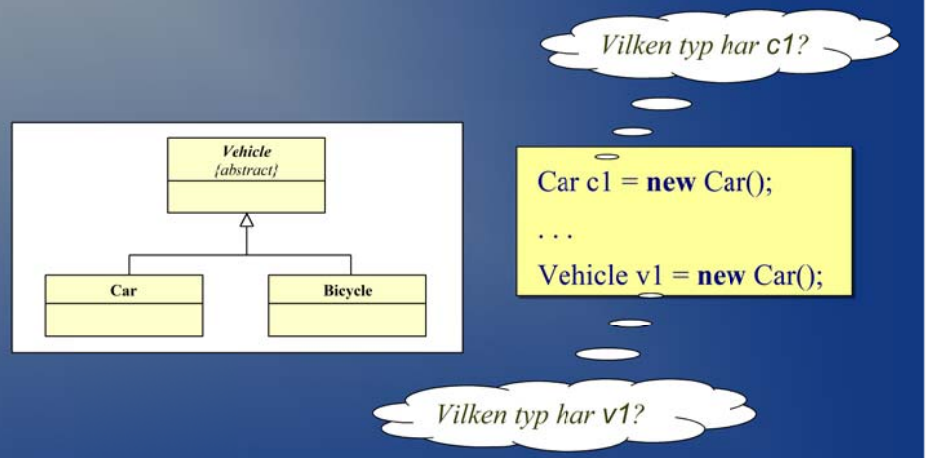
Föreläsning 4

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

Statiska och dynamiska typer



Statiska och dynamiska typer

- Den deklarerade typen hos 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 endast vid själva kompileringen, efter kompileringen är den borta.
- Under exekvering används den dynamiska typinformationen. Runtime-systemet gör *dynamisk typkontroll*.

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 där 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* hos ett objekt (=egenskaper som definieras i superklassen) 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.

Dependency Inversion Principle (DIP):

Depend on abstractions, not on concrete implementations.

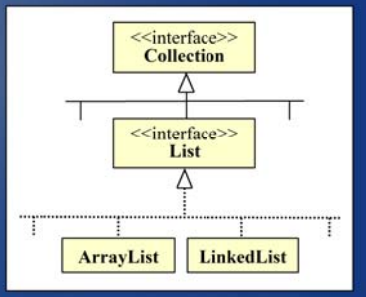
Betydelsen av polymorfism

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) {
   . . .
}
```

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();
```
list kan bara vara av typen LinkedList

Vidga typtillhörigheten

```
List list = new LinkedList();
```
list kan vara av godtycklig subtyp till interfacetypen List

Vidga typtillhörigheten

```
Collection list = new LinkedList();
```
list kan vara av godtycklig subtyp till interfacetypen Collection

Använd en factory-metod

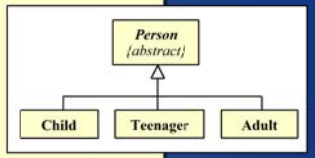
```
Factory factory = new Factory();
Collection list = factory.createNewList();
```
list kan vara av godtycklig subtyp till interfacetypen Collection.
I koden där list deklareras skapas objektet *utan kunskap om dess konkreta typ*.

```
public class Factory {
   . . .
   public Collection createNewList() {
      return new LinkedList();
   }
}
```
Objektet skapas av en *fabriksmetod*. Den konkreta subtypen nämns bara på ett ställe. Lätt att ändra!

Factories (fabriker)

En *Factory* är ett objekt 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);
   }
}
```



Fördelen med att använda en fabrik är att koden som använder fabriken inte påverkas om nya subtyper tillkommer. 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 med 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 specifika klasser.*

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

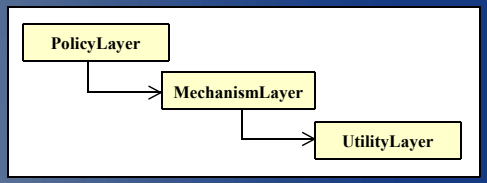
The Dependency Inversion Principle (DIP):

Depend on abstractions, not on concrete implementations.

Program against interfaces, not against concrete classes.

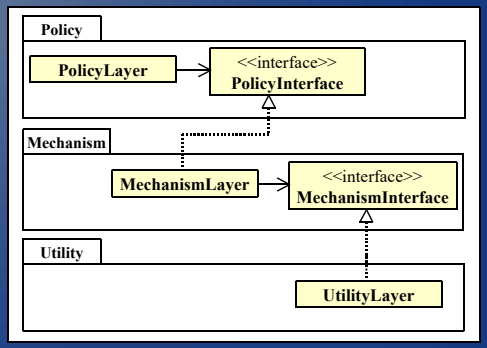
The Dependency Inversion Principle (DIP):

Design som strider mot DIP.



Design som följer DIP.

De konkreta klasserna kan under exekveringen bytas ut mot andra konkreta klasser som implementerar respektive gränssnitt.



Vad menas med dynamisk bindning?

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

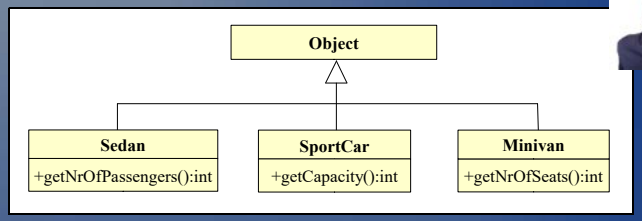
```
Vehicle v1 = new Car();
v1.turnLeft();
```

Vid *statisk bindning* bestäms vilken operation/metod som skall anropas *vid kompileringen*, d.v.s. den *statiska typen* (deklarationstypen) hos variabeln är avgörande för valet.

Vid *dynamisk bindning* bestäms vilken operation/metod som skall anropas *vid exekveringen*, d.v.s. variabelns *dynamiska typ* (det utpekade objektets typ) är avgörande för valet.

Avsaknad av dynamisk bindning

Antag att vi har de tre klasserna Sedan, Minivan och SportCar för att avbilda tre olika typer av bilar:



Klasserna Sedan, Minivan och SportCar har bl.a. metoder som returnerar hur många passagerare som ryms i respektive biltyp: getNrOfPassengers(), getCapacity() respektive getNrOfSeats().

Avsaknad av dynamisk bindning

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


```
Object[] fleet = new Object[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.

Avsaknad av dynamisk bindning

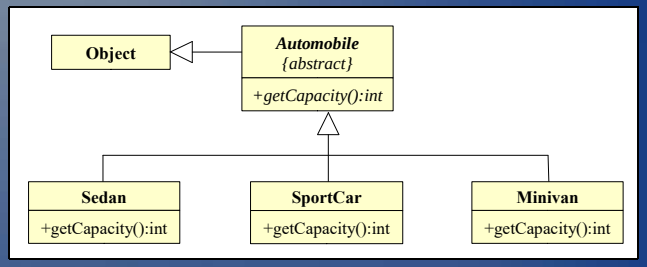
```
//--- VERSION 1: Bad code --//
int totalCapacity = 0;
for (int i = 0; i < fleet.length; i++) {
   if (fleet[i] instanceof Sedan)
      totalCapacity += ((Sedan) fleet[i]).getNrOfPassengers();
   else if (fleet[i] instanceof Minivan)
      totalCapacity += ((Minivan) fleet[i]).getCapacity();
   else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

Ta reda på vilken typ fältelementet har och anropa rätt metod

Koden fungerar, men är den bra?

Lösning med användning av dynamisk bindning

Refaktorera genom generalisering, d.v.s. inför en abstrakt superklass Automobile som innehåller de gemensamma delarna för subklasserna Sedan, Minivan och SportCar.



Klassen Automobile har den abstrakta metoden getCapacity() som returnerar antalet passagerare. Metoden getCapacity() överskuggas (*overrides*) i de tre subklasserna.

Lösning med användning av 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 som använder dynamisk bindning

Eftersom metoden getCapacity() är definierad i klassen Automobile kan dynamisk bindning användas för att få en elegant lösning:

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

dynamisk bindning

När programmet exekveras tittar Javas runtime system (JVM) efter vilken typ ett objekt har.

Detta innebär att **objektet** som refereras av fleet[0] vid exekveringen har typen Sedan och inte typen Automobile som är **referensens** deklarerade 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 än koden i VERSION 1:

- koden är kortare, enklare och mer överskådlig.
- koden uppfyller *The Open-Closed Principle*, man kan lägga till nya subklasser till Automobile, t.ex. SUV, utan att förändra i koden.

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

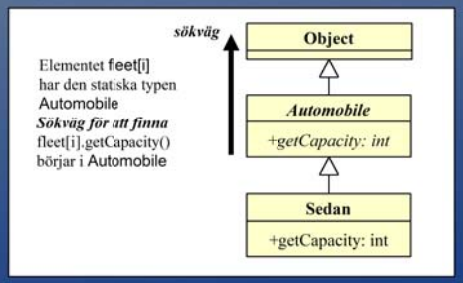
OCP är vårt mål och DIP den viktigaste mekanismen för att ta sig dit.

Hur fungerar dynamisk bindning?

Två steg som är inblandade när det gäller anropet av fleet[i].getCapacity():

- Det första steget, *typkontroll*, sker *under kompileringen*. Typkontrollen garanterar att en metod med rätt signatur existerar och kan anropas vid exekveringen.
- Det andra steget sker *under exekveringen,* då rätt metod väljs utifrån objektets typ.

När kompilatorn ser att fleet[i] är deklarerad av typen Automobile, börjar kompilatorn att söka efter metoden getCapacity() i klassen Automobile.



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

Hur fungerar dynamisk bindning?

Under exekveringen skall den korrekta/avsedda implementation av metoden getCapacity() väljas.

I vårt exempel är metoden getCapacity() överskuggad i samtliga subklasser till Automobile.

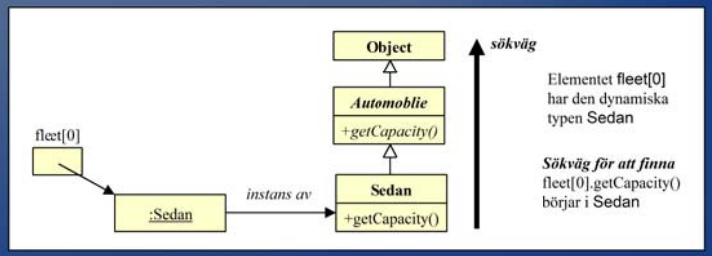
Det är denna överskuggning av metoden getCapacity() som är hela hemligheten till att vår lösning i VERSION 2 fungerar.

Vad hade hänt om någon subklass inte överskuggat metoden getCapacity()?

Hur fungerar dynamisk bindning?

Vid exekveringen påbörjas sökningen efter metoden getCapacity() i den klass till vilken *det anropande objektet* hör, d.v.s. den *dynamiska typen* hos referensvariabeln. Har denna klass ingen sådan metod genomsöks kedjan av klassens superklasser tills metoden påträffas. Metoden kommer med säkerhet att påträffas, annars hade kompileringen misslyckats. (Varför?)

För anropet fleet[0].getCapacity() påbörjas sökningen i klassen Sedan, eftersom objektet som fleet[0] refererar till 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

Java tillåter att returtypen för en överskuggande metod är en subtyp till returtypen för metoden som överskuggas.

Detta kallas *kovarians*.

Fördelen med kovarians är ökad typsäkerhet

- mindre behov av osäkra typomvandlingar 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 kovarians

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

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

Klient

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

explicit typomvandling behövs

Överskuggning och kovarianta returtyper

Exempel: Användning av kovarians

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

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

överskuggar aMethod genom att returnera en subtyp

Klient

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

ingen typomvandlig behövs

@Override-annotation

@Override-annotationen upplyser kompilatorn om att en metod har för avsikt att överskugga en metod i en superklass.

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

Exempel:

```
public class Base {
    public void f() { ... }
    public void f(int x) { ... }
    public void g(float x) { ... }
}
```

```
public class Sub extends Base {
    @Override
    public void f(int x) { ... }
    @Override
    public void g(int x) { ... }
}
```

Ok, f överskuggar Base.f(int)

Kompileringsfel! Felaktig överskuggning

Överlagring, överskuggning och hiding

Signaturen för en metod bestäms av vilka typer som parametrarna i metodens parameterlista har. Returtypen ingår *inte* i signaturen. Två signaturer är lika om *antal, typ och ordning* på parametrarna är identiska.

Om två metoder i samma klass har samma namn, men olika signaturer, är namnet på metoderna *överlagrat* (*overloaded*).

Överlagring (*overloading*) betyder alltså att ett metodnamn namnger två eller flera helt skilda metoder i *samma klass*.

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

Om en superklass och en subklass båda definierar en *klassmetod* med samma namn och signatur, kallas detta för *hiding*.

Överlagring

Exemel: 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 substring ovan).

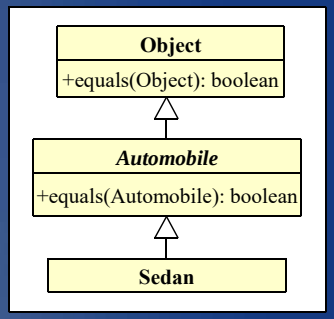
När parametrarna i de överlagrade metoderna är primitiva datatyper, som i substring-metoderna ovan, är det enkelt att förstå vilken metod som blir anropad.

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 klasserna Automobile och Sedan igen. Eftersom alla klasser i Java är subklasser till Object ärver Automobile bl.a. 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)

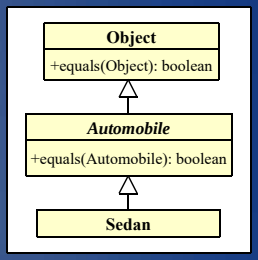
Observera att parametern till equals i Automobile har typen Automobile, medan parametern till equals i Object har typen Object.

Överlagring – ett exempel

Klasserna Automobile och Sedan har nu två metoder med namnet equals - en deklarerad i klassen Automobile och en deklarerad i klassen Object. Dessa båda metoder är alltså överlagrade (inte överskuggade).

Betrakta nedanstående kodavsnitt:

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



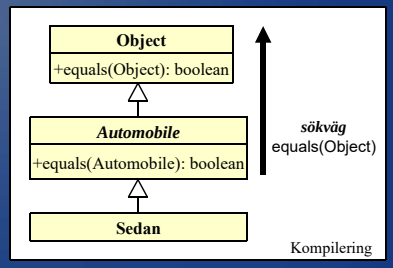
Kan du för vart och ett av de tre anropen av equals ovan avgöra om metoden i Automobile exekveras eller den ärvda metoden från Object?

Överlagring – hur det fungerar vid kompilering

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 (d.v.s. statiska) typerna som skickas i det aktuella anropet.

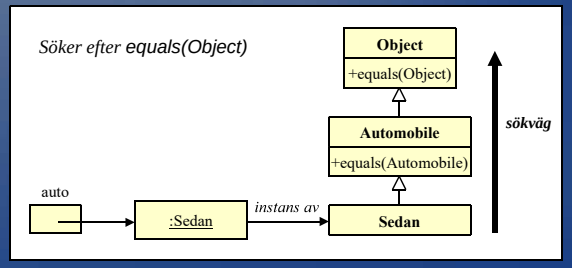
I anropet auto.equals(o), är auto av typen Automobile och o av typen Object. Parametertypen för equals-metoden i Automobile är ”to narrow”, eftersom denna är av typen Automobile. Således väljs equals(Object) i klassen Object.



Överlagring – hur det fungerar vid exekvering

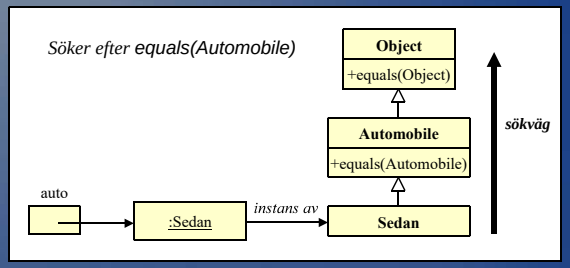
Vid runtime skall metoden med den signatur som kompilatorn valt letas upp. Sökningen startar i det anropande objektets klass. Har denna klass ingen sådan metod genomsöks kedjan av klassens superklasser tills metoden påträffas. (Den kommer att påträffas, annars hade det inträffat ett kompileringsfel.)

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



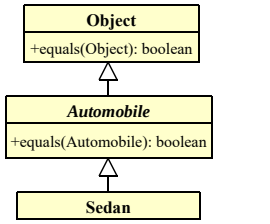
Överlagring – hur det fungerar vid exekvering

För anropet auto.equals(auto) kommer kompilatorn att välja metoden equals(Automobile). Vid exekveringen startar sökningen i Sedan. Metoden påträffas i Automobile och det är denna equals-metod som kommer att exekveras.



För anropet auto.equals(autoObject) är parametern autoObject deklarerad som Object och equals i Object väljs (pga av samma orsaker som var fallet för anropet auto.equals(o)).

Övning 1:



Object o = **new** Object();
Automobile auto = **new** Sedan();
Object autoObject = **new** Sedan();

Vilken metod väljs i nedanstående anrop:

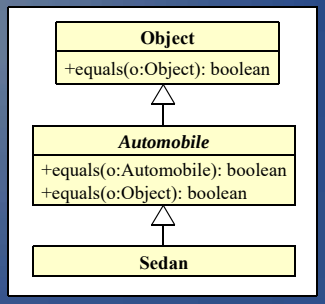
- o.equals(o)
- o.equals(auto)
- o.equals(autoObject)
- autoObject.equals(o)
- autoObject.equals(auto)
- autoObject.equals(autoObject)

Övning 2: Överskuggning

Låt oss införa metoden

public boolean equals(Object o) (3)

i klassen Automobile.

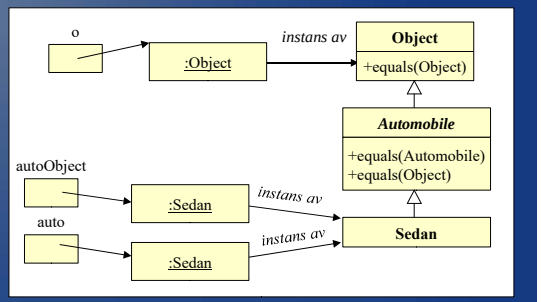


Detta innebär att Automobile överskuggar metoden som ärvs från Object.

Övning 2 forts: Överskuggning

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

Object o = **new** Object();
Automobile auto = **new** Sedan();
Object autoObject = **new** Sedan();
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);



Övning 2 forts: Överskuggning

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

```
Object o = new Object();
Automobile auto = new Sedan();
Object autoObject = new Sedan();
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)
```

Obs! equals skall naturligtvis inte överlagras såsom gjorts i detta exempel!!

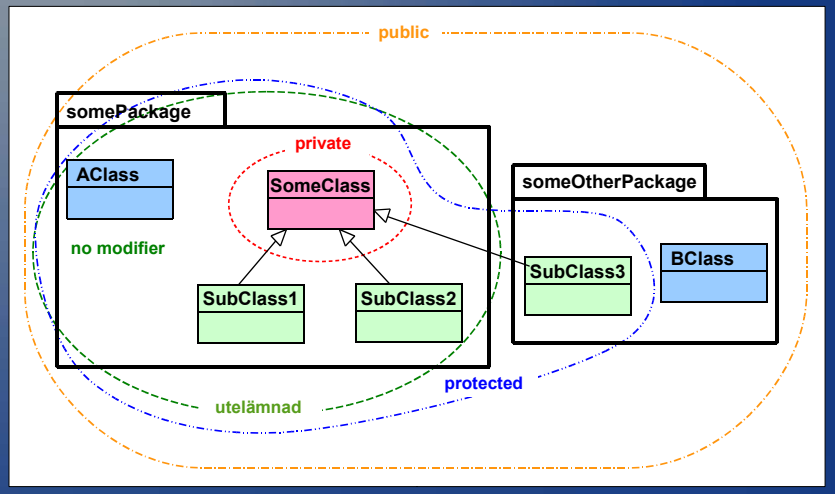
Åtkomstmodifierare

När man säger att en subklass ärver data och metoder (implementation) från sin superklass, betyder detta inte nödvändigtvis att subklassen kan få *direkt access* till dessa i superklassen.

I Java finns fyra olika synlighetsmodifierare:

public	tillåter access för alla klasser.
protected	tillåter access för alla klasser i samma paket samt för subklasser i andra paket.
utelämnad	tillåter access för alla klasser i samma paket (paketsynlighet, eng. package private).
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 superklass	Åtkomst i subklasser
public	**public**
protected	**protected** **public**
package private	package private **protected** **public**
private	ingen

Detta följer, som vi senare skall se, från *Liskov Substitution Principle*.

Vidgad synlighet eller inte?

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

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

Ett undantag är när man har ett paket som utgör en integrerad komponent (subsystem), då kan det ibland vara motiverat på grund av effektivitetsorsaker. 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 datat till en enhet. I objekt-orientering realiseras en sådan inkapslad enhet som en klass eller ett paket.

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

Inkapsling skall används så 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 Date {
   public int year;     //tillgänglig information
   public int month;    //tillgänglig information
   public int day;      //tillgänglig information
   public Date(int year, int month, int day) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
}
```

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

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

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

Hemlighåll information

Ny implementation av klassen Date:

```
//-- Bad code --//
public class Date {
   public int julian;     //tillgänglig information
   public Date(int year, int month, int day) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
}
```

Oförändrad kod i användarprogrammet

```
Date d = new Date(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 Date är inkapslade

```
public class Date {
   private int year;
   private int month;
   private int day;
   public Date(int year, int month, int day) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
}
```

måste koden i föregående exempel skrivas

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

Detta innebär att det är möjligt att byta implementation i klassen Date utan att klientkod 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 klienter behöver för att använda klassen begränsas till ett minimum.
- klienterna skall bara veta hur modulen används, inte hur modulen är implementerad.

Slutsats: Gör alla instansvariabler privata!

Information Hiding Principle

En klass kan jämföras med ett isberg:

90% döljs under vattnet och endast 10% är synligt över ytan.

- Klienter skall inte veta något om hur gränssnittet är implementerat
- Interna ändringar i klassen skall inte beröra klienterna.

Access-metoder och mutator-metoder

Vi gör en konceptuell skillnad mellan

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

Ett objekt som tillhör en klass som innehåller mutatormetoder kan ändra sitt tillstånd. En sådan klass är muterbar (*mutable*).

Ett objekt som bibehåller samma tillstånd (som det får när det skapas) under hela sin livstid kallas för icke muterbart (*immutable*). Icke muterbara objekt tillhör en icke muterbar klass. Ett nödvändigt (men, som vi senare skall se, inte tillräckligt) villkor för att en klass skall vara icke muterbar är att klassen saknar muterande metoder.

Klassen String och omslagsklasser som t.ex. Integer och Double, är exempel på icke muterbara klasser.

Access-metoder och mutator-metoder

```
public class Date {
   private int year;
   private int month;
   private int day;
   public Day(int year, int month, int day) {. . .}
   public int getYear() {. . .}
   public int getMonth() {. . .}
   public int getDay() {. . .}
}
```

Date saknar mutator-metoder. Skall vi lägga till metoderna

```
public void setYear(int year)
public void setMonth(int month)
public void setDay(int day)
```

i Date?

Access-metoder och mutator-metoder

Antag att klassen Date har mutator-metoderna

```
public void setYear(int year)
public void setMonth(int month)
public void setDay(int day)
```

och att en klient exekverar satserna

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

Vad händer?

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

Slutsats: Skapa inte slentrianmässigt set-metoder för alla instansvariabler! Eftersträva icke-muterbara klasser.