Open-closed principle

why is it important?

S.O.L.I.D

S Single Responsibility Principle

O Open-Closed Principle

L Liskov Substitution Principle

I Interface Segregation Principle

D Dependency Inversion Principle

software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

➡ *we should strive to write code that doesn't have to be changed every time the requirements change*

Why is this principle important?

- Software is written to implement specific requirements
- The only constant is change
- Write code that can be easily adapted
- Better create new code instead of affecting existing one

Simple example 1 (in Perl)

> perl writeFile.pl

```
#!/usr/bin/perl
my $sampleText = „Test“;
my $filename = 'C:\temp\testfile.txt';

open(FH, '>', $filename) or die $!;

print FH $sampleText;

close(FH);
```

new requirement:
Filename must be different

Simple example 2 (in Perl)

> perl writeFile.pl testfile.txt

```
#!/usr/bin/perl
my $filename = shift;
my $path = „C:\temp\“;
my $sampleText = „Test“;

open(FH, '>', $path.$filename) or die $!;

print FH $sampleText;

close(FH);
```

simple solution:
pass the filename as a parameter

TicTacToe example

```
public class WinningConditions {

private List<WinningCondition> winningConditions;
   public WinningConditions() {
      winningConditions = new ArrayList<>();
      winningConditions.add(new WinningCondition(Fields.TOP_LEFT, Fields.TOP_CENTER, Fields.TOP_RIGHT));
      winningConditions.add(new WinningCondition(Fields.CENTER_LEFT, Fields.CENTER_CENTER, Fields.CENTER_RIGHT));
      winningConditions.add(new WinningCondition(Fields.BOTTOM_LEFT, Fields.BOTTOM_CENTER, Fields.BOTTOM_RIGHT));
      winningConditions.add(new WinningCondition(Fields.TOP_LEFT, Fields.CENTER_LEFT, Fields.BOTTOM_LEFT));
   }
   public List<WinningCondition> getConditions() {
      return winningConditions;
   }
}
```

Insurance example

```
public enum InsuranceType {
        Standard = 1,
        HalfPrivate = 2,
        Private = 3,
}
```

```
public class InsuredPerson {
        public int id { get; set; }
        public String fullName { get; set; }
        public InsuranceType insuranceType{ get; set; }
}
```

```
public class PolicyHolder {
        public int id { get; set; }
        public String fullName { get; set; }

        public PolicyHolder generate(InsuredPerson insuredPerson) {
                PolicyHolder policyHolder = new PolicyHolder();
                policyHolder.id = insuredPerson.id;
                policyHolder.fullName = insuredPerson.fullName;

                switch(insuredPerson.insuranceType) {
                        case InsuranceType.Standard:
                                generateStandardInsurance(insuredPerson);
                                break;
                        case InsuranceType.HalfPrivate:
                                generateHalfPrivateInsurance(insuredPerson);
                                break;
                        …
                }
        }

        private void  generateStandardInsurance(PolicyHolder policyHolder) {
                // implementation
        }
        public void generateHalfPrivateInsurance(PolicyHolder policyHolder) {
                // implementation
        }
}
```

Insurance example

- Extend the functionality to provide the custom implementation for each insurance type
- Create separated classes
- Use interfaces

```
public class Standard :  IinsuredPerson {
        public int id { get; set; }
        public String fullName { get; set; }
        public IPlanInsurance insuranceType{ get; set; } = new StandardInsurance();
}
```