Föreläsning 8

Likhetsrelationer	**equals()**
Hashkoder	**hashCode()**
Jämförelserelationer	**compareTo()**
Kopiering av objekt	**clone()**

Klassen Object

I Java är alla klasser subklasser till java.lang.Object.

Klassen Object innehåller följande metoder

clone	skapar en kopia av objektet.
equals	jämför om objektet är lika med ett annat objekt.
getClass	returnerar vilken klass objektet tillhör vid runtime.
hashCode	returnerar hashkoden för objektet.
toString	returnerar en String-representation av objektet.
samt	ett antal metoder för synkronisering i multitrådade program.

För att en klass skall betraktas som fullständig skall klassen överskugga metoderna toString, clone, equals och hashCode. Särskilt gäller detta om klassen kommer att användas i, eller tillsammans med, ett klassbibliotek.

Metoden equals

I klassen Object har metoden equals följande utseende:

```
public boolean equals(Object obj) {
   return this == obj;
}
```

Detta betyder att två objekt betraktas som lika endast om referenserna till dem är *alias*.

Alla klasser behöver därför definiera vad som menas med att ett objekt är lika med ett annat objekt.

Metoden equals

equals-metoden används på många ställen i bl.a. Collection-klasserna.

Här är ett typiskt exempel på användning av equals:

```
private Object[] elementData  =  . . .;
. . .
public int indexOf(Object elem) {
   for (int i = 0; i < elementData.length ; i++)
      if (elem.equals(elementData[i])
         return i;
   }
   return -1;
}
```

Alla klasser behöver definiera vad som menas med att ett objekt av klassen är lika med ett annat objekt.

Metoden equals

Låt oss titta på klassen Triangle:

```
import  java.awt.Point;
public class Triangle {
   private Point p1, p2, p3;
   /**
      @pre:  p1 != null && p2 != null && p3 != null
   **/
   public Triangle(Point p1, Point p2, Point p3) {
      this.p1 = p1;
      this.p2 = p2;
      this.p3 = p3;
   }
   . . .
}
```


Metoden equals

Vid en första anblick kan det tyckas vara enkelt att avgöra likheten mellan två objekt av klassen Triangle – om samtliga tre hörnpunkter är lika i de båda trianglarna borde trianglarna vara lika. Detta leder således till att equals-metoden skulle få följande utseende:

```
//-- Felaktig --//
public boolean equals(Object otherObject) {
   if ( !(otherObject instanceof Triangle))
      return false;
   Triangle other = (Triangle) otherObject;
   return p1.equals(other.p1)
          && p2.equals(other.p2)
          && p3.equals(other.p3) ;
}
```

Metoden equals

Låt oss nu göra ett litet testprogram

```
Triangle t1 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
Triangle t2 = new Triangle(new Point(2,2), new Point(4,3), new Point(1,0));
Triangle t3 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
System.out.println(t1.equals(t2));
System.out.println(t1.equals(t3));
```

Utskriften blir vad vi förväntar oss:

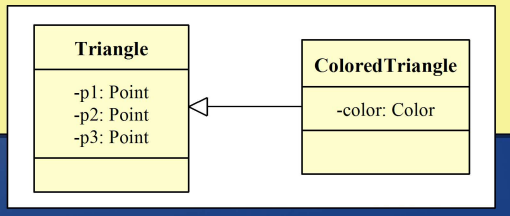
```
false
true
```

d.v.s. att t1 och t2 är olika, samt att t1 och t3 är lika.

Metoden equals

Låt oss nu krångla till det hela genom att införa en subklass ColoredTriangle till klassen Triangle.

```
import  java.awt.Point;
import  java.awt.Color;
public class ColoredTriangle extends Triangle {
   private Color color;
  /**
     @pre:  color != null && p1 != null && p2 != null && p3 != null
   **/
   public ColoredTriangle(Color color, Point p1, Point p2, Point p3) {
      super(p1, p2, p3);
      this.color = color;
   }
   . . .
}
```



Metoden equals

Låt oss nu göra ett nytt testprogram:

```
Triangle t1 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
Triangle t2 = new Triangle(new Point(2,2), new Point(4,3), new Point(1,0));
ColoredTriangle t3 = new ColoredTriangle(Color.GREEN, new Point(0,0),
                                         new Point(1,1), new Point(1,0));
System.out.println(t1.equals(t2));
System.out.println(t1.equals(t3));
```

Utskriften blir

```
false
true
```

d.v.s. att t1 och t3 är lika. Men är verkligen ett objekt av klassen Triangle lika med ett objekt av klassen ColoredTriangle??
Vi måste i metoden equals ta hänsyn till vilka typer som objekten i jämförelsen har.

Metoden equals

I klassen Object finns metoden getClass() som returnerar vilken runtime-klass ett objekt har. Denna metod kommer nu väl till pass

```
//-- Fortfarande felaktig --//
public boolean equals(Object otherObject) {
   if (otherObject.getClass() != this.getClass())
      return false;
   Triangle other = (Triangle) otherObject;
   return p1.equals(other.p1) && p2.equals(other.p2) && p3.equals(other.p3);
}
```

Testkör vi nu med samma exempel som tidigare får vi med denna variant av equals-metoden utskriften

```
false
false
```

Detta resultat är vad vi vill ha. Dock är equals-metoden fortfarande inte korrekt!

Metoden equals

I *The Java Language Specification* anges att equals-metoden skall ha följande egenskaper:

Skall vara *reflexiv*:	För varje icke-**null** referens x skall det gälla att x.equals(x) returnerar **true**.
Skall vara *symmetrisk*:	För alla referenser x och y skall det gälla att x.equals(y) returnerar **true** om och endast om y.equals(x) returnerar **true**.
Skall vara *transitiv*:	För alla referenser x, y och z skall gälla att om x.equals(y) returnerar **true** och y.equals(z) returnerar **true** så skall också x.equals(z) returnera **true**.
Skall vara *konsistent*:	Om objekten till vilka x och y refererar inte har förändrats skall upprepade anrop av x.equals(y) returnera samma värde.

För alla icke-**null** referenser x skall gälla att x.equals(**null**) skall returnera **false**.

Metoden equals

I equals-metoden för klassen Triangle måste vi ta hand om fallet då objektet som jämförelsen utförs mot är **null**.

```
//-- Slutlig version av equals i klassen  Triangle--//
public boolean equals(Object otherObject) {
   if (this == otherObject)
      return true;
   if (otherObject == null)
      return false;
   if (otherObject.getClass() != this.getClass())
      return false;
   Triangle other = (Triangle) otherObject;
   return p1.equals(other.p1) && p2.equals(other.p2)
          && p3.equals(other.p3) ;
}
```

Metoden equals

Vad händer när följande kod körs?

```
ColoredTriangle ct1 = new ColoredTriangle(Color.RED, new Point(0,0),
                                          new Point(1,1), new Point(1,0));
ColoredTriangle ct2 = new ColoredTriangle(Color.BLUE, new Point(0,0),
                                          new Point(1,1), new Point(1,0));
System.out.println(ct1.equals(ct2));
```

Utskriften blir

true

ColorTriangle ärver equals-metoden från Triangle, och metoden equals i Triangle beaktar inte komponenten color.

Även klassen ColoredTriangle måste överskugga equals-metoden!!

Metoden equals

Metoden equals i klassen ColoredTriangle får följande utseende:

```
//-- equals i klassen  ColoredTriangle--//
public boolean equals(Object otherObject) {
   return super.equals(otherObject) &&
          color.equals(((ColoredTriangle) otherObject).color) ;
}
```

Den som är observant, har naturligtvis konstaterat att vi definierat likhet mellan två trianglar genom att instansvariablerna p1 är lika, p2 är lika och p3 är lika. Men trianglarna kan vara lika även i andra situationer. Detta överlåtes dock som övning.

Ett dilemma

Antag att vi har tre variabler t1, t2, och t3 som är deklarerade av typen Triangle.

Antag också att de trianglar som dessa variabler refererar till, samtliga har samma hörnpunkter. Betraktade som objekt av typen Triangle är de således lika.

Men antag nu att t2 i verkligheten refererar till ett objekt av typen ColoredTriangle, utan att användaren är medveten om detta.

Detta innebär att t1.equals(t3) och t2.equals(t3) ger olika värden (**true** respektive **false**), vilket inte är vad användaren förväntar sig.

Våra equals-metoder bryter således mot *Liskov Substitution Principle*!

Lösning på dilemmat

Det finns ingen bra lösning på detta dilemma, utan är beroende på hur klasserna skall användas:

1. Ta bort equals-metoden i ColoredTriangle. Det innebär att två objekt av klassen ColoredTriangle betraktas som lika även om de har olika färg.
2. Ta bort equals-metoden i Triangle. Detta innebär att två objekt av klassen Triangle betraktas som lika endast om de är alias.
3. Ta bort superklass/subklass förhållandet mellan Triangle och ColoredTriangle. Innebär att vi förlorar möjlighet till polymorfism.
4. Acceptera att vi bryter mot LSP.

Överväg noga hur equals *implementeras när implementationsarv är involverat.*

Metoden hashCode

Metoden hashCode() skall alltid överskuggas när man överskuggar equals-metoden. Motiveringen är att när man lagrar ett objekt i en hashtabell skall två objekt som är lika ha samma hashkod för att hamna på samma plats i tabellen. Platsen ges av värdet som metoden hashCode returnerar, vilket är ett heltal.

Om x.equals(y), så är x.hashCode() == y.hashCode()

Har man en klass för vilken man skall omdefiniera metoden hashCode räcker det i allmänhet att nyttja attributens hashkoder och addera dessa. Möjligen att också multiplicera koderna med ett primtal innan additionen.

```
//-- hashCode för klassen Triangle  --//
public int hashCode() {
  return 7*p1.hashCode() + 11*p2.hashCode() + 13*p3.hashCode();
}

//-- hashCode för klassen ColoredTriangle  --//
public int hashCode() {
  return 17*super.hashCode() + 19*color.hashCode();
}
```


Metoden hashCode

De instansvariabler som används i metoden equals() kallas klassens *signifikanta variabler.*

Metoden hashCode() får *endast* bero av de signifikanta variablernas värden, men den måste inte bero på alla dessa.

```
public class Person {
   private String pnr;      // signifikant
   private String name;
   private String address;
   public boolean equals(Object o) {
      // Detta och det andra objektet är lika om personnumren är lika
   }
   public int hashCode() {
      // Värdet som returneras får endast bero på personnumret.
   }
}
```

- Varför får inte hashCode bero på andra egenskaper än personnumret ovan?

Metoden compareTo

Metoden compareTo() används för att rangordna objekt i någon form av storleksordning.

```
public interface Comparable<T> {
   int compareTo(T other);
}
```

a.compareTo(b) returnerar
-1 om a är mindre än b
 0 om a är lika med b
 1 om a är större än b

Rekommendation:

Om a.equals(b) returnerar **true** så bör a.compareTo(b) returnera 0

Det omvända måste inte nödvändigtvis gälla: Två objekt som i någon mening är lika stora måste inte vara likadana. T.ex. kan två geometriska figurer ha lika stora ytor men ha helt olika form.

Om objekten skall *lagras i samlingar* bör dock även gälla att

Om a.compareTo(b) returnerar 0 så bör a.equals(b) returnera **true**

Metoden compareTo

Exempel: Genom att låta klassen Person implementera gränssnittet Comparable kan vi t.ex. rangordna personobjekt så att namnen kommer i bokstavsordning (lexikografisk ordning). Jämförelsen delegeras till compareTo() i klassen String.

```
public class Person implements Comparable<Person>{
   private String pnr;
   private String name;
   private String address;
   public Person(String pnr,String name,String address) { ... }

   public int compareTo(Person other) {
      return name.compareTo(other.name);
   }

   public String toString() {
      return pnr + ", " + name + ", " + address;
   }
}
```

String.compareTo()

- Har detta sätt att göra objekt jämförbara någon uppenbar nackdel?

Metoden compareTo

Exempel: I klassen java.util.Arrays finns sorteringsmetoden

static void sort(Object[] o)

"Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements."

Eftersom klassen Person implementerar gränssnittet Comparable, så kan vi använda ovanstående metod för att sortera ett fält av Person-objekt.

```
import java.util.Arrays;
public class Main {
   public static void main(String[] arg) {
      Person[] group =
         { new Person("1234567-8901","Lisa","Kungsgatan 3"),
           new Person("345657-0123","Rut","Prinsgatan 5"),
           new Person("2345678-9021","Emil","Drottninggatan 4") };
      Arrays.sort(group);
      for ( Person p : group )
         System.out.println(p);
   }
}
```

Utskrift:

2345678-9021, **Emil**, Drottninggatan 4
1234567-8901, **Lisa**, Kungsgatan 3
345657-0123, **Rut**, Prinsgatan 5

Metoden compareTo

Exempel: Genom att låta klassen Rectangle implementera gränssnittet Comparable kan vi rangordna rektangelobjekt i storleksordning efter deras yta.

```
public class Rectangle implements Comparable<Rectangle> {
   private int width;
   private int height;

   public Rectangle(int width,int height) {
      this.width = width;
      this.height = height;
   }

   public int getArea() {
     return width*height;
   }

   public int compareTo(Rectangle other) {
      if ( getArea() < other.getArea() ) return -1;
      if ( getArea() > other.getArea() ) return 1;
      return 0;
   }
}
```

```
public boolean equals(Object otherObject) {
   if (this == otherObject)
      return true;
   if (otherObject == null)
      return false;
   if (otherObject.getClass() != this.getClass())
      return false;
   Rectangle other = (Rectangle) otherObject;
   return width = other.width && height == other.height;
}
```

```
Rectangle r1 = new Rectangle(10,20);
Rectangle r2 = new Rectangle(20,10);
Rectangle r3 = new Rectangle(20,30);
Rectangle r4 = new Rectangle(25,15);
System.out.println(r1.equals(r2));          false
System.out.println(r1.compareTo(r2));       0
System.out.println(r4.compareTo(r3));       -1
System.out.println(r2.compareTo(r4));       -1
System.out.println(r3.compareTo(r2));       1
```


Metoden clone

Metoden clone() används för att skapa en kopia av ett objekt.

Klassen Object definierar en standardimplementering av clone():

```
protected Object clone() throws CloneNotSupportedException {
   if (this instanceof Cloneable) {
      //Copy the instance fields
      . . .
   }
   else
      throw new CloneNotSupportedException();
}
```

Vi ser att metoden clone() är **protected** och att ett undantag av typen CloneNotSupportedException kastas om metoden anropas för ett objekt av en klass som inte implementerar gränssnittet Cloneable.

Metoden clone

En klass vars objekt skall gå att klona måste implementera gränssnittet Cloneable och överskugga metoden clone().

Det rekommenderas att överskuggade metoder av clone skall uppfylla följande villkor:

x.clone() != x

x.clone().equals(x)

x.clone().getClass() == x.getClass()

Observera att den sista egenskapen även skall gälla då clone ärvs av subklasser. Det innebär att **new** *aldrig får användas* för att skapa kopian i överskuggningar av clone.

Metoden clone

När Object.clone() anropas för ett objekt skapas en *grund kopia* av objektet.

Referenser kopieras "som de är" vilket i praktiken innebär att addressen till ett utpekat objekt kopieras – inte objektet självt.

Kopian får samma *dynamiska typ* som det anropande objektet och det gäller även subklassobjekt. En överskuggande clone-metod kan alltså typomvandla returvärdet från super.clone() till den egna klassen innan det returneras.

Metoden clone

Metoden Object.clone() returnerar alltid ett objekt av *samma dynamiska typ* som det anropande objektet:

```
public class SomeCloneableClass implements Cloneable {
   public Object clone() {
      ...
      x = super.clone();
      ...
   }
}
```

Statisk typ: Object
Dynamisk typ: SomeCloneableClass

Det är alltså typsäkert att omvandla typen hos returvärdet från **super**.clone till den dynamiska typen och vi kan utnyttja *kovarians*:

```
public class  SomeCloneableClass implements Cloneable {
   public SomeCloneableClass clone() {
      ...
      x = (SomeCloneableClass)super.clone();
      ...
}
```

Använd alltid denna möjlighet till kovarians i clone-metoder!

Grund kopiering (*shallow copy*)

Antag att vi vill kunna skapa kopior av objekt av klassen SomeClass nedan:

```
public class SomeClass {
   private int value;
   private boolean  status;
   private String str;
   public SomeClass(int value, boolean status, String str) {
      this.value = value;
      this.status = status;
      this.str = str;
   }
    . . .
}
```

Vi måste således låta klassen implementera interfacet Cloneable och överskugga metoden clone().

Vill man att clone() skall vara allmänt tillgänglig skall clone() deklareras som **public**.

Grund kopiering (*shallow copy*)

Metoden clone i klassen Object, skapar och returnerar en kopia av det aktuella objektet. Varje instansvariabel i kopian har *samma värde* som motsvarande instansvariabeln i originalet.

I klassen SomeClass är instansvariablerna value och status primitiva variabler, och instansvariabeln str är ett icke-muterbart objekt. Därför gör clone() i klassen Object allt som är nödvändigt för att skapa en kopia.

```
public class SomeClass implements Cloneable {
   private int value;
   private boolean  status;
   private String str;
   public SomeClass(int value, boolean status, String str) {
      this.value = value;
      this.status = status;
      this.str = str;
   }
}
```

kovariant returtyp

```
@Override
public SomeClass clone() {
   try {
      return (SomeClass)super.clone();
   }
   catch (CloneNotSupportedException e) {
      throw new InternalError();
   }
}
```

Grund kopiering (*shallow copy*)

```
public class SomeClass {
   private int value;
   private boolean  status;
   private String str;
   public SomeClass(int value, boolean status, String str) {
      this.value = value;
      this.status = status;
      this.str = str;
   }
   @Override
   public SomeClass clone() {
      try {
         return (SomeClass)super.clone();
      }
      catch (CloneNotSupportedException e) {
         throw new InternalError();
      }
   }
}
```

Vi har glömt att implementera Cloneable!
Vad händer?

```
. . .
SomeClass master = new SomeClass(5, true, "Hello ");
SomeClass copy = master.clone();
```


Djup kopiering (*deep copy*)

Grund kopiering är trivial, eftersom det bara är att nyttja metoden clone() i klassen Object.

Grund kopiering fungerar då attributen för objektet som klonas utgörs av *primitiva typer och icke-muterbara typer*. För primitiva typer skapas kopior och för icke-muterbara typer spelar det ingen roll att de delas, eftersom de inte kan förändras.

Om ett objekt har muterbara referenser måste man använda djup kopiering (*deep copy*) för att klona objektet.
Djup kopiering innebär att man skapar *kopior av de refererade objekten,* vilket kan vara komplicerat.

Djup kopiering (*deep copy*)

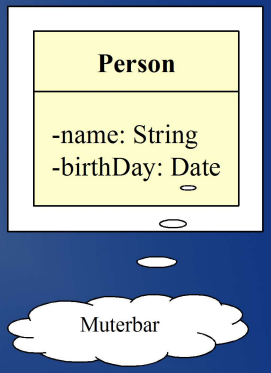
Principiellt gör man djup kopiering enligt följande:

```
public class SomeClass implements Cloneable {
   . . .
   @Override
   public SomeClass clone() {
      try {
         // First shallow copy all attributes.
         SomeClass result = (SomeClass) super.clone();
         // Then deep copy all mutable attributes.
         . . .
         return result;
      }
      catch (CloneNotSupportedException) {
         throw new InternalError();
      }
   }
}
```


Djup kopiering – enkelt exempel

Klassen Person har en referensvariabel birthDay, av typen java.util.Date, som är muterbar.

```
import  java.util.Date;
public class Person implements Cloneable {
   private String name;
   private Date birthDay;
   . . .
   @Override
   public Person clone() {
      try {
         Person result = (Person) super.clone();
         result.birthDay = (Date) birthDay.clone();
         return result;
      }
      catch (CloneNotSupportedException e) {
         throw new InternalError();
      }
   }
}
```



Kloning och arv

Vid arv anropas clone() i superklassen, varefter de muterbara referensvariablerna som deklareras i subklassen måste kopieras djupt.

```
import java.util.Date;
public class Member extends Person {
   private Date dayOfMembership;
   . . .
   @Override
   public Member clone() {
      Member result = (Member) super.clone();
      //add copies of sub class specified fields to result
      result.dayOfMembership = (Date) dayOfMembership.clone();
      return result;
   }
}
```

Djup kopiering av muterbart attribut

- *Varför behöver inte* Member *implementera* Cloneable*?*
- *Varför behövs inte något* ***try-catch****-block?*

Kloning och arv

Metoden clone bör överskuggas även om inte subklassen innehåller attribut som måste kopieras djupt. Annars krävs osäkra typomvandlingar vid kloning av subklassobjekt.

```
public class SomeCloneableClass implements Cloneable {
   public SomeCloneableClass clone() { ... }
}
public class CloneableSubClass extends SomeCloneableClass {
   // clone överskuggas ej i denna klass
}
```

```
CloneableSubClass original = new CloneableSubClass();
 ...
CloneableSubClass copy = (CloneableSubClass)original.clone();
```

Typomvandling krävs

Kloning och arv

Här överskuggas clone() i subklassen så att rätt typ returneras:

```
public class CloneableSubClass extends SomeCloneableClass {
   . . .
   @Override
   public CloneableSubClass clone() {
      return (CloneableSubClass)super.clone();
   }
}
```

```
CloneableSubClass original = new CloneableSubClass();
 ...
CloneableSubClass copy = original.clone();
```

Typomvandling behövs ej!

Kloning och arv

Exempel: Varför det är olämpligt att skapa kopian med **new** i clone().

Speciell varning utfärdas till C++-programmerare som är vana vid att använda s.k. kopieringskonstruktorer.

```
public class A {
   private int x;
   public A(int x) {
      this.x = x;
   }
   public A(A other) {
      x = other.x;
   }
   public Object clone() {
      return new A(this);
   }
}
```

```
public class B extends A {
   private int y;
   public B(int x,int y) {
      super(x);
      this.y = y;
   }
   public B(B other) {
      super(other);
      y = other.y;
   }
   public Object clone() {
      return new B(this);
   }
}
```

```
public class C extends A {
   public C(int x) {
      super(x);
   }
   public void wohoo() { ... }
}
```

Ärver clone() från A

public A(A other) och **public** B(B other) är kopieringskonstruktorer.

Kloning och arv

Klasserna A, B, C och Main nedan är typkorrekta men vi får ändå ett typfel *under exekveringen* då clone() anropas i bar():

```
public class Main {
   private static void foo(B obj) {
      B copy = (B)obj.clone();
   }
   private static void bar(C obj) {
      C copy = (C)obj.clone();
   }
   public static void main(String[] arg) {
      foo(new B(1,2));
      bar(new C(1));
   }
}
```

Ok, clone returns a B object

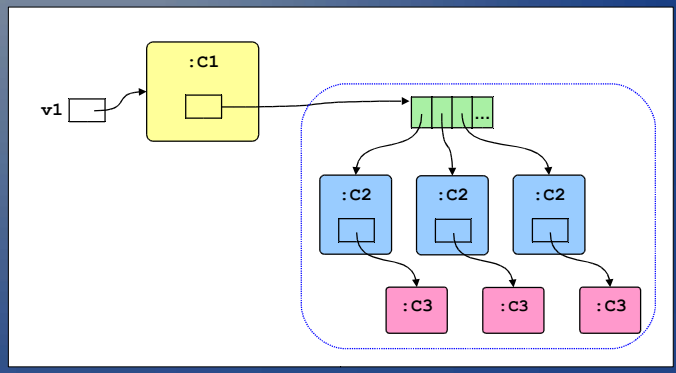
Class cast exception!

I den (osäkra) typomvandlingen omvandlas den statiska returtypen Object till C och vi lovar att objektet skall vara av typ C, eller någon subklass till C, men det kommer ett A-objekt från clone()!

- Varför kan inte ett A-objekt lagras i en variabel av typ C?

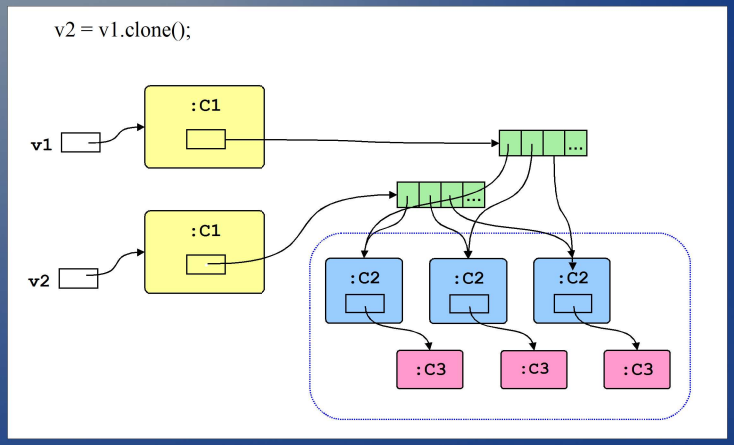
Djup kopiering – krångligare exempel

I nedanstående scenario vill vi göra en göra en djup kloning av objektet som refereras av variabeln v1.



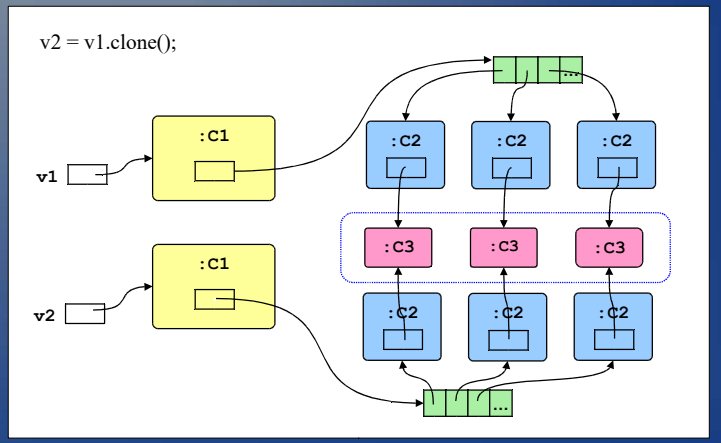
Djup kopiering – felaktig

I nedanstående scenario skapar metoden clone() i klassen C1 en kopia av sin instansvariabel, vilken är en lista. Men clone() skapar inte kopior av elementen i listan.



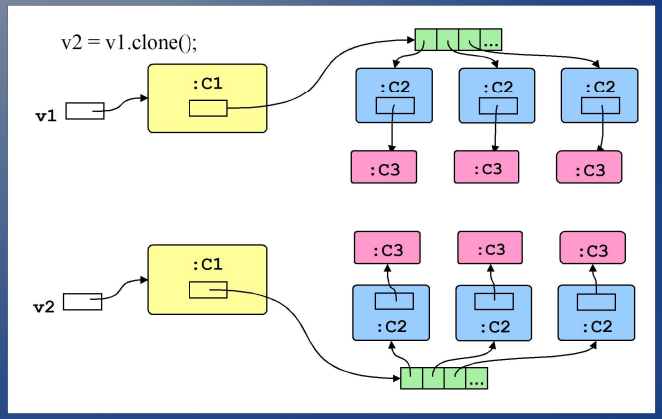
Djup kopiering – felaktig

I nedanstående scenario skapar metoden clone() i klassen C1 en djup kopia av sin instansvariabel (listan och elementen i listan) på ett korrekt sätt – men clone() i klassen C2 gör en grund kopia.



Djup kopiering – korrekt

I nedanstående scenario har vi en korrekt djup kopiering. Metoden clone() i klassen C1 skapar en kopia av listan och elementen i listan – och clone() i klassen C2 gör en djup kopiering.



Djup kopiering – samlingar

Metoden clone() i samlingar och arrayer använder grund kopiering. Detta betyder att alla instansvariabler som är samlingar eller arrayer måste klonas ”element för element” om elementen är muterbara.

```
public class SomeClass implements Cloneable {
   private List<Person>  list = new ArrayList<Person>();
   . . .
   @Override
   public SomeClass clone() {
      try {
         SomeClass result = (SomeClass) super.clone();
         result.list = (ArrayList<Person>)list.clone();
         for (int i = 0; i <= list.size(); i++)
               result.list.set(i,list.get(i).clone();
         return result;
      }
      catch (CloneNotSupportedException e) {
         throw new InternalError();
      }
   }
}
```


Förhindra kloning

Det finns situationer då man, av olika anledningar, vill förhindra kloning. Nedanstående scheman är då användbara:

```
public class NoCopy {
   private int value;
   private boolean  status;
   private String str;
   . . .
   public NoCopy clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException();
   }
}
```

```
public class NoCopySubClass extends SomeCloneableClass {
   private int value;
   private boolean  status;
   private String str;
   . . .
   public NoCopySubclass clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException();
   }
}
```


Problem vid kloning

Det finns många problem med att implementera clone():

- superklassen måste implementera clone()
- superklassens implementation måste vara korrekt
- många klasser saknar implementation av clone()
- många klasser har felaktig implementation av clone()
- alla instansvariabler som är samlingar eller arrayer måste klonas ”element för element”
- i cykliska strukturer och strukturer där objekt är delade, måste också motsvarande objekt vara delade i kopian
- . . .