an introduction to Spring boot and the power of weak communication

- Operation
- Collection injection (it can be a map, list etc.)

```
@Component
public class SampleCalculator {
    private final Collection<IIntBinaryOperator> k_operations;
```

```
public interface IIntBinaryOperator extends IntBinaryOperator {
    boolean isValid(char op);
}
```

- all classes that implements this interface will be automatically placed in the collection and take up "size"

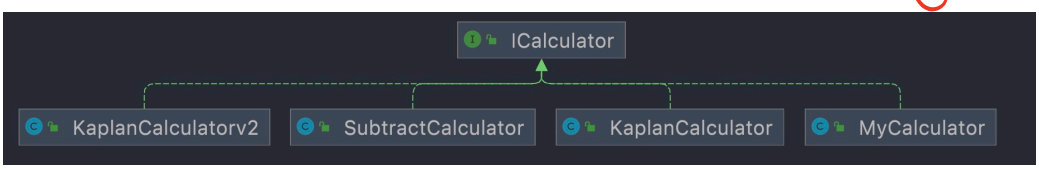
↳ these 5 classes are autowired, Sample Calculator has no idea what are they!!!

- let's look one of them, a simple 2 step process

```
@Component
public class MultiplyOperation implements IIntBinaryOperator{
    @Override
    public boolean isValid(char op) { return op =='*'; }

    @Override
    public int applyAsInt(int left, int right) { return left*right; }
}
```

• Calculation, properties and Configuration




```java
@FunctionalInterface
public interface ICalculator {
    void calculate() throws Exception;
}
```

properties

```
mycalc.a=30
mycalc.b=20
mycalc.op=*
```

```java
@Component("samplecalc")
public class SampleCalculator {
    private final Collection<IIntBinaryOperator> k_operations;

    public SampleCalculator(Collection<IIntBinaryOperator> operatio
        this.k_operations = operations;
        System.out.println("two classes are autowired, that's why
                " what are they !");
        System.out.println("this is how we do collection injection
        System.out.println(k_operations.size());
    }
    public void throwException(char op) { throw new UnsupportedOpe

    public void calculate(int a,int b,char op){
        k_operations.stream().filter(ibo ->ibo.isValid(op)).findFir
                ibo-> Console.writeLine( fmt: "%d %c %d = %d",a,op,b,
                ()->throwException(op));

    }
```

```java
@Component
public class MyCalculator implements ICalculator {
    private final SampleCalculator smpcalc;
    @Value("30")
    private int k_a;

    @Value("20")
    private int k_b;

    @Value("*")
    private char k_op;
    public MyCalculator(@Qualifier("samplecalc") SampleCalculator _smpcalc) { this.smpcalc = _smpc

    @Override
    public void calculate() { smpcalc.calculate(k_a,k_b,k_op); }
}
```

→ a singleton SampleCalculator obj is created

→ via dependency injection ↘ is called by MyCalculator

→ according to the "ICalculator" contract we call "calculate" clause

→ multiplyOperation is in collection injection and match with the k-op taken from properties file!

→ in this overided method we use desired method of the reference obtained with dependecy injection, using the parameters given by "properties" file.

Configure MyCalculator class

```java
@Configuration
public class MyCalcConfig {
    //doesn't need to know sample calculator!
    private final MyCalculator k_mycalc;

    public MyCalcConfig(MyCalculator _mycalc) { this.k_mycalc = _mycalc; }
    @Bean("minecalculator")
    public ApplicationRunner runCalculator() { return args -> k_mycalc.calculate(); }
}
```

→ via dependecy inj we get MyCalculator reference and run via AppRunner.

→ this is not a well implementation. I divided runner for substraction below

Divide Runner and see the power of ICalculator!

```java
@Configuration
public class IntCalcConfigv2 {

    private final SubtractCalculator subtractCalculator;

    public IntCalcConfigv2(SubtractCalculator _subtractCalculator) {
        this.subtractCalculator = _subtractCalculator;

    }

    @Bean("IntCalcConfigv2_via_ICalculator")
    public ICalculator createCalculator(){
        return subtractCalculator;
    }

}
```

```java
@Component
public class SubtractRunner implements ApplicationRunner {
    private final ICalculator m_calculator;

    public SubtractRunner(@Qualifier("IntCalcConfigv2_via_ICalculator")
                          ICalculator calculator)
    {
        m_calculator = calculator;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception
    {
        m_calculator.calculate();
    }
}
```

I can return everything that implements ICalculator

⇒ and use in runner via App Runner with the help of dependency inj

this runner doesn't know what type of calculation will be executed

and this is the power of weak/uncoupled communication!