Kopiering av objekt i Java

Först

När du läser detta papper bör du samtidigt studera dokumentationen för klasserna `Object`, `Cloneable` (`java.lang`) och `ArrayList` (`java.util`). Mycket blir klarare genom att köra exemplen nedan i BlueJ och använda Object inspector för att utforska objektstrukturerna genom att följa pekarna så långt det går. Filerna finns i `kloning.zip` på kurshemsidan.

Kopiering av objekt i Java

Ibland är grund kopiering av objekt önskvärd men oftast krävs djup kopiering.

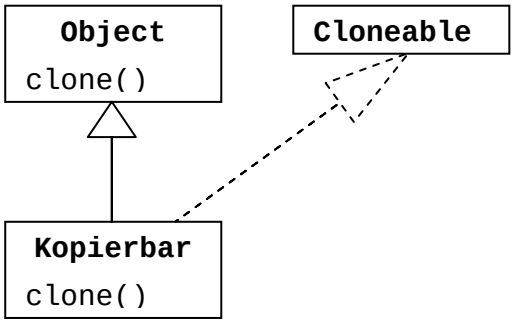
Metoden clone

I java finns standardmetoden `clone` för kopiering av objekt. Metoden definieras i superklassen `Object` och ärvs därför av alla klasser. Den ärvda `clone` returnerar en grund kopia av objektet den anropas för. Klasser som har ickeprimitiva instansvariabler (de innehåller alltså objektreferenser) måste definiera om `clone` så att den kopiar djupt. Görs detta rätt kommer klassen att tillåta kopiering med `clone` på ett standardmässigt sätt. Avvikelser bör dokumenteras, se t.ex `ArrayList.clone`. *Felaktig kopiering som lämnar objektstrukturer delade mellan flera objekt kan ge upphov till svårupptäckta fel!*

Gränssnittet Cloneable

En klass som implementerar gränssnittet (`interface`) `Cloneable` tillåter `Object.clone` att kopiera klassens instansvariabler grunt. Om `clone` anropas för ett objekt där klassen inte deklarerar att den implementerar `Cloneable` kastas undantaget `CloneNotSupportedException`.

En klass vars objekt skall vara kopierbara
1. implementerar gränssnittet `Cloneable`
2. överskuggar metoden `clone`



Definitionen får typiskt följande form

```
public class Kopierbar implements Cloneable {
    ...
    public Kopierbar clone() {
        Kopierbar kopia = (Kopierbar)super.clone();
        ...
        return kopia;
    }
    ...
}
```

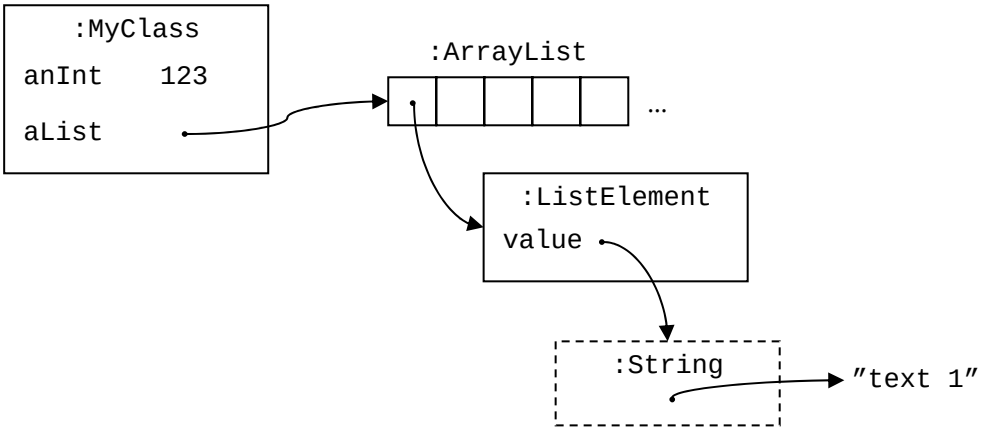
Om klassen endast har instansvariabler av primitiva datatyper, alltså inga objekttyper, räcker ovanstående eftersom grund och djup kopiering då ger samma resultat. Annars måste anropet `super.clone()` kompletteras med egen kopiering av de variabler som kopierats grunt av `super.clone()`. Observera typomvandlingen från `Object` till `Kopierbar` av returvärdet från `clone`, denna omvandling är nödvändig för att kunna hantera objektet lokalt. Omvandlingen är säker eftersom `super.clone()` alltid returnerar ett objekt av samma dynamiska typ som objektet den anropas för. Ibland är det nödvändigt att göra all kopiering själv. Följande exempel visar vilka problem som kan uppstå.
Anm. Överskuggade `clone`-metoder kan ha returtypen `Object`, eller den aktuella klassen (kovariant returtyp, se OH från den andra föreläsningen om arv).

Exempel

För att demonstrera hur kloning av objekt går till i Java definierar vi klasserna `MyClass` och `Listelement`. För att förenkla koden låter vi alla instansvariabler vara publika (huh!).

```
public class MyClass implements Cloneable {
    public int anInt;                       // en primitiv variabel
    public ArrayList<ListElement> aList;   // objekt, se upp!

    public MyClass () {
        anInt = 123;
        aList = new ArrayList<ListElement>();
        aList.add(new ListElement("text 1"));
    }

    public Object clone() {
        ...
    }
}

public class ListElement implements Cloneable {
    public String value;

    public ListElement(String value) {
        this.value = value;
    }

    public ListElement clone() {
        return (ListElement)super.clone();
    }
}
```

Ett objekt av klassen `MyClass` har följande struktur



Följande kodavsnitt antas exekveras i samtliga tre exempel som följer:

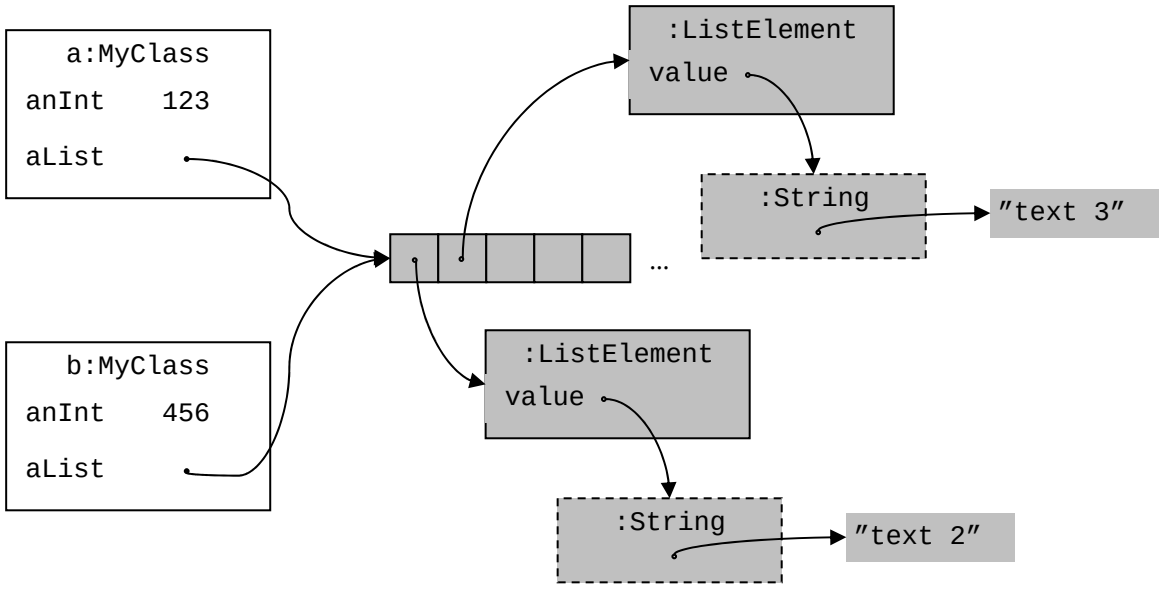
```
MyClass a,b;
MyClass a = new MyClass();      // ger strukturen i figuren ovan
MyClass b = a.clone();
b.anInt = 456;
b.aList.get(0).value = "text 2";
b.aList.add(new ListElement("text 3"));
```

`MyClass.clone()` - försök 1

Som nämndes ovan räcker det inte att definiera `clone` så här

```
public MyClass clone() {
    return (MyClass)super.clone();
}
```

`super.clone()` returnerar att nytt objekt som blir en grund kopia av det aktuella objektet. Resultatet blir att variablen `anInt` kopieras korrekt, eftersom det har typen `int`, medan `aList` i båda objekten kommer att referera till samma listobjekt. Efter att de tre sista satserna exekverats ser objektstrukturen ut som figuren på nästa sida visar.



Vi ser att listan delas mellan båda objekten, och det var ju inte meningen!

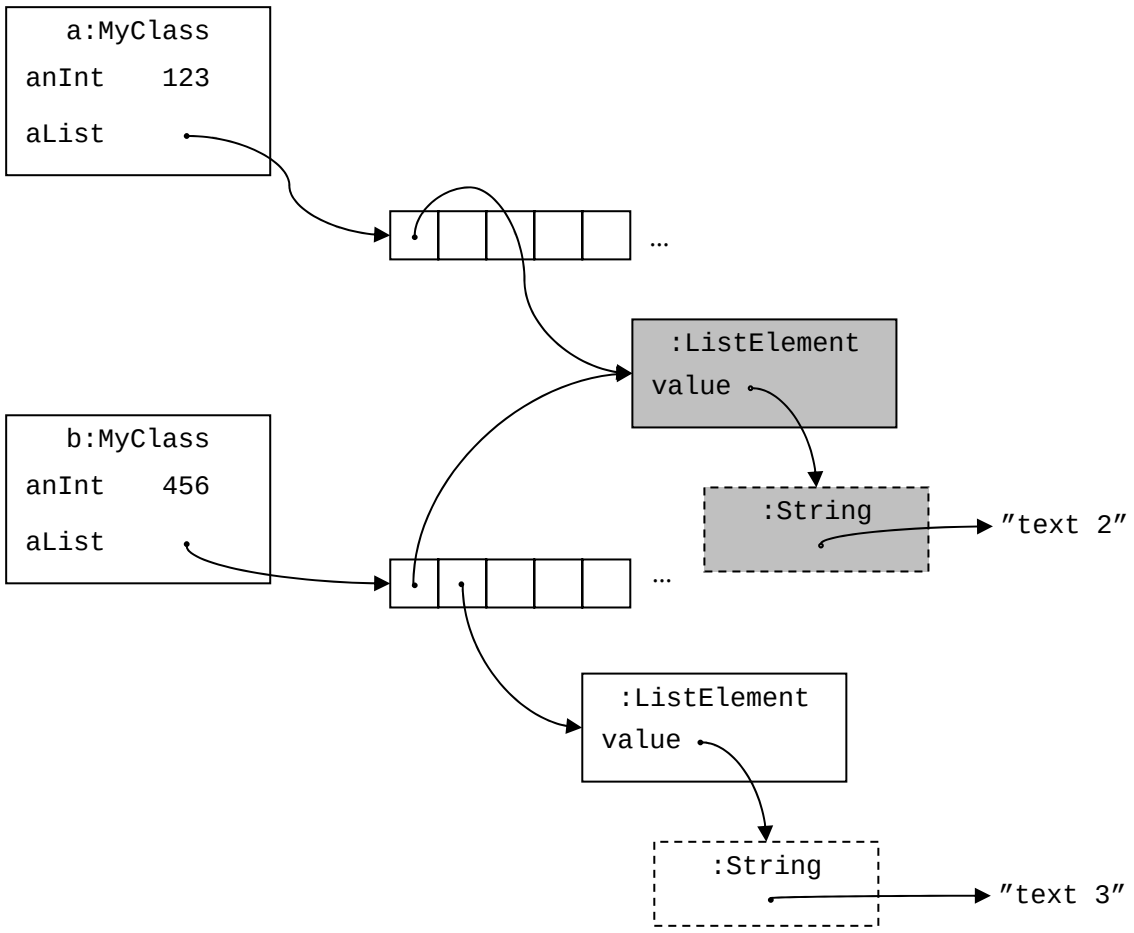
`MyClass.clone()` - försök 2

För att undvika delning av listan måste den klonas separat:

```
public MyClass clone() {

    try {
        MyClass copy = (MyClass)super.clone();
        copy.aList = (ArrayList<ListElement>)aList.clone();
        return copy;
    }
    catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

Nu borde väl allt stämma? Nej! Som framgår av nästa figur kopieras visserligen listan, men listelementen är också objekt och listklassens `clone` kan bara ta ansvar för själva liststrukturen, inte för elementen. Följden blir att de listelement som existerade före kopieringen av listan kommer att delas mellan originallistan och dess kopia.

Anm. När **`super`**`.clone` anropas ovan så anropas `clone` i klassen `Object`. Om denna metod av någon anledning misslyckas måste det bero på något internt fel i den virtuella maskinen. Ett sådant fel måste betraktas som mycket allvarligt och försök till återhämtning därför ej meningsfullt. Därför fångas undantaget och istället kastas `InternalError`. Det är inte meningen att `InternalError` skall fångas av klienten till `MyClass`. Ovanstående sätt att definiera `clone` kan användas för klasser som ärver från `Object`.

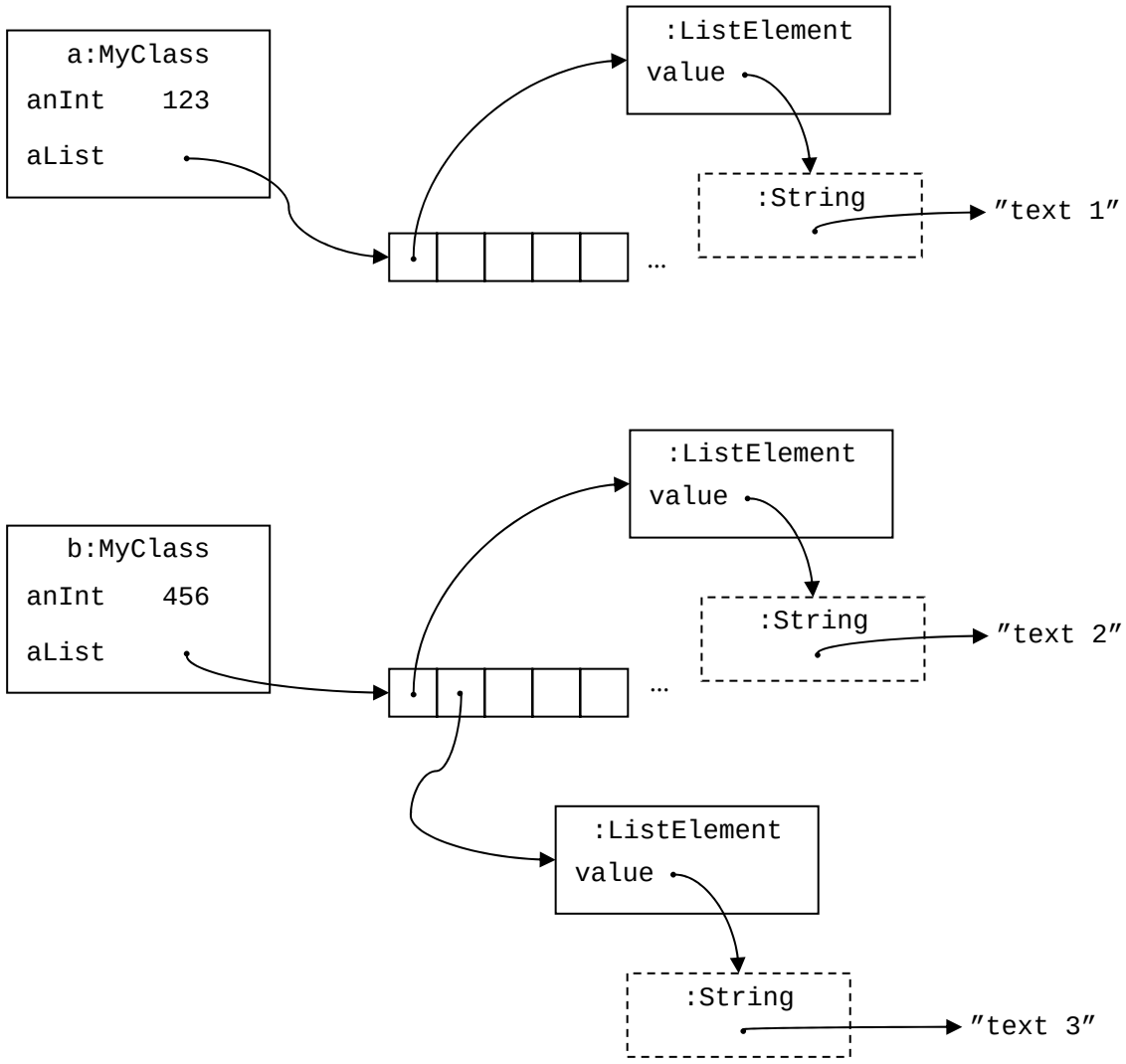


`MyClass.clone()` - försök 3

Vi går nu ett steg längre och kopierar även själva listelementen.

```
public MyClass clone() {
    try {
        // kopiera hela detta objekt grunt
        MyClass copy = (MyClass)super.clone();
        // kopiera listan djupt (elementen grunt)
        copy.aList = (ArrayList<ListElement>)aList.clone();
        // kopiera elementen djupt
        for ( int i = 0; i < aList.size(); i++ )
            copy.aList.set(i,aList.get(i).clone());
        return copy;
    }
    catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

Resultatet av ett anrop av denna sista version av `clone` (samt övriga satser) visas i nästa figur. Vi ser att såväl lista som element kopierats djupt.



Slutsats

Designprincip: Ansvarssdriven design.
Varje klass som definierar om metoden `clone` måste ta ansvar för kopiering av alla objektstrukturer som klassen *själv* äger, inget annat.

I vårt exempel finns två klasser, `MyClass` och `Listelement`. Notera att `MyClass` inte behöver, eller skall, kopiera interna objekt i `Listelement`, den skall *lita på* att `Listelement` sköter det. I de två första misslyckade varianterna av `MyClass.clone` föll strukturer som `MyClass` borde ansvara för ”mellan stolarna”.

Vid programmering i Java är det mycket viktigt att komma ihåg att alla objekt hanteras med referenser samt skillnaden mellan grund och djup kopiering.