (i.e. if we have a `connect()` method) and if we do, we do connect to the database.

The problem with this code is, that we're very specific in our `constructor()` method. The code we execute depends on which kind of database we're getting.

Hence we "depend on a concretion" in that example.

This is suboptimal because it means that we need to make this check whereever we get such a database and we also need to change our code whenever the database implementation changes.

Here's how we could restructure the code to follow the ISP:

```
class App {
  private database: Database;

  constructor(database: Database) {
    this.database = database;
  }

  saveSettings() {
    this.database.storeData('Some data');
  }
}

const sqlDatabase = new SQLDatabase();
sqlDatabase.connect('my-url');
const app = new App(sqlDatabase);
```

This snippet shows us why it's called the "Dependency Inversion Principle".

We ensure that we **depend on an abstraction** ("we just get some database") **instead of a concretion** ("we need to check whether we need to connect"). And we do that by **inverting the dependency**.

Instead of depending on the kind of database inside of `constructor()`, we "force" whoever is creating an `App` to give us a database which is already connected (if required).

## Use Common Sense

Just as in the "Functions" section of this course, I really want to emphase one important thing: Use your common sense.

It's easy to treat every tiny step as a single responsibility and single thing and if you do that, you end up with projects with 100s of classes that all contain only one method.

**This is NOT the goal and definitely NOT clean code!**

Yes, classes should be small - just like methods and functions.

But "small" does not mean "almost empty".

It is okay to do work in classes and methods. And whilst methods indeed probably shouldn't be dozens of lines long, the idea behind classes is to group related data and concepts together.

**Don't destroy that by breaking up everything!**

Instead, consider the various concepts and rules explained here. Apply them as good as you can but don't start breaking up everything because of them.