Föreläsning 6

Mer om klasser och objekt

Enkla variabler kontra referensvariabel

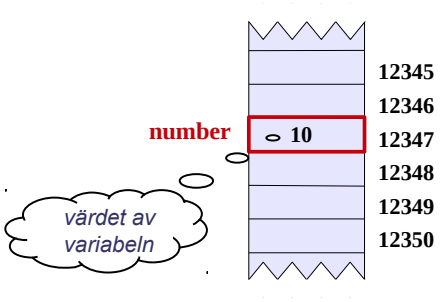
En variabel är ett *namngivet* minnesutrymme i datorns primärminne.

En variabel som används för att representera en primitiv datatyp kallas för ***enkel variabel***.

Deklarationssatsen

```
int number = 10;
```

innebär att en heltalsvariabel number deklareras och initieras till värdet 10. Detta kan illustreras med vidstående bild:

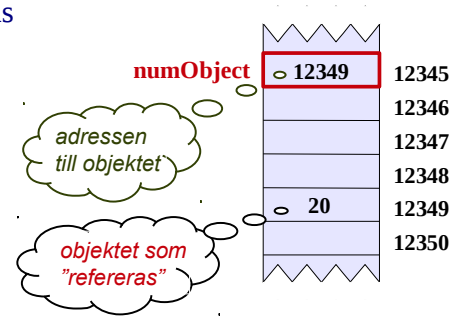


En variabel som används för att representera en instans av en klass kallas för ***referensvariabel***.

Deklarationssatsen

```
Integer numObject = new Integer(20);
```

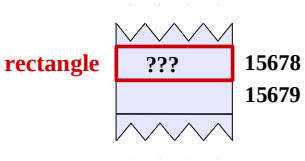
innebär att en referensvariabel numObject av typen Integer deklareras och initieras till att referera till ett objekt av typen Integer som har värdet 20. Detta kan illustreras med vidstående bild:



Referensvariabler

En referensvariabel är en variabel som kan lagra adresser och vars datatyp är en klass:

Rectangle rectangle;

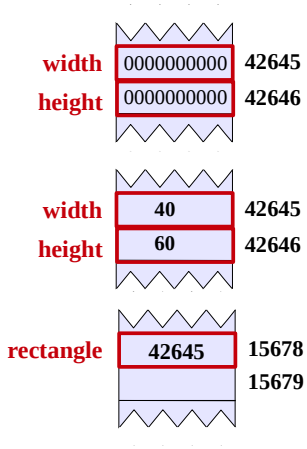


Enda sättet att skapa ett objekt är med operatorn **new** (och en konstruktor):

rectangle = **new** Rectangle(40, 60);

Följande händer vid anrop av konstruktorn:

1. Minne för objektets instansvariabler allokeras på någon *ledig* adress och detta minne *nollställs*.
2. Eventuella initieringar av instansvariablerna görs och satserna i konstruktorn exekveras.
3. Adressen till minnet som allokerades för objektet ges som returvärde från konstruktorn.



Omslagsklassen Integer

Constructors

Constructor and Description
`Integer(int value)` Constructs a newly allocated Integer object that represents the specified int value.
`Integer(String s)` Constructs a newly allocated Integer object that represents the int value indicated by the String parameter.

Method Summary

Methods

Modifier and Type	Method and Description
static int	`bitCount(int i)` Returns the number of one-bits in the two's complement binary representation of the specified int value.
byte	`byteValue()` Returns the value of this Integer as a byte.
static int	`compare(int x, int y)` Compares two int values numerically.
int	`compareTo(Integer anotherInteger)` Compares two Integer objects numerically.
static Integer	`decode(String nm)` Decodes a String into an Integer.
double	`doubleValue()` Returns the value of this Integer as a double.
boolean	`equals(Object obj)` Compares this object to the specified object.

Omslagsklassen Integer

int	**intValue**() Returns the value of this Integer as an int.
long	**longValue**() Returns the value of this Integer as a long.
static int	**lowestOneBit**(int i) Returns an int value with at most a single one-bit, in the position of the lowest-order ("rightmost") one-bit in the specified int value.
static int	**numberOfLeadingZeros**(int i) Returns the number of zero bits preceding the highest-order ("leftmost") one-bit in the two's complement binary representation of the specified int value.
static int	**numberOfTrailingZeros**(int i) Returns the number of zero bits following the lowest-order ("rightmost") one-bit in the two's complement binary representation of the specified int value.
static int	**parseInt**(String s) Parses the string argument as a signed decimal integer.

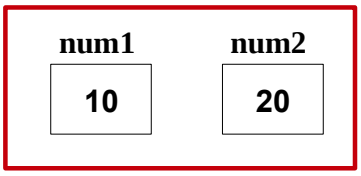
static String	**toHexString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 16.
static String	**toOctalString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 8.
String	**toString**() Returns a String object representing this Integer's value.
static String	**toString**(int i) Returns a String object representing the specified integer.

Skillnaden mellan en variabel och en referensvariabel

Om vi gör nedanstående deklarationer

```
int num1 = 10;
int num2 = 20;
```

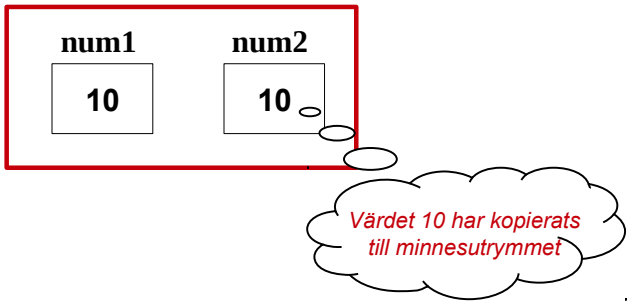
kan detta illustreras med följande bild



Görs sedan satsen

```
num2 = num1;
```

får vi bilden

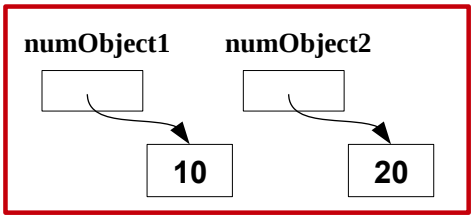


Skillnaden mellan en variabel och en referensvariabel

Om vi gör nedanstående deklarationer

```
Integer numObject1 = new Integer(10);
Integer numObject2 = new Integer(20);
```

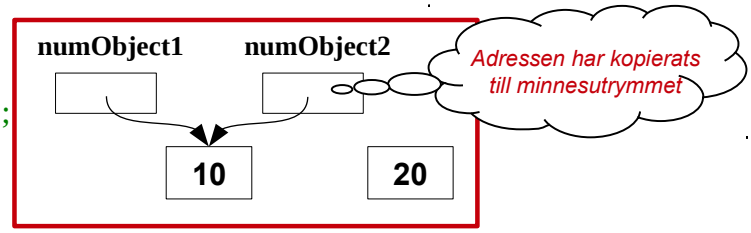
kan detta illustreras med följande bild:



Görs sedan satsen

```
numObject2 = numObject1;
```

får vi bilden:



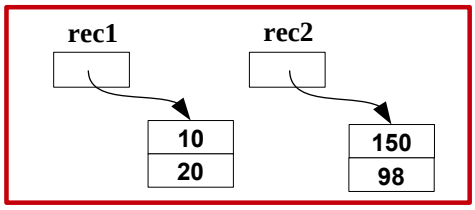
Ett litet program

```
public class TestRectangle {
   public static void main (String[] args) {
      Rectangle rec1 = new Rectangle(10, 20);
      Rectangle rec2 = new Rectangle(150, 98);
      System.out.println("Rec1:\n" + rec1 + "\nRec2: \n" + rec2);
      rec2 = rec1;
      rec1.setWidth(50);
      rec2.setHeight(200);
      System.out.println("Rec1:\n" + rec1 + "\nRec2: \n" + rec2);
   }//main
}//TestRectangle
```

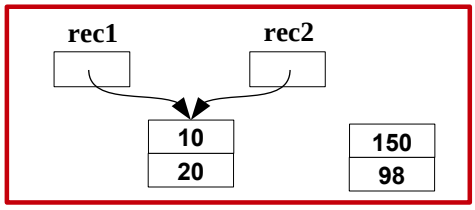
Vad blir utskriften?

Förklaring

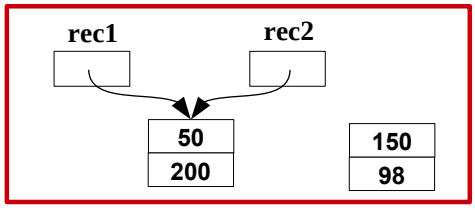
```
Rectangle rec1 = new Rectangle(10, 20);
Rectangle rec2 = new Rectangle(150, 98);
```




```
rec2 = rec1;
```




```
rec1.setWidth(50);
rec2.setHeight(200);
```

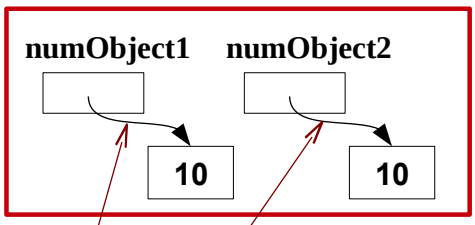


När är två objekt lika?

Om vi gör följande deklarationer

```
Integer numObject1 = new Integer(10);
Integer numObject2 = new Integer(10);
```

kan detta illustreras med följande bild:



Vad kommer jämförelsen

```
(numObject1 == numObject2)
```

att få för värde?

Resultatet blir **false**, eftersom det är *referenserna* till objekten som jämförs!!

Operatorn '==' jämför på *identitet*, inte på likhet!

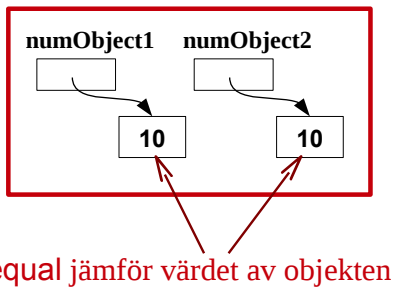
Metoden equals

För att jämföra likhet på själva objekten tillhandahåller klassen Integer instansmetoden equals.

Anropet

numObject1.equals(numObject2)

kommer att returnera värdet **true**.



equal jämför värdet av objekten

Alla klasser ärver metoden equals från klassen Object, men denna metod jämför på identitet (==). Vill man jämföra två objekt av en viss klass på likhet måste klassen *överskugga* instansmetoden equals.

Uppgift:

Utöka klassen Rectangle med instansmetoden equals som jämför om två objekt är lika! Hur skall likhet definieras?

Implementation av equals i Rectangle

```
public class Rectangle {
  private double width;
  private double height;
  . . .
  public boolean equals(Object otherObject) {
    if (otherObject == null)
      return false;
    if (otherObject.getClass() != this.getClass())
      return false;
    Rectangle other = (Rectangle) otherObject;
    return this.width == other.width && this.height == other.height;
  }//equals
}//Rectangle
```

*Konstruktorer och metoder har i en klass access till instansvariablerna i alla objekt av klassen! Det aktuella objektet (**this**) har således access till instansvariablerna hos objektet other även om dessa är angivna som **private**!*

Default-konstruktor

Om man i en klass inte definierar någon konstruktor, skapas automatiskt en parameterlös **default-konstruktor**. Denna konstruktor är ekvivalent med den explicita definitionen:

```
public SomeClass(){ }
```

Om *någon* konstruktor definieras i klassen skapas ingen default-konstruktor. Om man då vill ha tillgång till en parameterlös konstruktor måste denna definieras i klassen:

```
public SomeClass(int value){
   //some stuff
}
public SomeClass(){ }
```

Om klassen endast har behov av den parameterlösa konstruktorn är det bättre att göra en explicit definition av denna än att nyttja default-konstruktorn.

Klassvariabler och klassmetoder

En klass *kan* innehålla klassvariabler och klassmetoder.

Det som skiljer en klassvariabel från en instansvariabel och en klassmetod från en instansmetod är att:

- klassvariabler och klassmetoder är *gemensamma för alla objekt som tillhör en klass*
- instansvariabler och instansmetoder finns i *unika uppsättningar i varje objekt som tillhör en klass.*

Detta betyder att:

- metoder som påverkar någon instansvariabel eller behöver tillgång till någon instansvariabel för att kunna fullgöra sin uppgift, *måste* implementeras som instansmetoder
- metoder som *inte* påverkar eller behöver tillgång till någon instansvariabel (måste inte men) *skall* implementeras som klassmetoder.

Klassvariabler och klassmetoder

Exempel:

Antag att vi har en klass Person för att avbilda personer. Antag vidare att vi i ett program skapat instanserna kalle och stina av klassen Person

```
Person kalle = new Person("Kalle");
Person stina = new Person("Stina");
```

I klassen Person finns vidare metoderna getPhoneNumber() och getPhoneNumber(Person p) som t.ex kan anropas på följande sätt:

```
PhoneNumber number1 = kalle.getPhoneNumber();
PhoneNumber number2 = kalle.getPhoneNumber(stina);
```

som levererar svaren på frågorna:

Kalle, vad har du för telefonnummer?

respektive

Kalle, vilket telefonnummer har Stina?

Metoden getPhoneNumber() måste vara en instansmetod, medan metoden getPhoneNumber(Person p) skall vara en klassmetod. **Varför?**

Klassvariabler och klassmetoder

Modifieraren **static** används för att ange att en variabel är en klassvariabel respektive att en metod är en klassmetod:

```
private static double rate;
public static final double G = 6.6743e-11;
public static double getPhoneNumber(Person p) {
   . . .
}//getPhoneNumber
public static void main(String[] arg) {
   . . .
}//main
```

Klassvariabler och klassmetoder

Access till publika klassattribut fås genom att ange klassens namn, följt av en punkt '.', följt av attributets namn

```
Math.PI
Interger.MAX_VALUE
```

Access till publika klassmetoder fås genom att ange klassens namn, följt av en punkt '.', följt av metodens namn och argumentlista.

```
String indata = JOptionPane.showInputDialog("Ge talet:");
double tal1 = Double.parseDouble(indata);
double tal2 = Math.pow(tal1, 3);
System.out.println("Det beräknade talet blev: " + tal2);
```

Har man skapat objekt av en klass kan, men skall inte, klassvariabler och klassmetoder refereras via dessa objekt. Varför inte? Ger otydlig kod!

```
~~PhoneNumber number2 = kalle.getPhoneNumber(stina);~~
PhoneNumber number2 = Person.getPhoneNumber(stina);
```

Exempel: Klassvariabler och klassmetoder

Skapa en klass Account för att avbilda bankkonton.

Ett konto skall innehålla uppgifter om

- vem som äger kontot
- kontonummer
- saldo
- räntesats

På ett konto skall det vara möjligt att

- göra transaktioner (insättningar eller uttag)
- ta reda på räntesatsen
- ta reda på saldo
- ta reda på kontots ägare
- ta reda på kontots kontonummer
- ändra räntesats
- lägga till årsräntan

Alla konton skall ha samma räntesats.

Alla konton skall ha unika kontonummer, vilket är ett löpnummer i den ordning konton skapas.

Implementation av klassen Account

```
public class Account {
   private static double rate;              //klassvariabel
   private static int currentNr = 0;        //klassvariabel
   private double balance;                  //instansvariabel
   private String owner;                    //instansvariabel
   private int number;                      //instansvariabel

   public Account(String owner) {
      this.owner = owner;
      currentNr = currentNr + 1;
      number = currentNr;
   } //constructor

   public static void setRate(double newRate) {
      rate = newRate;
   } //setRate

   public static double getRate() {
      return rate;
   } //getRate

   public double getBalance() {
      return balance;
   } //getBalance

   public int getNumber() {
      return number;
   } //getNumber

   public String getOwner() {
      return owner;
   } //getOwner

   public void transaction(double amount) {
      if (amount < 0 && balance + amount < 0)
         System.out.println("Not money enought!!");
      else
         balance = balance + amount;
   } //transaction

   public void addInterest(double interest) {
       balance = balance + balance*interest;
   }//addInterest

   //toString och equals bör också implementeras
} //Account
```

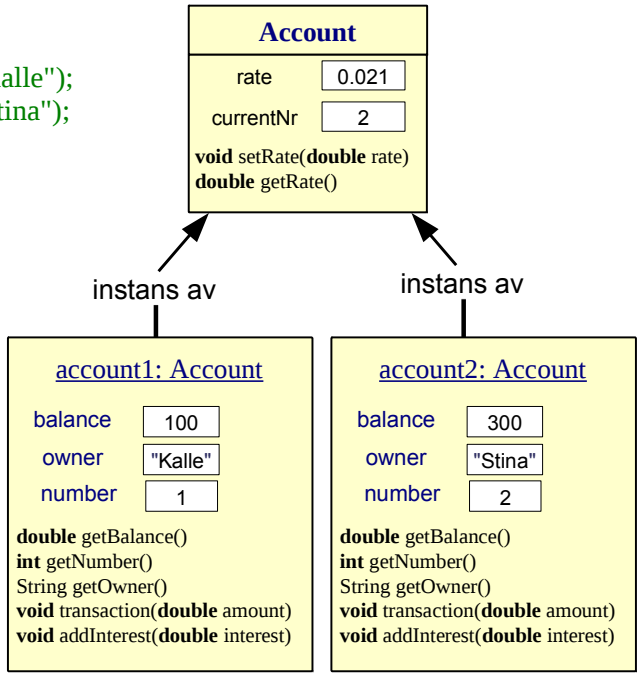
Klassvariabler och klassmetoder

Antag att följande satser har gjorts:

```
Account account1 = new Account("Kalle");
Account account2 = new Account("Stina");
Account.setRate(0.024);
account1.transaction(100);
account2.transaction(300);
Account.setRate(0.021);
```

Vidstående bild illustrerar vilka objekt som finns samt vilka värden respektive instansvariabel och klassvariabel har.

En klassmetod får inte referera till någon instansvariabler! Varför?



Oföränderliga objekt

```
public class Location {
    private int row;
    private int col;
    public Location(int row, int col) {
        this.row = row;
        this.col = col;
    } //constructor
    public int getRow() {
        return row;
    } //getRow
    public int getCol() {
        return col;
    } //getCol
```

Om en klass saknar omformare kan inte tillståndet hos ett objekt av denna klass förändras. Objektet är ***oföränderligt*** (*icke-muterbart*).

Det tillståndet som objektet initieras till när det skapas kommer att kvarstå under objekts hela livstid.

```
    public boolean equals(Object otherObject) {
        if (otherObject == null)
            return false;
        if (otherObject.getClass() != this.getClass())
            return false;
         Location other = (Location) otherObject;
        return row == other.getRow() && col == other.getCol();
    } //equals
    public String toString() {
        return "(" + row + ", " + col + ")";
    } //toString
} //Location
```

Man skall inte slentrianmässigt tillhandahålla omformare i alla klasser.

Problemexempel

Uppgift

Utveckla en klass för att avbilda tärningar.

Frågor att ta ställning till:

Vad skall vi kalla klassen?

- vi döper klassen till Die

Vilket beteende har en tärning (= vad vill man kunna göra med en tärning)?

- slå tärningen
- läsa av värdet på tärningen

Vilka tillstånd kan en tärning ha?

- värdet, dvs vilken siffra har ovansidan på tärningen

Specifikation av klassen Die

För att handha tärningens värde behövs en instansvariabel, som vi kan kalla dots. Eftersom värdet är ett heltal skall dots vara av typen **int**:

```
private int dots;
```

För att kasta tärningen behövs en instansmetod som slumpmässigt ändrar tärningens värde, låt oss kalla denna metod för roll:

public void roll()	kastar tärningen

För att läsa av tärningens värde behövs en instansmetod som returnerar värdet av instansvariabeln dots:

public int getValue()	returnerar tärningens värde

Vi väljer att låta klassen ha två konstruktorer. Den första konstruktorn sätter ett slumpmässigt startvärde (mellan 1 och 6) på tärningen. Den andra konstruktorn tillåter användaren att initiera tärningens startvärde.

public Die()	skapar en tärning med en slumpmässig sida upp
public Die(**int** dots)	skapar en tärning med sidan dots upp

Metoden roll använder ett objekt av klassen Random för att erhålla ett slumpmässigt värde. Detta objekt görs till en klassvariabel.

```
private static Random diceRandom = new Random();
```

Implementering av klassen Die

```
import java.util.Random;
public class Die {
   private int dots;
   private static Random dieRandom = new Random();
   public Die() {
      roll();
   }//constructor
   public Die(int dots) {
      if (dots >= 1 && dots <= 6)
         this.dots = dots;
      else
         roll();
   }//constructor
   public int getValue() {
      return dots;
   } //getValue
   public void roll() {
      dots = dieRandom.nextInt(6) + 1;
   } //roll
} //Die
```

Varför har dieRandom implementerats som en klassvariabel?

Finns det något snyggare sätt att lösa problemet som uppstår då användaren ger ett otillåtet värde på parametern?

Varför finns det ingen metod setValue för att ändra värdet på instansvariabeln dots?

Alternativ implementation: kastar ett exception

```
import java.util.Random;
public class Die {
   private int dots;
   private static Random dieRandom = new Random();
   public Die() {
      roll();
   }//constructor
   public Die(int dots) {
      if (dots < 1 || dots > 6)
         throw new IllegalArgumentException();
      else
         this.dots = dots;
   }//constructor
   public int getValue() {
      return dots;
   } //getValue
   public void roll() {
      dots = dieRandom.nextInt(6) + 1;
   } //roll
} //Die
```

Exceptions kommer att diskuteras något mer utförligt senare i kursen.

Problemexempel

Använd klassen Die för att skriv ett program som genom simulering beräknar sannolikheten för att erhålla den sammanlagda summan 7 när man kastar två tärningar.

Analys:

Diskussion: För att beräkna sannolikheten görs ett tillräckligt stort antal kast med tärningarna. Vi räknar hur många gånger de båda tärningarnas sammanlagda värde blir 7. Detta antal dividerat med totala antalet tärningskast ger sannolikheten för att erhålla värdet 7.

Utdata: Sannolikheten att erhålla värdet 7.

Design:

Algoritm:

1. Sätt *nrOfSeven* till 0.
2. Upprepa *nrOfTurns* gånger
   2.1. Kasta tärning *t1*.
   2.2. Kasta tärning *t2*.
   2.2. Läs av tärningarnas värden och lagra den sammanlagda summan i *sum*.
   2.3. Om *sum* har värdet 7 så
      öka *nrOfSeven* med 1.
3. Skriv ut värdet av *nrOfSeven / nfOfTurns*

Datarepresentation:

Tärningarna *t1* och *t2* är av klassen *Die*.

nrOfSeven, *nrOfTurns* och *sum* är av typen **int**.

Implementation:

```
// Programmet beräknar genom simulering sannolikheten för att erhålla
// värdet 7 vid kast med två tärningar
public class TestTwoDie {
   public static void main (String[] args) {
      Die die1 = new Die();
      Die die2 = new Die();
      final int nrOfTurns = 100000;
      int sum;
      int nrOfSeven = 0;
      for (int i = 1; i <= nrOfTurns; i = i + 1) {
         die1.roll();
         die2.roll();
         sum = die1.getValue() + die2.getValue();
         if (sum == 7)
            nrOfSeven = nrOfSeven + 1;
      }
      System.out.println("Sannolikheten att få värdet 7 är " + ((double) nrOfSeven) / nrOfTurns);
   } // main
} //TestTwoDie
```

Använd top-down design och generalisera så att sannolikheten för andra värden än 7 också kan beräknas.

Implementation:

```
public class TestTwoDie {
   public static void main (String[] args) {
      Die d1 = new Die();
      Die d2 = new Die();
      double percentage = Utility.computeProbability(d1, d2, 7, 100000);
      System.out.println("Sannolikheten att få värdet 7 är " + percentage);
    }//main
}//TestTwoDie
```

```
public class Utility {
    public static double computeProbability(Die die1, Die die2, int target, int nrOfTurns) {
      int sum;
      int nrOfTarget = 0;
      for (int i = 1; i <= nrOfTurns; i = i + 1) {
         die1.roll();
         die2.roll();
         sum = die1.getValue() + die2.getValue();
         if (sum == target)
            nrOfTarget = nrOfTarget + 1;
      }
      return (double) nrOfTarget / nrOfTurns;
   }//computeProbability
}//Utility
```

Interface

En tärning förknippas vanligtvis med 6 sidor, som i klassen Die. Men det finns tärningar med annat antal sidor än 6.

Det som är gemensamt för alla tärningar, oavsett antalet sidor, är att de kan kastas och att man kan avläsa vilket värde som kom upp. *De har samma gränssnitt.*

I Java finns en särskild konstruktion **interface** som kan ses som specifikationen för en klass, där gränssnittet för klassen anges men implementationen saknas.

```
public interface Rollable {
   public void roll();
   public int getValue();
}//Rollable
```

Interface

En alternativ realisering av klassen Die är att låta klassen implementera interfacet Rollable.

Enda ändringen i koden är att ange
implements Rollable
i klasshuvudet till Die, eftersom klassen redan har implementationer av de metoder som specificeras i Rollable.

Både en klass och ett interface definierar en *datatyp*. Om en klass implementerar ett interface blir klassens datatyp en ***subtyp*** till datatypen för interfacet.

En instans av klassen Die kommer därför att tillhöra både datatypen Die och datatypen Rollable.

```
import java.util.Random;
public class Die implements Rollable {
   private int dots;
   private static Random dieRandom = new Random();
   public Die() {
      roll();
   }//constructor
   public Die(int dots) {
      if (dots < 1 || dots > 6)
         throw new IllegalArgumentException();
      else
         this.dots = dots;
   }//constructor
   public int getValue() {
      return dots;
   } //getValue
   public void roll() {
      dots = dieRandom.nextInt(6) + 1;
   } //roll
} //Die
```

Interface

Om vi betraktar metoden computeProbability, ser vi att metoden har två referenser av typen Die. För dessa referenser anropar computeProbability metoderna roll() och getValue().

```
public static double computeProbability(Die die1, Die die2, int target, int nrOfTurns) {
   int sum;
   int nrOfTarget = 0;
   for (int i = 1; i <= nrOfTurns; i = i + 1) {
      die1.roll();
      die2.roll();
      sum = die1.getValue() + die2.getValue();
      if (sum == target)
         nrOfTarget = nrOfTarget + 1;
   }
   return (double) nrOfTarget / nrOfTurns;
} //computeProbability
```

Eftersom metoderna roll() och getValue() specificeras av interfacet Rollable, kan datatypen på referenserna die1 och die2 bytas till Rollable:

```
public static double computeProbability(Rollable die1, Rollable die2, int target, int nrOfTurns)
```

Interface

```
public static double computeProbability(Rollable die1, Rollable die2, int target, int nrOfTurns) {
   int sum;
   int nrOfTarget = 0;
   for (int i = 1; i <= nrOfTurns; i = i + 1) {
      die1.roll();
      die2.roll();
      sum = die1.getValue() + die2.getValue();
      if (sum == target)
         nrOfTarget = nrOfTarget + 1;
   }
   return (double) nrOfTarget / nrOfTurns;
} //computeProbability
```

I Java gäller att:

där det efterfrågas ett värde av datatypen C får man även ge ett värde som är en subtyp till C

Det vi åstadkommit är att metoden computeProbability kan användas för alla instanser av klasser som implementerar interfacet Rollable.

Antag att vi har en klass EightSideDie, som avbildar en tärning med 8 sidor, som implementerar interfacet Rollable. För att ta reda på sannolikheten för att få värdet 12 när man kastar två tärningar där den ena tärningen har 6 sidor och och den andra tärningen 8 sidor kan detta göras genom att anropa metoden computeProbability enligt:

```
double percentage = Utility.computeProbability(new Die(), new EightSideDie(), 12, 100000);
```

Relationer mellan objekt

I objektorienterad programmering samverkar ett antal objekt för att lösa ett givet problem. Objekten kan sinsemellan ha olika *relationer*.

”Är”-relationen: Innebär att en instans av en klass, förutom att tillhöra datatypen som definieras av klassen, även är en subtyp till en annan datatyp.

Används för att bygga hierarkiska relationer.

Åstadkoms genom att en klass implementerar ett interface (***specifikations arv***) eller att en klass utökar en annan klass (***implementations arv***). Arv kommer att behandlas utförligare senare i kursen.

Exempel: Ett objekt av typen Die är ett objekt av typen Rollable.

En partiledare är politiker.

En politiker är en person.

Relationer mellan objekt

”Har”-relationen: Innebär att ett objekt är uppbyggt av andra objekt. Är ett sätt att bygga upp ett system från delar där delarna göms för omgivningen. Endast helheten är synlig.

Kallas även för komposition

Exempel: En bil har en motor

En bil har en ratt


```
public class Car  {
   private Engine engine = new Engine(120);
   private SteeringWheel wheel = new SteringWheel();
   //stuff not shown here
   public void start() {
      engine.turnOn();
   }
}//Car
```

Relationer mellan objekt

”Känner till”-relationen: Innebär att det finns en relation mellan två objekt som har skapats oberoende av varandra och som kan fortleva utan varandra.

Om objekt A ”känner till” objekt B är objekt B, på ett eller annat sätt, synligt utanför objekt A.

Exempel:

En bil ”känner till” sin förare


```
public class Car  {
   private Engine engine = new Engine(120);
   private SteeringWheel wheel = new SteringWheel();
   private Person driver;
   //stuff not shown here
   public void changeDriver(Person driver) {
      this.driver = driver;
   }
}//Car
```