Mittuniversitetet
MID SWEDEN UNIVERSITY

Läsanvisningar

Börja med att läsa kapitlen i kursboken, därefter resurserna på Internet och till sist själva lektionsmaterialet.

Kursboken (5:e upplagan): Kapitel 3 och 10 (ej 10.10 och framåt)
Kursboken (6-7:e upplagan): Kapitel 3 och 10 (ej 10.11 och framåt)
Kursboken (8:e upplagan): Kapitel 3 (Mer om typer och klasser) och 10 (Mer om arv). (ej 10.9 (funktionsgränssnitt och lambda-uttryck) och 10.10 (Jämförbara objekt))

Internet: The Java Tutorials, Trail: Learning the Java Language - Object-Oriented Programming Concepts
(https://docs.oracle.com/javase/tutorial/java/concepts/index.html)

Internet: The Java Tutorial, Trail: Learning the Java Language - Nested Classes (inte det om Lambda Expressions)
(https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html)

Internet: The Java Tutorial, Trail: Learning the Java Language - Enum types
(https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html)

Internet: The Java Tutorial, Trail: Learning the Java Language - Interfaces and Inheritance
(https://docs.oracle.com/javase/tutorial/java/IandI/index.html)

Inkapsling och synlighet

Objektorienterad analys och design handlar om att försöka efterlikna detaljer i världen genom modeller som till exempel en bil, ett djur och så vidare. En modell kan också bestå av flera mindre modeller som ratten i en bil, bilstereon i bilen med mera. För att bilstereon skall fungera behövs kanske inte information om hur mycket bensin som finns i bilen, inte heller behöver bilstereon veta hur motorn omvandlar bensin till kraft och får bilen att rulla. Men bilen kan däremot behöva information från stereon, exempelvis för att på instrumentpanelen visa vilken volym som stereon har för tillfället eller visa namnet på låten som spelas, eller för att kontrollera volymen via ratten.

Bilen behöver inte veta exakt hur till exempel bilstereon internt fungerar när den ändrar sin volym, byter låtar eller andra interna funktioner. Bilen behöver endast ett sätt att säga åt bilstereon att ändra sin volym och kan strunt i hur det sen görs i praktiken. Det är här order inkapsling kommer in. Vad det betyder är hur data inom en modell skyddas från åtkomst från utsidan. Att den egna modellen har kontroll över sina data och bara den egna modellen. Modellen kan välja att tillhandahålla ett gränssnitt för andra att komma åt viss data som modellen själv väljer.

Ordet synlighet används också i dessa sammanhang. Det har att göra med hur man väljer att tillåta åtkomst till data och funktioner inom modellen. Målet är att tillhandahålla funktioner (metoder när vi pratar Java) för att avläsa data från modellen och förhindra direkt åtkomst. För att beskriva olika åtkomstmöjligheter används nyckelorden public, private och protected.

Public innebär att man ger full tillgång till metoden eller variabeln. De kan ses både internt inom modellen (den egna klassen) såväl som externt av andra.

Private innebär att man ger full tillgång inom den egna klassen men nekar helt åtkomst utifrån. Exempelvis kan man inte komma åt inställningar för volym genom ratten eller någon annan del inne i bilen, utan enbart direkt på själva bilstereon.

Protected kan ses som en blandning av private och public. Om man skapar en ny modell som ärver egenskaper och metoder från en annan befintlig modell är dessa tillgängliga från den nya modellen men inte från andra externa modeller. Mer om detta senare.

Låt oss titta på en enkel klass för att illustrera några av dessa nyckelord.

```
public class CarStereo {
  private int nrSongs;
  private String currentSongName;
  boolean isOn;

  public CarStereo(boolean isOn) {
    this.isOn = isOn;
  }

  private void nextSong() {
    // increment to the next song
  }

  public String songPlaying() {
    return songName;
  }
}
```

För att förklara lite av vad denna klass gör tittar vi närmare på några rader kod.

```
private int nrSongs;
```

Här deklareras en privat instansvariabel nrSongs av den primitiva typen int (heltal). Bara metoder inom samma klass har nu åtkomst till denna variabel. Det går alltså inte att få direkt åtkomst till variabeln efter att ha skapat ett objekt av klassen CarStereo.

```
CarStereo sony = new CarStereo(true);
System.out.println("songName: " + sony.songName); //ej möjligt!
```

Det går heller inte att anropa sony.nextSong() för att byta till nästa låt. Men för att få tag i namnet på låten som spelar går det att anropa sony.songPlaying(); eftersom den är deklarerad som publik.

Vad händer med den instansvariabel i klassen ovan vi deklarerade med obestämd synlighet?

```
boolean isOn;
```

Instansvariabler och metoder som deklareras utan något av nyckelorden public, private eller protected får då paketåtkomst (som tidigare inte nämnts). Vad detta innebär är att enbart modeller som tillhör samma grupp (paket) av modeller har tillgång till instansvariablerna och/eller metoderna. Annars fungerar den exakt som den deklarerats med privat.

I kurslitteraturen finns en bra figur som visar vilken synlighet de olika nyckelorden har (sida 99 i upplaga 6 och sida 95 i upplaga 5).

Klass- och instansvariabler

Som namnet antyder är en instansvariabel en variabel som gäller för varje instans (objekt) som skapas av klassen. Varje objekt av klassen har en egen minnesplats för varje instansvariabel. Ibland kan det vara användbart att deklarera en variabel som är gemensam för alla objekt av en klass. Det vill säga det skapas enbart en minnesplats som sen används av alla objekt. Denna typ av variabel kallas klassvariabel och deklareras med nyckelordet static.

Nedan följer ett exempel där klassvariabler används.

```
public class Elevator {
  public static final int UP = 1, DOWN = 2, STILL = 0;
  private int currentDirection = STILL;

  public int getDirection() {
    return currentDirection;
  }
}
```

I klassen ovan har bland annat konstruktorn valts bort för att kunna visa de viktiga delarna i bättre detalj. En hiss kan befinna sig i tre olika tillstånd. Antingen står den stilla, rör sig uppåt eller så rör den sig nedåt. Vi ger dessa tillstånd värden som vi godtyckligt bestämmer själva. Dessa värden är samma för alla hissar och för att spara minne deklareras variablerna UP, DOWN och STILL därför som klassvariabler (static). Värdet kommer inte heller att förändras så vi deklarerar dem även som konstanter (final).

Om vi nu vill kolla om en hiss åker uppåt kan man göra koden mer lättbegriplig genom att använda dessa variabler i stället för numeriska värden. Det vill säga det är bättre att göra så här

```
Elevator myElevator = new Elevator();
if (myElevator.getDirection() == Elevator.UP) {
  // do something
}
```

istället för att använda

```
Elevator myElevator = new Elevator();
if (myElevator.getDirection() == 1 ) {
  // do something
}
```

där man inte vet vad 1 betyder såvida man inte kollat upp hur klassen Elevator är skriven. Detta är följaktligen att bra sätt att göra sin kod tydligare. Ytterligare ett exempel visar hur man använder detta i en switch:

```
Elevator myElevator = new Elevator();

switch (myElevator.getDirection()){
  case Elevator.UP:
    // do something
    break;
  case Elevator.DOWN:
    // do something
    break;
  case Elevator.STILL:
    // do something
    break;
```

```
}
```

Klassmetoder

Ibland kan det kännas onödigt att behöva skapa ett objekt av en klass bara för att komma åt en viktig metod i klassen I stället vill vi kunna anropa metoden direkt på klassen. I till exempel klassen java.lang.Math finns ett stort antal av sådana metoder., bland annat abs(a) vilket ger absolutbeloppet av värdet a. Precis som med klassvariabler deklareras metoden med modifieraren static.

Det skulle kunna se ut enligt följande:

```
public class MyMath {
  public static int myOwnAbs(int value) {
    return Math.abs(value);
  }
}

System.out.println("abs av -10 är: " + MyMath.myOwnAbs(-10));
```

Som du ser behöver vi inte skapa ett objekt av klassen MyMath innan metoden anropas. Utan den anropas direkt via klassen. Något annat som är viktigt att nämna är att klassmetoder inte kan använda sig av instansvariabler utan måste hålla sig till klassvariabler. Tvärtom går däremot bra, det vill säga ”vanliga” metoder kan använda klassvariabler.

```
public class MyMath {
  private static double PI = 3.14;
  private double myPI = 3.14;

  public static double myOwnAbs(int value) {
    if (value < 0) {
      return -value * PI;
    }
    else {
      return value * myPI; // << ERROR >>
    }
  }
}

System.out.println("abs av -10 är: " + MyMath.myOwnAbs(-10));
```

Exemplet ovan leder till ett kompileringsfel. Det går inte att använda myPI, vilket är en instansvariabel, från en klassmetod. Anledningen är att anropet till en klassmetod inte är kopplat till ett visst objekt av klassen och därför går det inte heller att använda instansvariabler.

Uppräkningstyper

I tidigare exempel skapade vi tre konstanter för hissens färdriktning. Det finns ett annat sätt att göra detta på, nämligen med en uppräkningstyp (enum).

```
enum Direction {DOWN, STILL, UP};
```

Dessa kan senare användas genom att skriva Direction.UP, Direction.DOWN och Direction.STILL. Det vi gör är att skapa en helt ny typ som kan användas i koden på samma sätt som andra typer till

exempel primitiva typer som int och double samt referens typer (klasser). Vi kan deklarera variabler av typen, använda den som argument i metoder samt som returvärde i metoder.

```
public class Elevator {
  public enum Direction {UP, DOWN, STILL};
  private Direction currentDirection = Direction.STILL;

  public Direction getDirection() {
    return currentDirection;
  }
}
```

Det som skiljer att använda en enum från att använda konstanter som vi såg från tidigare exempel av klassen Elevator är att en enum har de värden som angetts i deklarationen och inga andra. En konstant skulle vi kunna jämföra mot vilket annat värde som helst så länge som typen är densamma. I det gamla exemplet är följande möjligt.

```
if (myElevator.getDirection() == Calendar.HOUR_OF_DAY) {}
```

Denna rad ser minst sagt en aning ologisk ut, men är ändå helt korrekt då Calendar.HOUR_OF_DAY är av typen int (värdet 11 enligt API). Med en enum går det inte att göra på detta sätt och man förhindrar ologiska rader i sin kod.

```
if (myElevator.getDirection() == Direction.STILL) {}
```

Ett annat exempel som visar hur man skulle kunna använda enum är följande:

```
public class MyMenu {
  private enum MenuOption {NEW_CUSTOMER, MODIFY_CUSTOMER, DELETE_CUSTOMER,
    LOGIN, LOGOFF, EXIT};
  private MenuOption[] mainMenu = {MenuOption.LOGIN, MenuOption.EXIT};
  private MenuOption[] customerMenu = {MenuOption.MODIFY_CUSTOMER,
    MenuOption.DELETE_CUSTOMER, MenuOption.LOGOFF};
  private java.util.Scanner input = new java.util.Scanner(System.in);

  public void printMainMenu() {
    System.out.println("1. Login");
    System.out.println("2. Exit");
    int menuChoice = input.nextInt();
    handleMenuChoice(mainMenu[menuChoice - 1]);
  }

  public void printCustomerMenu() {
    System.out.println("1. Modify customer");
    System.out.println("2. Delete Customer");
    System.out.println("3. Log off");
    int menuChoice = input.nextInt();
    handleMenuChoice(customerMenu[menuChoice - 1]);
  }

  public void handleMenuChoice(MenuOption choice) {
    switch ( choice ) {
      case NEW_CUSTOMER:     // ask for customer data, etc...
        break;
      case MODIFY_CUSTOMER:  // modify current customer...
        break;
      case DELETE_CUSTOMER:  // delete current customer...
        break;
      case LOGIN:            // go to level 2 menu...
```

```
        break;
      case LOGOFF:            // go to level 1 menu...
        break;
      case EXIT:              // exit program
        break;
    }
  }
}
```

Metoden handleMenuOption tar en parameter av typen MenuOption och utifrån värdet gör vi sen olika saker. Det finns många sätt att skriva en menyklass på, denna visar enbart hur man skulle kunna ta till vara på enum för att förtydliga menyvalen.

En enum är faktiskt en egen klass i Java, en klass som ärver Object. Det är därför möjligt att i en enum överskugga metoder som toString samt lägga till en konstruktor. Men så avancerat behöver du inte lära dig i denna kurs. Bara att ovan nämnda möjligheter finns och råds att användas vid behov.

Arv och polymorfi

Arv innebär kort att man använder en befintlig klass som grund för att skapa en ny klass. Den nya klassen ärver egenskaper och metoder från den första klassen. Första klassen kallas för superklass och den nya klassen för subklass.

För att ange vilken klass som ska ärvas används nyckelordet extends.

```
public class Vehicle {
  protected Color color;
  private int nrWheels;

  public Vehicle(Color color, int nrWheels) {
    this.color = color;
    this.nrWheels = nrWheels;
  }

  public int getNrWheels() {
    return nrWheels;
  }
}

public class Car extends Vehicle {
  // Här behöver inte variablerna color och nrWheels deklareras
  // de ärvs från superklassen (Vehicle).

  public Car(Color color) {
    // Här går det att anropa superklassens konstruktor för att skapa
    // initiera värden.
    super(color, 4);  // super utan något annat anropar konstruktor
    super.getNrWheels();  // super med punkt anropar superklassens metod
  }
}

Car myVolvo = new Car(Color.blue);
```

Här skapas ett objekt myVolvo av typen Car, som i sin tur är en utbyggnad utav Vehicle. Om vi nu skulle skapa en tredje klass som heter Bike kan den också vara en utbyggnad utav Vehicle. En cykel har några saker gemensamt med alla typer av fordon, som färg och antal hjul, men har andra

egenskaper som är unika just för en cykel. Sådant som är gemensamt för många olika klasser kan vi samla i en superklass och sen låta andra klasser ärva och utöka med sitt eget.

Polymorfism innebär att något har många formen. I Java betyder det att en metod, med samma namn och samma antal och typ av parametrar, finns i många olika klasser. Klasserna måste ärva samma superklass eller implementera samma gränssnitt.

```
public class Account {
  protected double balance;

  public Account() {
  }

  public void withdraw(double amount) {
  if (amount <= balance) {
    balance -= amount;
  }
}

public class SavingsAccount extends Account {

  public SavingsAccount() {
    super();
  }
}

public class CreditAccount extends Account {
  private double credit;

  public CreditAccount(double credit) {
    super();
    this.credit = credit;
  }

  @Override
  public void withdraw(double amount) {
    if (balance <= amount) {  // lägg till koll för kreditdel!
      balance -= amount; // måste ta kreditdelen i beaktning
    }
  }
}

Account account1 = new SavingsAccount();
Account account2 = new CreditAccount(1000.0);
account2.withdraw(500.0);
```

I detta exempel är metoden withdraw polymorfisk. Den finns i flera varianter i olika klasser. Sättet att göra ett uttag på skiljer sig beroende på om det är ett vanligt konto eller ett kreditkonto.

En av fördelarna med arv är att man kan skapa en referensvariabel (objekt) till superklassen men tilldela denna ett objekt av någon av dess subklasser. När vi i koden ovan anropar metoden whitdraw på objektet account2, vilken variant av metoden är det då egentligen som körs? Är det whitdraw i Account eller är det withdraw i CreditAccount?

Det är metoden i CreditAccount som kommer att anropas. Först kontrolleras om det finns någon metod som matchar anropet i den klass som det faktiskt skapas ett objekt av (det vill säga CreditAccount). Finns det en metod som matchar anropas den, annars kontrolleras det om det finns någon matchande metod i närmaste superklassen och så vidare.

Du kanske undrar över @Override som används ovan metoddeklarationen för withdraw. Kort kan man säga att det är en flagga till kompilatorn att metoden som deklareras överskuggar en metod i superklassen. När vi överskuggar en metod måste vi vara väldigt noga att skriva exakt rätt namn på metoden och exakt rätt namn och typ på parametrarna, annars lägger vi bara till en ny metod istället för att överskugga den befintliga. Om det inte finns någon matchande metod att överskugga i superklassen kommer kompilatorn, om @Override används, ge en varning.

Abstrakta klasser och metoder

Om vi tittar på exemplet angående bankkonton ser vi att vi kommer att ha flera upplagor av metoden withdraw. Om vi skulle glömma att överskugga withdraw i alla ärvda klasser av konton gör det egentligen inte så mycket då anropet skulle falla tillbaka på superklassens metod. Detta är kanske inte önskvärt alla gånger och kan skapa stora problem. Det är bättre om vi i superklassen skulle kunna tvinga alla subklasser att tillhandahålla sin egen variant av withdraw. För att åstadkomma detta använder vi nyckelordet abstract och deklarerar metoden withdraw som abstrakt.

```
public abstract class Account {
  protected double balance;

  public Account() {
  }

  public abstract void withdraw(double amount);

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

En abstrakt metod inte har någon kod utan avslutas med ett semikolon. Eftersom metoden saknar kod tvingas subklasser att själva implementera metoden (eller deklarera den abstrakt i sin tur och låta ytterligare subklasser implementera den). Vi ser även att om en klass innehåller en abstrakt metod måste även klassen deklareras som abstrakt.

Gränssnitt

Något som inte är möjligt i Java är multipelt arv som till exempel är möjligt i C++. I Java kan man inte låt en klass ärva från två eller fler klasser på samma gång. För att delvis komma runt detta problem kan man använda interface (gränssnitt). Det är mallar för vilka metoder som måste finnas med i en klass. Ett interface är en klass som endast kan innehålla konstanter och abstrakta metoder. Ett gränssnitt kan man inte skapa objekt av, men däremot går det bra att deklarera variabler av en gränssnittstyp. Det man gör med gränssnitt är att låta andra klasser implementera gränssnittet. Då tvingas den klassen att tillhandahålla en definition av alla abstrakta metoder i gränssnittet.

Vi kan t.ex. skapa gränssnittet Eatable för att indikera att något är ätbart. I denna definieras metoden eat för att kunna äta det som nu är ätbart.

```
public interface Eatable {
  void eat(); // Blir automatiskt public och abstract
}
```

Vi skriver nu en klass Apple som implementerar gränssnittet Eatable.

```
public class Apple implements Eatable {
  private String color;

  public Apple(String color) {
    this.color = color;
  }

  public String getColor() {
    return color;
  }

  @Override
  public void eat() {
    System.out.println("Eating a " + color + " apple.");
  }
}
```

Om vi i klassen Apple skulle glömma bort att implementera alla metoder i gränssnittet skulle vi få följande kompileringsfel: Apple is not abstract and does not override abstract method eat() in Eatable. Vi måste med andra ord antingen implementera de saknade metoderna eller deklarera hela klassen som abstrakt (eftersom klassen saknar implementation för en eller flera metoder).

Vi skapar även en klass Potato som också implementerar gränssnittet Eatable:

```
public class Potato implements Eatable {
  private String variety;

  public Potato(String variety) {
    this.variety = variety;
  }

  public String getVariety() {
    return variety;
  }

  @Override
  public void eat() {
    System.out.println("Eating a " + variety + " potato.");
  }
}
```

När metoden eat är implementerad kan vi nu skapa objekt av klassen Apple och Potato på följande sätt och anropa metoden eat:

```
Apple redApple = new Apple("red");
Eatable greenApple = new Apple("green");
Potato kingEdward = new Potato("King Edward");
Eatable bintje = new Potato("Bintje");

redApple.eat();
greenApple.eat();
kingEdward.eat();
bintje.eat();
```

Observera att när vi deklarerar en variabel av en gränssnittstyp kan vi endast anropa de metoder det gränssnittet definierar. I exemplet med greenApple är det därför endast möjligt att anropa metoden

eat och inte metoden getColor. Försöker vi göra det senare får vi ett kompileringsfel som säger att metoden getColor inte kan hittas i variabeln greenApple som är av typen Eatable.

Vi skrev ovan att ett gränssnitt endast kan innehålla konstanter och abstrakta metoder. Från och med version 8 av Java har det gjorts tillägg till detta. Ett gränssnitt kan nu mera innehålla även något som kallas för default-metoder och statiska metoder.

Default-metoder i ett gränssnitt

Default-metoder gör det möjligt för oss att lägga till ny funktionalitet till ett gränssnitt utan att göra alla de klasser som implementerar detta gränssnitt okompileringsbara. Låt säga att vi i ett senare skeda kommer på att vi vill lägga till en metod i gränssnittet Eatable för att kunna tillaga (cook) det som är ätbart. I ett första försök gör vi följande:

```
public interface Eatable {
  void eat();  // Blir automatiskt public och abstract
  void cook(); // Blir automatiskt public och abstract
}
```

Gränsnittet ovan går att kompilera utan problem. Däremot får vi problem om vi försöker kompilera om t.ex. klassen Potato: Potato is not abstract and does not override abstract method cook() in Eatable. Det vi måste göra är att gå igenom alla klasser som implementerar det förändrade gränssnittet och implementera de nya metoderna. I vårt exempel är det inte något stort problem eftersom det bara är två klasser. Större problem blir det om du t.ex. skapat ett eget API som hundratals andra utvecklare använder. Dessa utvecklare får vi anta inte skulle bli så glada över dina förändringar i gränssnittet. Då är det väldigt många som måste in i totalt sett väldigt många klasser och ändra.

Lösningen på detta är att använda en default-metod i stället. Default-metoder definieras i gränssnittet med nyckelordet default samt ges en implementation direkt i gränssnittet, en så kallad default-implementation:

```
public interface Eatable {
  void eat();  // Blir automatiskt public och abstract
  default void cook() { // Blir automatiskt public
    System.out.println("Cooking the eatable.");
  }
}
```

Nu kan båda våra klasser Apple och Potato kompileras utan att ha definierat metoden cook. Vi kan nu använda metoden på t.ex. följande sätt:

```
Eatable greenApple = new Apple("green");
Eatable kingEdward = new Potato("King Edward");

greenApple.cook();
greenApple.eat();
kingEdward.cook();
kingEdward.eat();
```

Vilket ger utskriften:

```
Cooking the eatable.
Eating a green apple.
```

```
Cooking the eatable.
Eating a King Edward potato.
```

Om vi vill kan vi i våra klasser som implementerar ett gränssnitt som har en eller flera default-metoder, ge en egen definition av default-metoden. Då är det denna definition som kommer att användas. Exempelvis kan vi göra följande tillägg i klassen Apple:

```
@Override
public void cook() {
  System.out.println("An apple doesn't need to be cooked.");
}
```

Använder vi samma testkod blir utskriften nu i stället:

```
An apple doesn't need to be cooked.
Eating a green apple.
Cooking the eatable.
Eating a King Edward potato.
```

Statiska metoder i ett gränssnitt

Från och med Java 8 kan ett gränssnitt ha statiska metoder. En statisk metod är annars associerad med en klass inte med objekten av klassen. Statiska metoder fungerar ofta som hjälpmetoder. Så om vi deklarerar en statisk metod i ett gränssnitt, är det lätt för oss att organisera våra hjälpmetoder. Vi slipper skapa nya klasser för dessa metoder utan kan samla dem direkt i det gränssnitt som har att göra med metoden. Statiska metoder liknar default-metoder, förutom att vi inte kan överskugga dem i klasser som implementerar gränssnittet.

Låt säga att vi vill ha en metod att mixa två ätbara objekt med varandra. Normalt hade vi då skapat en klass på följande sätt:

```
public class EatableHelper {
  public static String mix(Eatable eatable1, Eatable eatable2) {
    return eatable1.getClass().getName() + eatable2.getClass().getName();
  }
}
```

Denna kan vi sen använda på följande sätt:

```
Eatable greenApple = new Apple("green");
Eatable kingEdward = new Potato("King Edward");

String mixedEatable = EatableHelper.mix(greenApple, kingEdward);
System.out.println("Mixing an apple and a potato gives us a: " + mixedEatable);
```

Vilket ger utskriften:

```
Mixing an apple and a potato gives us a: ApplePotato
```

I stället för att skapa en ny hjälpklass kan vi nu definiera metoden direkt i gränssnittet på följande sätt:

```
public interface Eatable {
  void eat(); // Blir automatiskt public och abstract
```

```
  default void cook() { // Blir automatiskt public
    System.out.println("Cooking the eatable.");
  }

  static String mix(Eatable eatable1, Eatable eatable2) {
    return eatable1.getClass().getName() + eatable2.getClass().getName();
  }
}
```

Som kan användas på samma sätt som tidigare men med skillnaden att vi anropar metoden mix direkt på gränssnittet i stället:

```
Eatable greenApple = new Apple("green");
Eatable kingEdward = new Potato("King Edward");

String mixedEatable = Eatable.mix(greenApple, kingEdward);
System.out.println("Mixing an apple and a potato gives us a: " + mixedEatable);
```

Utskriften av denna testkod blir densamma som tidigare.