Bilder Vecka 6

TDA548/Joachim von Hacht

Klasser

Eget Arv: Extends

```
public class Pet {
     private String name;
     private int age;
     public String getName() {return name;}
     public void setName(String name) { this.name = name;}
     public int getAge() { return age;}
     public void setAge(int age) { this.age = age;}
}

public class Dog extends Pet {
   ...
}

public class Cat extends Pet {
  ...
}

Dog d = new Dog(...);
Cat c = new Cat(...);
out.println(d.getName()); // Call inherited methods
out.println(c.getAge());
```

3

Alla klasser ärver vissa metoder implicit från klassen Object. Men vi kan även låta våra egna klasser ärva från varann.

- Genom att ange extends vid klassdeklarationen kommer klassen **(subklassen**) att ärva allt (icke-privat) från en annan angiven klass **(superklassen, basklassen**)
  - D.v.s. i klasserna Cat och Dog kan vi använda alla (ärvda) metoder från Pet.
  - Konstruktorer ärvs inte.
- För att komma åt de privata instansvariablerna måste vi använda set/get (vilka skall vara publika)
- Denna typ av arv kallas **implementationsarv** eftersom vi ärver körbar kod.
- Vi använder denna typ av arv för att undvika redundant kod!
  - Det som är gemensamt för flera klasser bryts ut och samlas i en basklass (metoder/instansvariabler).
  - Därefter får alla klasser som behöver det ärva basklassen.
- Java kan bara använda implementationsarv från en superklass (man kan bara använda extends en gång)!
  - Java har inte multipelt implementationsarv.

Abstrakta Basklasser

```
public abstract class Pet {
    private String name;
    private int age;
    public String getName() {return name;}
    ...
}

public class Dog extends Pet {
   ...
}


Dog d = new Dog(); // Ok
Pet p = new Pet(); // No!
```

4

Ofta är det inte tänkt att man skall kunna skapa instanser av basklassen

- Vi skapar Dog och Cat-objekt men inte Pet-objekt (ett sällskapsdjur är inte ett djur, det är en klassifikation)!
- För att man inte skall kunna skapa objekt anges att klassen är **abstract**!
    - Kan inte använda new-operator på en abstrakt klass.

Ett annat fall då man måste deklarera klassen som abstrakt är då den innehåller en abstrakt metod.

- Se Metoder.

super

```
public class FacingCreep extends
                        AbstractFacingCreep {

  @Override
  public void move() {
     super.move(); // First run move in superclass
     facing = getDir(); // Then run this
  }


}
```


Super, i koden, förekommer bara i samband med arv och syftar på basklassobjektet.

- I detta fallet har vi en överskuggad metod men vill i denna köra basklassobjektets metod först.
- Super är inte en referens (vilket ju this är)

Konstruktorer och super(...)

```
public class Pet {
     private String name;
     private int age;
     public Pet(String name, int age) {
          this.name = name;
          this.age = age;
     }
}
```

```
public class Dog extends Pet {
   public Dog(String name, int age) {
       super(name, age);
   }
}
```

```
Dog d = new Dog("Fido", 3);
```

6

Om superklassen har en konstruktor med parametrar måste subklassens konstruktor anropa denna.

- För att initiera superklassens instansvariabler.
  - Vi kan se det som att superklassobjektet är en del av subklassobjeket
- **super(..)** syftar på den direkta superklassens konstruktor (jämför this(...)).
  - Måste stå först i subklassens konstruktor.
- Detta innebär att i en arvshierarki så initieras klasserna uppifrån ner (alltid superklasser först)
  - Jämför också static och initiering!

Standard för Konstruktorer

```
// Evil game character
public class Creep extends AbstractCreep {

   // Constructor, setting all values (set some default in super)
   public Creep(Path path, double width, double height) {
       super(path, width, height, 1, 30, 10, 10);
   }

   // Overloaded constructor
   public Creep(Path path) {   // Remove some params to simplify
       this(path, 5,5);        // Call another constructor with
                                // default values
   }
}
```


Ett vanligt mönster för konstruktorer är att:

- Ha en baskonstruktor som sätter alla värden (och ev. anropar super)
- Ha en eller flera konstruktorer med förvalda värden (av bekvämlighetsskäl).
    - Dessa konstruktorer anropar alltid baskonstruktorn.

final Klasser

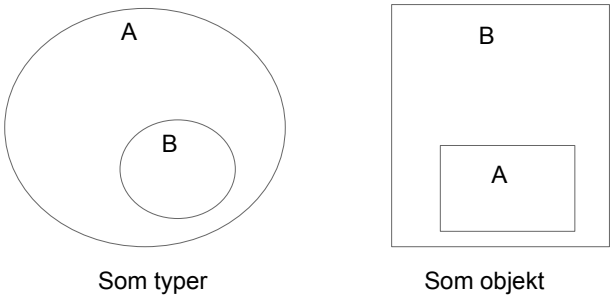
```
public final class Dog extends Pet {
   ...
}
```

Klasser som är final kan man inte ärva. En del av Java's klasser är final t.ex. String

- enum är alltid final

Olika Syn på Arv

B <: A



Antag B är subtyp till A (B <: A)

Utifrån typer är en subtyp en delmäng till supertypen.

- Det finns fler operationer för subtypen.

Utifrån objekt finns ett delobjekt av typ A i objektet B

- Man kan anropa alla metoder för A objektet på ett B objekt.

Typer

Arv och Subtyp

```
public class Pet  {
   private String name;
   public String getName() {
      return name;
   }
}
```

Dog <: Pet

```
public class Dog extends Pet {
   // Inherited
   public String getName(){...}
}
```

11

Om en klass ärver (extends) en annan klass skapas en super/subtyp relation!

- Klassen som ärver blir subtyp
- Logiskt: Eftersom klassen som ärver kan minst lika mycket som klassen den ärver från!
  - Alla operationer som vi kan göra med superklassen kan vi göra med subklassen (men inte tvärtom).
  - Som tidigare: Subtypen kan mer!
- Se även Klasser.

Typomvandlingar vid Arv

```
public class Pet  {
  ...
}

public class Dog extends Pet {
   ...
}

Pet p = new Dog(); // Implicit sub to super, ok

Dog d = (Dog) p;  // Explicit super to sub, allowed
                  // but possibly ClassCastException
                  // at runtime
```


Typomvandling från sub till super sker automatiskt vid behov.

- Tvärtom får man (på egen risk) explicit typomvandla (undvik).

Typomvandlingar med Gränssnittstyper

```
public interface ISayable  {
   String say();
}
```

```
public class Car {
   // Class has nothing
   // to do with ISayable!!!
}
```

```
public class TalkingCar extends Car implements ISayable {
   @Override
   public String say() { … }
}
```

```
Car car = new Car();
ISayable s = (ISayable) car;   // Compiles, why is this allowed ??? ...

car = new TalkingCar();    // Because could be like this! TalkingCar <: Car
ISayable s = (ISayable) car;     // ... sub class may implement!
```

13

Vilket typ som helst kan explicit typomvandlas till en gränssnittstyp!!!

- Trots att objektet kanske inte implementerar gränssnittet d.v.s. inte är en subtyp.
  - Kompilering tillåten eftersom någon subklass kanske implementerar!
- Om objektet inte implementerar blir det körningsfel, undantag.

I Bilden: Ett TalkingCar-objekt kan sägas ha flera typer (TalkingCar, Car eller ISayable). Gäller generellt med OO-språk.

Sammanfattning <:

```
'1' <: 1 <: 1.0            // Primitive values
int i NOT <: double d      // Primitive variables
int[] i NOT <: double[]    // Primitive arrays
Integer i NOT <: Double d  // Class (reference) types
none (except null) <: enum  // enum a class type
Integer[] <: Double[]       // Reference arrays!!!
any type <: Object
null <: any reference type
interface B NOT <: interface A    // Interface types
List<Integer> NOT <: List<Number>  // Generic types
B implements A <: interface A      // Implements
B extends A <: class A             // Extends
```

Förklaring rad för rad:

1. För värden av primitiva värden gäller att: char <: int <: long <: float <: double
2. Variabler är inte subtyper eftersom vänster sida vid tilldelning inte kan bytas ut mot subtypen (vi måste ha super = sub)
3. Arrayer av primitiv typ har inget <: förhållande (samma motivering som 2).
4. Referenstyper har inget <: förhållande (samma som Player NOT <: Dog)
5. enum har inga subtyper alls (någonsin) utom den namnlösa null-typen (null-värdet)
6. Arrayer av referenstyp är subtyper!!! "The array loophole". Under körning sker kontroll av vad vi stoppar in i arrayen, kan leda till ArrayStoreException
7. Object är supertyp till allt (primitiv typer boxas)
8. null är subtyp till vilken referenstyp som helst (lite konstigt eftersom vi sagt att subtypen kan mer,... null kan inget. Vi får se det som ett undantag)
9. Gränssnitt har inte någon super/sub-relation.
10. Generiska typer har inget <: förhållande (trots att t.ex. Integer <: Number)
11. En klass som implementerar ett gränssnitt blir subtyp till

1. gränssnittstypen.
2. En klass som ärver blir subtyp till klassen den ärver ifrån.

Interface och getClass()

```
public interface ISayable  {
   String say();
}
```

```
public class Car {
   // Class has nothing
   // to do with ISayable!!!
}
```

```
   public class TalkingCar extends Car implements ISayable {
      @Override
      public String say() { … }
   }
```

```
Car c = new Car();
TalkingCar tc = new TalkingCar();
ISayable s = tc;

out.println(c instanceof Car);  // true
out.println(tc instanceof Car); // true
out.println(c instanceof TalkingCar); // false

out.println(c.getClass() == Car.class);  // What's the type of the object?
out.println(s.getClass() == TalkingCar.class);
```

15

För att under körning undersöka ett objekts typ kan man använda

- instanceof: Sant om objektet är av typen eller någon subtyp
- getClass(): Ger objektets typ (klassobjektet för objektet)

Undvik, ger oflexibel kod, föredra override se Klasser och Metoder.

- Dock, för att säkert kunna typomvandla från super till subtyp kan man undersöka om detta går m.h.a. instanceof (slipper ClassCastException)

Statisk och Dynamisk Typ

```
// Declared variable type (Object) is compatible with any
// reference type. No change of object "fia", it's still a
// string object
Object o = "fia";

o.lenght();          // Bad! No length method in object

// Declared type Object, there is a method equals in
// Object so ok, … **BUT** method in String executed!!!
// String overrides Objects equals. Objecttype decides
// which method
o.equals("pelle");
```

Variable/static type (Object) = there is a method!
Object/dynamic type (String) = which method to run!

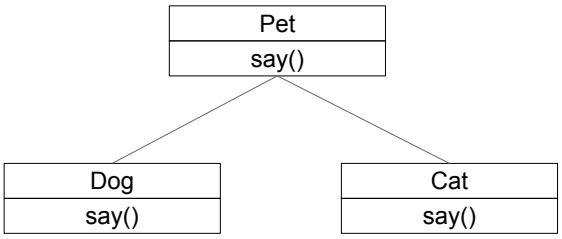
Huruvida typen för referensen till objektet (typen för objektet säger vi) och typen för variabeln är kompatibla bestämmas av ett stor antal komplicerade regler, varav den viktigaste är super/subtyp relationen

- Typen på variabeln och typen på objektet kan alltså var olika!
- Typen för variabeln (statiska, deklarerade) typen bestämmer vilka operationer över huvud taget kan göra (vilka operationer kompilatorn godkänner, samma som för primitiva typer).
- Typen på objektet (objekttypen, dynamiska typen, runtime-typen) bestämmer vilken "version" av operationen (metoden) som skall köras. Den dynamiska typen ges av getClass().
    - Ev. är metoden överskuggad (override). Se Klasser och Metoder.
    - Kallas också **sen bindning (lat/dynamic binding)**

Sammanfattning:

- En primitiv variabel innehåller ett värde av exakt den primitiva typen.
- En referensvariabel kan innehålla en referens till den deklarerade typen eller någon subtyp.

Polymorfism



17

Ett sammanfattande begrepp för “att det som händer beror på de inblandade typerna” är
polymorfism. Vi har sett följande

- Vilken metod som körs beror på parametrarnas typer, överlagring/overloading
- Vilken metod som körs beror på objektets typ, överskuggning/override
- Generiska typer.

Anm: Det som vanligen avses när man talar om polymorfism i OO sammanhang är override.

- Metoden say() i bilden.

Referenser

Metodreferenser

```
// References to method with int param and return type
Function<Integer, Integer> addRef = this::add;

// Reference to void method with int param
Consumer<Integer> doItRef = this::doIt;

// Sadly can't do like this … must use methods as below
// addRef(6);
// doItRef(6);

out.println(addRef.apply(6));  // Execute referenced method
doItRef.accept(6);             // Execute referenced method

void doIt(int i) {
   out.println(i);
}

int add(int i) {
   return i + 1;
}
```

19

Man kan ha referenser till metoder i Java

- Vi går inte in på detaljerna här (man kan spara referenser i variabler d.v.s. det finns typer för olika sorters metoder m.m.)
- En metodreferens skrivs: objektet :: metoden (dubbla kolon)
  - Objektet vi använder är this.

Användning

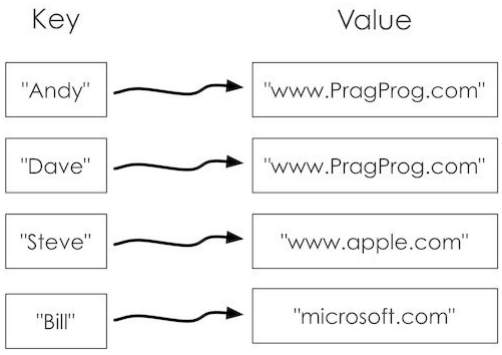
- För intern iteration av samlingar (forEach)
  - Man anger vilken metod som skall anropas för varje element i samlingen (elementet skickas som argument till metoden, d.v.s. metoden måste ha en parameter av elementtypen)
- I JavaFX då vi skapar händelsestyrda program
- Finns i sista labben (given kod).

Händelsestyrda Program

(Se bilder JavaFXHändelser)

Samlingar

Avbildningstabell

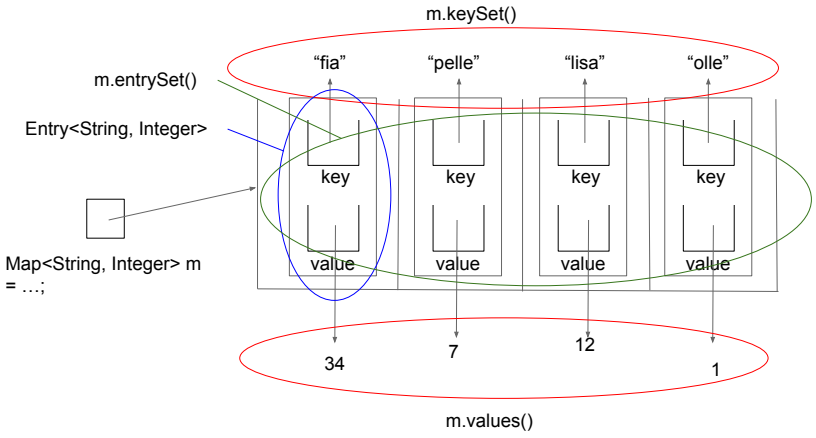


22

En avbildningstabell kopplar en nyckel och ett värde.

- Som en uppslagsbok
- Genom att använda nyckeln (key) kan man slå upp värdet (value)
    - OBS! Åt andra hållet är inte lika lätt.

Avbildningstabell i Java



23

En avbildningstabell i Java har typen Map<Key, Value> där båda Key och Value skall var någon referenstyp (Map<String, Integer> i bilden).

- Vanligaste operationerna är:
  - put(key, value) sparar värdet under nyckeln i tabellen
  - get(key), avläser och returnerar värdet för en nyckel (ändrar inte i tabellen)

Förutom metoderna put() och get() kan man i Java behöva följande metoder från Map:

- m.keySet(), ger en samling med alla nycklar
- m.values(), ger en samling med alla värden
- m.entrySet, ger en samling med Entry (par av nyckel/värde)

Metoder i Map<K,V>

```
// A Map with key type String and value Integer
// NOTE Variable naming: name + Count (used for names + frequency of name)
Map<String, Integer> nameCount = new HashMap<>();

// Add frequency of names
nameCount.put("fia", 23);
nameCount.put("sven", 31);
nameCount.put("lisa", 29);

int nFia = nameCount.get("fia");  // Look frequency for fia
out.println(nFia == 23);
out.println(nameCount.get("sven") == 31);

// Traversing
for( String s: nameCount.keySet()){    // All keys
   out.println(s);
}
for( Integer i : nameCount.values()){    // All values
   out.println(i);
}
for( Map.Entry<String, Integer> e : nameCount.entrySet()){  // Both key and value
   out.println(e.getKey() + ":" + e.getValue());
}

nameCount.remove("fia");
out.println(nameCount.size() == 2);
```


Gränssnitten/Klasserna ingår inte i JCF (om du saknar dessa i den tidigare bilden, … spelar ingen roll för oss)