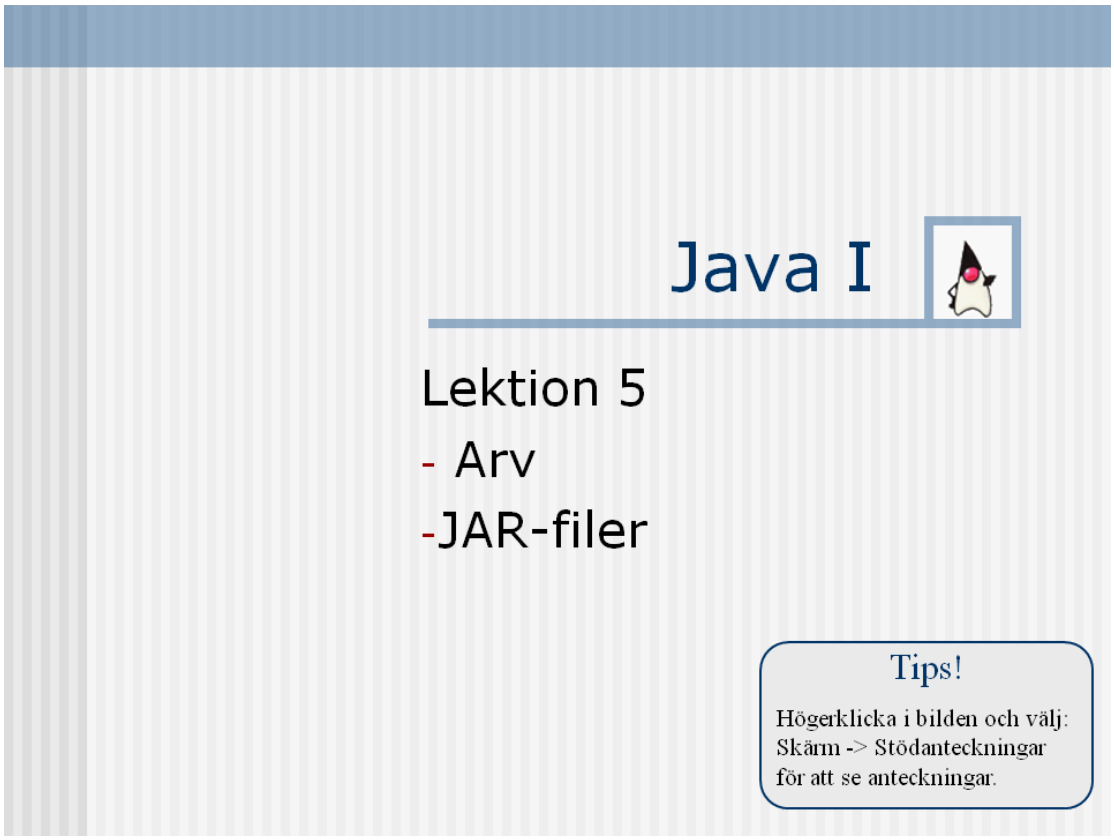
Mittuniversitetet
MID SWEDEN UNIVERSITY

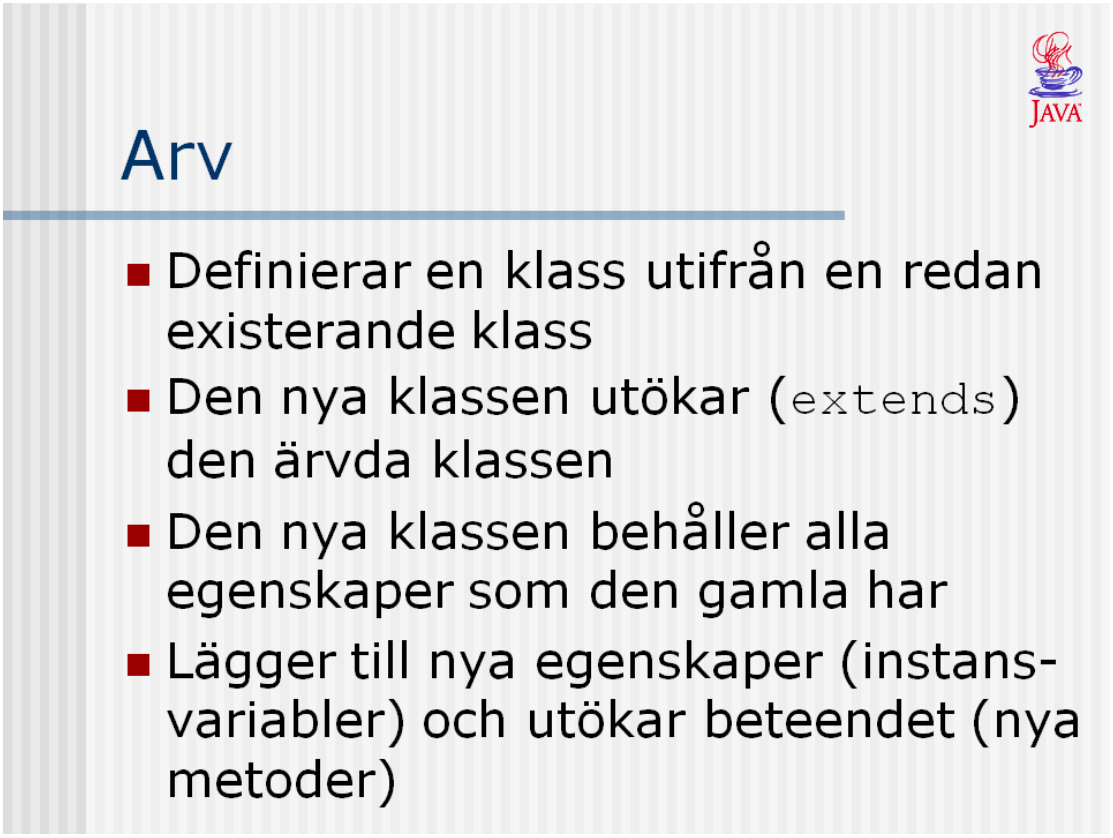
Lektion 5

Datateknik GR(A), Java I, 7,5 högskolepoäng

Syfte:	Att lära sig utöka befintliga klasser med hjälp av arvs-mekanismen och skapa egna klasshierarkier. Att lära sig använda verktyget jar för att paketera Javafiler.
Att läsa:	Kursboken, Kapitel 3.1 (det om jar) Kursboken, Kapitel 4.5.3 (Är-relationen) Kursboken, Kapitel 10 (fram till jämförbara objekt) The Java™ Tutorial, Packaging Programs in JAR Files https://docs.oracle.com/javase/tutorial/deployment/jar/ De två första avsnitten (Using JAR… och Working with…) The Java™ Tutorial, Inheritance https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html



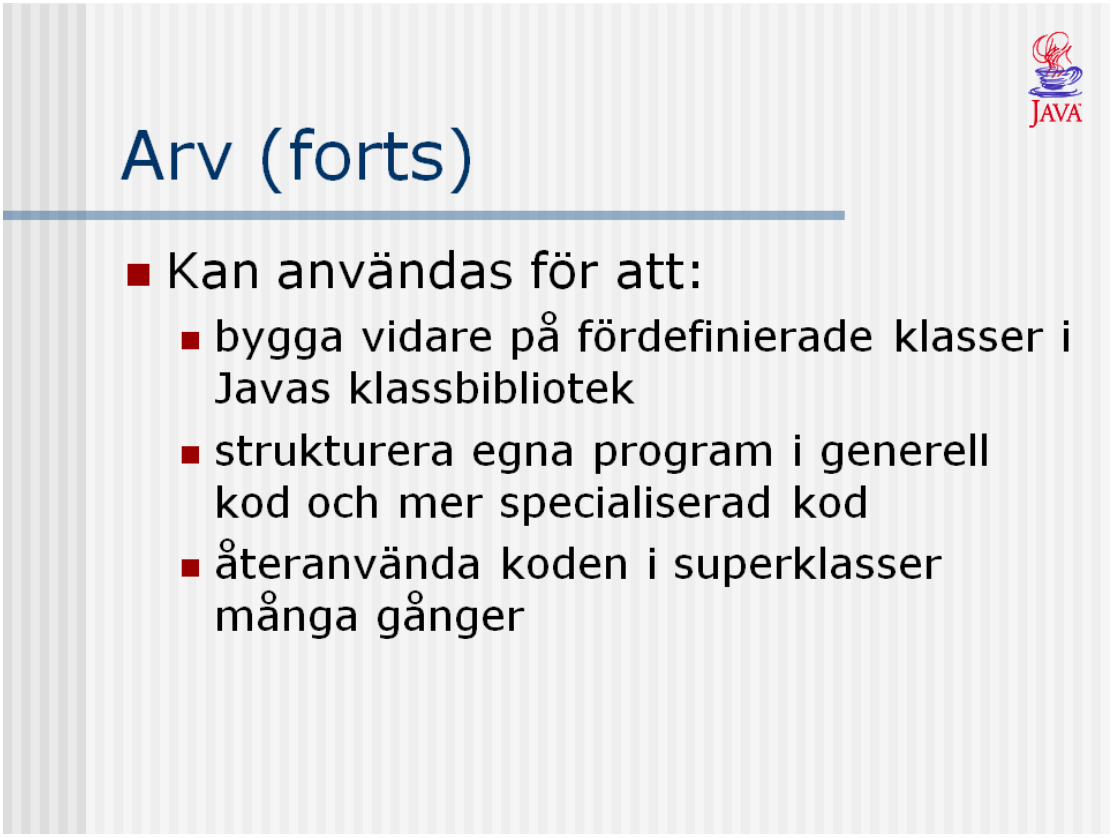
I denna lektion ska vi titta närmare på hur vi med hjälp av arvsmekanismen i Java kan återanvända befintliga klasser när vi skapar en ny klass samt hur vi kan skapa egna klasstrukturer. Avslutningsvis tittar vi på verktyget jar som används för att "packa" ihop de klasser som ingår i en applikation.



Med arvsmekanismen är det möjligt att definiera en klass (en s.k. subklass) utifrån en redan existerande klass (superklass). Man kan alltså skapa en helt ny klass utifrån en klass som redan existerar.

Den nya klassen säger man utökar den ärvda klassen. D.v.s. den nya klassen återanvänder den gamla klassens definition och implementation (instansvariabler och metoder), men utökar den med nya definitioner och/eller implementationer (med fler instansvariabler, fler metoder och kod).

En klass som ärver (subklass) en annan klass (superklass) kan man säga innehåller exakt samma instansvariabler och metoder som superklassen har, men att subklassen utöver de instansvariabler och metoder den ärver också tillhandahåller egna. På detta sätt utökar subklassen de egenskaper/beteende som superklassen har.

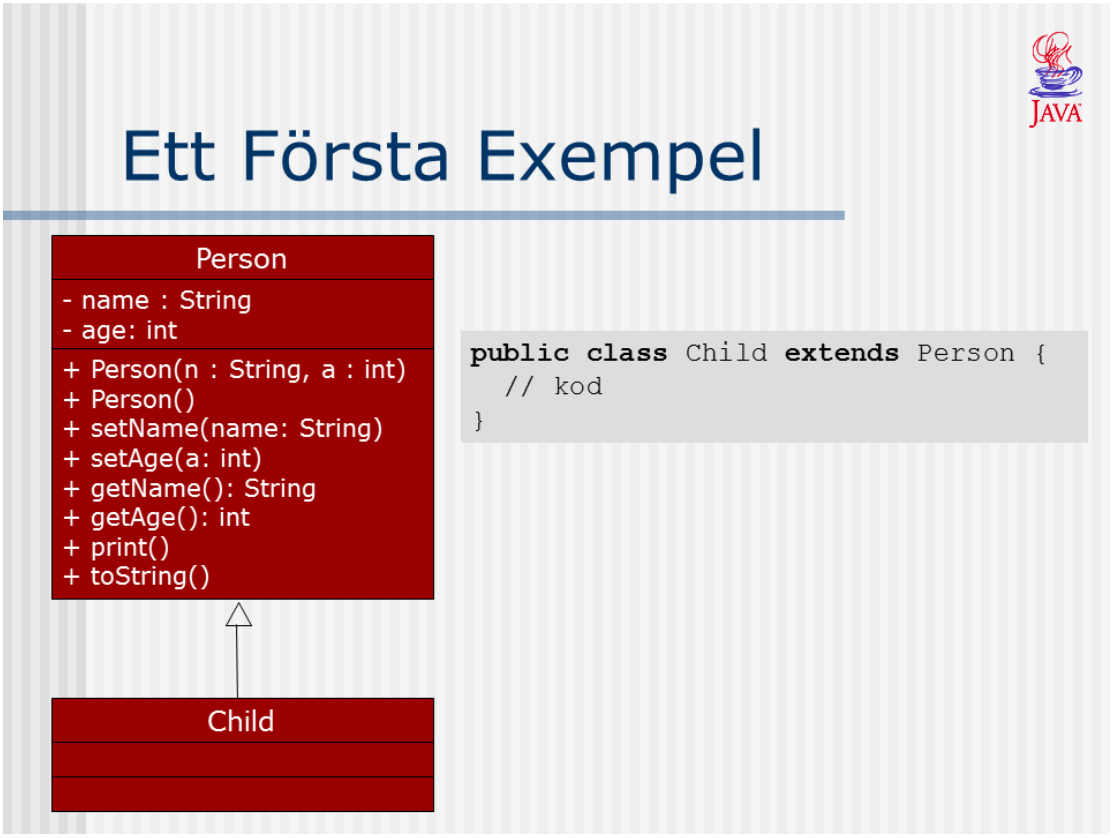


Arv kan man använda på lite olika sätt:

I Java finns det ett stort antal färdiga klasser som följer med i klassbiblioteket. Dessa är oftast väldigt användbara som de är, men det finns tillfällen då dessa klasser inte riktigt uppfyller de behov man har. Man kan då skapa en klass som ärver någon av dessa ”färdiga” klasser och utöka den med kod som bättre passar det vi vill utföra. T.ex. kan man ärva klassen JButton (en klass i paketet javax.swing och som används i grafiska användargränssnitt) och ge den egna egenskaper, t.ex. ändra färgen m.m. Skapandet av grafiska användargränssnitt bygger för övrigt alltid på arv.

Det är också vanligt att använda arv för klasser i egna applikationer, för att strukturera upp de begrepp som används och för att skilja ut generell kod från mer specialiserad kod. Vi vill kanske skriva ett program som hanterar olika flygplanstyper och där varje klass beskriver en viss typ av flygplan. Alla flygplan har alltid någonting gemensamt (t.ex. motor, kabin, vingar etc.) och i stället för att upprepa dessa i varje klass (för varje typ av flygplan) är det bättre att samla sådant som är gemensamt i en superklass (innehåller alltså generell kod). Därefter kan de olika flygplanstyperna ärva dessa egenskaper och fylla på med mer specialiserad kod för just den aktuella flygplanstypen.

På detta sätt kan vi återanvända koden i superklassen många gånger i olika subklasser och vi behöver inte upprepa koden.



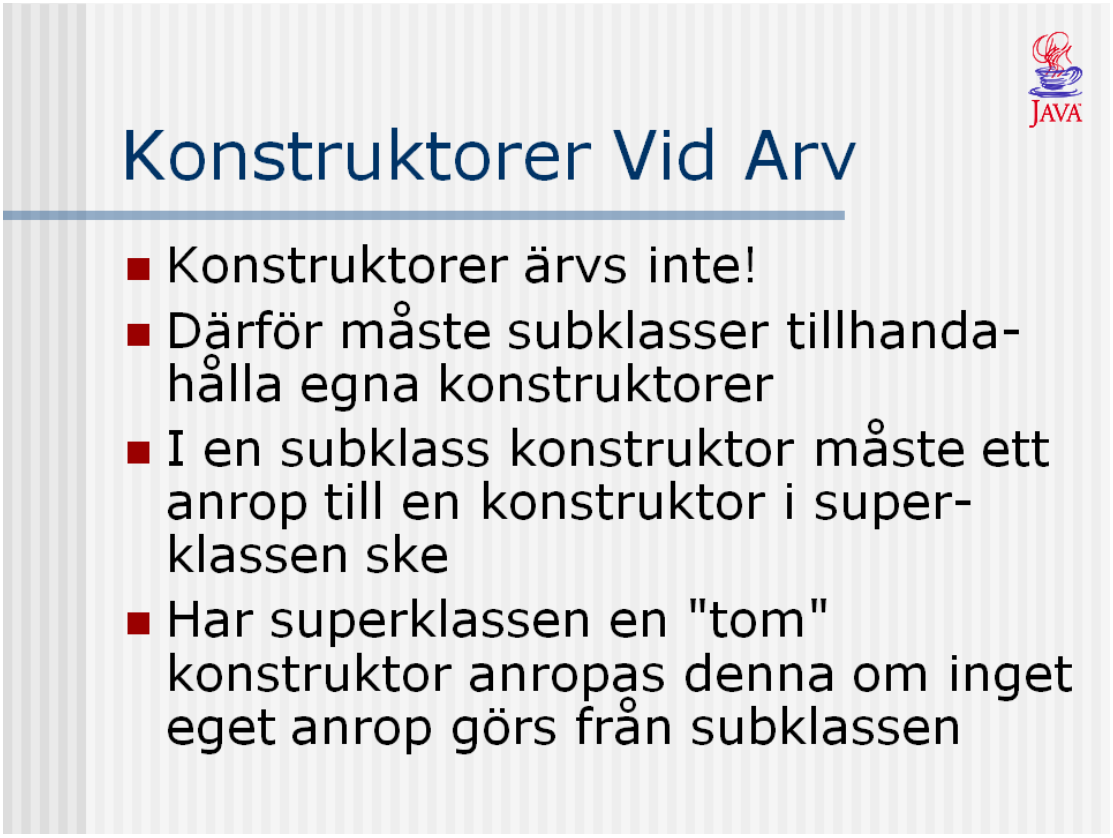
Vi ska nu titta på ett första exempel på arv och utgår från en av de första versionerna av klassen `Person` som använts i tidigare exempel. I vår applikation som vi tänker skriva vill vi kunna skapa både `Person`-objekt och `Child`-objekt. Ett barn ska ha exakt samma egenskaper som en person och i stället för att i stort sett kopiera alla instansvariabler och metoder från `Person`-klassen till en ny `Child`-klass utnyttjar vi arv.

I UML visar vi arv genom en pil som pekar från klassen som ärver (subklass) mot klassen den ärver ifrån (superklass). Eftersom vi i det här läget inte utökar klassen `Person` med nya instansvariabler eller metoder är klassen `Child` helt ”tom” i klassdiagrammet.

I klassdeklarationen använder vi nyckelordet `extends` följt av namnet på den klass vi ska ärva egenskaperna ifrån. Eftersom vi inte skulle utöka klassen `Person` består klassen `Child` av endast klassdeklarationen. Det är nu möjligt att skapa objekt av klassen `Child` och anropa metoder som finns deklarerade i `Person` enligt följande:

```
Child myChild = new Child();
myChild.setName("Melinda");
myChild.setAge(2);
myChild.print();
```

Observera att vi inte kan skapa ett `Barn`-objekt genom att använda konstruktorn som tar två parametrar i klassen `Person`. Anledningen till detta kommer vi till strax.
Titta nu på exemplen **Person.java**, **Child.java** och **ChildTest.java**.



Som vi såg i `ChildTest` måste vi ge `Child`-objekt dess värden genom de `set`-metoder som finns. Anledningen till att vi i klassen `Child` inte kan skapa ett objekt genom att skriva `Child daughter = new Child("Stina", 8);` är att superklassens konstruktorer inte ärvs av subklasserna. Som nämnts i tidigare lektion anses inte konstruktorn vara en medlem i klassen och det är endast medlemmar i klassen som ärvs.

Av den anledningen måste alla subklasser tillhandahålla egna konstruktorer för att vi ska kunna skapa objekt på ett smidigt sätt. Precis som vanligt tillhandahålls en konstruktor, `Child()` som inte tar några argument, om vi inte skriver en egen konstruktor. Därför kan vi i `ChildTest` skriva `Child daughter = new Child();`

I konstruktorn i en subklass måste man placera ett anrop på någon av superklassens konstruktorer. Detta måste göras för att samtliga delar av det objekt som skapas ska initieras korrekt. Namn och ålder för ett barn är deklarerade i superklassen och måste därför initieras där. Och det är ju konstruktorns uppgift att skapa och initiera ett objekt riktigt. Ett `Child`-objekt 'är-ett' `Person`-objekt med några modifikationer. Därför måste vi se till att både konstruktorn i `Child` och konstruktorn i `Person` körs så att objektet skapas riktigt.

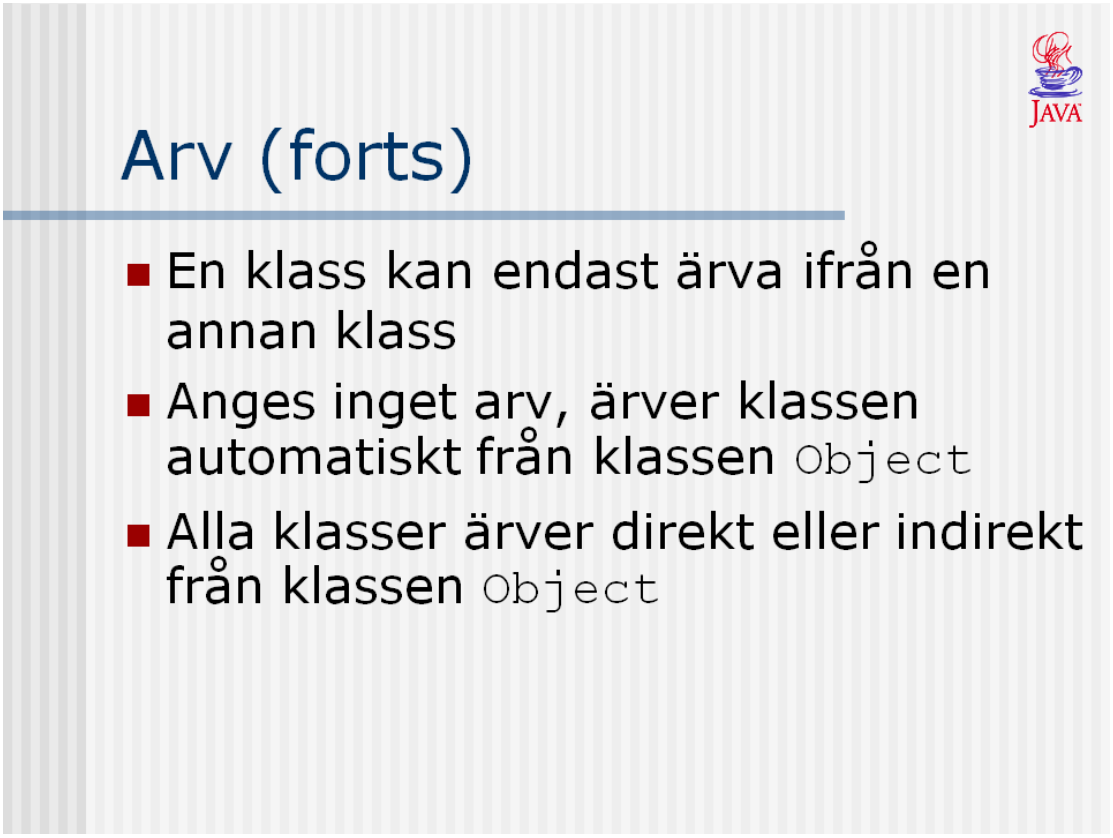
Det enda tillfälle när subklassens konstruktor inte behöver anropa superklassens konstruktor är när superklassen har en "tom" konstruktor som inte tar några argument. Den kommer då att anropas automatiskt och eftersom inga argument krävs så krävs inte heller anropet i subklassen. Det är därför namnet sätts till "okänd" och ålder till -1 när vi skapar ett nytt `Child`-objket och inte använder `set`-metoderna.

För att anropa en konstruktor i superklassen från subklassens konstruktor använder vi nyckelordet `super` och innanför parenteserna skickar vi de argument som konstruktorn i superklassen behöver. Jämför detta med hur vi använder this för att anropa en konstruktor i den egna klassen.

Ett anrop till en konstruktor i superklassen måste vara det första som sker i subklassens konstruktor, d.v.s. vi måste placera super-anropet först, före andra programsatser i subklassens konstruktor. Gör vi inte det får vi ett felmeddelande när vi kompilerar klassen.

I exemplet i bilden har vi en konstruktor i klassen Barn som tar två parametrar. När vi skapat ett nytt `Child`-objekt vill vi ange namnet och åldern på barnet. Eftersom det är i superklassen `Person` dessa ska sättas gör vi ett anrop med `super` och skickar med name och age som argument.

Nu när vi har en konstruktor som tar argument kan vi inte längre skapa `Child`-objekt genom att använda den tomma konstruktorn (om vi inte själva skriver en förstås).



I Java kan man (tillskillnad från en del andra programmeringsspråk) endast ärva från en superklass (enkelt arv). Varje klass i Java har med andra ord en, och endast en direkt superklass (utom klassen `Object` som inte ärver någon annan klass)

Om man i en klassdeklaration inte anger något arv (d.v.s. `extends` saknas) kommer ändå klassen som deklarerats att ärva automatiskt från klassen `Object`. `Object` är en fördefinierad klass i Java som innehåller en del generella metoder som alla klasser måste ha.

På ett eller annat sätt ärver alla klasser de metoder som finns i klassen `Object`. Det spelar ingen roll i vilket led i arvet vi befinner oss i så kan vi ändå anropa de metoder som finns definierad i `Object`.

Prova gärna att i **Child.java** att ärva från två klasser samtidigt. T.ex. `extends Person, Employee`.

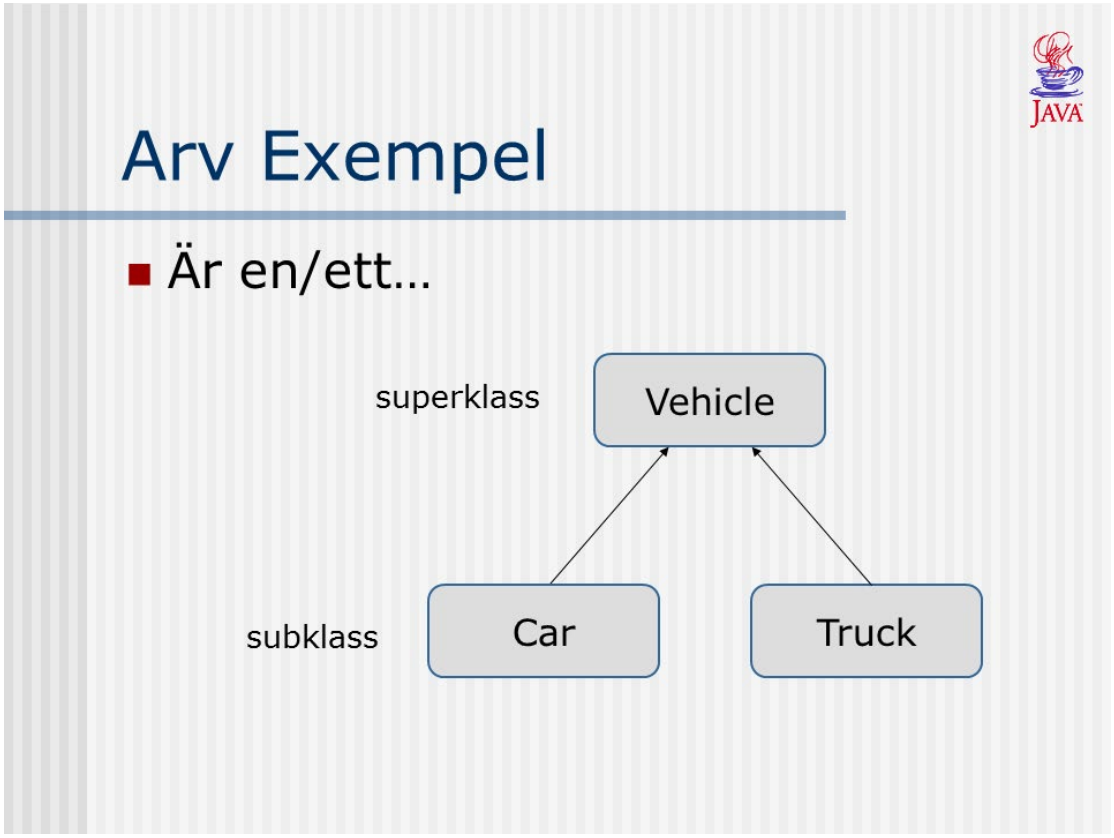
Alla klasser ärver alltså direkt eller indirekt ifrån klassen `Object`. `Object` fungerar som en generell superklass för samtliga klasser. Detta gäller oavsett om en faktisk deklaration finns av arvet eller inte. Därför är ovanstående exempel helt lika. I båda fallen ärver `Person` ifrån klassen `Object`.

De metoder som finns i `Object` är således extra viktiga eftersom de kan användas och omdefinieras på samtliga klasser. Det innebär också att det är möjligt att genom att deklarera en variabel av typen `Object` referera till samtliga objekt i ett program, eftersom alla klasser ärver från `Object`. Dock kan naturligtvis endast metoderna i klassen `Object` anropas via en sådan variabel, vilket är en betydande restrektion.

Det är alltså inte möjligt att i exemplet ovan anropa metoder som återfinns i klassen `Person` genom att använda objektet `dad`. Detta objekt är deklarerat av typen `Object` och vi kan endast anropa metoder som finns i klassen `Object`. Vi kan alltså alltid deklarera ett objekt att vara av typen ”superklass”, men att vi initierar objektet med en ”subklass” till superklassen. Vi kan t.ex. göra följande:

```
Object dad = new Person("Kalle", 33);
Person daughter = new Child("Stina", 8);
```

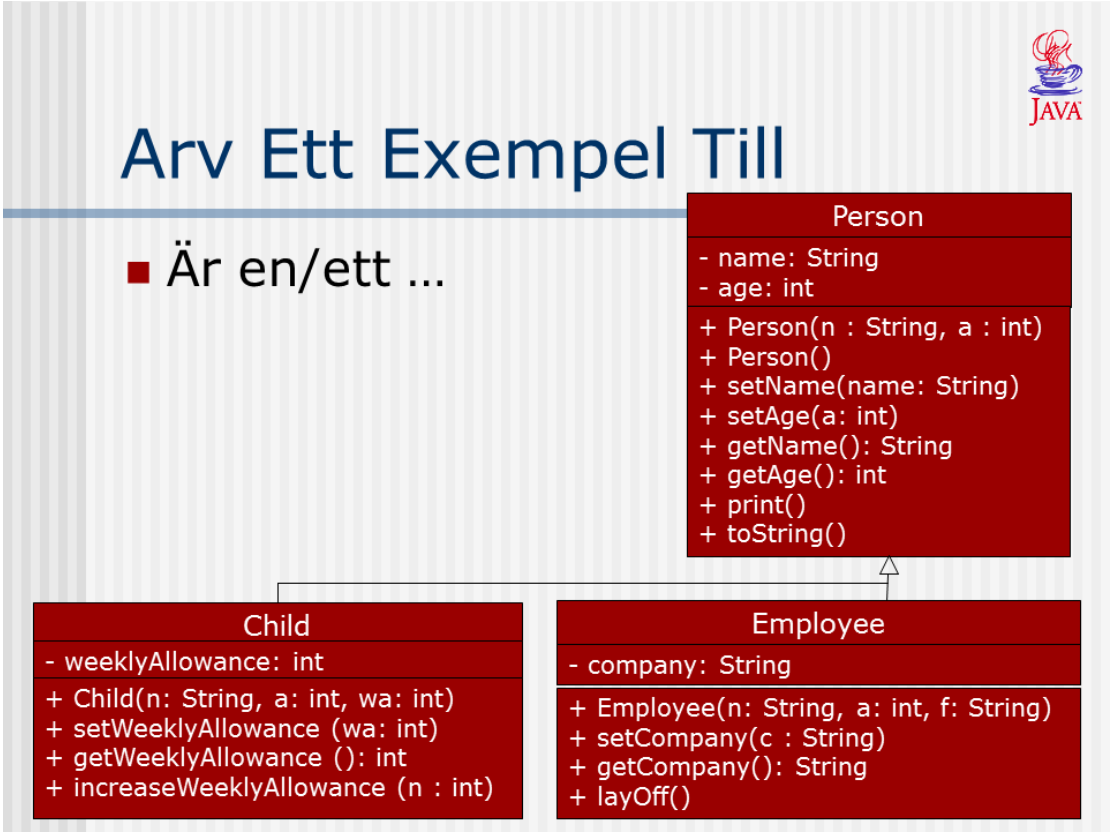
Detta kan vara väldigt användbart i vissa situationer (tas upp i Java II). I exempel **ChildTest1.java** demonstreras deklarationerna ovan.



Det är viktigt att använda arvsmekanismen på rätt sätt. Arv betecknar en subtypsrelation, där den nya klassen (subklassen) ska bibehålla samtliga egenskaper som den gamla klassen (superklassen) har. Subklassen kan utöka och/eller omdefiniera en eller flera metoder (dvs. ha en annan implementation för en metod, men beteendet för vad denna metod ska utföra ska inte förändras).

Ett bra sätt att testa om ett arv är lämpligt är att testa meningen **är-en** eller **är-ett** sett ifrån subklassen mot superklassen. Ett exempel kan vara en subklass allemanskonto som **är-ett** bankkonto, personbil som **är-ett** fordon, dialogfönster som **är-ett** fönster. Om det inte går att använda detta talesätt ifrån subklassen till superklassen är arvet sannolikt olämpligt (eller så har mycket dålig namnsättning gjorts på klasserna).

I exemplet i bilden har vi en superklass `Vehicle` som innehåller generella egenskaper för alla typer av fordon. En bil **är-ett** fordon och även lastbil **är-ett** fordon, så här är arv lämpligt att använda. Vi samlar sådant som är gemensamt för både personbilar och lastbilar i superklassen `Vehicle` och sådant som är unikt för en personbil lägger vi i `Car` och sådant som är unikt för en lastbil lägger vi i klassen `Truck`.



I detta exempel bygger vi vidare på vår arvshierarki av `Person` och `Chil`, med ytterligare en klass; nämligen `Employee`. Här är ett arv väldigt lämpligt att använda eftersom både ett barn och en anställd har ett namn och en ålder. I stället för att upprepa koden för instansvariablerna `name` och `age` samt metoderna `setAge` och `getName` m.fl. i de båda klasserna låter vi i stället dessa klasser ärva från `Person`. Ett barn **är-en** person och anställd **är-en** person.

Utöver de medlemmar i `Person` som dessa två subklasser ärver kommer vi att lägga till några egna medlemmar så att klasserna passar oss. Vi kommer alltså att utöka beteendet i `Person`.

`Child` kommer att ha en egen instansvariabel för att hålla koll på vilken veckopeng barnet får. Det kommer även att finnas `set`- och `get`-metoder för denna instansvariabel, samt en metod för att öka veckopengen när barnet blir äldre.

För en anställd är vi intresserad av att veta vilket företag den anställde jobbar på och lägger därför till en instansvariabel för detta. För att kunna ändra/hämta företaget har vi `get`- och `set`-metoder. Det finns även en metod `layOff` som används för att avskeda den anställde från företaget (endast en utskrift kommer att ske).

Ta en titt på exemplen **Child1.java**, **Employee.java** och **InheritanceTest.java**. Child1.java och Employee.java är skrivna enligt klassdiagrammet ovan. I testklassen skapas olika objekt och de olika metoderna testas. Observera att vi i subklasserna inte direkt kommer åt instansvariablerna i `Person` utan måste använda `get`- och `set`-metoderna.

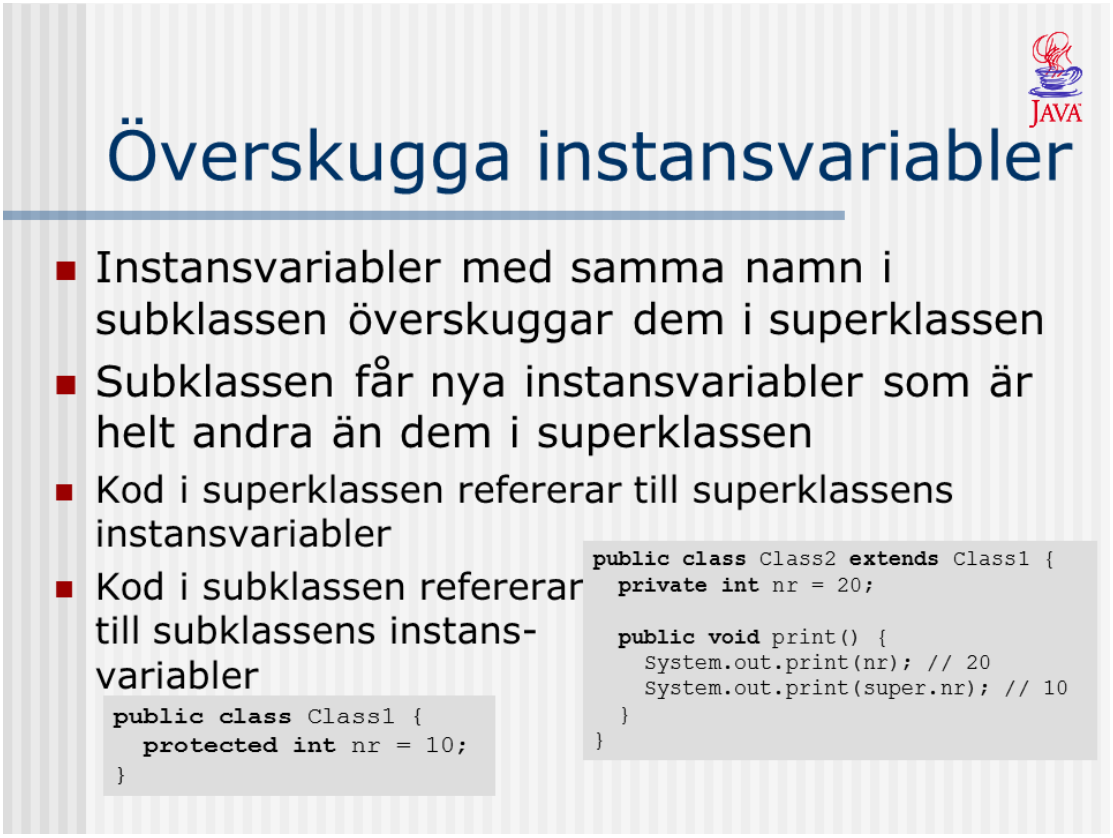
Som du märkte kan vin inte från subklasserna `Child` och `Employee` direkt komma åt instansvariabler deklarerade i superklassen `Person`. Man kan tycka att subklasserna trots allt ärver superklassens instansvariabler och metoder och borde väl ha tillgång till dessa medlemmar? Så är dock inte fallet.

De medlemmar (instansvariabler och metoder) som har deklarerats som `private` eller `public` bibehåller samma åtkomstregler som tidigare, d.v.s. privata medlemmar är inte tillgängliga utanför klassen de är deklarerade i (kan ej användas i subklasserna) medan publika medlemmar är tillgängliga. Detta gäller även gentemot subklassen och i detta fall så har subklassen alltså inte tillgång till instansvariablerna `name` och `age` eftersom dessa är deklarerade som `private` i `Person`. En subklass ärver alltså inte medlemmar i en superklass som är deklarerad som `private`. Däremot ärver subklassen superklassens alla publika medlemmar.

Det finns dock ytterligare en åtkomstform, skyddad åtkomst, som är speciellt intressant i samband med arv. En medlem som deklarerats som skyddad (`protected`) fungerar som privat gentemot andra klasser, men publik gentemot klassens subklasser. Det innebär att om instansvariablerna `name` och `age` istället deklareras som `protected` blir de åtkomliga i subklasserna, och i detta fall är det nog en lämpligare lösning eftersom dessa instansvariabler behöver refereras från subklassen (i t.ex. metoden `increaseWeeklyAllowance`).

Vill man helt förhindra att en klass kan ärvas kan man deklarera klassen som `final`, ex `public final class Person`

Prova nu i **Child1.java** ändra metoden `increaseWeeklyAllowance` så att du direkt använder instansvariabeln `name` i stället för anropet till metoden `getName`. Kompilera och notera vilket felmeddelande som ges. Ändra sen åtkomsten till instansvariablerna `name` och `age` till `protected` i `Person` (kompilera). Kompilera sen Child1.java igen.

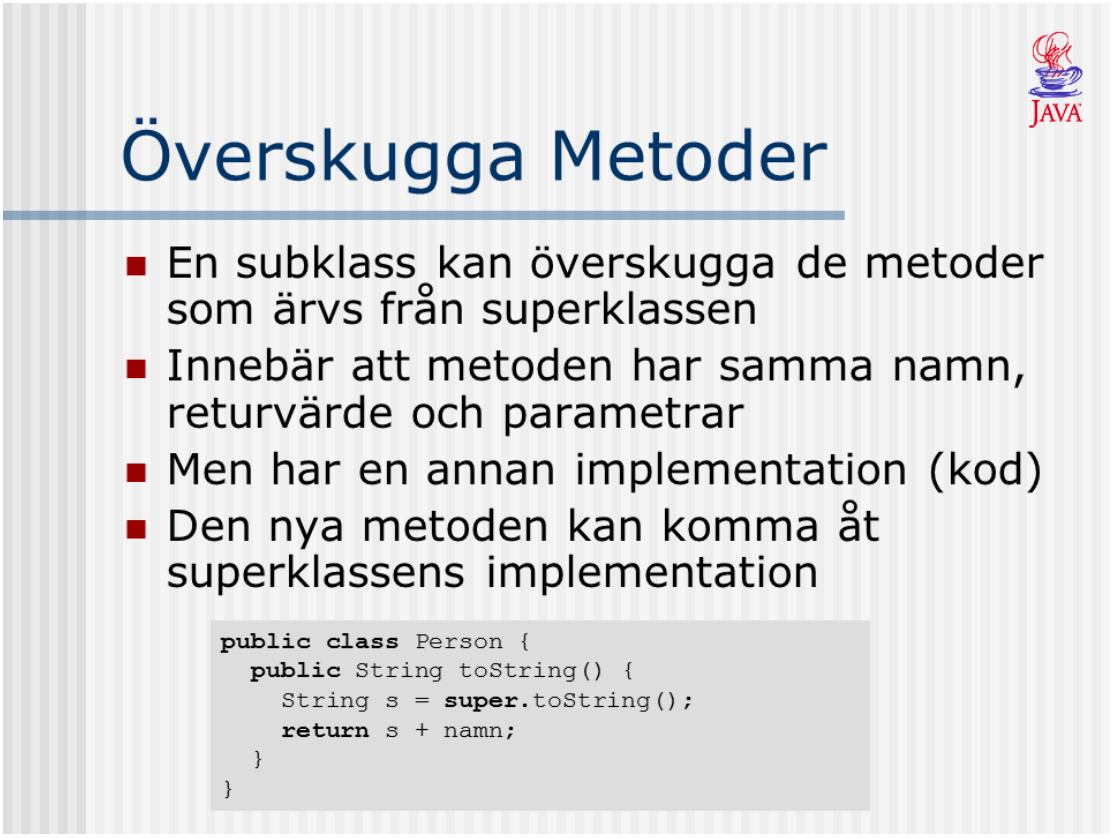


Ett instansvariabel kan inte omdefinieras eller tas bort i en subklass, endast överskuggas. Om man i klassen `Child` och `Employee` deklarerar två instansvariabler `name` och `age`, vilket är samma namn instansvariablerna i superklassen har, kommer detta inte att omdefiniera superklassens instansvariabler utan i stället lägga till två nya instansvariabler i subklassen. Dessa nya instansvariabler råkar ha samma namn som dem i superklassen men är helt separerade från de i superklassen (jämför med instansvariabler och parametrar i en metod med samma namn).

Ett objekt av subklassen (`Child` eller `Employee`) kommer att ”innehålla” både superklassens instansvariabler `name` och `age` (om de är `public` eller `protected`) och subklassens instansvariabler `name` och `age`.

Kod som i superklassen refererar till dessa instansvariabler gör det med definitionen de har där, medan kod i subklassen gör det med den definition som de har i subklassen. Detta är naturligtvis förvirrande och bör i största mån undvikas – det är svårt att finna några bra anledningar till varför instansvariablerna i subklassen ska döpas till samma namn som i superklassen.

Instansvariablerna i superklassen finns ändå tillgängliga i subklassen via nyckelordet `super` (se exempel i bilden).

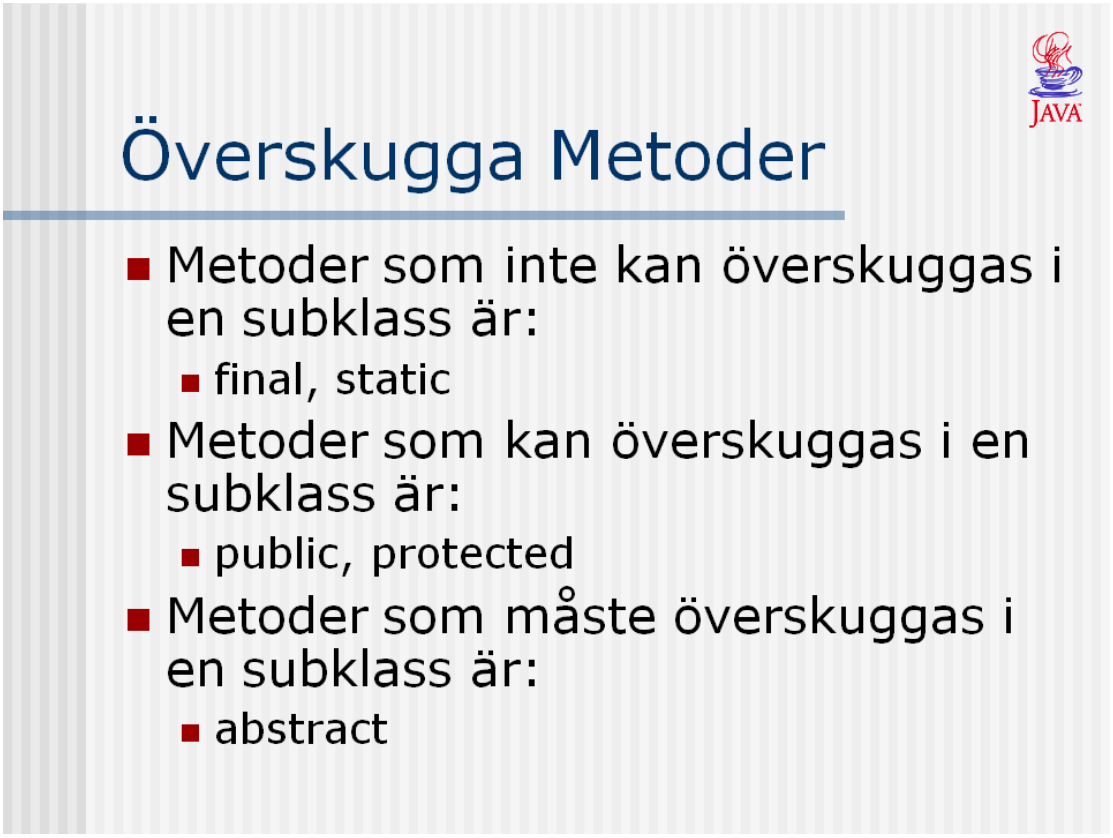


Det är möjligt att i en subklass överskugga metoder i en superklass. D.v.s. de metoder en subklass ärver från en superklass kan subklassen välja att ha en annan innebörd. Den överskuggade metoden måste ha samma namn, returvärde och parametrar som superklassens definition. Men den har alltså en annan implementation, d.v.s. koden för hur metoden ska utföras är annorlunda i subklassen. Det är dock principiellt viktigt att bibehålla betydelsen av vad metoden gör. Ett exempel på överskuggning av metoder har vi redan sett när vi använder metoden `toString` i våra klasser (`toString` ärvs direkt eller indirekt från klassen `Object`).

Trots att subklassen har omdefinierat betydelsen av metoden `toString` kan vi från subklassen komma åt superklassens metod. Vi kan ifrån den omdefinierade metoden anropa den metod som ursprungligen ärvdes. Detta görs med nyckelordet `super` (ungefär som vi kommer åt överskuggade instansvariabler). Detta för att möjliggöra att den omdefinierade metoden (nya metoden) ska kunna använda den gamla metodens implementation om den vill.

I exemplet i bilden kan vi i `toString` använda nyckelordet `super` för att komma åt klassen `Object` och dess implementation av `toString`. Vi använder punktnotation precis som om vi haft ett objekt att anropa metoden med.

I exemplen **Child2.java** och **Employee1.java** har jag omdefinierat metoden `print` så att även information som tillhör respektive subklass skrivs ut (veckopeng för `Child` och företag för `Employee`). Ta en titt på dessa klasser tillsammans med **InheritanceTest1.java**.

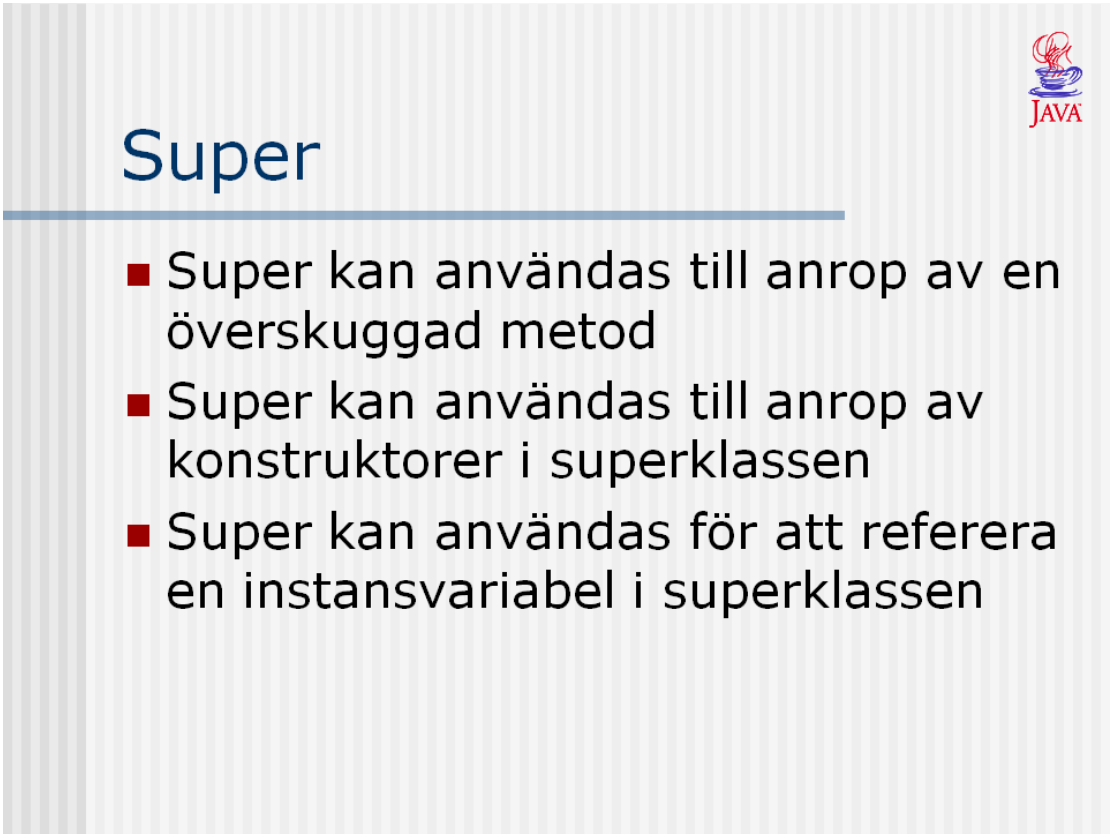


En subklass kan inte överskugga metoder som är deklarerade som `final` i superklassen. Om man försöker omdefiniera en metod som är `final` kommer kompilatorn att generera ett felmeddelande.

En subklass kan inte heller omdefiniera en metod som är deklarerad som `static` i superklassen. Med andra ord, en subklass kan inte omdefiniera en s.k. klassmetod.

En subklass kan välja att omdefiniera metoder som i superklassen är deklarerad som `public` eller `protected`. Men detta är inget krav, vilket vi sett exempel på i de tidigare bilderna.

En subklass måste däremot omdefiniera metoder som är deklarerade som `abstract`, om inte måste även hela klassen deklareras som `abstract`. Och vad abstrakta klasser och metoder är tas upp i nästa kurs (Java II).

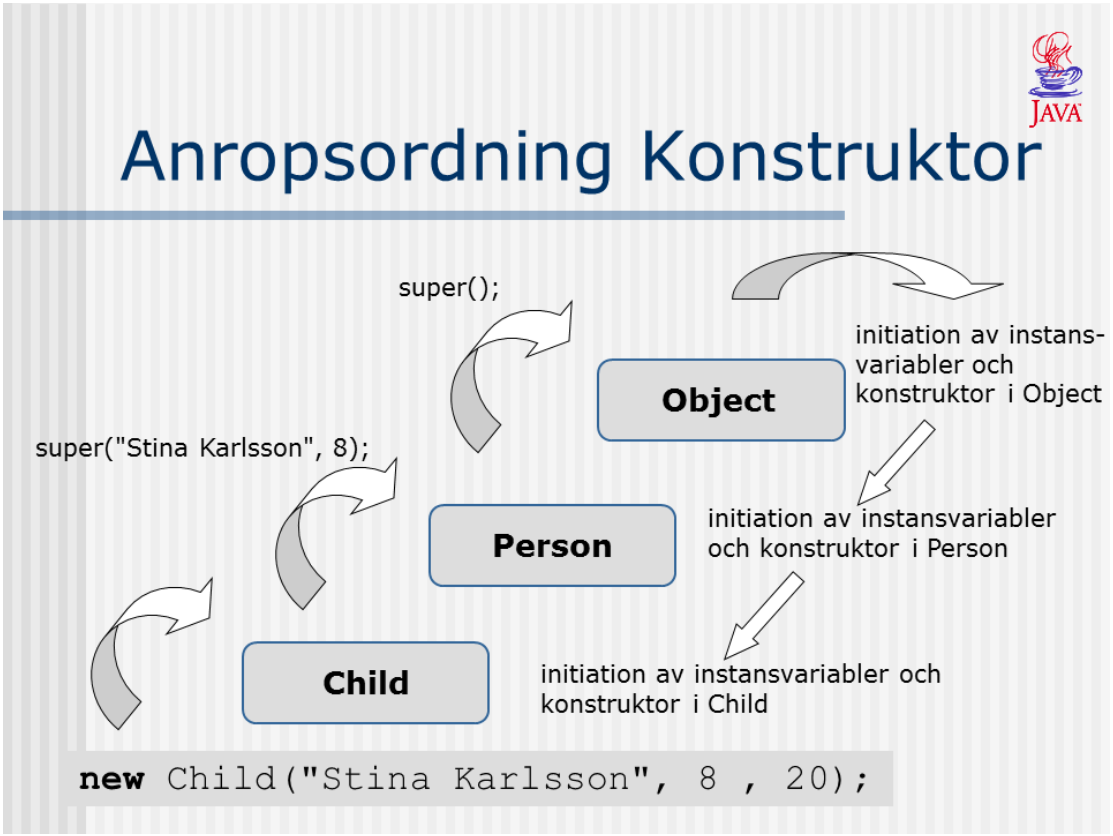


Innan vi går in på nästa del i lektionen vill jag ge en sammanställning av hur vi kan använda nyckelordet `super`. Ett sätt är om vi i en subklass har överskuggat en metod i superklassen. För att komma åt metoden i superklassen kan vi då anropa denna med `super`.

För att anropa en konstruktor i superklassen kan vi använda nyckelordet `super` och skicka med de argument som behövs till konstruktorn i superklassen.

Vi kan också använda `super` för att komma åt instansvariabler som är överskuggade. D.v.s. om vi i subklassen har en instansvariabel med samma namn som i superklassen kommer vi åt superklassens instansvariabel genom att ange nyckelordet `super` framför namnet på instansvariabeln.

Jämför vi med nyckelordet `this` finns det stora likheter. Skillnaden är att `this` använder vi när vi vill referera till något som finns i klassen (instansvariabler, konstruktor) medan `super` använder vi när vi vill referera till något som finns i superklassen (instansvariabler, konstruktorer, metoder).



Den exakta anropsordningen för hur ett objekt av en given klass skapas är:

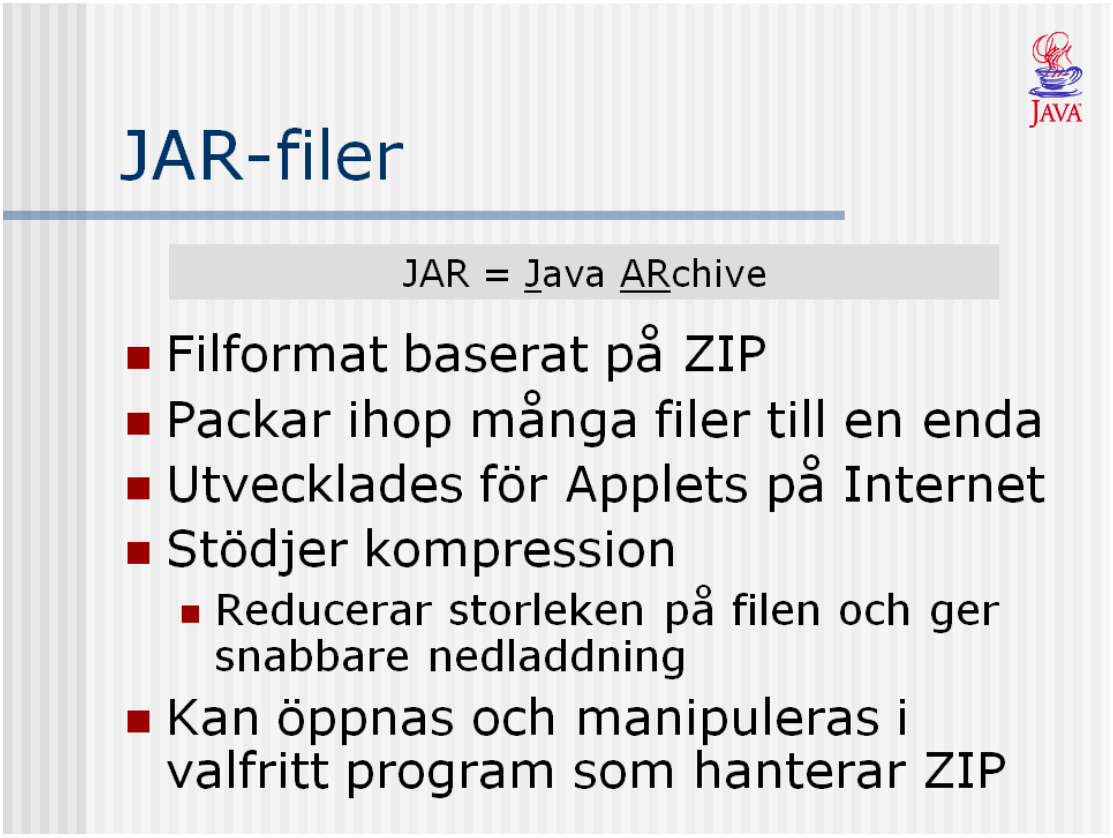
0. Objektet av klassen skapas med nyckelordet new och klassens konstruktor, som matchar de argument som anges, körs.
1. Superklassens konstruktor anropas (via `super`).
2. Instansvariablerna i superklassen initieras via sina initieringssatser (d.v.s. om vi tilldelar våra instansvariabler ett värde direkt vid deklarationen, om inte ges instansvariablerna sina default-värden).
3. Den egna klassens konstruktor (koden efter super-satsen) exekveras.

Denna ordning tillämpas rekursivt, d.v.s. om en superklass i sin tur ärver en annan klass så kommer konstruktorn i denna andra superklass att anropas först. Som en konsekvens av ovanstående regler ser man också att om en instansvariabel både initieras direkt vid deklarationen och senare i konstruktorn så exekveras konstruktorns tilldelning senare och blir alltså den som slutgiltigt bestämmer instansvariabelns värde.

När ett nytt Child-objekt skapas sker det i denna ordning:

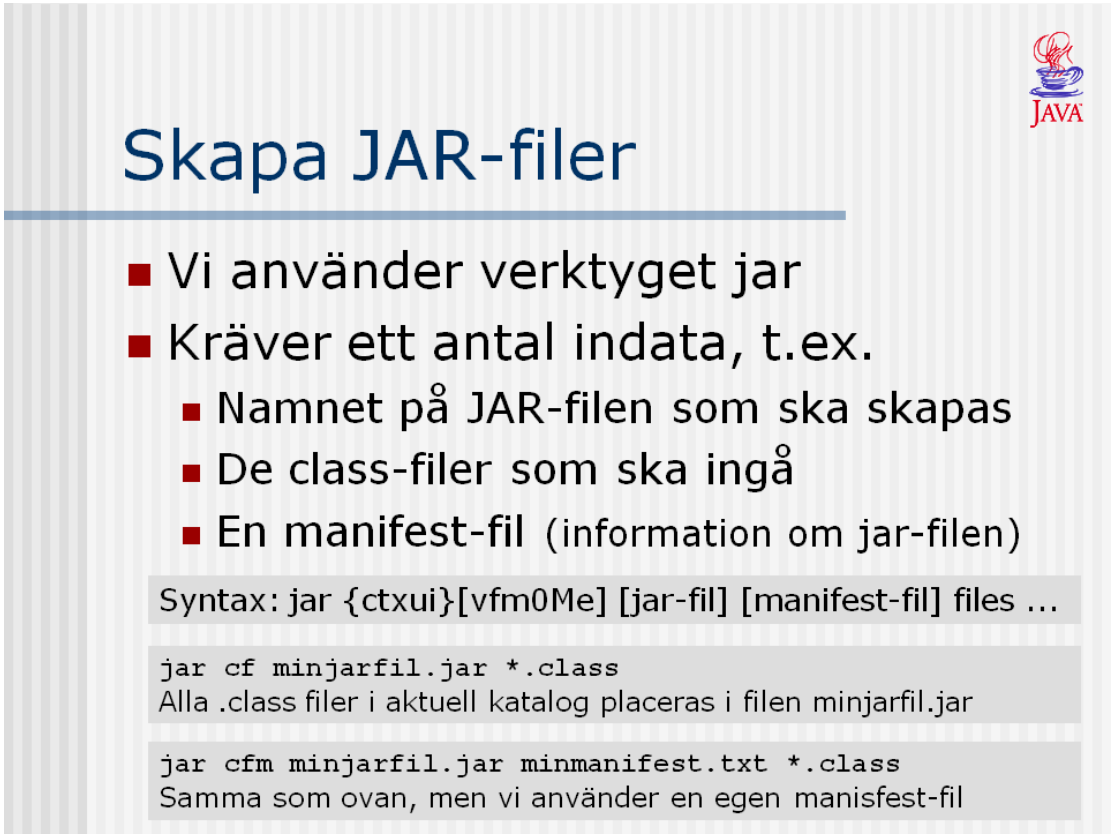
I samband med `new` anropas den konstruktor i klassen Barn som matchar bäst, men först initieras alla instansvariabler i klassen Child (weeklyAllowance initieras till dess defaultvärde 0).
Det första som sker efter att instansvariablerna initierats är ett anrop på superklassens konstruktor och som argument skickar vi med namn och ålder (som ju Person's konstruktor behöver).
Innan konstruktorn i Person körs initieras alla instansvariabler i klassen Person (age ges värdet 0 och name ges en referens till null). Därefter är det den "tomma" konstruktorn i klassen Object som anropas.
Minns att en default-konstruktor anropas automatiskt från subklassen om ingen egen konstruktor finns eller om inget `super`-anrop görs.
När Object är klar återgår exekveringen till konstruktorn i klassen Person. Här tilldelar vi instansvariablerna name och age de värden som skickades till konstruktorn som argument från Child.
När Person är klar med sin konstruktor återgår exekveringen till raden nedanför super-anropet i Child. Där tilldelar vi veckopengen för barnet.

Nu är ett objekt skapat av klassen Child och vi kan börja anropa eventuella metoder som klassen har.



JAR-filer är ett filformat med vilken vi kan lägga ihop flera separata filer till en enda fil. Filformatet är helt plattformsoberoende och baseras på det populära zip-filformatet. I grund och botten är en jar-fil en zip-fil som innehåller en speciell katalog för metadata. En jar-fil kan ”öppnas” med alla program som normalt kan hantera zip-filer (t.ex. med 7-Zip).

Tack vare att många filer kan grupperas ihop till en enda kan t.ex. en Applet (bestående av många filer) laddas ner i en enda överföring vilket minskar nerladdningstiden. Att filformatet baseras på zip innebär att filerna som grupperas ihop kan komprimeras vilket gör att den totala filstorleken blir mindre (och nerladdningstiden minskar ännu mer).



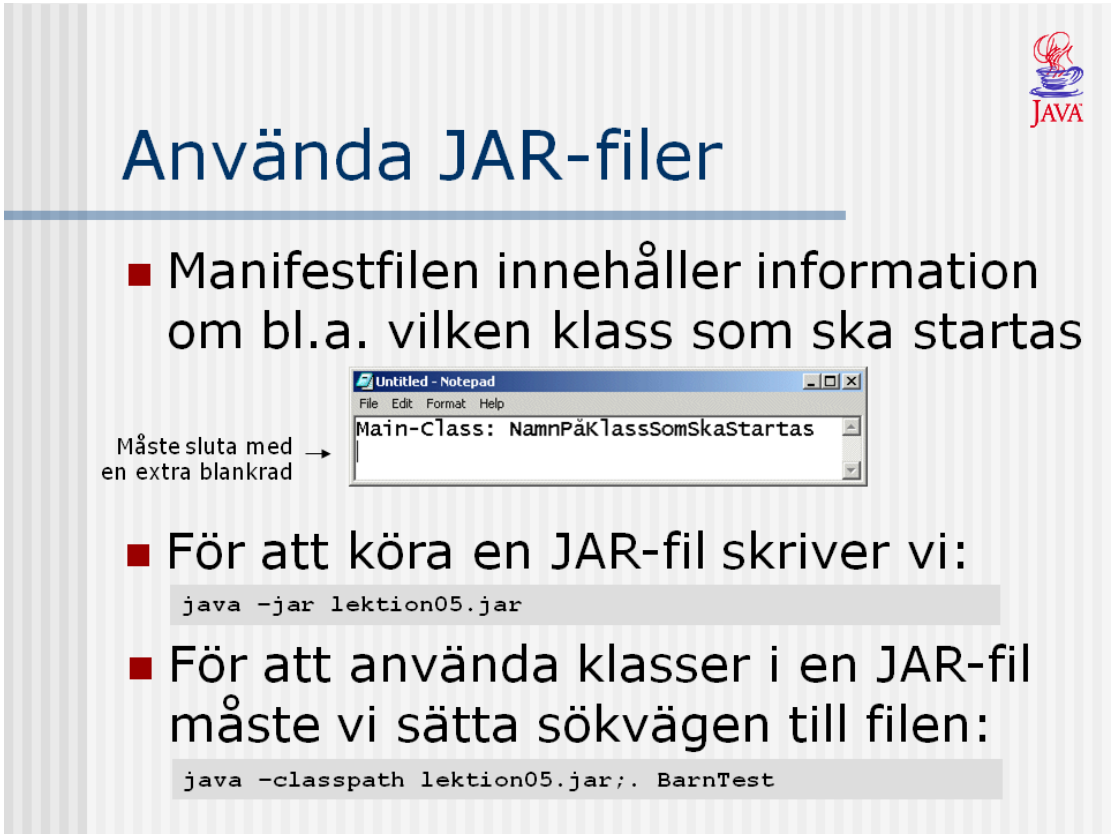
För att skapa en jar-fil använder vi verktyget **jar** som följer med i installationen av JDK. Jar används i ett kommandofönster och kräver att antal olika indata. Det är t.ex. namnet på den jar-fil som ska skapas och vilka filer som ska ingå i jar-filen (normalt endast .class och eventuella bilder/ljud/andra resurser som dessa behöver). Normalt krävs även en s.k. manisfest-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** ges nedan:

Syntax: **jar {ctxui}[vfm0Me] [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 ska ske till kommandofönstret över vad som händer, f 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 editor, t.ex. Anteckningar. För att arkivera filerna **Child2.java**, **Employee1.java**, **Person.java** och **InheritanceTest.java** i en jar-fil med namnet **lektion05.jar** gör vi på följande sätt:

Eftersom namnet på klassen som innehåller main-metoden i vår applikation är InheritanceTest1.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: InheritanceTest1
```

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: InheritanceTest1 den sista raden och tolkas därför inte när jar-filen skapas).

Vi skapar därefter jar-filen med namnet lektion05.jar genom att ge följande kommando i kommandofönstret:

```
jar cvmf manifest.txt lektion05.jar Person.class Child2.class Employee.class
InheritanceTest1.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 skriver vi följande i kommandofönstret:

```
java -jar lektion05.jar
```

Det enda som skiljer mot för att exekvera vanliga applikationer är att vi använder växeln –jar för kommandot java och därefter hela namnet på jar-filen. Tack vare att vi i manifestfilen angett vilken klass som är huvudklassen (innehåller main-metoden) kommer applikationen (InheritanceTest1) att starta.

Om vi vill använda filerna i en jar-fil i en annan applikation (t.ex. kunna skapa Child- och Person-objekt) måste vi inkludera sökvägen till jar-filen i något som heter classpath. Classpath är en eller flera sökvägar i vilken den virtuella maskinen letar efter klasser som ska användas. Används flera sökvägar separeras dessa med ett semikolon. Det är viktigt att komma ihåg att inkludera . (en punkt) när vi anger en classpath. Denna indikerar nämligen att klasser även ska letas i aktuell katalog.

Prova att öppna den skapade jar-filen med programmet 7-Zip (eller liknande) och utforska innehållet. Ta framför allt en titt på innehållet i manifest-filen.