
kyh

ObjektOrienterad Programmering (OOP)

Objektorienterad programmering är ett så kallat **programmerings paradigm.**

När man pratar om paradigm inom programmering så menar man en viss grundläggande stil eller metod för att utveckla programvara. Det representerar en generell filosofi eller metodologi som vägleder hur program ska utformas och kodas.

Olika paradigm har olika styrkor och svagheter och lämpar sig därför olika bra till olika typer av problemlösning.

På motsvarande sätt så är olika programmeringsspråk utformade för att passa till olika paradigm.

Java är exempelvis skapat med det objektorienterade paradigmet i åtanke.

Några vanliga paradigm

Procedurell programmering: Fokuserar på procedurer eller funktioner som utför en serie instruktioner för att utföra en uppgift. Det paradigm vi i huvudsak har jobbat med hittills.

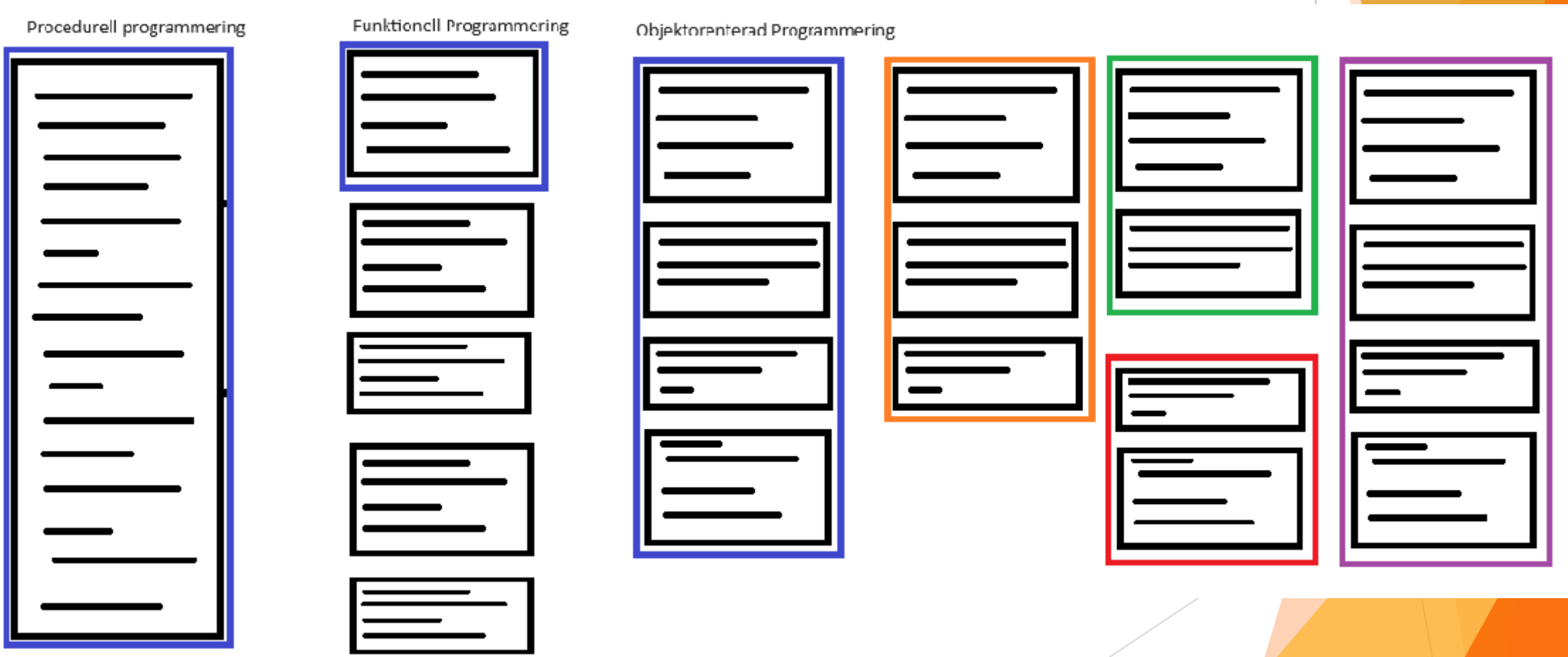
Funktionell programmering: Betonar användningen av funktioner som första klassens medborgare. Funktioner betraktas som matematiska funktioner som mappar indata till utdata. (Sten, sax, påse exemplet vi körde i förra veckan)

Objektorienterad programmering (OOP): Centreras kring konceptet av objekt, som är instanser av klasser. Objekten innehåller både data (egenskaper) och metoder (funktioner) som kan manipulera dessa data.

Det finns många fler paradigm än dessa men de går vi inte in på här. (googla på **programmerings paradigm** för fler exempel)

Värt att nämna är också att man inte måste hålla sig till ett enstaka paradigm, utan ofta kan man använda en kombination av paradigmer.

Några vanliga paradigm



Klasser och Objekt

Klasser och objekt utgör kärnaspekterna inom OOP (Objekt Orienterad Programmering)

Hittills har vi bara haft enkla program där koden läses uppifrån och ner i en maintråd och vi utför olika operationer på data genom att anropa olika metoder etc.

Med OOP så skapar man istället klasser och objekt som håller reda på sin egen data och manipulerar dessa.

Koden blir då mer strukturerad och oftast snabbare att köra

Klasser

En klass kan sägas vara som en **mall** för objekt

Ex.

Klass: Husdjur

- katt
- hund
- fågel

Klass: Bil

- volvo
- saab
- mercedes

Klasser -attribut

Variabler i en klass kallas attribut

Klass: djur

attribut:

antal ben

kan flyga

djur: hund

antal ben: 4

kan flyga: nej

djur: fågel

antal ben: 2

kan flyga: ja

Klasser -attribut

Olika objekt av samma klass kan mao ha olika värden på sina attribut

Klasser -attribut

```
class Animal {                          //vi utgår från en hund i detta exempel
        int numberOfLegs = 4;
        boolean canFly = false;
}

class Main{
        public static void main(String[] args){
                Animal dog = new Animal();
                System.out.println(dog.numberOfLegs);
                System.out.println(dog.canFly);
}
```

Klasser -attribut

Vi kan ändra attribut i ett objekt

```
dog.numberOfLegs = 3;
```

OBS. men INTE om vi satt access modifier till **private**, i så fall kan vi endast ändra attributen inifrån samma objekt.

Klasser -metoder

En klass kan innehålla egna metoder

en **public** metod kan anropas överallt i programmet, en private metod kan endast anropas från samma klass

```
class Animal {
        int numberOfLegs = 1;
        boolean canFly= false;

        public void printData(){
                System.out.println(canFly);
                System.out.println(numberOfLegs);
}
```

Klasser -konstruktor

```
class Scratch {
    public static void main(String[] args) {
        Animal dog = new Animal();

    }
}
```

En konstruktor är en speciell metod som anropas när man skapar ett objekt av klassen

Man skapar en konstruktor genom att skapa en metod med **samma namn som klassen**

```
class Animal {
    int legs;
    boolean canFly;

    public Animal(){
        legs = 4;
        canFly = false;
    }
}
```

Klasser konstruktor

Med hjälp av konstruktorer så kan vi även **skicka in argument** när vi skapar ett nytt objekt. På så sätt kan vi exempelvis skapa olika djur i vårt exempel

```
class Animal {
    int legs;
    boolean canFly;

    public Animal(boolean flying, int numberOfLegs){
        legs = numberOfLegs;
        canFly = flying;
    }
}
```

```
class Scratch {
    public static void main(String[] args) {

        Animal dog = new Animal( flying: false,  numberOfLegs: 4);
        Animal bird = new Animal( flying: true,  numberOfLegs: 2);
    }
}
```

Klasser -encapsulation

Ofta vill man “skydda” data från att ändras av misstag i en klass.

därför sätter man oftast attributen till **private**

Sen skapar man speciella **public** metoder som kan ändra på dessa, kallas för **Setters** och **Getters**

Detta förfarande kallas **Encapsulation**

Klasser -getters och setters

```java
class Animal {
    private int legs;
    private boolean canFly;

    public Animal(boolean flying, int numberOfLegs){
        legs = numberOfLegs;
        canFly = flying;
    }

    public int getLegs() {
        return legs;
    }

    public void setLegs(int legs) {
        this.legs = legs;
    }

    public boolean isCanFly() {
        return canFly;
    }

    public void setCanFly(boolean canFly) {
        this.canFly = canFly;
    }
}
```

```java
class Scratch {
    public static void main(String[] args) {

        Animal dog = new Animal( flying: false,  numberOfLegs: 4);
        dog.setCanFly(true);
        System.out.println("antal ben: " + dog.getLegs());
    }
}
```

this

Eftersom vi kan skapa flera objekt av samma klass så vill vi ofta ha en referens till det egna objektet. Exempelvis om vi vill skicka in ett objekt som argument till en metod. Nyckelordet **this** är ett objekts referens till sig själv.

Detta används exempelvis i konstruktorer när man tilldelar värden till attributen. Då kan man lätt skilja på lokala variabler och medlemsvariabler (klass variabler) även om de har samma namn (vilket de bör ha)

```
class Animal {
    private int legs;
    private boolean canFly;

    public Animal(boolean canFly, int legs){
        this.legs = legs;
        this.canFly = canFly;
    }
```

toString

När vi skapar ett objekt av en klass så är objektet en sk **referensvariabel.** Värdet på själva variabeln pekar mao på en plats i minnet på samma sätt som Arrayer gör.

Försöker vi skriva ut objektet får vi därför en konstig utskrift som default.

Vill vi däremot skriva ut objektets värden kan vi skapa en egen **toString()** metod som skriver över (override) defaultutskriften.

```
@Override
public String toString() {
    return "Animal{" +
            "legs=" + legs +
            ", canFly=" + canFly +
            '}';
}
```