PST: Objektorienterad programmering

- En föreläsning om:
  - Objektorienterad programmering
  - Avvägningar vid design av objektorienterade språk
  - Stöd för objektorientering i ert favoritspråk Java

Början ...

- Historik
  - Simula 67
  - Smalltalk 80
- Procedurorienterad programmering
  - Subprogram
  - Programbibliotek
- Dataorienterad programmering
  - Abstrakta datatyper
  - Objektbaserade språk, föregångare till...

Objektorienterade programmeringsspråk

- Smalltalk, Eiffel och Java är rena objektorienterade språk
- C++ och Ada stöder objektorientering
- Det går att programmera objektorienterat i C eller assembler för den delen
  - Språket måste stödja någon form av inkapsling,
  - T.ex. på fil-nivå

Krav

- Tre krav måste vara uppfyllda:
  - Abstrakt datatyp
  - Arv
  - Polymorfism och dynamisk bindning
- Den abstrakta datatypen kallas ofta klass
  - Data kallas **attribut**, funktionaliteten finns i **metoder**
  - Anrop av en metod kallas att skicka ett **meddelande** till ett objekt

Inkapsling

- Inkapsling brukar också tas upp som ett krav på ett objektorienterat språk
- Java har två sorters inkapsling
  - Class scope, med tre nivåer
    - private
    - protected
    - public
  - Package scope
    - Allt som inte är private är synligt i klassens package
- C++ har friends i stället

Tillbaka till 80-talet ...

- Problem med objektbaserade språk
  - Abstrakta datatyperna borde kunna återanvändas
  - Alla typdefinitionerna oberoende och på samma nivå
- Lösningen heter Arv
  - Data och funktionalitet kan ärvas och modifieras
  - Återanvändning utan ändringar i redan skriven kod

Arv ...

- En subklass ärver från en basklass (kallas också superklass)
  - Åtkomst av basklassens attribut och metoder kan styras
  - Metoder kan omdefinieras (override)
  - Två sorters metoder och två sorters attribut
    - Instansmetoder och attribut
    - Klassmetoder och attribut

Klass och instansattribut i Java

```
class Klass
{
   private int instansAttribut;
   static private int klassAttribut;

   private int instansMetod()
   {
      return instansAttribut + klassAttribut;
   }

   static private int klassMetod()
   {
      return klassAttribut; // Ej instansAttribut !
   }
}
```


Polymorfism och dynamisk bindning

- Polymorfa referenser
  - Är av basklassens typ
  - Men kan referera till objekt av någon subklass' typ
- Dynamisk bindning vid metodanrop
  - 'Rätt' metod binds vid anropet, dvs objektets typ avgör vilken metod som anropas

Dynamisk bindning i Java

```
class Publikation
{
   private String titel;

   public String info()
   {
      return titel;
   }
}
```

```
class Bok extends Publikation
{
   private String författare;

   public String info()
   {
      return super.info() + författare;
   }
}
```

```
Publikation p1 = new Publikation ...
Publikation b1 = new Bok ...

System.out.println(p1.info());   // Info om en Publikation
System.out.println(b1.info());   // Info om en Bok
```


Bindningar i Java

- Default är dynamisk bindning
  - Modifieraren **final** gör att en metod inte kan omdefinieras, den får då automatiskt statisk bindning
- Jämför med C++ som har statisk bindning som default
  - Modifieraren **virtual** anger att bindningen skall vara dynamisk
- Imperativa språk måste använda switch

Computing with an Object-Oriented Language

- Ett objektorienterat program som körs kan ses som en samling datorer/objekt som kommunicerar via meddelanden.
- Varje objekt är en abstraktion av en dator i den meningen att det lagrar data och kan manipulera det.
- Objektorienterad programmering är att lösa problem genom att identifiera objekten i problemet och sedan simulera dem, dess processer och den nödvändiga kommunikationen.

Avvägningar vid design av ett objektorienterat språk

- Tre sätt att angripa problemet med typsystemet:
  - Den 'rena' modellen:
    - Allt är objekt, från det minsta heltal till det största datasystem
    - + Elegant form
    - - De enklaste operationer måste ske genom metodanrop
  - Utgå från ett imperativt språk och lägg till en objektmodell
    - + Snabbare
    - - Mycket större, mer komplext system

Tre sätt ...

- Objektmodell i botten men med 'imperativ stil' för primitiva datatyper
  - + snabbt för heltal och andra primitiva typer
  - - Komplikationer när metoder bara kan ta objekt som parametrar -> Wrapper Classes
- Java hör till den senare typen
- Allt är objekt utom de primitiva datatyperna som int, double, ...
- String och fält [] är objekt

Är subklasser också subtyper ?

- Det är subtyper som har ett 'är en' - förhållande till sin bastyp
- För att en subklass skall vara en subtyp krävs att
  - Subklassen omdefinierar metoder på 'ett kompatibelt sätt', dvs utan att skapa typfel
  - Ett sätt är att ha precis samma signatur och returtyp
  - Mindre restriktiva regler kan förekomma
  - Speciellt otillåtet är att ha metoder som inte kan ärvas

Mera precist uttryckt ...

- Den omdefinerande metodens parametrar måste vara supertyper av den omdefinierade (ärvda) metodens parametrar
  - Contravariance rule
- Den omdefinierande metodens returtyp måste vara en subtyp av den ärvda metodens returtyp
  - Covariance rule
- Emerald enda språket
- Java, C++ kräver exakt samma typer

Två sorters inkapsling vid arv

- Implementation Inheritance
  - Alla detaljer angående implementationen (t.ex. attribut) är synliga i subklassen
  - Protected i Java
- Interface Inheritance
  - Bara vissa metoder är synliga i subklassen, 'interfacet' (inte att förväxla interface)
  - Alla attribut är private i basklassen
  - Tar längre tid pga metodanrop, bättre inkapsling
- Bästa sättet är kanske att erbjuda båda ?

Implementation Inheritance (jmf förra exemplet)

```
class Publikation
{
   protected String titel;

   public String info()
   {
      return titel;
   }
}
```

```
class Bok extends Publikation
{
   private String författare;

   public String info()
   {
      return titel + författare;
   }
}
```

Sämre inkapsling, en förändring i Publikation kan få stora effekter i Bok

Statisk eller dynamisk typkontroll ?

- Om det språket skall vara starkt typat:
  - -> Statisk typkontroll
  - Restriktioner vid polymorfism
  - Två sorters typkontroll vid metodanrop:
    - Parametrarna måste stämma
    - Returtypen likaså
  - Kanske tillåtet med 'assignment compatibility'
- Alternativet är dynamisk typkontroll
  - Vänta med kontrollen tills metoden anropas
  - Kostsamt och försenar typkontrollen

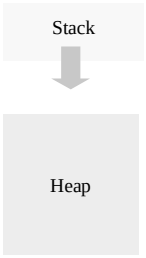
Multiplet arv, någon ?

- Behovet finns ibland, t.ex i Java (Applets som vill vara trådar också)
- Varför inte ?
  - Kompexitet:
    - Namnkollisioner
  - Prestanda
    - En extra addition, inget att bråka om ?
  - Svårt att designa klasserna rätt
  - Större beroenden mellan klasser
- Java har interface, en 'lättviktstyp'

Allokering/deallokering av objekt

- Var allokera objekten ? Tre sätt :
  - Statiskt av kompilatorn
  - Dynamiska objekt på stacken
  - Eller på heapen med new
- Om new det enda sättet -> uniform metod
  - inga pekare som måste derefereras a la C++
- C++ har alla sätten - och alla problemen
- Java har bara new (allt på heapen)

Stack och heap



Dynamiska variabler, t.ex. parametrar, lokala variabler
Ints, Doubles, referenser

Allt som skapas med new, dvs alla objekt, inklusive fält och strängar

Risk för fragmentering ->
Garbage Collector

Vi använder **new**, sen då ?

- Hur deallokera objekten ?
- Implicit ->
  - Vi behöver Garbage Collection i någon form
  - Java har GC
- Explicit ->
  - Problem med 'dangling pointers'
  - C++ har **delete**

Dynamisk eller statisk bindning ?

- Dynamisk bindning är grundläggande för objektorientering
  - Skall vi göra alla bindningar dynamiska ?
    - Prestandaskäl talar kanske emot, varför dynamisk bindning där statisk räcker ?
  - Användaren får välja -> mer komplicerat språk
- I Java är bindningarna dynamiska som deafult - final gör bindningen till metoden statisk