Föreläsning 14

- Relation mellan klasser
- this-referensen
- Objekt som parameter
- Överlagring av metoder
- Ett program med flera klasser

JF: 5.6-5.11

Association

En klass som använder en annan klass på något sätt är associerad till klassen. Exempel på detta är:

1. En metod i ClassB anropas från ClassA
2. En klassvariabel i ClassC används i ClassA
3. ClassA har en instansvariabel av typen ClassD

```
public ClassA {
   private int attr = new ClassD();          // punkt 3 ovan

   public void method() {
      ClassB loc = new ClassB();
      if( ClassC.pos > 10 ) {                // punkt 2 ovan
         loc.changeDirection();              // punkt 1 ovan
      }
   }
}
```

Association

```
public class Person {
   private String name;                                    // uses String
   private String id;  // Swedish personal number          // uses String
   private Address address;                                // uses Address

   public Person(String name, String id, Address address) {
      this.name = name;
      this.id = id;
      this.address = address;
   }

   // div set- och get-metoder

   public int getAge() {
      int year, yearOfBirth, age;
      Calendar cal = Calendar.getInstance();               // uses Calendar
      year = cal.get(Calendar.YEAR) % 100;
      yearOfBirth = Integer.parseInt(id.substring(0,2));   // uses Integer
      age = year - yearOfBirth;
      if( age<0) {
         age += 100;
      }
      return age;
   }
}
```

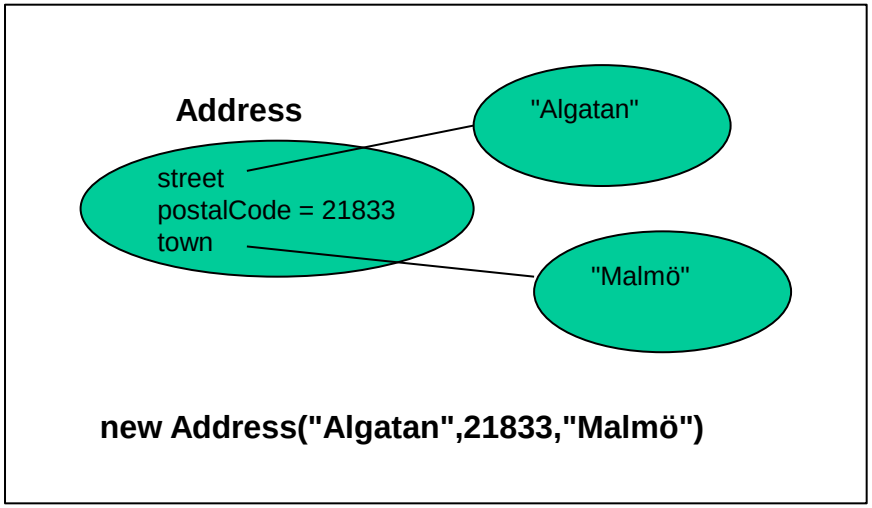
Person.java

PersonEx1.java

Aggregering - Har-relation

Om en klass har ett attribut (instansvariabel) till en annan klass föreligger en starkare association, en s.k. aggregering (***har***-relation). Klassen Address *har en gata* och *har en ort*. Båda attributen är av typen String.

```
public class Address {
   private String street;              // En adress har en gata
   private int postalCode;
   private String town;                // En adress har en ort

   public Address(String street, int postalCode, String town) {
      this.street = street;
      this.postalCode = postalCode;
      this.town = town;
   }

   public String getStreet() {
      return street;
   }

   public int getPostalCode() {
      return postalCode;
   }

   public String getTown() {
      return town;
   }
}
```



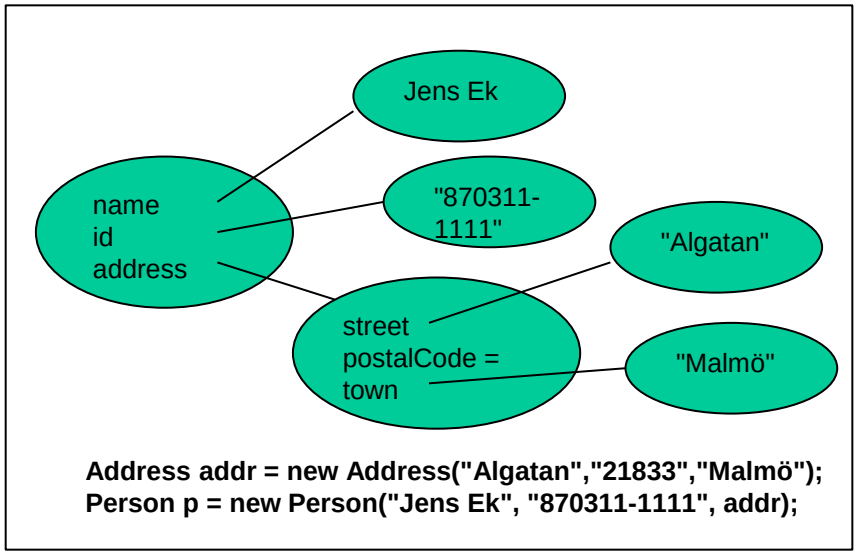
Address.java

Aggregering - Har-relation

Om en klass har ett attribut (instansvariabel) till en annan klass föreligger en starkare association, en s.k. aggregering (***har***-relation). Klassen Person *har ett namn*, *har ett personnummer* och *har en adress*. Namn och personnummer är av typen String och adressen av typen Address.

```
public class Person {
   private String name;                          // En person har ett namn
   private String id;  // Swedish personal number // En person har ett id-nummer
   private Address address;                      // En person har en adress

   public Person(String name, String id, Address address) {
      this.name = name;
      this.id = id;
      this.address = address;
   }

   public void setAddress(Address address) {
      this.address = address;
   }

   public Address getAddress() {
      return address;
   }

   public String getName() {
      return name;
   }

   // diverse metoder
}
```

PersonEx1.java



Aggregering - Har-relation

Hur ser metoder som ***getStreet***, ***getPostalCode*** och ***getTown*** ut i klassen Person?

```
public class Person {
   private String name;                              // En person har ett namn
   private String id;  // Swedish personal number    // En person har ett id-nummer
   private Address address;                          // En person har en adress
   :
   public String getStreet() {
      return address.getStreet();
   }

   public int getPostalCode() {
      return address.getPostalCode();
   }

   public String getTown() {
      return address.getTown();
   }
   :
}
```

PersonEx2.java

Aggregering - Har-relation

Hur ser konstruktorerna ut i klassen Person?

```
public class Person {
   private String name;                              // En person har ett namn
   private String id;  // Swedish personal number    // En person har ett id-nummer
   private Address address;                          // En person har en adress

   public Person(String name, String id, Address address) {
      this.name = name;
      this.id = id;
      this.address = address;
   }

   public Person(String name, String id, String street, int postalCode, String town) {
      this.name = name;
      this.id = id;
      this.address = new Address(street, postalCode, town);
   }
   :
}
```

Den sista konstruktorn skapar ett Address-objekt i vilket adressdatan lagras.

PersonEx3.java

Minneshantering vid exekvering

När ett program ska exekvera så har den virtuella maskinen (JVM) tillgång till en viss mängd internminne. I minnet ska bl.a. programmets kod, variabler och objekt lagras. Minnet delas i två delar:

Heapen

På heapen lagras alla **objekt** som du skapar i ett program. Objekt som inte används längre kan "städas bort" genom garbage collection. Med detta menas att det minne som de bortstädade objekten använder kan återanvändas av andra objekt som skapas i programmet.

Stacken

På stacken lagras **parametrar** vid metodanrop och **lokala variabler** som deklareras i en metod. Mär en metod exekverat färdigt återlämnas det minne som parametrar och lokala variabler reserverar.

Exekvering av ett litet program 1

```
public class Memory1 {
   public void main( String[] args ) {
      int a = 4;
      double b = 3.25
      int c = (int)(a * b);

      System.out.println( a );
      a = c * 2;
   }
}
```

När programmet Memory1 exekverar så skapas utrymme på stacken för variablerna ***a***, ***b*** och ***c***.

När variabeln ***a*** används i programmet så används utrymmet för ***a*** på stacken.

Instruktionen

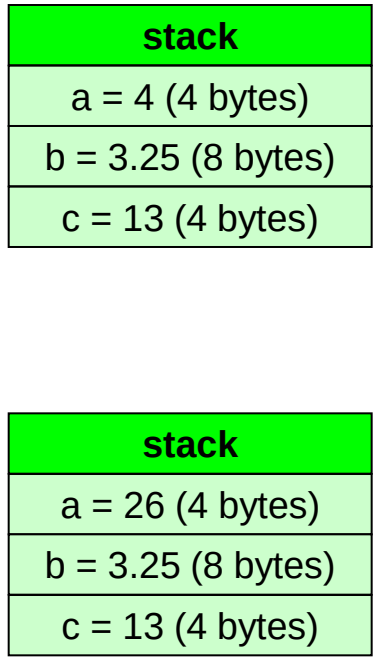
System.out.println(a);

ger därför utskriften 4.

Instruktionen

a = c * 2;

ändrar innehållet i utrymmet för ***a*** till 26.



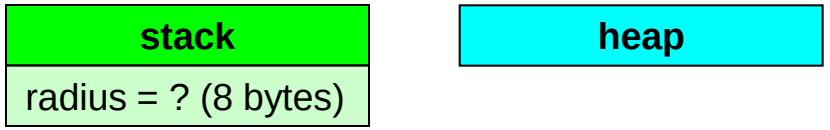
Exekvering av ett litet program 2a

```
1   public class TestCircle1 {
2      public static void main( String[] args ) {
3         double radius;
4         Circle c = new Circle();
5         c.setRadius( 10 );
6         radius = c.getRadius();
7      }
8   }
```

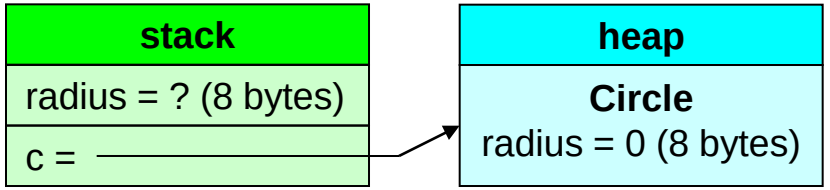
```
9   public class Circle {
10     private double radius;
11
12     public void setRadius( double radius ) {
13        this.radius = radius;
14     }
15
16     public double getRadius() {
17        return this.radius;
18     }
19  }
```

När programmet TestCircle exekverar så används både stacken och heapen.

```
3         double radius;
```



```
4         Circle c = new Circle();
```



Exekvering av ett litet program 2b

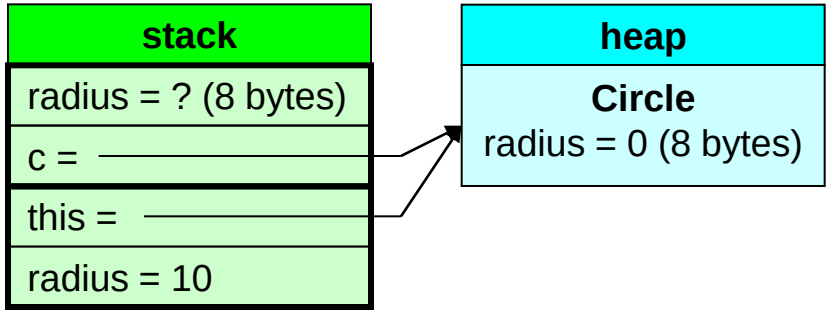
```
1   public class TestCircle1 {
2      public static void main( String[] args ) {
3         double radius;
4         Circle c = new Circle();
5         c.setRadius( 10 );
6         radius = c.getRadius();
7      }
8   }
```

```
9   public class Circle {
10     private double radius;
11
12     public void setRadius( double radius ) {
13        this.radius = radius;
14     }
15
16     public double getRadius() {
17        return this.radius;
18     }
19  }
```

forts

5 `c.setRadius( 10 );`

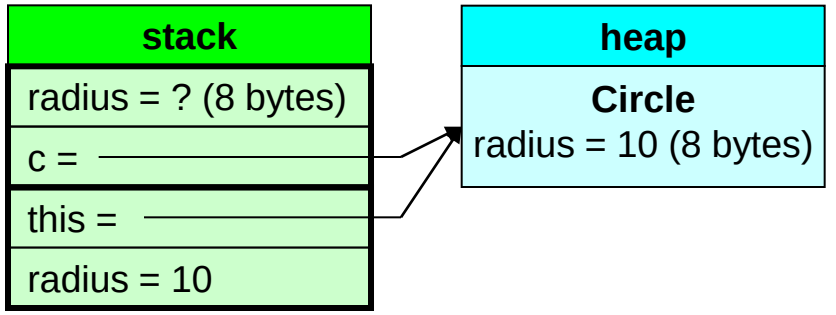
this får samma referens som ***c***.
Parametern ***radius*** i metoden ***setRadius*** får värdet 10.



---

13 `this.radius = radius;`

Värdet i parametern radius kopieras till instansvariabeln radius.



Exekvering av ett litet program 2c

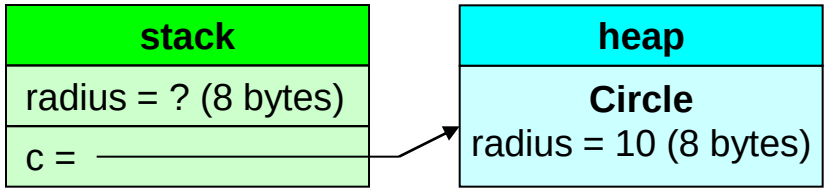
```
1   public class TestCircle1 {
2      public static void main( String[] args ) {
3         double radius;
4         Circle c = new Circle();
5         c.setRadius( 10 );
6         radius = c.getRadius();
7      }
8   }
```

```
9   public class Circle {
10     private double radius;
11
12     public void setRadius( double radius ) {
13        this.radius = radius;
14     }
15
16     public double getRadius() {
17        return this.radius;
18     }
19  }
```

forts

Metoden setRadius har exekverat färdigt

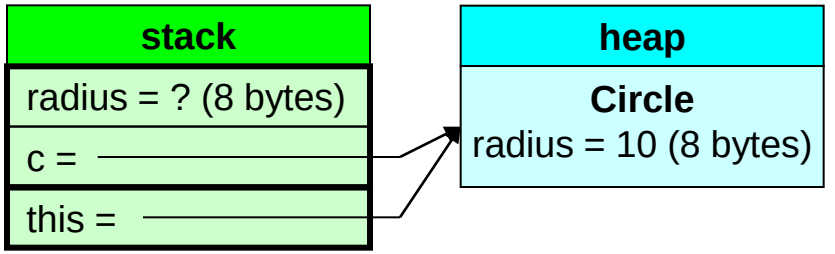
this och ***radius*** tas bort från stacken



---

6 radius = c.getRadius();

this får samma referens som ***c***.



Exekvering av ett litet program 2d

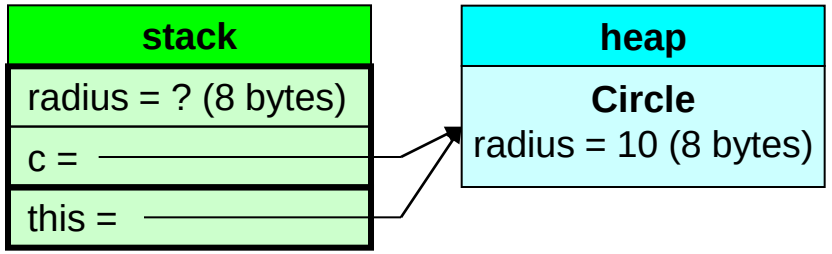
```
1   public class TestCircle1 {
2      public static void main( String[] args ) {
3         double radius;
4         Circle c = new Circle();
5         c.setRadius( 10 );
6         radius = c.getRadius();
7      }
8   }
```

```
9   public class Circle {
10     private double radius;
11
12     public void setRadius( double radius ) {
13        this.radius = radius;
14     }
15
16     public double getRadius() {
17        return this.radius;
18     }
19  }
```

forts

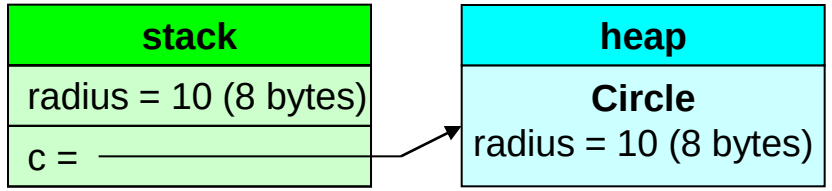
17 **return this.radius;**

värdet 10 returneras



Metoden getRadius har exekverat färdigt
radius tilldelas värdet 10

this får samma referens som ***c***.



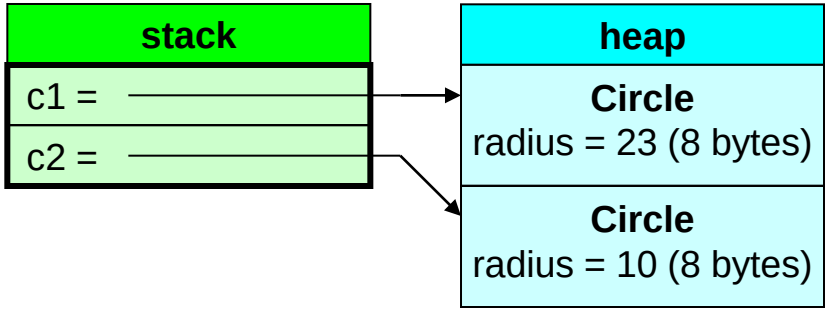
Exekvering av ett litet program 3

```
1   public class TestCircle2 {
2      public static void main( String[] args ) {
3         Circle c1 = new Circle();
4         Circle c2 = new Circle();
5         c2.setRadius( 10 );
6         c1.setRadius( 23 );
7      }
8   }
```

```
9   public class Circle {
10     private double radius;
11
12     public void setRadius( double radius ) {
13        this.radius = radius;
14     }
15
16     public double getRadius() {
17        return this.radius;
18     }
19  }
```

- Hur ser stack och heap ut när rad 3 exekverat färdigt?
- Hur ser stack och heap ut när rad 4 exekverat färdigt?
- Vad händer när rad 5 exekveras?
- Vad händer när rad 6 exekveras?

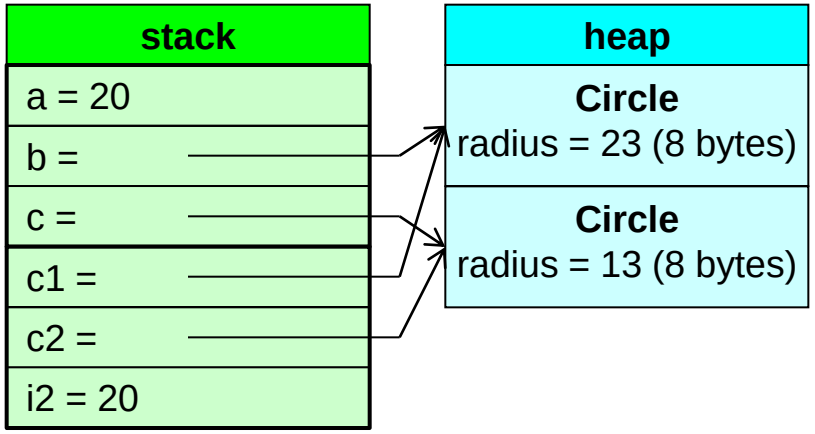
- Hur ser stack och heap ut när rad 6 exekverat färdigt?



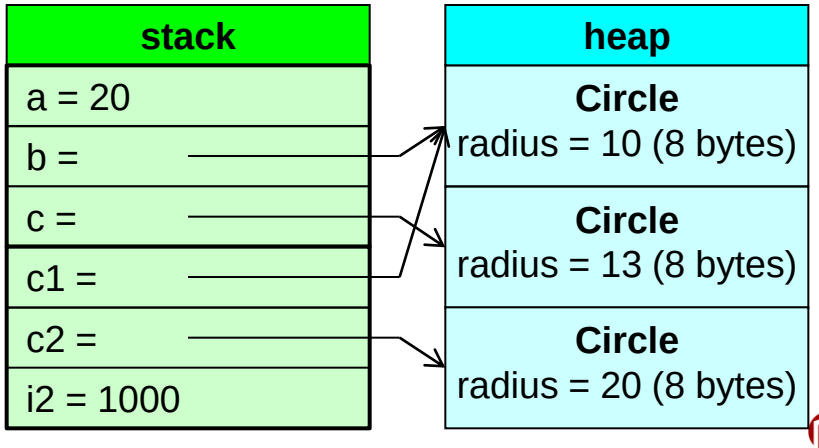
Objekt som parameter

När man överför data till en metod sker detta genom kopiering. Det är fullt möjligt att ändra i objekt som används som parameter vid metodanrop.

```
public class Parameter {
   public void change( Cirkle c1, Circle c2, int i1 ) {
      c1.setRadius( 10 );
      c2 = new Circle( 20 );
      i1 = 1000;
   }
}
-----------------------------------------------------
Parameter prog = new Parameter();
int a = 20;
Circle b = new Circle( 23 );
Circle c = new Circle( 13 );
prog.change( b, c, a );
System.out.println("a="+a);
System.out.println("b="+b.getRadius());
System.out.println("c="+c.getRadius());
```




```
c1.setRadius( 10 );
c2 = new Circle( 20 );
i1 = 1000;
```



Körresultat

```
a=20
b=10.0
c=13.0
```

Överlagra metoder

Det går bra att skriva flera metoder med samma namn i en klass. Men parameterlistorna måste skilja sig åt. Java väljer den metod där parametrarna matcher.

```
public class Overload {
   private int nbr = 10;

   public void setNbr(double realValue) {
      this.nbr = (int)realValue;
   }

   public void setNbr(int nbr) {
      this.nbr = nbr;
   }

   public void setNbr( String nbrStr ) {
      this.nbr = Integer.parseInt( nbrStr );
   }
}
--------------------------------
Overload ov = new Overload();
ov.setNbr(20);          // anrop av setNbr( int )
ov.setNbr(20.4);        // anrop av setNbr( double )
ov.setNbr("104");       // anrop av setNbr( String );
```

Metoder kan ge struktur

I klassen ***EstimateTimeVer1*** består ***action***-metoden av:

1. Slumpning av tid
2. Uppskattning av tid
3. Meddelande av resultat

```
public class EstimateTimeVer1 {

   public void action() {
      Stopwatch watch = new Stopwatch();
      Random rand = new Random();
      long estimation, rndTime = (rand.nextInt(4) + 2) * 1000;                    1.
      String res, txt = "Du ska uppskatta "+(rndTime/1000)+" sekunder.\n\n"+
            "Tryck på OK för att starta tidtagningen.";

      JOptionPane.showMessageDialog( null, txt );                                 2.
      watch.start();
      JOptionPane.showMessageDialog( null, "Tryck på OK för att stoppa tidtagningen" );
      watch.stop();

      estimation = watch.getMilliSeconds();
      res = "Uppskattning: " + estimation + "\nFel: " + (estimation - rndTime) + " ms";   3.
      JOptionPane.showMessageDialog( null, res );
   }
   :
}
```

EstimateTimeVer1.java

Metoder kan ge struktur

```
public class EstimateTimeVer2 {
   private long getRandomTime() {
      Random rand = new Random();
      return (rand.nextInt(4) + 2) * 1000;
   }

   private long estimation(long time) {
      Stopwatch watch = new Stopwatch();
      String txt = "Du ska uppskatta "+(time/1000)+" sekunder.\n\n"+
            "Tryck på OK för att starta tidtagningen.";
      JOptionPane.showMessageDialog( null, txt );
      watch.start();
      JOptionPane.showMessageDialog( null, "Tryck på OK för att stoppa tidtagningen" );
      watch.stop();

      return watch.getMilliSeconds();
   }

   private void result(long rndTime, long estimatedTime) {
      String res = "Uppskattning: " + estimatedTime + "\nFel: " +
                   (estimatedTime - rndTime) + " ms";
      JOptionPane.showMessageDialog( null, res );
   }

   public void action() {
      long rndTime, estimatedTime;
      rndTime = getRandomTime();
      estimatedTime = estimation( rndTime );
      result( rndTime, estimatedTime );
   }
   :
```

1.

2.

3.

EstimateTimeVer2.java

EstimateTime har Random och Stopwatch

```
public class EstimateTime {
   private Random rand = new Random();
   private Stopwatch watch = new Stopwatch();

   private long getRandomTime() {
      return (this.rand.nextInt(4) + 2)*1000;
   }

   private long estimate( long timeToEstimate ) {
      String txt = "Du ska uppskatta "+(timeToEstimate/1000)+" sekunder.\n\n"+
            "Tryck på OK för att starta tidtagningen.";

      JOptionPane.showMessageDialog( null, txt );
      this.watch.start();
      JOptionPane.showMessageDialog( null, "Tryck på OK för att stoppa tidtagningen" );
      this.watch.stop();
      return this.watch.getMilliSeconds();
   }

   private void showResult( long timeToEstimate, long estimation ) {
      String res = "Uppskattning: " + estimation + "\nFel: " + (estimation - timeToEstimate) + " ms";
      JOptionPane.showMessageDialog( null, res );

   }

   public void action() {
      long time = getRandomTime();
      long estimation = estimate( time );
      showResult( time, estimation );
   }
   // main-metod här
}
```

Variablerna *rand* och *watch* initieras när EstimateTime-objektet skapas.

EstimateTime.java