Mittuniversitetet
MID SWEDEN UNIVERSITY

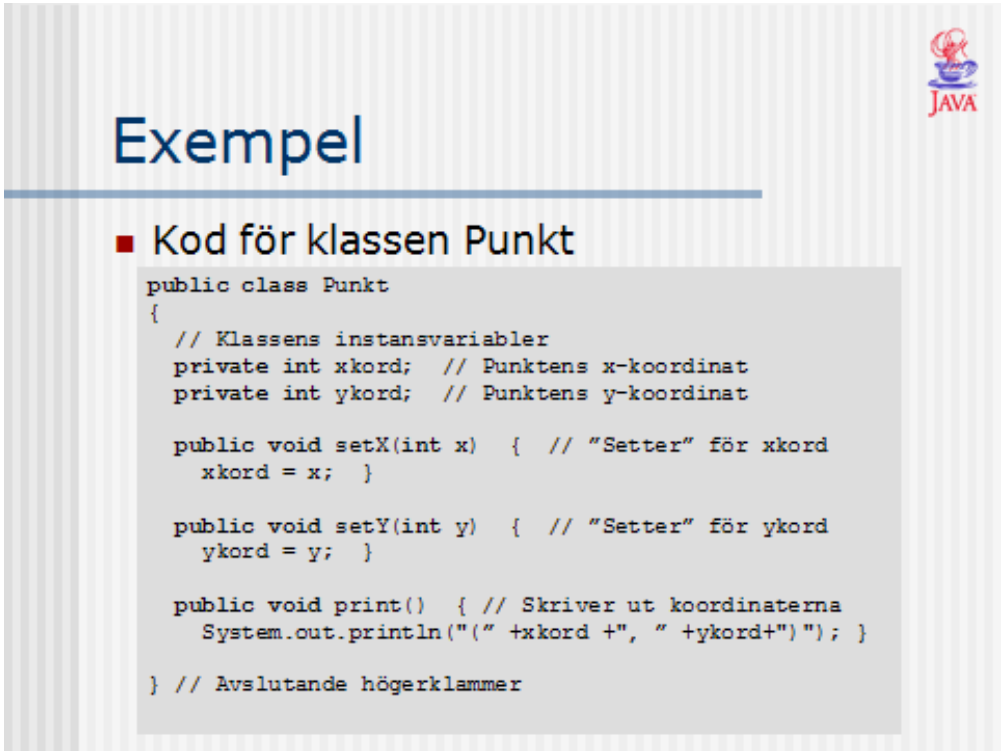
Lektion 3 – Klasser och interface

Java för C++-programmerare

Syfte:	Presentation av klasser och interface i Java: semantik och syntax
Att läsa:	Java direkt, kap 3, 4 samt andra relevanta avsnitt
	Relevant API-Dokumentation

Klassdeklarationen skiljer sig från C++ bl.a. genom att

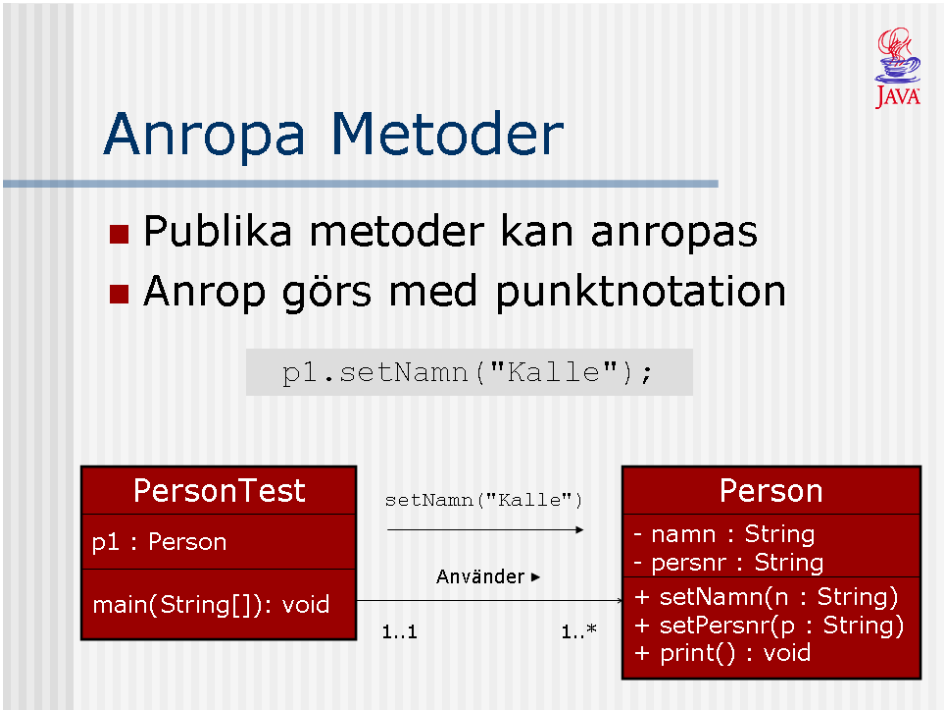
- avslutande semikolon saknas
- kolon efter åtkomstspecifikation saknas



- Observera att åtkomstmodifierarna vanligen skrivs ut för varje klassmedlem. Det är inte nödvändigt. Om man utelämnar en åtkomstmodifierare så gäller paketåkomst.

- Det normala är att använda `public` som åtkomstmodifierare för klasser. Detta innebär att andra klasser kan skapa objekt av klassen. Observera att vi aldrig kan använda åtkomstmodifieraren `private` i en klassdeklaration.

- Andra klassmodifierare och arv behöver inte anges. Om inget arv anges med `extends` kommer klassen implicit (d.v.s. automatiskt) att ärva från den abstrakta klassen `Object` som är all Java-klassers superklass (basklass). För alla klasshierarkier i Java kommer alltså klassen `Object` att vara den verkliga basklassen längst upp i hierarkin.



Att anropa en metod på ett objekt gör vi det genom punktnotation där vi anger namnet på det objekt, vars metod vi vill anropa, följt av namnet på metoden. Om metoden vi anropar finns i samma klass räcker metodens namn.

PersonTest

```
public class PersonTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person();
    p1.setNamn("Kalle");
    p1.setNamn(Kalle); // ej tillåtet
    p1.setNamn(10);    // ej tillåtet
    p1.setNamn();      // ej tillåtet
  }
}
```

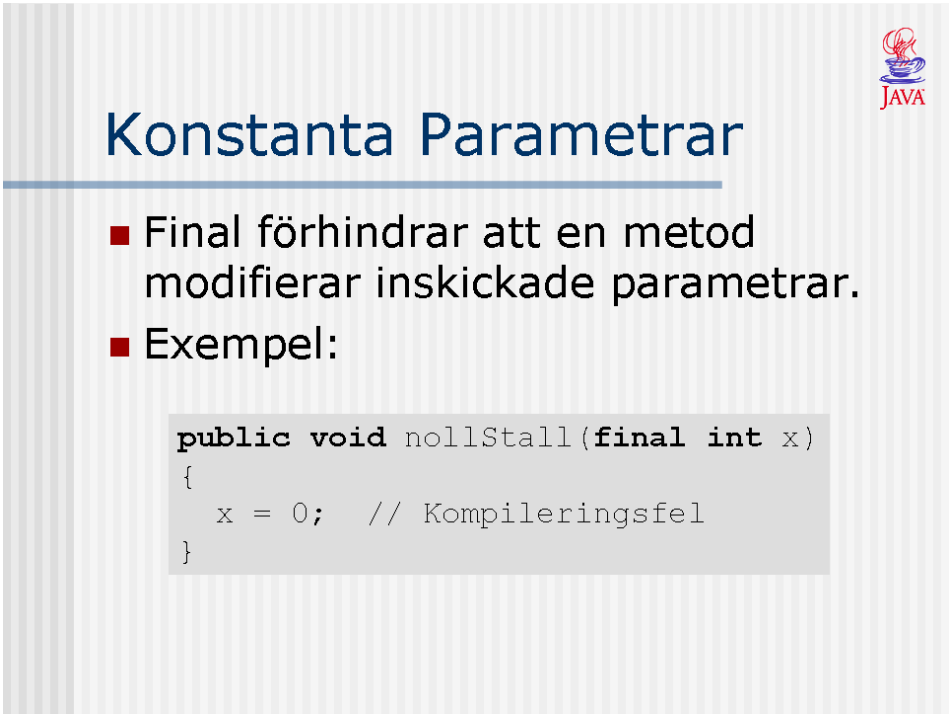
Alla ”riktiga” objekt, d.v.s. instanser av klasser måste allokeras med `new`. Någon motsvarighet till ”lokala objekt på stacken” som i C++ finns inte.

Metoder - Parametrar

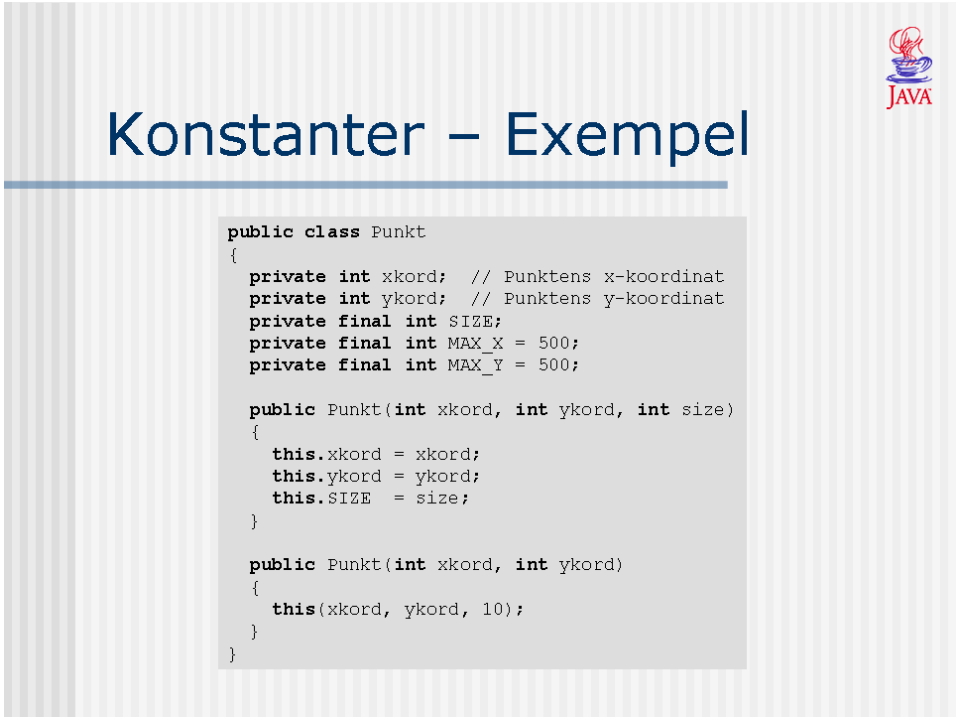
- Parametrar överförs via värdeanrop d.v.s endast en **kopia** skickas till metoden
- För de primitiva typerna innebär det att kopian är skiljd från originalet
- För objektreferenser refererar dock kopian till samma objekt som originalet

I Java överförs alla parametrar till metoder via s.k värdeanrop (”call-by-value”). Detta innebär att när en variabel skickas som parameter till en metod så överförs endast en kopia av parameterns värde till metoden, och metoden kan inte ändra den ursprungliga variabelns värde.
För de primitiva datatyperna (`boolean`, `int`, `char`, `double` etc) innebär detta att om vi gör en förändring på den *formella* parametern så sker inte denna förändring på den *aktuella* parametern (argumentet).
Eftersom alla variabler som är deklarerade att innehålla ett objekt i Java är objektreferenser (läs pekare) så är det en kopia på referensen som passas till en metod. Via denna referens kan det refererade objektet påverkas, det är alltså referensen som överförs via värdeanropet och inte själva objektet. Däremot kan inte själva referensen ändras.

När det gäller objekt som skickas som argument till en metod är ”kopian” och ”originalet” alltså två referenser som refererar till samma objekt i minnet. Det som överförs vid värdeanrop för ett objekt är alltså en kopia på referensen. I metoden `bytNamn` lagras denna objektreferens i p. När vi sedan använder p för att byta namn så byter vi namn på objektet som p refererar till. I och med att originalet p1 refererar till samma objekt som kopian p är namnet bytt även på p1.



Nyckelordet `final` är Javas motsvarighet till `const` i C++. Det kan användas på formella parametrar och instansvariabler. En medlemsfunktion i C++ kan vara `const` men en metod i Java kan inte vara `final` i den bemärkelsen att den inte kan förändra värde på någon instansvariabel. I Java har `final` tillämpat på en metod en helt annan innebörd som vi återkommer till.



Överlagring - Exempel

signatur

```
public void skrivUt(String s)
{
  // kod för att skriva ut strängen s
}
```

signatur

```
public void skrivUt(String s, Color c)
{
  // kod för att skriva ut strängen s i färgen c
}
```

signatur

```
public void skrivUt(String s, Color c, Font f)
{
  // kod för att skriva ut strängen s
  // i färgen c och med fonten f
}
```

Överlagring av metoder fungerar på samma sätt som i C++: flera metoder i samma klass kan ha samma namn så länge de skiljer sig åt i signaturen, d.v.s. antalet parametrar och/eller parmetrarnas typer.

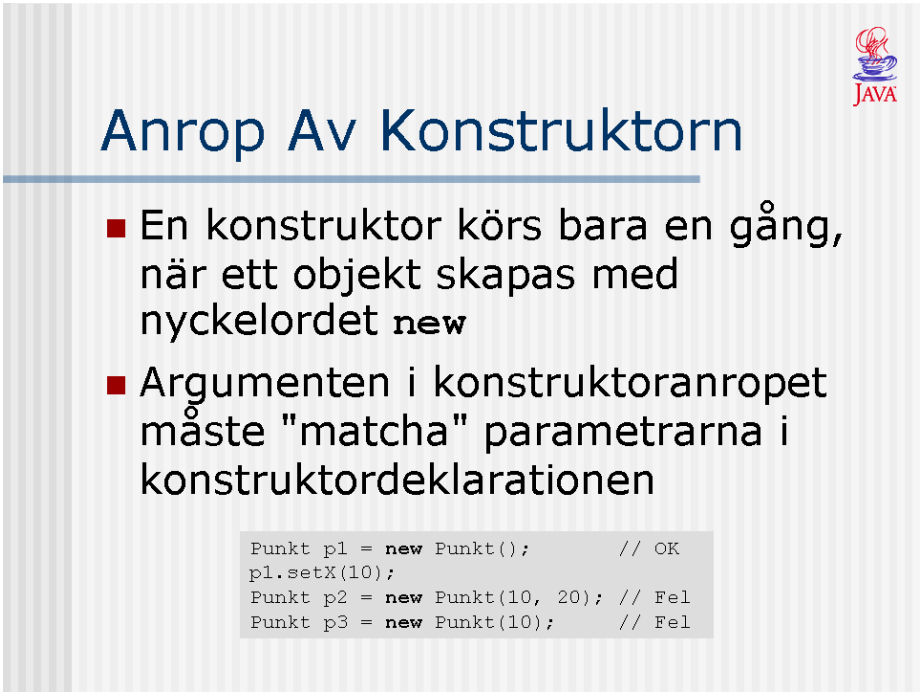
Defaultkonstruktorn

- Om ingen konstruktor anges, kommer en default-konstruktor att tillhandahållas.
  - Tar inga parametrar
  - Innehåller ingen kod

```
public Punkt()
{
}
```

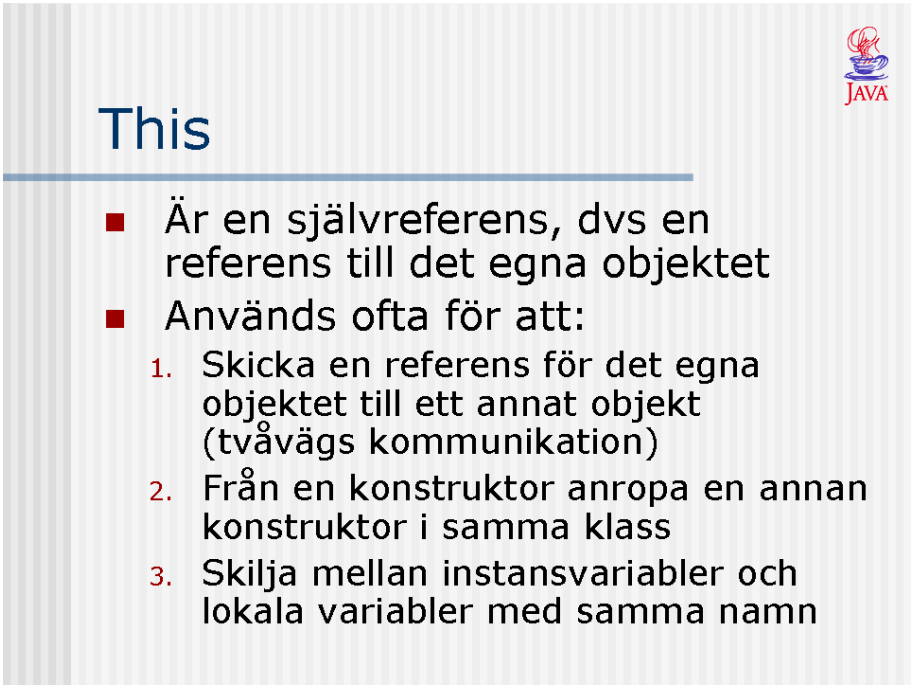
- Förklarar varför vi kunnat skriva:

```
Punkt p1 = new Punkt();
```



En Java-konstruktor fungerar som en C++-konstruktor med följande undantag:

- Anropsparanteserna ( ) *måste* skrivas ut även om konstruktorn är parameterlös.
- Java saknar det som kallas initieringslista i C++. Alla värden på instansvariablerna sätts med vanliga tilldelningssatser i konstruktorn eller direkt vid deklarationen.
- Initiering av en basklass i Java görs med nyckelordet `super(...)` som första sats i konstruktorn. Mer om detta längre fram.



Alla objekt av klasstyp i Java har en `this`-referens (pekare) som är en direkt motsvarighet till `this`-pekaren i C++.

Klassvariabler

- Finns endast i ett exemplar för klassen, som alla objekt delar
- Lagras och används på klassnivå istället för på objektnivå

```
public class Punkt
{
  ...
  private static int antalPunkter = 0;

  public Punkt(int xkord, int ykord, int size)
  {
    antalPunkter = antalPunkter + 1;
    ...
  }
}
```

Klassmetoder

- Statiska metoder tillhör klassen och anropas med namnet på klassen

```
public class Punkt
{
  ...
  private int xkord;
  private static int antalPunkter = 0;

  public static getAntalPunkter()
  {
    // kan inte komma åt icke statiska instans-
    // variabler i en statisk metod
    // xkord ej tillgänglig här!!!
    return antalPunkter;
  }
  ...
}

// Anrop av statisk metod från en testklass
int antal = Punkt.getAntalPunkter();
```

OBS!

Det behövs inget Punkt-objekt för att anropa den statiska metoden.

Statiska medlemmar fungerar med samma semantik som de gör i C++.

- En statisk datamedlem implementerar något som lagras och används på klassnivå istället för objektnivå och kallas därför *klassvariabel*. En klassvariabel delas av alla instanser av klassen. Accessas med `Klassnamn.variabelnamn;`

- En statisk metod kallas för en *klassmetod* och kan endast referera klassvariabler. Det går alltså inte att i en klassmetod referera till instansvariabler i klassen. Det innebär att om vi från main()-metoden (som är statisk) vill anropa en annan metod i samma klass måste även den andra metoden vara deklarerad som static.
- Eftersom en klassmetod inte verkar på ett visst objekt så saknar den också this-pekare.
- En klassmetod anropas med `Klassnamn.metodNamn();`

Både klassvariabler och klassmetoder existerar i och med att klassen är deklarerad, inget objekt behöver vara instansierat.

Till skillnad från C++ har Java en speciell konstruktion för att initiera statiska data. Initieringar och satser inom blocket

```
static {

}
```

sker en gång, vid starten av programmet.

toString()

```
public String toString()
```

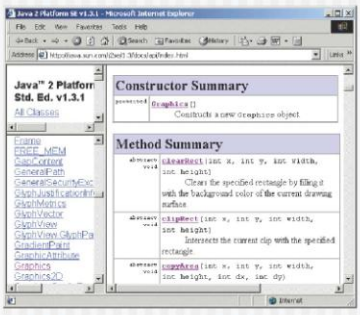
- Alla klasser har en `toString` metod
- Används för att skapa en sträng-representation av aktuellt objekt
- Bör alltid omdefinieras

```
// i klassen Punkt
public String toString()
{
  return "(" + xkord + ", " + ykord + ")");
}

// anrop av metoden ger t.ex. följande utskrift
// (12, 234) eller (198, 32)
```

Alla klasser har en `toString`-metod som är ärvd från `Object`. Denna metod bör omdefinieras av alla klasser.

Dokumentationskommentarer

- Används för att automatgenerera dokumentation
- Beskriver klasser, metoder och fält
- `javadoc` genererar HTML-kod
-
```
/**
 * Dokumentation
 */
```



Klasser och fält

- Första meningen ska vara en summering

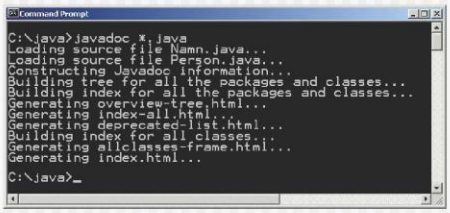
```
/**
 * Representerar en tvådimensionell punkt. Klassen är tänkt
 * att användas i olika geometriska klasser såsom cirklar,
 * rektanglar etc. Innehåller information om punktens x- och
 * y-koordinat.
 *
 * @author Robert Eriksson
 * @version 1.0
 */
public class Punkt
{
  /** Punktens x-koordinat. */
  private int xkord;

 /** Punktens y-koordinat. */
  private int ykord;
}
```

Konstruktor & metod

```
/**
 * Sätter för- och efternamn på personen. Utifrån det för- och
 * efternamn. som skickas som argument till metoden, skapas
 * ett nytt <code>Namn</code> objekt. Referensen till detta
 * objekt tilldelar vi vårat fält <code>namn</code>.
 * @param  f personens förnamn
 * @param  e personens efternamn
 */
public void setNamn(String f, String e)
{
  namn = new Namn(f, e);
}

/**
 * Returnerar förnamnet på personen.
 * @return personens förnamn
 */
public String getFornamn()
{
  return fornamn;
}
```

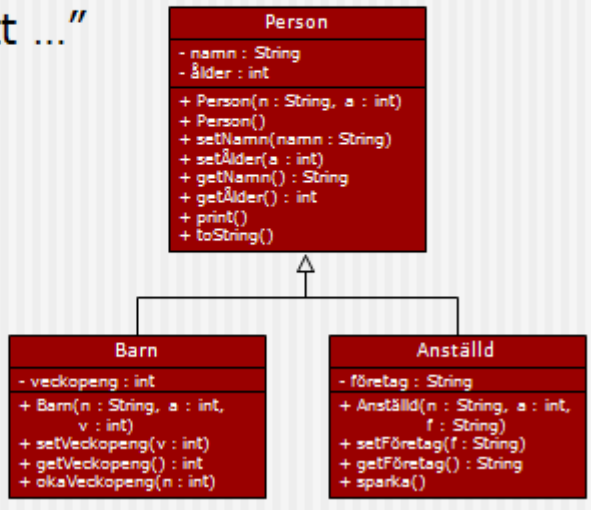
Javadoc

- Javadoc skapar HTML-filer utifrån deklarationer och dokumentations
  1. Samla alla filer i en katalog
  2. Öppna en DOS-prompt och skriv `javadoc *.java` i katalogen
  3. Öppna filen index.html i en webbläsare



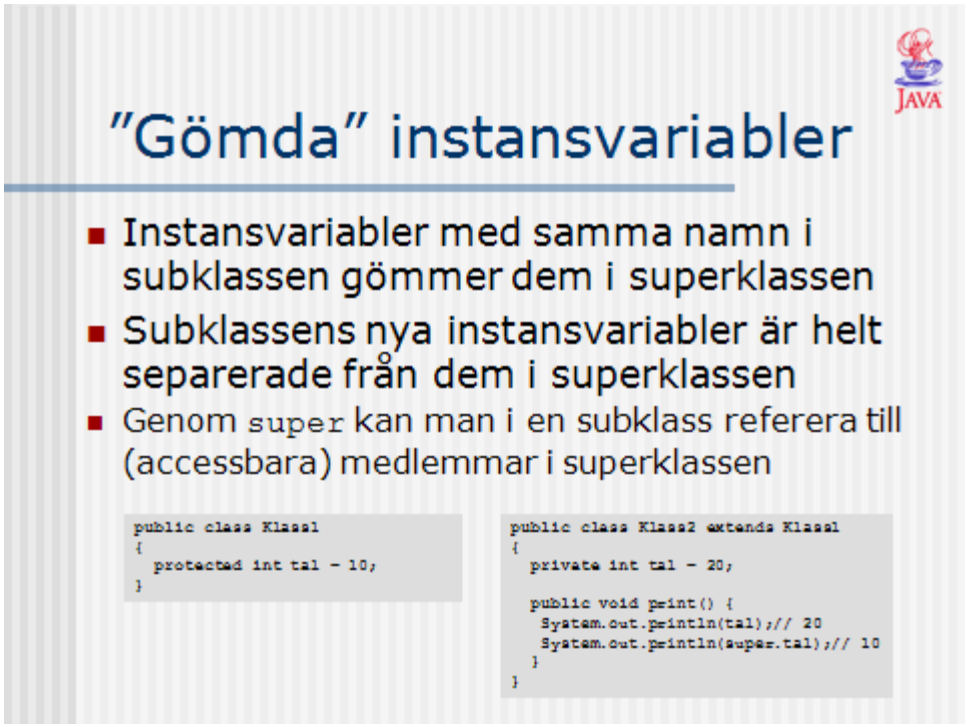
Arv – ett exempel

- ”Är en/ett …”

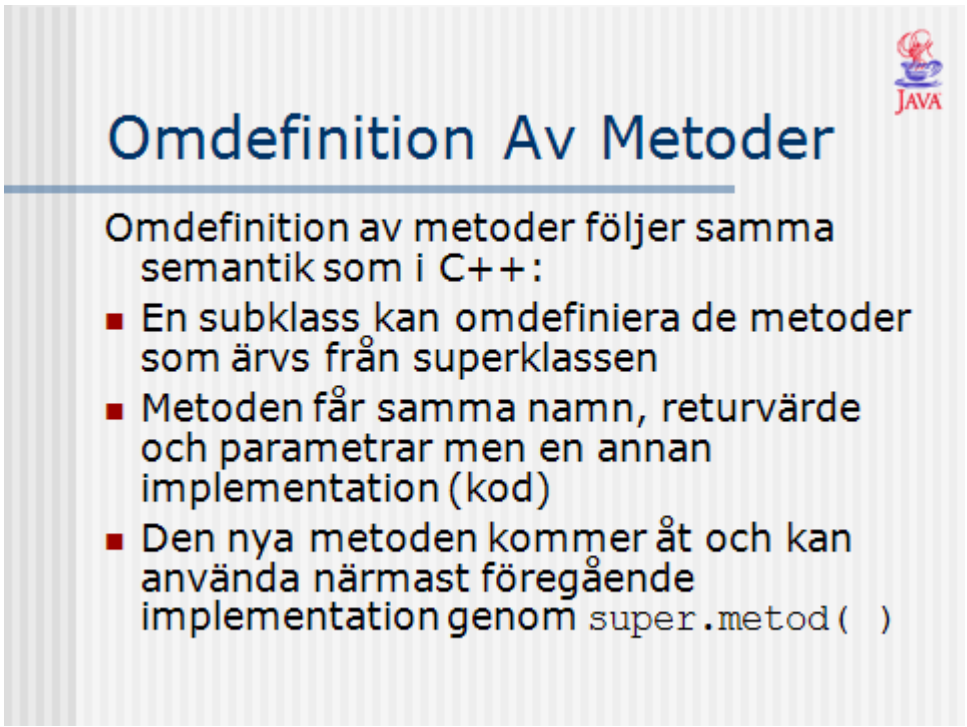


Klassen Person kan vara basklass för Barn och Anställd; Ett Barn eller en Anställd är ju ”en sorts Person”. Istället för att upprepa koden för setÅlder, getNamn etc i både Barn och Anställd ärver vi istället från Person. Utöver de medlemmar i Person som dessa två subklasser ärver kommer vi att utöka beteendet i Person.
Barn får en egen instansvariabel för att hålla koll på veckopengen. Det kommer även finnas set- och get-metoder för denna instansvariabel, samt en metod för att öka veckopengen när barnet blir äldre.
För en anställd finns en instansvariabel för företaget som denne är anställd hos. Observera att vi i subklasserna inte direkt kommer åt instansvariablerna i Person utan måste gå via get och set-metoderna.

För `private`, `public` och `protected` gäller samma åtkomstregler som för C++.
Vill man förhindra att en klass kan ärvas kan man deklarera klassen som final, ex
`public final class Person {...}`



I Java används `super` för att referera till närmaste superklass, jämför med `::` i C++.



Omdefinition Av Metoder

- Metoder som *inte kan omdefinieras* av en subklass är:
  - final, static
- Metoder som *kan omdefinieras* av en subklass är:
  - public, protected
- Metoder som *måste omdefinieras* av en subklass är:
  - abstract

Abstrakta klasser

- En abstrakt klass ska alltid kvalificeras med `abstract`.
- En klass kvalificerad med `abstract` är abstrakt *även* om alla metoder är implementerade.
- I en abstrakt klass måste även varje metod som saknar implementation kvalificeras med `abstract`.

Exemplet här nedan visar en abstrakt klass, en deriverad konkret klass och klientkod, allt definerat i filen Client.java.

```
abstract class AbstractClass {

     public String toString() {
        return "AbstractClass"; //can be used in derived classes
     }

     public abstract void myMethod(); // must be implemented
}
```

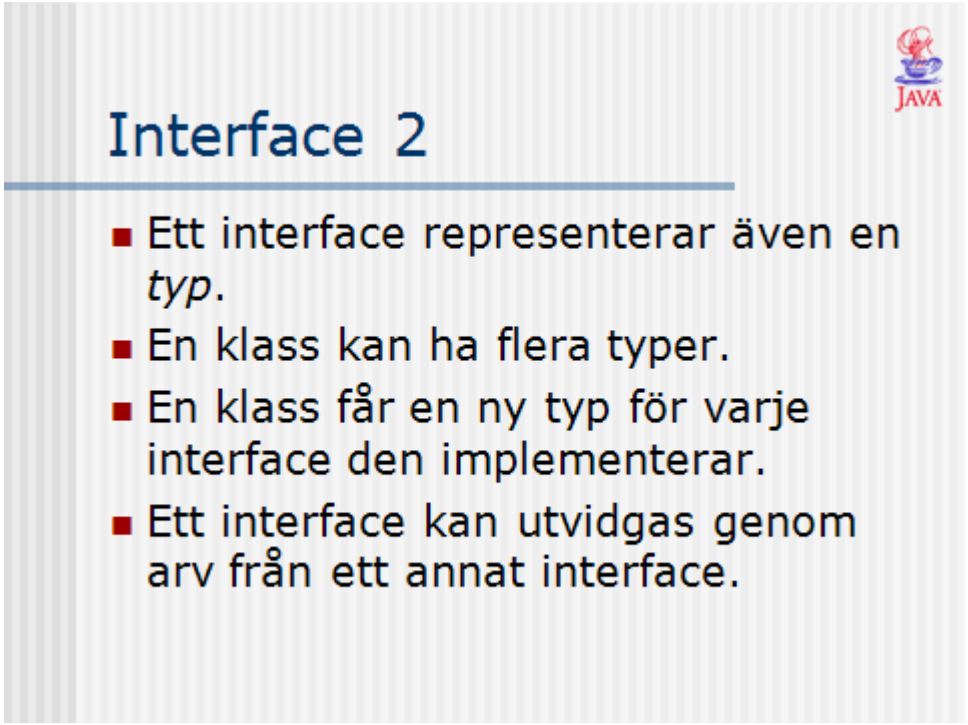
En deriverad klass måste implementera myMethod för att vara konkret:

```
class ConcreteClass extends AbstractClass {

     public void myMethod() {
          System.out.println("This is myMethod in
                                              ConcreteClass.");
     }

     public String toString() {
          return "ConcreteClass, derived from " +
                                         super.toString();
     }
}
```

```
public class Client {

  public static void main(String args[]) {

     AbstractClass ac = new ConcreteClass();
     ac.myMethod();
     System.out.println(ac); //ac.toString();
  }
}

Output:
     This is myMethod in ConcreteClass
     ConcreteClass, derived from AbstractClass
```

Ett javainterface motsvarar en abstrakt ”pure virtual class” i C++:

- bara metod-deklarationer, inga implementationer
- en klass som implementerar ett interface måste förse metodprototyperna med fullständiga implementationer



Genom att implementera ett interface får en klass ännu en typ.
En klass kan alltså flera olika typer, förutom klassens egen typ även de typer som ges av de interface som klassen implementerar. Namnet på ett interface brukar börja med ett I.

Exempel (MyDrawables.java)

```
interface IEatable {
     boolean isAnimal();
     boolean hasTail();
     float getWeight();
}

interface IHuntableAndEatable extends IEatable {
     boolean canFly();
     float getEscapeSpeed();
}

interface IDrawable {
     void drawMe(int x, int y);
}
```

`IEatable` deklarerar tre metoder. `IHuntableAndEatable` ärver dessa tre och deklarerar ytterligare två metoder. Interfacet `IDrawable` deklarerar en metod.

```
class Rat implements IHuntableAndEatable, IDrawable {
     // IHuntableAndEatable
     public boolean isAnimal() { return true; }
     public boolean hasTail() { return true; }
     public float getWeight() { return 0.025f; }
     public boolean canFly() { return false;}
     public float getEscapeSpeed() { return 10.0f; }
     // IDrawable
     public void drawMe(int x, int y) {
          System.out.println("I am a rat.");
     }

     public String toString() { return "rat"; }
}
```

Klassen `Rat` implementerar `IHuntableAndEatable` och `IDrawable` (och därigenom också `IEatable`). Ett Rat-objekt kan därför refereras av flera olika referenstyper:

```
Rat Jerry1 = new Rat( );    // Treat Jerry1 as a Rat
IDrawable Jerry2 = new Rat();  // Treat Jerry2 as an IDrawable;
IEatable Jerry3 = new Rat();     // Treat Jerry3 as an IEatable
IHuntableAndEatable Jerry4 = new Rat(); // ...IHuntableAndEatable
```

Skillnaden mellan dessa olika `Rat`-objekt består i vilka metoder som finns tillgängliga. *Metoderna är knutna till, och begränsas av, den aktuella typen.* Om vi accessar ett `Rat`-objekt genom en `IDrawable`-referens så finns bara metoden `drawMe(int x, int y)` tillgänglig. För `Jerry4` finns alla metoder från interfacet `IHuntableAndEatable` och för `Jerry1` finns samtliga metoder i `Rat` tillgängliga.

```
abstract class Feline // Family of cat species
{     abstract void speak();
      abstract boolean canEat(IEatable aFood);
      abstract boolean canCatch(IHuntableAndEatable aVictim);

}

class HouseCat extends Feline implements IDrawable {
      public void drawMe(int x, int y) {
            System.out.println("I am a housecat");
      }

      public void speak() {
            System.out.println("mjau");
      }

      boolean canEat(IEatable aFood) { // Good enough?
            if(aFood.isAnimal() && aFood.hasTail() &&
               aFood.getWeight() < 0.25)  // A rat maybe...?
               return true;
            return false;
      }

      boolean canCatch(IHuntableAndEatable aVictim) {
            if(!aVictim.canFly() && aVictim.getEscapeSpeed() < 12.0)
                  return true; // Not a bird.
            return false;
      }
}
```

Klassen `HouseCat` implementerar de abstrakta metoderna i `Feline` och blir därmed en konkret klass. Dessutom implementeras interfacet `IDrawable`. `SpaceAlien` är ett annat (hittills okänt) kattdjur:

```
class SpaceAlien extends Feline implements IDrawable{
      public void drawMe(int x, int y) {
            System.out.println("But I am a space-alien");
      }

      public void speak() { System.out.println("uhuh"); }

      boolean canEat(IEatable aFood) { return true; } //Eats anything

      boolean canCatch(IHuntableAndEatable aVictim) { return true; }
}

public class MyDrawables {
      private IDrawable[] drawables = new IDrawable[3];

      public MyDrawables()  // Create and "draw" the IDrawables {
            drawables[0]=new HouseCat();
            drawables[1]=new SpaceAlien();
            drawables[2]=new Rat();

            for (int i=0;i<drawables.length;i++)
                  drawables[i].drawMe(i*5,i*5);
      }

      public IDrawable getDrawable(int idx) {
            return drawables[idx];
      }

      public static void main(String[] args){
            MyDrawables theDrawables = new MyDrawables();

            IHuntableAndEatable food = new Rat(); // A cat eats a rat?
            Feline cat = new HouseCat();

            if(cat.canEat(food))
                  System.out.println("A cat can eat a " + food);
            else
                  System.out.println("No, a cat cannot eat a " + food);

            // Well then, can a cat catch a rat??
            if(cat.canCatch(food))
               System.out.println("A cat can catch a " + food);
            else
               System.out.println("No, a cat cannot catch a " + food);

      }
}
```

När `MyDrawables` instansieras skapas tre `IDrawable`-objekt och `drawMe()` anropas för varje objekt. Ett nytt HouseCat-objekt och ett nytt Rat-objekt skapas men med andra typer av referenser till dessa objekt. Sedan kan vi testa om en katt verkligen kan fånga och äta en råtta…

```
. . .
IHuntableAndEatable food = new Rat();
Feline cat = new HouseCat();

if(cat.canEat(food)) // A cat eats a rat?
      System.out.println("A cat can eat a " + food);
. . .
```

Output:

```
I am a housecat
But I am a space-alien
I am a rat.
A cat can eat a rat
A cat can catch a rat
```

I C++ finns RTTI-mekanismer som t.ex. operatorn `dynamic_cast` för att under runtime kunna testa om det går att göra ett (potentiellt farligt) downcast i en hierarki. Behovet av en sådan mekanism finns naturligtvis också i Java.

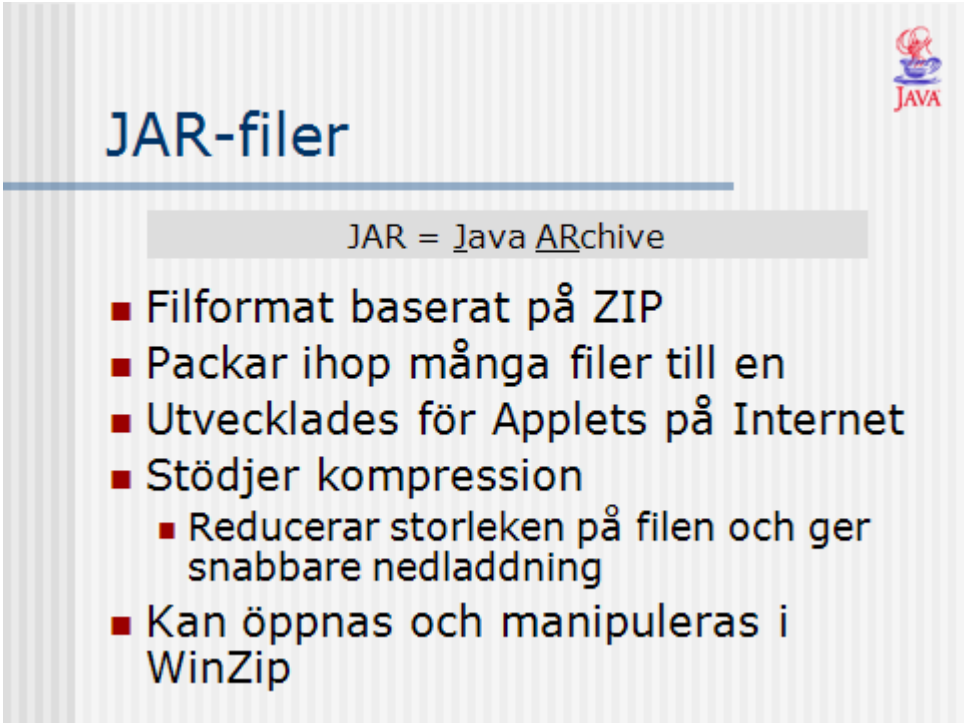
I Java kan man testa om ett objekt har en viss typ genom operatorn `instanceof`.
I
Vi lägger till följande kod i main i föregående exempel:

```
if(food instanceof IEatable)
      System.out.println("IEatable");

if(food instanceof IHuntableAndEatable)
      System.out.println("IHuntableAndEatable");

if(food instanceof IDrawable)
      System.out.println("IDrawable");

if(food instanceof Rat)
      System.out.println("Rat");
```

Vi körning blir utskriften

```
IEatable
IHuntableAndEatable
IDrawable
Rat
```

Detta visar att `Rat`-objektet som `food` refererar till egentligen har fyra typer även om `food` i sig är en referens till en `Rat`. Genom ett typecast (som i C++) kan man vid behov sätta typmekanismen ur spel och själv specificera hur objektet ska betraktas.



JAR-filer är ett filformat med vilket vi kan lägga ihop flera filer till en enda. Filformatet är helt plattformsoberoende och baseras på zip-filformatet. I grund och botten är en JAR-fil en ZIP-fil som innehåller en speciell katalog för metadata.

För att skapa en jar-fil använder vi verktyget jar som följer med i installationen av JDK. Jar används i ett kommandofönster och kräver att antal olika indata. Det är t.ex. namnet på den jar-fil som ska skapas och vilka filer som ska ingå i jar-filen (normalt endast .class och eventuella bilder/ljud/resurser som dessa behöver). Normalt krävs även en s.k. manifest-fil som innehåller information om den jar-fil som skapas. I denna anger vi t.ex. vilken klass som är huvudklass i applikationen (den som innehåller main()-metoden). Syntaxen för verktyget jar ser du nedan:
Syntax: jar {ctxu}[vfm0M] [jar-fil] [manifest-fil] [-C dir] files ...
Alternativ:

-c skapa nytt arkiv
-t visar innehållsförteckning för arkiv
-x extraherar namngivna (eller alla) filer i arkivet
-u uppdaterar befintligt arkiv
-v genererar utförliga utdata vid standardutmatning
-f anger arkivfilsnamn
-m inkluderar manifestinformation från angiven manifestfil
-0 (siffran noll)lagrar enbart; använder inte ZIP-komprimering
-M skapar inte någon manifestfil för posterna
-i genererar indexinformation för de angivna jar-filerna
-C växlar till den angivna katalogen och inkluderar följande fil

files , de filer som ska ingå i jar-filen. Flera filer separeras med mellanslag.
Om någon fil i fil-listan är en katalog bearbetas den rekursivt.
Det är viktigt att vi anger namnen på manifest- och arkivfilerna i samma följd som 'm'- och 'f'-flaggorna har angetts.

Exempel 1: så här arkiverar du två klassfiler i ett arkiv med namnet klasser.jar (en manifest-fil skapas automatiskt):

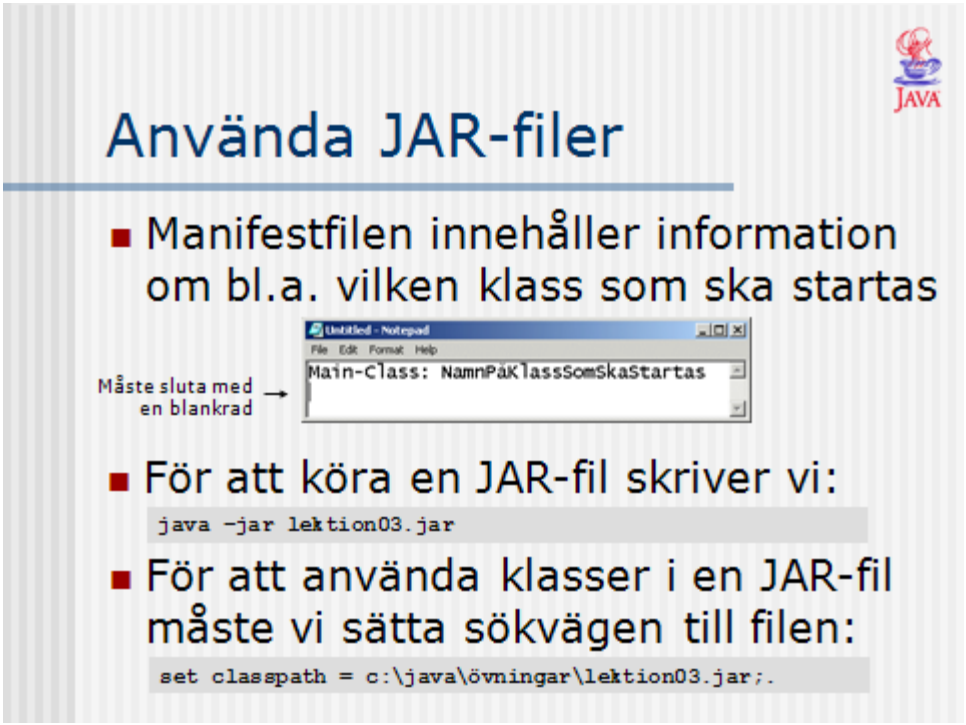
```
jar cvf klasser.jar Foo.class Bar.class
```

c står för att vi ska skapa en jar-fil, v för att vi vill att en utskrift till kommandofönstret , f för att filnamn på arkivfilen ska anges.

Exempel 2: så här använder du den befintliga manifestfilen 'minmanifestfil' och arkiverar samtliga filer i foo/-katalogen tilli 'klasser.jar':

```
jar cvfm klasser.jar minmanifestfil -C foo/ *
```

c står för att vi ska skapa en jar-fil, v att en utskrift ska ske, f att filnamn på arkivfilen ska anges, m att vi anger namn på en befintlig manifest-fil som ska användas, -C att vi ska byta katalog till foo, * att samtliga filer ska arkiveras.



Ha för vana att alltid skriva en egen manifest-fil när du skapar jar-filer (i alla fall om det är en applikation du arkiverar). Manifest-filen innehåller bl.a. information om vilken klass som ska startas när en jar-fil används. Din manifest-fil kan skapas i valfri texteditor, t.ex. Notepad. För att arkivera filerna Barn2.java, Anställd1.java, Person.java och ArvTest2.java i en jar-fil med namnet lektion03.jar gör man på följande sätt.

Eftersom namnet på klassen som innehåller main()-metoden i vår applikation är ArvTest1.java måste vi skapa en manifest-fil för att ange detta när vi skapar jar-filen. Vi skapar därför först en fil med namnet manifest.txt som innehåller följande:

```
Main-Class: ArvTest1
```

OBS! Textfilen måste sluta med en tom rad eftersom sista raden i manifest-filen inte tolkas när jar-filen skapas (har vi inte en tomrad i vårt exempel är raden med Main-Class: ArvTest1 den sista raden och tas därför inte med när jar-filen skapas).
Vi skapar därefter jar-filen med namnet lektion03.jar genom att ge följande kommando i kommandofönstret:
jar cvmf manifest.txt lektion03.jar Person.class Barn2.class Anställd1.class ArvTest1.class

c anger att vi ska skapa en ny jar-fil
v att vi vill att en utskrift ska ske till kommandofönstret
m att vi anger namnet på den manifestfil som ska användas
f att vi anger namnet på den jar-fil som ska skapas

Därefter följer en lista på de filer vi vill ska ingå i jar-filen. Observera att vi inte tar med källkodsfilerna (.java) eftersom vi normalt inte vill dela med oss av den till andra.
För att sen exekvera innehållet i jar-filen:

```
java –jar lektion03.jar
```

Det enda som skiljer mot för att exekvera vanliga applikationer är att vi använder växeln `–jar` . Om vi vill använda filerna i en jar-fil i en annan applikation (t.ex. kunna skapa Barn- och Person-objekt) måste vi inkludera sökvägen till jar-filen i miljövariabeln CLASSPATH (glöm inte bort att vi alltid ska sätta en **.** (en punkt) i CLASSPATH för att klassfiler ska letas i aktuell katalog).