Tärningarna ska kastas

Upphovsrätt för detta verk

Detta verk är framtaget i anslutning till kursen Inledande programmering med C# vid Linnéuniversitetet.

Du får använda detta verk så här:

Du har ett problem

- Du ska skriva ett C#-program som simulerar tärningskast med en eller flera tärningar som har sex sidor.

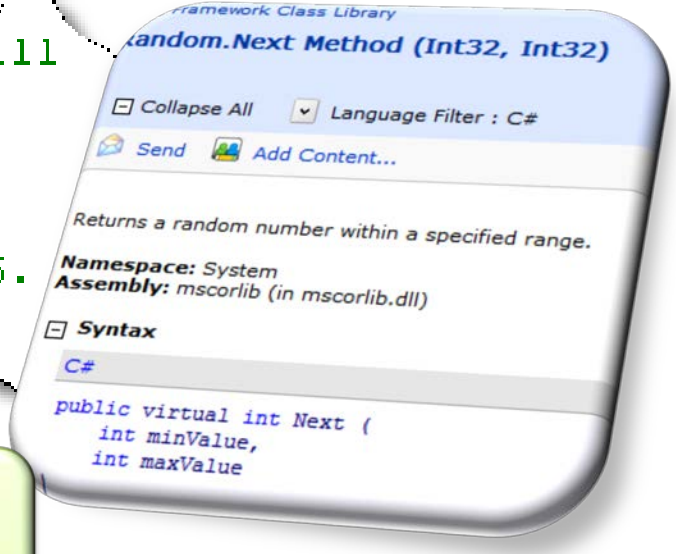
Du måste först lösa hur du gör för…

- ✓ …att skapa ett slumptal som har värdet 1, 2, 3, 4, 5, eller 6.

Klassbiblioteket i dotnetramverket innehåller klassen `Random` som du kan använda till att generera slumptal.

```
// Referensvariabeln random refererar till
// det instansierade Random-objektet.
Random random = new Random();

// Ger slumptalet 1, 2, 3, 4 , 5 eller 6.
int value = random.Next(1, 7);
```



Metoden `Next(int minValue, int maxValue)` som `Random`-objektet anropar via referensvariabeln returnerar ett slumptal inom det angivna området (inklusive minvärdet, exklusive maxvärdet).

Problemet löst men…

```
// Referensvariabeln random r
// det instansierade Random-ob
Random random = new Random();

// Ger slumptalet 1, 2,          eller 6.
int value = random.Nex
```

Hur enkelt är det att förstå att koden handlar om ett tärningskast?

Inte intuitivt uppenbart!

Går det inte att efterlikna ett verkligt tärningskast mer?

En idé! Skulle jag inte…

- …kunna skapa ett tärningsobjekt, på samma sätt som `Random`-objektet?

Vad utmärker en tärning?

- För att kunna skriva en tärningsklass måste du först identifiera vad som gör en tärning till en tärning.
  - Något som beskriver tärningen? (attribut)
    - Antalet prickar som visas. (*eng. face value*)
  - Något du kan göra med tärningen? (operation)
    - Slå tärningen. (*eng. throw*)

Det finns andra attribut, som färg, storlek, etc. men de är inte intressanta att lägga in i en klass i detta fall.

…och man kan ju t.ex. skaka tärningen (en operation), men det är inte heller intressant.

Ett första utkast till en tärningsklass

I C# definieras en ny klass med nyckelordet `class`, ett namn och ett par klammerparenteser.

```
class Die
{
    public int _faceValue;

    public int Throw()
    {
        Random random = new Random();
        _faceValue = random.Next(1, 7);
        return _faceValue;
    }
}
```

Klassen har ett fält som beskriver antalet prickar tärningen visar.

Metoden `Throw()` simulerar ett tärningskast genom att det genererade slumptalet sparas i fältet `_faceValue`. Värdet som `_faceValue` tilldelats returneras därefter.

Fungerar klassen bra nu? Eller…

```
// Instansierar ett Die-objekt. Referens-
// variabeln myDie refererar till det nya
// Die-objektet
Die myDie = new Die();

// Kastar tärningen och lagrar antalet
// prickar metoden returnerar i en
// lokal variabel.
int value = myDie.Throw();

// Presenterar antalet prickar.
Console.WriteLine(value);

// Presenterar antalet prickar genom
// att använda fältet.
Console.WriteLine(myDie._faceValue);
```

Nu är det enklare att förstå att det handlar om tärningskast. Så långt så bra…

Fältet `_faceValue` är publikt och därmed helt oskyddat.

Av ”misstag” skulle det vara möjligt att ändra antalet prickar till ett värde mindre än 1 eller större än 6. INTE BRA!

Skydda datat!

'RollTheDie.Die._faceValue' is inaccessible due to its protection level

Fältet deklareras som ett privat fält och är inte tillgängligt utanför klassen.

```
class Die
{
    private int _faceValue;

    public int Throw()
    {
        Random random = new Random();
        _faceValue = random.Next(1, 7);
        return _faceValue;
    }

    public int FaceValue
    {
        get { return _faceValue; }
        set { _faceValue = value; }
    }

}
```

Via en egenskap görs det privata fältet `_faceValue` tillgängligt.

En egenskap är en blandning mellan ett fält och en metod, och innehåller två metoder som inleds med nyckelorden get och set.

- `get`-metoden innehåller kod som körs då egenskapen läses.
- `set`-metoden innehåller kod som körs då något tilldelas egenskapen. `value` är en dold parameter av samma typ som egenskapen.

Ett steg i rätt riktning, men fortfarande går det av ”misstag” att ändra antalet prickar via egenskapen till ett värde mindre än 1 eller större än 6.

Kontrollera datat

```
public int FaceValue
{
    get { return _faceValue; }
    set
    {
        if (value < 1 ||
            value > 6)
        {
            throw new ArgumentOutOfRangeException();
        }
        _faceValue = value;
    }
}
```

Genom att låta `set`-metoden i egenskapen kasta ett undantag om värdet är mindre än 1 eller större än 6 är det helt omöjligt att tilldela fältet `_faceValue` ett ogiltigt värde med hjälp av egenskapen.

Bra! Nu är fältet som representerar resultatet av ett tärningskast skyddat.

Fungerar klassen bra nu då? Eller…

```
// Instansierar ett Die-objekt.
Die myDie = new Die();

// Presenterar antalet prickar genom
// att använda egenskapen.
Console.WriteLine(myDie.FaceValue);

// Leder till att ett undantag kastas,
// varför satsen är bortkommenterad.
//myDie.FaceValue = 13;

// Kastar tärningen. (Behöver inte ta hand
// om det metoden returnerar.)
myDie.Throw();

// Presenterar antalet prickar.
Console.WriteLine(myDie.FaceValue);
```

Ingen större skillnad mot förra försöket. Nu är det dock helt omöjligt att skapa ett `Die`-objekt med ett annat värde än 1, 2, 3, 4, 5 eller 6. Eller?

Vad är det som skrivs ut? Var kommer 0 ifrån?

`Die`-objektet har inte initierats till ett giltigt värde. Fältet `_faceValue` får värdet 0 då ett nytt objekt instansieras. Du måste använda en konstruktor!

En egen standardkonstruktor

```
class Die
{
    private int _faceValue;

    public Die()
    {
        Throw();
    }

    public int FaceValue
    {
        get { return _faceValue; }
        set
        {
            if (value < 1 ||
                value > 6)
            {
                throw new ArgumentOutOfRangeExcepti
            }
            _faceValue = value;
        }
    }

    public int Throw()
    {
        Random random = new Ra
        _faceValue = random.Next(1,
        return _faceValue;
    }
```

Klassen har kompletterats med en egen standardkonstruktor, en konstruktor som inte tar några parametrar.

Konstruktorn är en speciell publik metod som har samma namn som klassen men kan inte returnera något värde (inte ens `void`).

Genom att anropa `Throw()` tilldelas `_faceValue` ett giltigt värde.

När du skapar ett nytt objekt med new, skapas objektet av ”Common Language Runtime” (CLR) som då använder klassdefinitionen.

Minne allokeras till fälten som definieras av klassen och sedan anropas konstruktorn för att utföra den initiering av fälten som krävs.

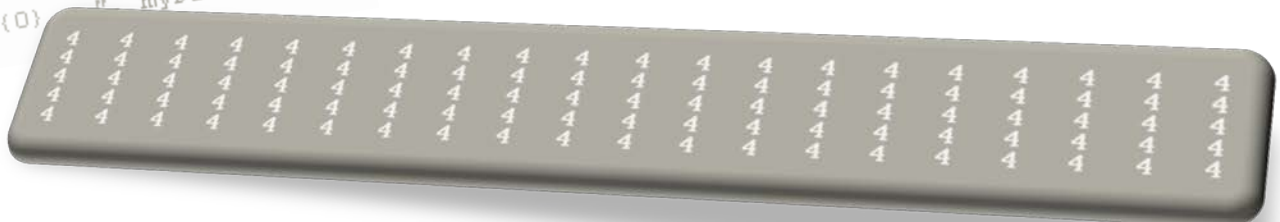
Ett stort men…

Mats har gjort fel! Klassen är i och för sig komplett gällande fält, egenskaper, konstruktorer och metoder. Men den fungerar inte bra…

Ja, jag vet. Problemet är metoden `Throw()` som ju ger samma resultat om jag så anropar den 100 gånger.

```
Die myDie = new Die();

for (int i = 0; i < 100; ++i)
{
    Console.Write("{0} ", myDie.Throw());
}
```



Felet identifierat…

```
public int Throw()
{
    Random random = new Random();
    _faceValue = random.Next(1, 7);
    return _faceValue;
}
```

Problemet är att ett nytt `Random`-objekt skapas varje gång metoden anropas.

Då metoden anropas i en ”for”-sats går det så fort att datorns klocka inte hinner gå, och klassen `Random` använder datorns tid för att skapa det första slumptalet. Samma tid ger samma slumptal!

Aha! OK! Då förstår du säkert att du bara behöver se till att det skapas ett `Random`-objekt per tärning. Eller hur!!?

…och åtgärdat

```
class Die
{
    private int _faceValue;
    private Random _random;

    public Die()
    {
        _random = new Random();
        Throw();
    }

    public int FaceValue
    {
        get { return _faceValue; }
        set
        {
            if (value < 1 ||
                value > 6)
            {
                throw new ArgumentOutOfRangeException();
            }
            _faceValue = value;
        }
    }

    public int Throw()
    {
        _faceValue = _random.Next(1, 7);
        return _faceValue;
    }
}
```

Genom låta `Random`-objektet bli ett fält och instansiera objektet i konstruktorn kommer det bara att finnas ett enda `Random`-objekt per `Die`-objekt, och…

…nu kommer olika värden returneras då metoden `Throw()` anropas i en ”for”-sats.

Ett litet men…

Det där med Random är ju fixat. Men varför fungerar det inte nu då?

Vet inte. Men problemet är att då jag skapar två Die-objekt får jag samma serie!

```
Die myDie = new Die();
Die myOtherDie = new Die();

for (int i = 0; i < 100; ++i)
{
    Console.WriteLine("myDie: {0}\tmyOtherDie: {1}",
        myDie.Throw(), myOtherDie.Throw());
}
```

```
myDie: 2        myOtherDie: 2
myDie: 2        myOtherDie: 2
myDie: 1        myOtherDie: 1
myDie: 1        myOtherDie: 1
myDie: 6        myOtherDie: 6
myDie: 2        myOtherDie: 2
myDie: 2        myOtherDie: 2
myDie: 4        myOtherDie: 4
myDie: 5        myOtherDie: 5
myDie: 5        myOtherDie: 5
myDie: 2        myOtherDie: 2
myDie: 2        myOtherDie: 2
myDie: 5        myOtherDie: 5
myDie: 4        myOtherDie: 4
```

Ännu ett fel identifierat…

```
Die myDie = new Die();
Die myOtherDie = new Die();

for (int i = 0; i < 100; ++i)
{
    Console.WriteLine("myDie: {0}\tmyOther
        myDie.Throw(), myOtherDie.Throw());
}
```

Problemet är att då två `Die`-objekt skapas omedelbart efter varandra utgår `Die`-objektens respektive `Random`-objekt från samma tidpunkt, samma slumptalsfrö.

Rätt slutsats! Istället för att använda datorns klocka, varför då inte slumpa slumptalsfröet som `Random`-objekten ska utgå ifrån?

…och åtgärdat

```
class Die
{
    private static readonly Random _randomSeed = new Random();

    private int _faceValue;
    private Random _random;

    public Die()
    {
        _random = new Random(_randomSeed.Next());
        Throw();
    }

    public int FaceValue
    {
        get { return _faceValue; }
        set
        {
            if (value < 1 ||
                value > 6)
            {
                throw new ArgumentOutOfRangeException();
            }
            _faceValue = value;
        }
    }

    public int Throw()
    {
        _faceValue = _random.Next(1, 7);
        return _faceValue;
    }
}
```

Klassen har kompletterats med ett statisk ”*read-only*” `Random`-referens.

- ✓ Att referensen är statiskt innebär att den är gemensamt för alla objekt som instansieras av klassen. Det finns med andra ord bara ett enda `_randomSeed`-objekt.
- ✓ `readonly` har i princip samma effekt som `const`.

Då ett nytt `Die`-objekt skapas instansieras samtidigt ett nytt `Random`-objekt med ett framslumpat slumptalsfrö.

Inga fler men

```
class Die
{
    private static readonly Random _randomSeed = new Random();

    private int _faceValue;
    private Random _random;

    public Die()
    {
        _random = new Random(_randomSeed.Next());
        Throw();
    }

    public int FaceValue
    {
        get { return _faceValue; }
        set
        {
            if (value < 1 ||
                value > 6)
            {
                throw new ArgumentOutOfRang
            }
            _faceValue = value;
        }
    }

    public int Throw()
    {
        _faceValue = _random.Next(
        return _faceValue;
    }
}
```

```
Die myDie = new Die();
Die myOtherDie = new Die();

for (int i = 0; i < 100; ++i)
{
    Console.WriteLine("myDie: {0}\tmyOtherDie: {1}",
        myDie.Throw(), myOtherDie.Throw());
}
```

```
myDie: 6    myOtherDie: 4
myDie: 1    myOtherDie: 5
myDie: 6    myOtherDie: 3
myDie: 4    myOtherDie: 5
myDie: 2    myOtherDie: 2
myDie: 6    myOtherDie: 5
myDie: 5    myOtherDie: 5
myDie: 4    myOtherDie: 2
myDie: 2    myOtherDie: 2
myDie: 2    myOtherDie: 1
myDie: 2    myOtherDie: 3
myDie: 5    myOtherDie: 1
myDie: 4    myOtherDie: 3
myDie: 1    myOtherDie: 1
myDie: 6    myOtherDie: 3
myDie: 4    myOtherDie: 3
myDie: 1    myOtherDie: 5
myDie: 6    myOtherDie: 2
myDie: 6    myOtherDie: 4
myDie: 2    myOtherDie: 3
myDie: 3    myOtherDie: 5
myDie: 2    myOtherDie: 2
myDie: 1    myOtherDie: 4
myDie: 5    myOtherDie: 2
myDie: 1    myOtherDie: 3
```

Nu ska du kunna

- ✓ Definiera en klass innehållande data och metoder.
- ✓ Skapa objekt med `new` och en konstruktor.
- ✓ Förstå hur `private` och `public` används.
- ✓ Använda egenskaper för att kapsla in fält.
- ✓ Skapa data som delas mellan alla instanser av samma klass, med `static`.