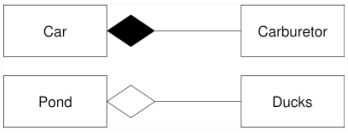
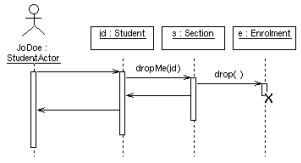
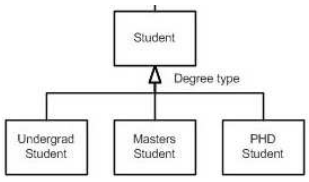
Föreläsning 4
Polymorfism
Dynamisk bindning
Inkapsling
Information hiding
Access-metoder och mutator-metoder

Grundbegreppen i objektorienterad design

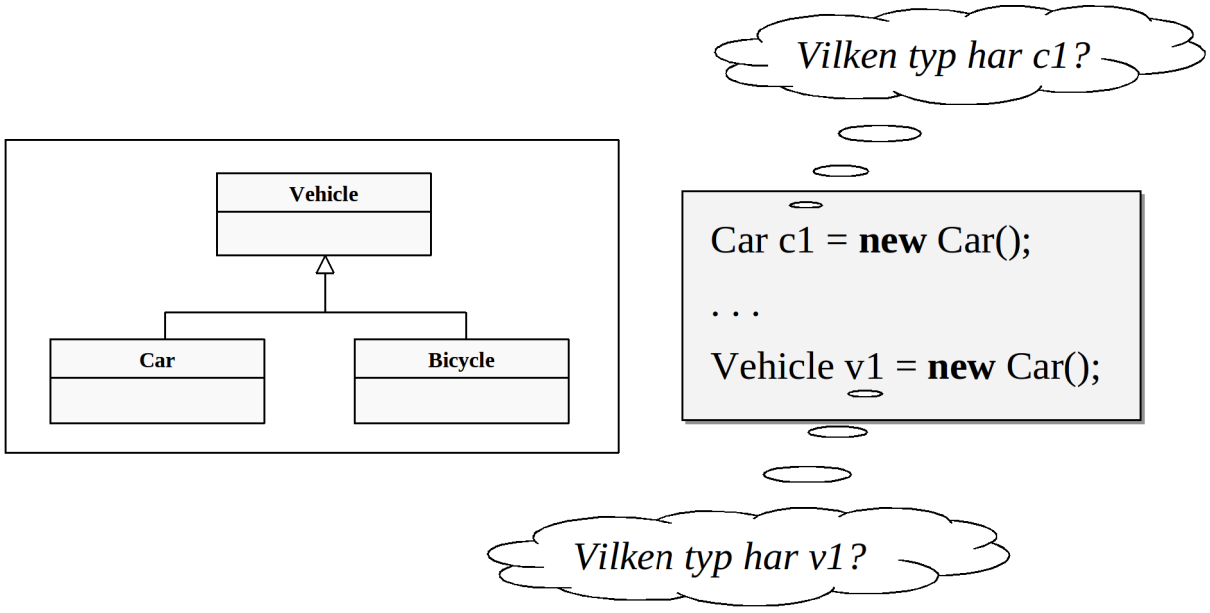
Encapsulation	Abstraction & Information Hiding
Composition	Nested Objects
Distribution of Responsibility	*Separation of concerns*
Message Passing	*Delegating responsibility*
Inheritance	*Conceptual hierarchy, polymorphism and reuse*







Statiska och dynamiska typer



Statiska och dynamiska typer

- Den deklarerade typen av en variabel är variabelns *statiska typ*.
- Typen på det objekt som en variabel refererar till är variabelns *dynamiska typ*.
- Typinformationen används vid två olika tillfällen – vid kompilering (*compile time*) och vid exekvering (*run time*).
- Det är kompilatorns uppgift att utföra statisk typkontroll. Den statiska typinformationen används vid själva kompileringen, efter kompileringen är den borta .
- Under exekvering används den dynamiska typinformationen.

Polymorfism

Java tillåter att man till en variabel av en viss typ C kan tilldela objekt som är av typ C eller är av en subtyp till C.

> *Egenskapen att ett objekt som är en subtyp till typen C legalt kan användas varhelst ett objekt av typen C förväntas kallas polymorfism.*
>
> *Polymorfism är ett av de viktigaste koncepten i objektorientering och är den grundläggande tekniken i objektorientering för att åstadkomma återanvändbar kod.*

Polymorfism kommer från grekiska *poly morf*, som betyder *många former*.

Polymorfism möjliggör att en användare som endast utnyttjar generella egenskaper av ett objekt inte behöver känna till exakt vilken typ objektet tillhör.

Betydelsen av polymorfism

När man diskuterar hur bra designen hos en mjukvara är, kommer ofta konceptet *plug-and-play* upp.

Idén med plug-and-play är att en komponent skall kunna pluggas in i ett system och kunna användas direkt, utan att omkonfigurera systemet.

Detta kan göras med hjälp av polymorfism genom att låta deklarerade variabler ha så vida typer som möjligt, dvs vara supertyper och inte subtyper.

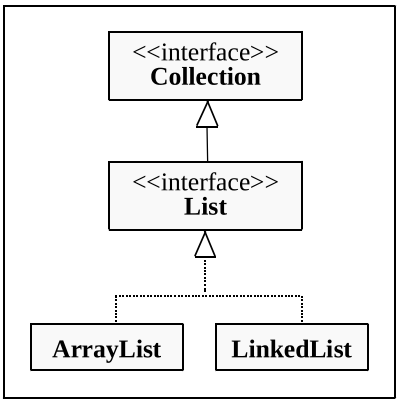
Betydelsen av polymorfism

Antag att vi på många ställen i ett programsystem skapar och använder objekt av den konkreta klassen LinkedList enligt:

```
private LinkedList list = new LinkedList();
. . .
public void someMethod(LinkedList list) {
   . . .
}//someMethod
```

En del av klasserna och interfacen i Javas Collection-Framework visas i bilden bredvid.

Kan vi med den kunskap som bilden ger, göra vårt programsystem mer flexibelt och mindre känsligt med avseende på förändringar?



Betydelsen av polymorfism

```
LinkedList list = new LinkedList();
```

Vidga typtillhörigheten

```
List list = new LinkedList();
```

Vidga typtillhörigheten

```
Collection list = new LinkedList();
```

Använd en factory-metod

```
Factory factory = new Factory();
Collection list = factory.createNewList();

public class Factory {
   . . .
   public Collection createNewList() {
      return new LinkedList();
   }//createNewList
}//Factory
```

list kan bara vara av typen LinkedList

list kan vara godtycklig subtyp till interfacetypen List

list kan vara godtycklig subtyp till interfacetypen Collection

list kan vara godtycklig subtyp till interfacetypen Collection.

Klassen där list deklareras skapar inget konkret objekt.

Factories (fabriker)

En *Factory* är en metod som tillverkar och returnerar objekt som är av en viss typ (d.v.s. är av typen själv eller subtyper till denna).

```
public class PersonFactory {
  public Person createPerson(String name, int age) {
    if (age < 13) {
      return new Child(name, age);
    } if (age < 20) {
       return new Teenager(name, age);
    }
    return new Adult(name, age);
  }//createPerson
}//PersonFactory
```

Fördelen med att använda en fabrik är att koden som använder fabriken inte påverkas om vi vill byta dynamisk. Eventuella förändringar koncentreras till fabriken. Exempelvis om vi inför en ny subklass `MiddleAged` till klassen `Person` i exemplet ovan.

Vi återkommer till fabriker när vi senare i kursen diskuterar *designmönster*.

Återanvändbar kod

Att en variabel kan deklareras som en supertyp (d.v.s. typen för ett interface eller en superklass) och referera till objekt som är subtyper till den deklarerade supertypen (d.v.s. tillhör någon klass som realiserar interfacet eller som är en subklass till superklassen), innebär att vi kan *minska beroendet av en specifik klass*.

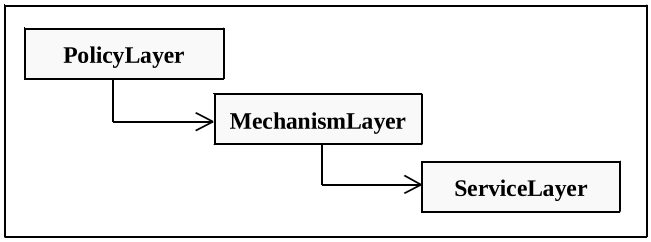
Detta ger oss möjlighet att skriva kod som är mycket mer flexibel och återanvändbar än vad som annars skulle vara fallet.

The Dependency Inversion Principle (DIP):

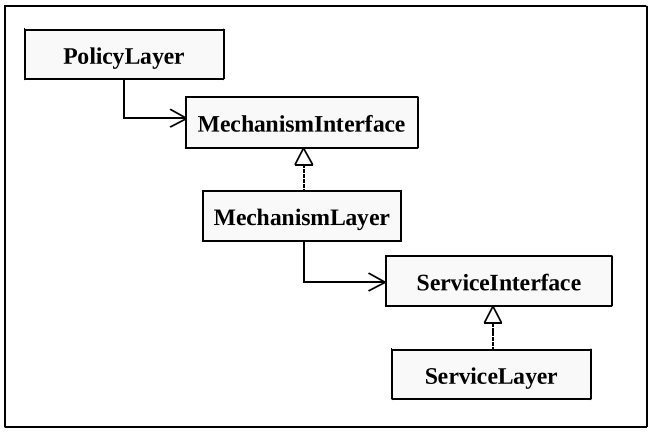
Depend upon abstractions, not upon concrete implementations.
Program against interfaces, not concrete classes.

The Dependency Inversion Principle (DIP):

Design som strider mot DIP.



Design som följer DIP.



Vad menas med dynamisk bindning

I objektorientering används generella namn på operationer/metoder som skall kunna appliceras på objekt av olika typer:

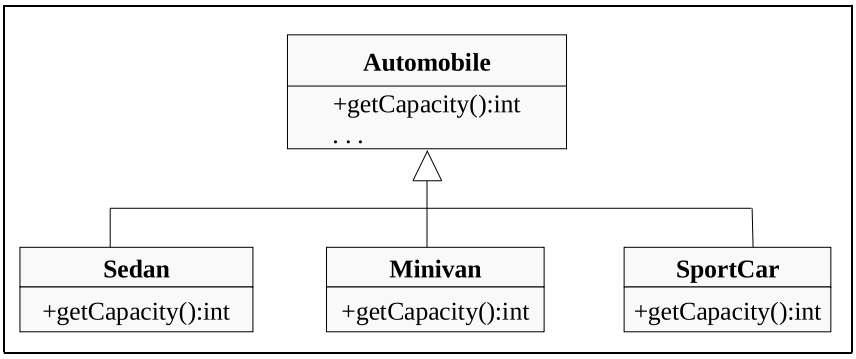
```
Vehicle v1 = new Car();
v1.turnLeft();
```

När valet av vilken operation/metod som skall anropas bestäms *vid exekveringen* kallas detta för *dynamisk bindning*, d.v.s. den dynamiska typen på objektet är avgörande för valet.

Dynamisk bindning skiljer sig ifrån *statisk bindning* då exakt vilken operation/metod som skall anropas bestäms redan *vid kompileringen*.

Dynamisk bindning

Antag att vi har en abstrakt superklass Automobile samt att denna har de tre konkreta subklasserna Sedan, Minivan och SportCar.



Klassen Automobile har den abstrakta metoden getCapacity() som överskuggas (*overrides*) av de tre subklasserna.
Metoden getCapacity() returnerar hur många passagerare som ryms i respektive biltyp.

Dynamisk bindning

Eftersom Sedan, Minivan och SportCar är subtyper till Automobile är följande kod legal:

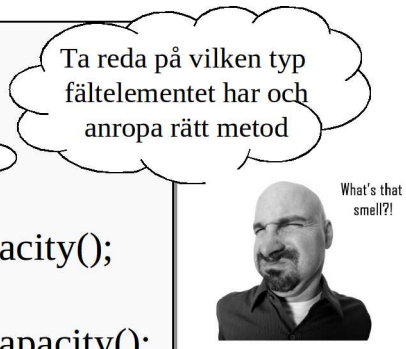
```
Automobile[] fleet = new Automobile[3];
fleet[0] = new Sedan();
fleet[1] = new Minivan();
fleet[2] = new SportCar();
```

Fältet fleet innehåller således element från tre olika klasser.

Antag nu att vi vill ta reda på den totala passagerarkapaciteten för objekten som finns i fältet fleet.

Lösning utan nyttjande av dynamisk bindning

```
//--- VERSION 1: Bad code --//
int totalCapacity = 0;
for (int i = 0; i < fleet.length; i++) {
   if (fleet[i] instanceof Sedan)
      totalCapacity += ((Sedan) fleet[i]).getCapacity();
   else if (fleet[i] instanceof Minivan)
      totalCapacity += ((Minivan) fleet[i]).getCapacity();
   else if (fleet[i] instanceof SportCar)
      totalCapacity += ((SportCar)fleet[i]).getCapacity();
}
```



Koden fungerar, men är den bra?

Lösning som använder dynamisk bindning

Genom att dra nytta av dynamisk bindning kan vi få en elegant lösning.

```
//--- VERSION 2: Elegant code --//
int totalCapacity = 0;
for (int i = 0; i < fleet.length; i++)
   totalCapacity += fleet[i].getCapacity();
```

Javas runtime system (JVM) tittar efter vilken dynamisk typ ett objekt har när programmet exekveras och inte på vilken deklarerad typ det har.

Detta innebär att objektet som refereras av `fleet[0]` vid exekveringen har typen `Sedan` och inte typen `Automobile` som det har som deklarerad typ. Således anropas metoden `getCapacity()` i klassen `Sedan` för elementet `fleet[0]`, metoden `getCapacity()` i klassen `Minivan` för elementet `fleet[1]` och metoden `getCapacity()` i klassen `SportCar` för elementet `fleet[2]`.

Dynamisk bindning

Koden i VERSION 2 är mycket elegantare och återanvändbar än koden i VERSION 1:

- koden är kortare, enklare och mer överskådlig.
- viktigast är dock att koden inte behöver förändras om man senare behöver lägga till en ny subklass till `Automobile`, t.ex. subklassen `SUV`.

Koden i VERSION 2 uppfyller *The Open-Closed Principle*, vilken är en av de viktigaste principerna i OOP.

> ***The Open-Closed Principle (OCP):***
>
> *Software modules should be both open for extension and closed for modification. (Bertrand Meyer)*

OPC är vårt mål och DIP den viktigaste mekanismen för att ta sig dit.

Hur fungerar dynamisk bindning

Vi har två delar som är inblandade när det gäller anropet av `fleet[i].getCapacity()`.

- Den första delen sker under kompileringen. När kompilatorn ser att fleet[i] är deklarerad av typen `Automobile`, söker kompilatorn efter metoden `getCapacity()` i klassen `Automobile`.
- Det andra steget sker under exekveringen när den korrekta implementation av metoden `getCapacity()` skall väljas.

Detta innebär att metoden `getCapacity()` måste vara deklarerad både i klassen `Automobile` och i alla subklasser till `Automobile` för att det hela skall fungera.

Det är alltså överskuggningen av metoden `getCapacity()` som är hela hemligheten.

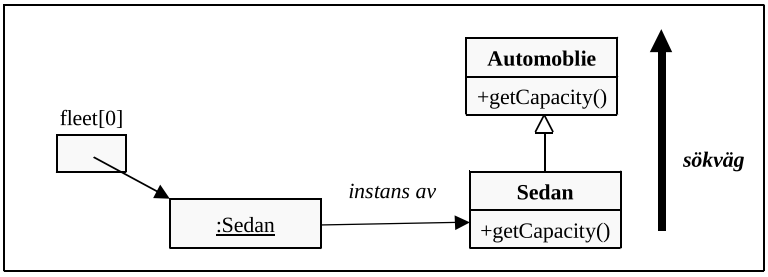
> *Vad hade hänt om metoden `getCapacity()` inte funnits i klassen `Automobile`?*

Hur fungerar dynamisk bindning

Vid exekveringen skall i vårt exempel metoden `getCapacity()` letas upp.

Sökningen påbörjas i den klass till vilken *objektet som skickar anropet* tillhör, d.v.s. den *dynamiska typen* på referensvariabeln. Har denna klass ingen sådan metod genomsöks kedjan av klassens superklasser tills metoden påträffas.

För anropet `fleet[0].getCapacity()` påbörjas sökningen i klassen `Sedan`, eftersom objektet `fleet[0]` är ett objekt av klassen `Sedan`. Metoden `getCapacity()` finns i klassen `Sedan`, således är det denna metod som kommer att exekveras.



Överskuggning och kovarianta returtyper

I Java är det tillåtet att returtypen för en överskuggande metod är en subtyp till returtypen för metoden som överskuggas.

Detta kallas för *kovarians*.

Fördelen med kovarians är typsäkerheten

- mindre behov av osäker typomvandling när överskuggade metoder anropas.

Överskuggning och kovarianta returtyper

```
public class A {}
public class B extends A {}

public class Base {
   public A f() { return new A(); }
   public Base g() { return new Base(); }
}


public class Sub extends Base {
   public A f() { return new A(); }
   public A f() { return new B(); }
   public B f() { return new B(); }

   // In particular, covariance can be
   // applied to the class itself:
   public Base g() { return new Base(); }
   public Base g() { return new Sub(); }
   public Sub g() { return new Sub(); }
}
```

Samtliga dessa är en korrekt överskuggning av f

Samtliga dessa är en korrekt överskuggning av g

Överskuggning och kovarianta returtyper

Exempel: Ingen kovariant

```
public class Base {
   . . .
   public Base aMethod() {
      . . .
   }//aMethod
}//Base
```

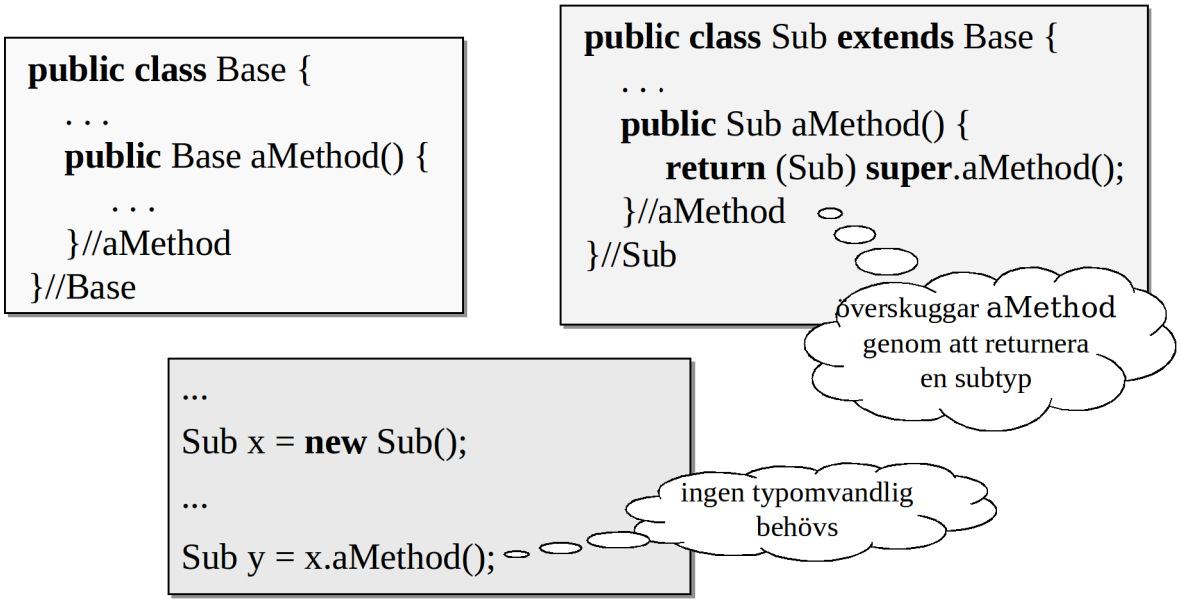
```
public class Sub extends Base {
   . . .
   public Base aMethod() {
      return super.aMethod();
   }//aMethod
}//Sub
```

```
...
Sub x = new Sub();
...
Sub y = (Sub) x.aMethod();
```

explicit typomvandling behövs

Överskuggning och kovarianta returtyper

Exempel: Användning av kovariant

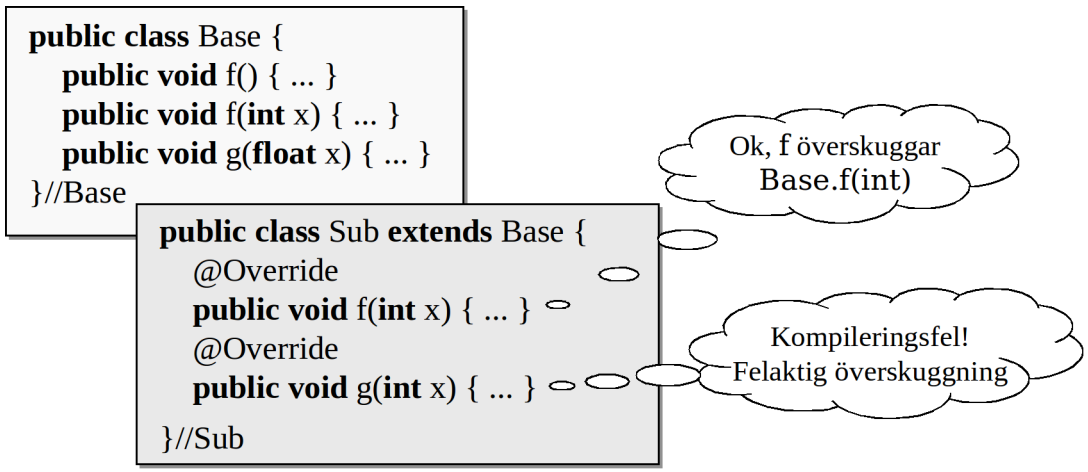


@Override-annotation

@Override-annotationen anger till kompilatorn att metoden har för avsikt att överskugga en metod i superklassen.

Om en metod är annoterad med @Override, men metoden inte överskuggar en metod i superklassen, uppstår ett kompileringsfel.

Exempel:



Överlagring kontra överskuggning

Om det finns två metoder i samma klass som har samma namn, men med olika signaturer, innebär detta att namnet på metoderna är överlagrade (*overloaded*).

Signaturen för en metod bestäms av vilka typer som parametrarna i metodens parameterlista har (namnen på parametrarna är irrelevant). Två signaturer är lika om *antal, typ och ordning* på parametrarna är identiska.

Överlagring betyder alltså att ett metodnamn används av två eller flera helt skilda metoder i *samma* klass.

Överskuggning (*overriding*) däremot involverar en *superklass* och en *subklass*, och berör två metoder som har samma metodsignatur – den ena metoden finns lokaliserad i superklassen och den andra metoden finns lokaliserad i subklassen.

Överlagring

I klassen `String` finns metoderna:

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```

Dessa metoder är överlagrade - de har samma namn men olika signaturer.

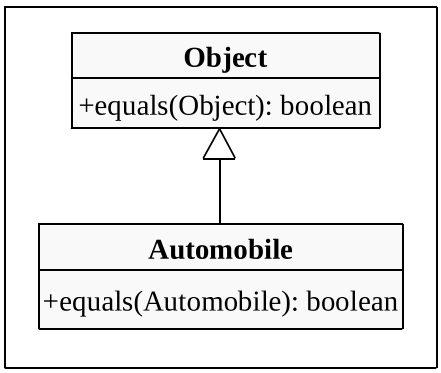
Naturligtvis skall överlagring användas om och endast om de överlagrade metoderna i allt väsentligt gör samma sak (som med metoderna `substring` i klassen `String` ovan).

När parametrarna i de överlagrade metoderna är primitiva datatyper, som `substring`-metoderna ovan, är det enkelt att förstå vilken metod som blir anropad.

Det är svårare att förstå vilken metod som verkligen blir anropad när parametrarna utgörs av referenstyper och därför kan anropas med subtyper till de typer som anges i parameterlistan.

Överlagring – ett exempel

Låt oss titta på klassen Automobile igen. Eftersom alla klasser i Java är subklasser till klassen Object betyder detta att klassen Automobile bl.a. ärver en metod med följande metodhuvud:



public boolean equals(Object obj) (1)

Antag nu att vi lägger till en metod i klassen Automobile som har metodhuvudet:

public boolean equals(Automobile auto) (2)

Observera att parametern till metoden equals i klassen Automobile är Automobile, medan parametern till metoden equals i klassen Object är Object. Vi har alltså överlagring (och inte överskuggning).

Överlagring – ett exempel

Klassen Automobile, har nu två metoder med namnet equals - en deklarerad lokalt i klassen och en ärvd från Object. Dessa båda metoder är alltså överlagrade.

Betrakta nedanstående kodavsnitt:

```
Object o = new Object();
Automobile auto = new Automobile();
Object autoObject = new Automobile();
auto.equals(o);
auto.equals(auto);
auto.equals(autoObject);
```

Kan du för vart och ett av de tre anropen av equals i koden ovan tala om ifall det är den lokala metoden i klassen Automobile som kommer att exekveras eller om det är den ärvda metoden från Object?

Överlagring – hur fungerar det

När kompilatorn ser ett anrop t.ex. `auto.equals(o)` händer följande:

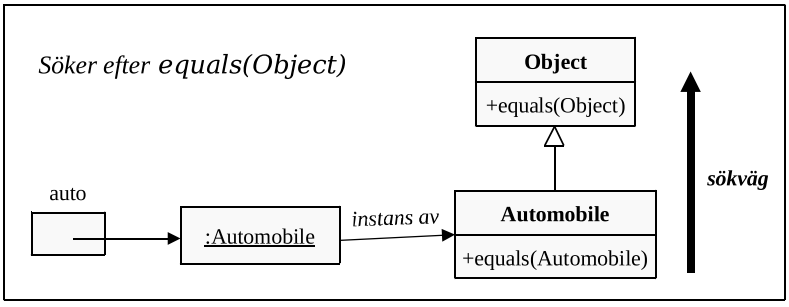
1. Kompilatorn tar reda på vilken deklarerad typ `auto` har.
2. Kompilatorn tittar i den aktuella i klassen (eller interfacet) och i dess superklasser (eller superinterface) efter alla metoder med namnet `equals`.
3. Kompilatorn tittar på parameterlistorna för dessa metoder och väljer den parameterlista vars typer *bäst överensstämmer* med de deklarerade typerna som skickas i det aktuella anropet.

I anropet `auto.equals(o)`, är `auto` av typen `Automobile` och `o` av typen `Object`. Parametertypen för den lokala `equals`-metoden i klassen är `Automobile` och är således ”to narrow”, eftersom denna är av typen `Automobile`. Således väljs `equals`-metoden i klassen `Object`, som har en parameter av typen `Object` (d.vs. `equals(Object)`).

Överlagring – hur fungerar det

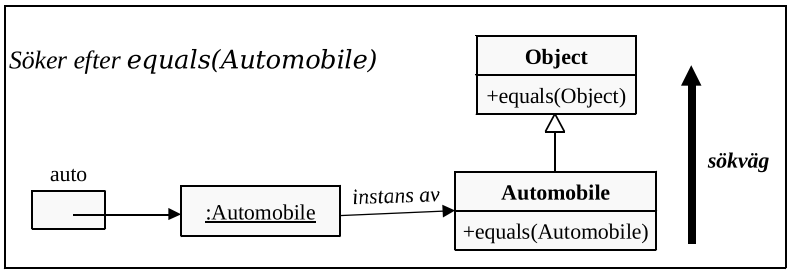
Vid runtime skall metoden med den signatur som kompilatorn valt letas upp. Sökningen startar i den klass till vilket objektet som skickar anropet tillhör. Har denna klass ingen sådan metod genomsöks kedjan av klassens superklasser tills metoden påträffas. (Den kommer att påträffas, annars hade det inträffat ett exekveringsfel.)

För anropet `auto.equals(o)` påbörjas sökningen i klassen `Automobile` och vi letar efter `equals(Object)`-metoden som påträffas i klassen `Object`, således är det denna metod som kommer att exekveras.



Överlagring – hur fungerar det

För anropet `auto.equals(auto)` kommer kompilatorn att välja metoden `equals(Automobile)`. Vid exekveringen startar sökningen i klassen Automobile, där metoden påträffas. Allstå kommer `equals`-metoden i klassen `Automobile` att exekveras.



För anropet `auto.equals(autoObject)` är parametern `autoObject` deklarerad som `Object` och metoden `equals` i klassen `Object` väljs (pga av samma orsaker som var fall för anropet `auto.equals(o)` ovan).

Fråga:

Vilken metod väljs i nedanstående anrop

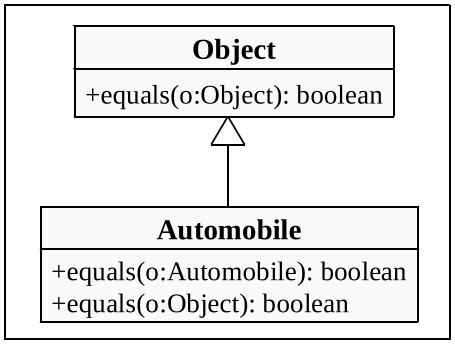
o.equals(o)
o.equals(auto)
o.equals(autoObject)
autoObject.equals(o)
autoObject.equals(auto)
autoObject.equals(autoObject)

Överskuggning

Låt oss nu införa metoden

public boolean equals(Object o) (3)

i klassen Automobile.

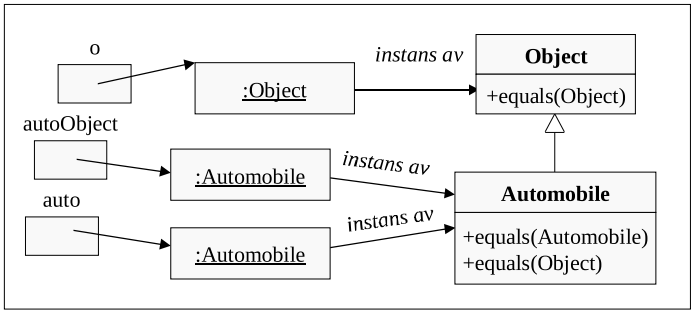


Detta innebär att klassen Automobile överskuggar metoden som ärvs från klassen Object.

Överskuggning

Vad inträffar nu för de nio olika anropen vi diskuterade tidigare?

```
Object o = new Object();
Automobile auto = new Automobile();
Object autoObject = new Automobile();
auto.equals(o);
auto.equals(auto);
auto.equals(autoObject);
o.equals(o);
o.equals(auto);
o.equals(autoObject);
autoObject.equals(o);
autoObject.equals(auto);
autoObject.equals(autoObject);
```



Överskuggning

Vad inträffar nu för de nio olika anropen vi diskuterade tidigare?

```
Object o = new Object();
Automobile auto = new Automobile();
Object autoObject = new Automobile();
auto.equals(o);                    (3)
auto.equals(auto);                 (2)
auto.equals(autoObject);           (3)
o.equals(o);                       (1)
o.equals(auto);                    (1)
o.equals(autoObject);              (1)
autoObject.equals(o);              (3)
autoObject.equals(auto);           (3)
autoObject.equals(autoObject);     (3)
```

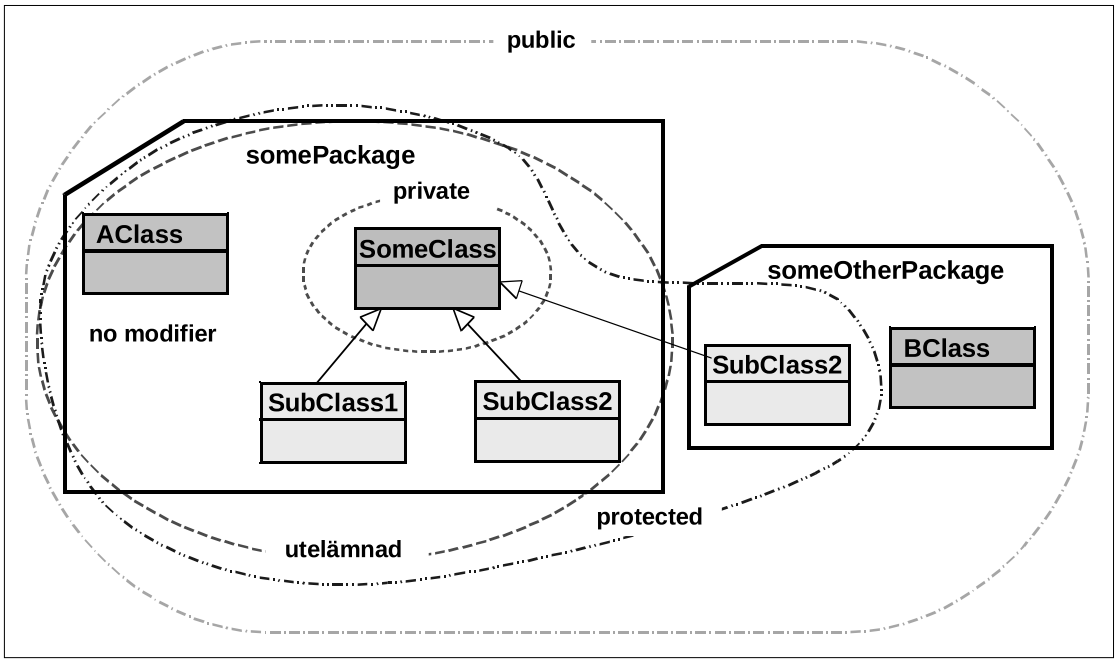
Åtkomstmodifierare

När man säger att en subklass ärver all data och alla metoder från sin superklass, betyder nödvändigtvis inte detta att subklassen direkt kan få access till all data och alla metoder i superklassen.

I Java finns fyra olika synlighetsmodifierare:

public	tillåter access för alla andra klasser.
protected	tillåter access för alla andra klasser i samma paket och för subklasser i andra paket.
utelämnad	tillåter access för alla andra klasser i samma paket.
private	tillåter access endast inom klassen själv.

Åtkomstnivåer



Överskuggning och åtkomstnivåer

När en subklass överskuggar en metod måste metoden i subklassen vara åtminstone lika synlig som den överskuggade metoden i superklassen.

Åtkomst i superklassen	Tillåten åtkomst i subklassen
public	**public**
protected	**protected** **public**
package private	package private **protected** **public**

Åtkomstmodifierare

När skall man använda **protected**- eller paketåtkomst?

En bra tumregel är aldrig, eftersom den interna implementationen exponeras för subklasser respektive klasser i paketet.

Ett undantag är när man har ett paket som utgör en integrerad komponent, då kan det ibland på grund av effektivitetsorsaker vara motiverat. Dock kan detta innebära att en förändring i en klass kräver förändringar i andra klasser i paketet, men förändringarna är åtminstone begränsade till paketet.

Inkapsling och informationsdöljande

Begreppet inkapsling (*encapsulation*) betyder att man sammanför data och de operationer som kan utföras på denna data till en enhet. I objekt-orientering realiseras en sådan inkapslad enhet som en klass, paket eller subsystem.

Begreppet informationsdöljande (*information hiding*) innebär att man endast tillhandahåller den information som en användare behöver veta.

Inkapsling skall används på så sätt att all information som en användare inte behöver skall hållas dold för användaren. Informationsutbyte med användaren skall endast kunna ske via ett väldefinierat gränssnitt.

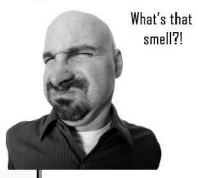
Keep it secret! Keep it safe!

Don't let anyone else play with you. (Joseph Pelrine)

Hemlighåll information

Antag att vi har en klass som använder publika instansvariabler

```
//-- Bad code --//
public class Day {
   public int year;      //tillgänglig information
   public int month;   //tillgänglig information
   public int date;      //tillgänglig information
   public Day(int aYear, int aMonth, int aDate) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
   . . .
}//Day
```



En programmerare som använder klassen Day har i ett program skrivit koden

```
Day d = new Day(2009, 10, 30);
. . .
int month = d.month;
```

Vad händer som man bestämmer sig för att byta implementation i klassen Day, t.ex. att avbilda en dag som ett Julianskt datum?

Hemlighåll information

Ny implementation av klassen Day:

```
//-- Bad code --//
public class Day {
   public int julian;     //tillgänglig information
   public Day(int aYear, int aMonth, int aDate) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
   . . .
}//Day
```

Oförändrad kod i användarprogrammet

```
Day d = new Day(2009, 10, 30);
. . .
int month = d.month;
```

kommer nu att resultera i ett fel! Komponenten d.month finns inte längre.

Hemlighåll information

Om instansvariablerna i klassen Day är inkapslade

```
public class Day {
   private int year;
   private int month;
   private int date;
   public Day(int aYear, int aMonth, int aDate) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
   . . .
}//Day
```

måste koden i föregående exempel skrivas

```
Day d = new Day(2009, 10, 30);
. . .
int month = d.getMonth();
```

Detta innebär att det är möjligt att byta implementation i klassen Day utan att program som använder klassen behöver förändras.

Information Hiding Principle

En moduls interna struktur skall skyddas mot extern åtkomst

- enklare att hålla systemet i ett konsistent/legalt tillstånd
- implementationen av modulen kan ändras utan att det påverkar klienterna.

Varje modul skall ha ett väldefinierat publikt gränssnitt genom vilket all extern åtkomst måste ske:

- informationen som en klient behöver för att använda klassen begränsas till ett minimum.
- klienterna skall bara veta hur modulen används, inte veta hur modulen är implementerad.

Slutsats: Gör alla instansvariabler privata!

Access-metoder och Mutator-metoder

Vi gör en konceptuell skillnad mellan

- mutator-metoder (*mutator methods*): metoder som förändrar tillståndet i ett objekt. Kallas också för setter-metoder
- access-metoder (*accessor methods*): metoder som returnerar värdet av en instansvariabel. Kallas även för getter-metoder.

En klass som innehåller mutator-metoder är en muterbar (*mutable*) klass. Objekt som tillhör en muterbar klass kan ändra sitt tillstånd.

En klass som är en icke-muterbar (*immutable*) kan inte ändra sitt tillstånd. En icke-muterbar klass har inga mutator-metoder. Ett objekt som tillhör en icke-muterbar klass bibehåller under hela sin livstid det tillstånd som objektet får när det skapas.

Klassen String är ett exempel på en icke-muterbar klass.

Access-metoder och Mutator-metoder

```
public class Day {
   private int year;
   private int month;
   private int date;
   public Day(int aYear, int aMonth, int aDate) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
   . . .
}//Day
```

Klassen Day är icke-muterbar. Skall vi lägga till metoderna

```
public void setYear(int aYear)
public void setMonth(int aMonth)
public void setDate(int aDate)
```

i klassen Day?

Access-metoder och Mutator-metoder

Antag att klassen Day har mutator-metoderna

```
public void setYear(int aYear)
public void setMonth(int aMonth)
public void setDate(int aDate)
```

och man i ett användarprogram ger koden

```
Day firstdDeadline = new Day(2010,1, 1);
firstDeadline.setMonth(2);
Day secondDeadline = new Day(2010,1, 31);
secondDeadline.setMonth(2);
```

Vad händer?

Objektet secondDeadline hamnar i ett ogiltigt tillstånd, eftersom 31/2 inte är ett giltigt datum!

Slutsats: Tillhandahåll inte slentrianmässigt set-metoder för alla instansvariabler!