Programmering 2

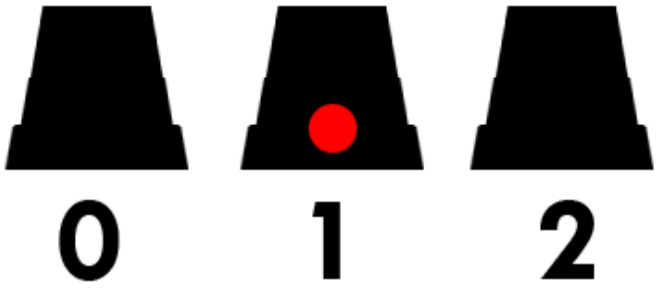
Övning 2 – Bibliotekarien

I kursens andra uppgift, ”Bibliotekarien”, så ska vi arbeta mer med konceptet av inkapsling och polymorfism. Tillsammans med Arv är dessa tre koncept de tre grundläggande för objektorienterar programmering. Mer information om inkapsling finns i ett separat PDF-dokument, men i det här dokumentet kommer vi titta på några exempel.

Inkapsling

Precis som tidigare nämnt så arbetat programmering utifrån att försöka efterlikna den riktiga världen. Ibland kan det vara lämpligt att dölja information, för att göra interaktion med en klass enhetlig och enkel.

Säg att vi skapar det klassiska spelet där vi gömmer en boll under en av tre möjliga koppar. Användaren vet här inte vilken kopp som bollen ligger under.



Här kan det vara användbart att använda sig av inkapsling. Det här är information som är dold, och det enda sättet att veta vilken kopp som bollen ligger inom ska vara genom att lyfta på koppen. Låt oss göra ett exempel av det här. Vi börjar med den grundläggande klassen;

```
public class KoppSpel
{
    private  int BollPlats;

    public KoppSpel(Random rng)
    {
        BollPlats = rng.Next(0, 3);
    }
}
```

Här skapar vi en klass kallad Bollspel som initieras med ett Random-objekt. Vi skapar en privat int-variabel kallad BollPlats, som sparar om bollen ligger på plats 0, 1 eller 2 (se bild). Vi kan inte komma åt bollplatsen genom att anropa den direkt. Användaren kan inte se, direkt, var den ligger. Istället måste vi lyfta på kopparna. Därför behöver vi en metod där vi låter användaren lyfta på en kopp, och se efter om bollen finns där. Exempelvis såhär;

```
public bool LyftaKopp(int kopp)
    {
        if (kopp < 1 || kopp > 3)
        {
            Console.WriteLine("\tDu får bara gissa mellan 1 till 3!");
            return false;
        }
        else if ((kopp - 1) == BollPlats)
            return true;
        else
            return false;
    }
```

Här skapar vi då metoden LyftaKopp. Den tar emot användarens gissning (en int där de gissar 1-3, för användarvänlighetens skull), och vi testar värdet mot värdet där bollen är sparad. Om de har gissat rätt så returneras ett sant boolvärde, och i alla övriga fall returneras ett falskt värde. På det här sättet kan vi skriva hela klassen såhär;

```
public class KoppSpel
{
    private int BollPlats;

    public KoppSpel(Random rng)
    {
        BollPlats = rng.Next(0, 3);
    }

    public bool LyftaKopp(int kopp)
    {
        if (kopp < 1 || kopp > 3)
        {
            Console.WriteLine("\tDu får bara gissa mellan 1 till 3!");
            return false;
        }
        else if ((kopp - 1) == BollPlats)
            return true;
        else
            return false;
    }

}
```

Polymorfism

Om vi arbetar vidare på samma exempel, så kan det hända att vi kanske har ett program med flera olika spel. Kanske finns det flera svårighetsgrader. Säg att vi vill ha ett spelläge där användaren istället får gissa mellan 5 olika koppar, istället för 3. Istället för att skriva en klass med helt andra namn så kan vi istället använda Polymorfism. Konceptet bygger på att samma Metodnamn får olika funktioner beroende på kontexten där det åkallas.

Så säg att vi vill göra en ny underklass av BollSpel kallad SvårtBollSpel, som ska fungera likadant, men ge lite andra meddelanden och ha lite annorlunda egenskaper. Vi börjar med att skapa underklassen (se nästa sida)…

```
public class SvårtKoppSpel : KoppSpel
```

… sedan lägger vi till konstruktorn, och ändrar koden till att passa ett större spel…

```
public class SvårtKoppSpel : KoppSpel
{
    public SvårtKoppSpel(Random rng) : base(rng)
    {
        BollPlats = rng.Next(0, 6);
    }
}
```

Här kommer dock vårt första fel. Då ”BollPlats” är en privat int så kan vi *inte* komma åt den utifrån. Dock så vill vi att underklasser ska kunna göra det. Vi kan då kolla igenom listan av möjliga åtkomstnivåer och finna att ”protected” fungerar lite som private, förutom att den tillåter ärvda klasser att påverka egenskapen. Därför ändrar vi åtkomstnivån av BollPlats, i originalklassen, till protected istället för private.

Sedan lägger vi till vår modifierade metod.

```
public override bool LyftaKopp(int kopp)
    {
        if (kopp < 1 || kopp > 6)
        {
            Console.WriteLine("\tDu får bara gissa mellan 1 till 6! Det här är ett
svårare spel.");
            return false;
        }
        else if ((kopp - 1) == BollPlats)
            return true;
        else
            return false;
    }
```

Vi kan se att den här koden ser lite annorlunda ut. Vi har ändrat utskriften och villkoren. Vi kan även se ett nytt ord; override. Override betyder i kort att den här metoden blir kallad över grundmetoden, förutsatt att vi kallar på den här metoden från underklassens objekt. Så om vi kallar på BollSpel.LyftaKopp() så vill vi att grundklassens metod kallas, medan om vi istället vill kalla på SvårtBollSpel.LyftaKopp() så ska underklassens metod kallas. Därför anger vi override i underklassens LyftaKopp-metod.

Här kommer dock vårt andra fel; vi kan *inte* ange en override av en metod som inte är markerad som antingen Abstract eller Virtual. Så vad är dessa, och vad bör vi använda?

En Abstract-metod är en metod som används som en grund för underklassernas override. Om det är en metod som vi inte kan kalla på, men som vi vill ha som en mall för andra metoder. I det här fallet vill vi dock kunna kalla på BollSpel.LyftaKopp, så då är det bättre att vi använder oss av Virtual.

Virtual fungerar som Abstract, med en skillnad; den går att kalla på. Vi skriver om grundklassens LyftaKopp-metod till att vara en `public virtual bool`.

Det här är ett exempel hur vi kan arbeta med polymorfism. Vi kan även använda override för att skriva över allmänna metoder, exempelvis ToString(). På det sättet kan vi bestämma vilket meddelande som skrivs ut när vi kallar på objektet metoden tillhör. Härefter följer båda klasserna, med var sin ToString-override. Testa gärna att skriva ett spel som använder klasserna!

```
public class KoppSpel
{
    protected int BollPlats;
    public KoppSpel(Random rng)
    {
        BollPlats = rng.Next(0, 3);
    }
    public virtual bool LyftaKopp(int kopp)
    {
        if (kopp < 1 || kopp > 3)
        {
            Console.WriteLine("\tDu får bara gissa mellan 1 till 3!");
            return false;
        }
        else if ((kopp - 1) == BollPlats)
            return true;
        else
            return false;
    }
    public override string ToString()
    {
        return "Det här är ett spel där du gissar på ett värde mellan 1-3";
    }
}
public class SvårtKoppSpel : KoppSpel
{
    public SvårtKoppSpel(Random rng) : base(rng)
    {
        BollPlats = rng.Next(0, 6);
    }
    public override bool LyftaKopp(int kopp)
    {
        if (kopp < 1 || kopp > 6)
        {
            Console.WriteLine("\tDu får bara gissa mellan 1 till 6!" +
            "\n\tDet här är ett svårare spel.");
        return false;
        }
        else if ((kopp - 1) == BollPlats)
            return true;
        else
            return false;
    }
    public override string ToString()
    {
        return "Det här är ett spel där du gissar på ett värde mellan 1-6";
    }
}
```