Föreläsning 9

Tobias Wrigstad

tobias.wrigstad@it.uu.se

Interface

Arv och subtypspolymorfism

- I Java används arv för *återanvändning* och för att skapa *subtyper*

  (Notera att en subklass inte behöver vara en subtyp, men vi undviker det här)

- Förmågan hos ett objekt av en typ B att användas istället för objekt av en typ A om B är en subtyp till A

  Shape x = **new** Square(); *// OK om Square ärver av Shape*

- Javas statiska typning och avsaknad av multipelt implementationsarv begränsar kraftfullheten hos subtypspolymorfism

  Om jag har en B som ärver av A och jag vill kunna göra:

  C c = **new** B(); *// Ex. C = ColouredShape och B = Square*

  måste jag göra så att B också ärver C, men B ärver ju redan A!

Möjliga lösningar

- Ersätt B extends A med B extends C

  B tappar alla medlemmar ärvda från A (B fungerar/kompilerar nog inte nu)

- Stoppa in C i arvskjedjan så B extends C och C extends A

  Endast möjligt om vi har koden till C och C inte ärver något D ≠ Object

- Använd ett interface

  Ger fördelarna med multipelt arv med avseende på subtypspolymorfism

  Undviker problemen med multipelt arv av olika implementationer

Interface

- En specifikation av ett protokoll, d.v.s. ett antal metodsignaturer

  Kan även ses som ett kontrakt

- Exempel:

```
public interface Coloured {
    public Colour getColour();
    public void setColour(Colour c);
    public void paintSameAs(Coloured obj)
}
```

Multipelt arv mellan interface

- Interface kan ärva av varandra
- Det finns inget ”rotinterface” på samma sätt som Object är en rotklass
- Multipelt arv (även från samma klass) är oproblematiskt och därför tillåtet

  Dock ej cykler!

- Specialisering (overriding) finns inte eftersom implementationer ej finns i interface

```
interface A extends B { ... }
interface B {}
interface C extends A, B { ... }
interface D extends B, C { ... }
```

OK!

UU

Koppling mellan klasser och interface

- En klass kan *implementera* ett interface

  En nominell relation i form av en implements-deklaration in klasshuvudet

```
public class Square implements Coloured {
    private double R;
    private double G;
    private double B;
    public Colour getColour() { return new Colour(R,G,B); }
    public void setColour(Colour c) {
        R = c.getRedComponent();
        G = c.getGreenComponent();
        B = c.getBlueComponent();
    }
    public void paintSameAs(Coloured obj) {
        this.setColour(obj.getColour());
    }
    ...
```

UU

Koppling mellan klasser och interface

- Varför fungerar följande kod inte?

```
public class Square implements Coloured {
    private double R;
    private double G;
    private double B;
    ...
    public void paintSameAs(Coloured obj) {
        this.R = obj.R;
        this.G = obj.G;
        this.B = obj.B;
    }
    ...
```

Koppling mellan klasser och interface

- Partiell implementation av ett interface

```
public interface Coloured {
    public Colour getColour();
    public void setColour(Colour c);
    public void paintSameAs(Coloured obj);
}

public class Circle implements Coloured {
    private Point center;
    private int radius;
    public Colour getColour() { ... }
    public void setColour(Colour c) { ... }
}
```

// Saknas!

Koppling mellan klasser och interface

- Partiell implementation av ett interface

```
public interface Coloured {
    public Colour getColour();
    public void setColour(Colour c);
    public void paintSameAs(Coloured obj);
}

public class Circle implements Coloured {
    private Point center;
    private int radius;
    public Colour getColour() { ... }
    public void setColour(Colour c) { ... }
}
```

// Saknas!

Circle.java:1: Circle is not abstract and does not override abstract method paintSameAs(Coloured) in Coloured

Abstrakta klasser

- Klasser som hjälper till att bygga arvshiearkier och inte är avsedda att instantieras

  new A(); går ej om A är abstakt klass

- En klass K kan deklareras som abstrakt med nyckelordet abstract **`abstract`**:

  `public abstract class` K ...

- En klass K måste deklareras abstrakt om

  K implementerar ett interface partiellt, eller

  K har en metod M som är deklarerad abstrakt, eller

  K ärver en abstrakt metod M utan att override:a/specialisera den

Interface och subtypspolymorfism

- Att göra Coloured till ett interface löser vårt tidigare problem

  Dock måste varje klass själv implementera funktionaliteten — den ärvs ej!

```
public interface Coloured { ... }
public class Shape { ... }
public class Square extends Shape implements Coloured { ... }

Square s = new Square();
Shape x;
Coloured c;
x = s;
c = s;
x = c; // Does not compile! Why?
c = x; // Does not compile! Why?
```

Parametrisk Polymorfism i Java

Parametrisk polymorfism i Java – ”Generics”

- Ibland behöver man skriva kod som fungerar på samma sätt för objekt av flera olika typer – varför är det dåligt att kopiera kod som vi har gjort här?

```
public class IntList {
    private class Link {
        Link next;
        int value;
    }
    private Link first;
}
```

```
public class Base...List {
    private class Link {
        Link next;
        BaseballPlayer value;
    }
    private Link first;
}
```

```
    private class Link {
        Link next;
        bool value;
    }
    private Link first;
}
```

UU

En naiv lösning

- ...och den enda före Java 1.5 (för många många år sedan)
- Hur skiljer sig denna implementation från de på föregående sida?

```
public class List {
    private class Link {
        Link next;
        Object value;
    }
    private Link first;
}
```

Typsäkerhet

```
IntList list1 = new IntList();
BaseballPlayerList list2 = new BaseballPlayerList();
BooleanList list3 = new BooleanList();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
bool v3 = false;

list1.add(v1); // ok
list1.add(v2); // does not compile
list1.add(v3); // does not compile
list2.add(v1); // does not compile
list2.add(v2); // ok
list2.add(v3); // does not compile
list3.add(v1); // does not compile
list3.add(v2); // does not compile
list3.add(v3); // ok
```

En lista av Object kan innehålla allting...

```
List list1 = new List();
List list2 = new List();
List list3 = new List();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
bool v3 = false;

list1.add(v1); // ok
list1.add(v2); // compiles, but is it safe?
list1.add(v3); // compiles, but is it safe?
list2.add(v1); // compiles, but is it safe?
list2.add(v2); // ok
list2.add(v3); // compiles, but is it safe?
list3.add(v1); // compiles, but is it safe?
list3.add(v2); // compiles, but is it safe?
list3.add(v3); // ok
```

UU

Fungerar detta eller ej?

```
List list1 = new List();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
bool v3 = false;

list1.add(v1); // ok
list1.add(v2); // compiles, but is it safe?
list1.add(v3); // compiles, but is it safe?

int v4 = (int) list1.get(1);
BaseballPlayer v5 = (BaseballPlayer) list1.get(2);
bool v6 = (bool) list1.get(3);
```

Fungerar detta eller ej?

```
List list1 = new List();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
bool v3 = false;

list1.add(v1); // ok
list1.add(v2); // compiles, but is it safe?
list1.add(v3); // compiles, but is it safe?

int v4 = (int) list1.get(1);
BaseballPlayer v5 = (BaseballPlayer) list1.get(2);
bool v6 = (bool) list1.get(3);
```

ClassCastException on line ...

UU

Parametrisk polymorfism

- Introducerades i Java 1.5
- Implementationen något begränsad på grund av bakåtkompatibilitet

```
public class List <ElementType> {
    private class Link {
        Link next;
        ElementType value;
    }
    private Link first;
}
```

- En klass introducerar en typ
- En parametriskt polymorf klass introducerar en typkonstruktor som kan användas för att skapa typer

UU

Parametrisk polymorfism

- Introducerades i Java 1.5
- Implementationen något begränsad på grund av bakåtkompatibilitet

```
public class List <ElementType> {
    private class Link {
        Link next;
        ElementType value;
    }
    private Link first;
}
```

```
List<Person>
List<String>
List<Object>
```

- En klass introducerar en typ
- En parametriskt polymorf klass introducerar en typkonstruktor som kan användas för att skapa typer

UU

Parametriskt polymorfa typer

```
List<Integer> list1 = new List<Integer>();
List<BaseballPlayer> list2 = new List<BaseballPlayer>();
List<Boolean> list3 = new List<Boolean>();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
bool v3 = false;

list1.add(v1); // ok
list1.add(v2); // does not compile
list1.add(v3); // does not compile
list2.add(v1); // does not compile
list2.add(v2); // ok
list2.add(v3); // does not compile
list3.add(v1); // does not compile
list3.add(v2); // does not compile
list3.add(v3); // ok
```

Parametriskt polymorfa typer

```
List<Integer> list1 = new List<Integer>();
List<BaseballPlayer> list2 = new List<BaseballPlayer>();
List<Boolean> list3 = new List<Boolean>();

int v1
Baseba
bool v3

list1.a
list1.a
list1.a
list2.a
list2.a
list2.a
list3.a
list3.a
list3.add(v3); // ok
```

Utvikning: varför `Integer` och `Boolean` och inte **`int`** och **`bool`**?

Svar: en **`int`** är en primitiv typ, och Java stöder inte primitiva typargument till typkonstruktorer.

Java konverterar automatiskt mellan primitiver (t.ex. **`int`**) och deras objektmotsvarigheter (t.ex. `Integer`) varför denna kod fungerar!
(Detta kallas för autoboxing.)

Att kedja typparametrar

- Om vår lista inte använt en inre klass...

```
public class List {
    private Link first;
}
public class Link {
    private Link next;
    private Object value;
}
```

UU

Att kedja typparametrar

- Om vår lista inte använt en inre klass...

```
public class List {
    private Link first;
}
public class Link {
    private Link next;
    private Object value;
}
```

```
public class List<E> {
    private Link<E> first;
}
public class Link<E> {
    private Link<E> next;
    private E value;
}
```

UU

Att kedja typparametrar

- Om vår lista inte använt en inre klass...

```
public class List {
    private Link first;
}
public class Link {
    private Link next;
    private Object value;
}
```

```
public class List<E> {
    private Link<E> first;
}
public class Link<E> {
    private Link<E> next;
    private E value;
}
```

Parameter

Argument

Parameter

Argument

Användande som typ

Manipulation av objekt av typparameter-typ

- Vad kan man göra med en variabel vars typ är okänd?

  Eller – bättre uttryckt – vilken typ har en variabel vars typ är en typparameter?

```
public class List<E> {
    private Link first;
    private class Link {
        Link next;
        E value;
        void m() {
            value.frob(); // kompilerar?
        }
    }
}
```

UU

Rotklassen till undsättning

- Under huven expanderas...

```
public class List<E> {
    private class Link {
        Link next;
        E value;
    }
    private Link first;
}
```

- ...till...

```
public class List<E extends Object> {
    private class Link {
        Link next;
        E value;
    }
    private Link first;
}
```

Rotklassen till undsättning

- En övre gräns (upper bound) för en typparameter låter oss bättre resonera om vad den kan bindas till

- I listan till höger kan E bindas till alla typer som ärver av Object

- Mer specifika typer är också möjliga, som här:

  Nu kan man anropa metoder på value som finns i Shape-klassen

- Priset är att `List<String>` ej längre är möjlig då `String` inte är en subtyp till `Shape`

```
public class List<E extends Object> {
    private class Link {
        Link next;
        E value;
    }
    private Link first;
}
```

```
public class List<E extends Shape> {
    private class Link {
        Link next;
        E value;
    }
    private Link first;
}
```

En ”svag” implementation

- Under huven kompileras...

```
public class List<ElementType> {
    private class Link {
        Link next;
        ElementType value;
    }
    private Link first;
}
```

- ...ned till...

```
public class List {
    private class Link {
        Link next;
        Object value;
    }
    private Link first;
}
```

UU

En ”svag” implementation

- Under huven kompileras...

```
public class List<ElementType> {
    private class Link {
        Link next;
        ElementType value;
    }
    private Link first;
}
```

- ...ned till...

```
public class List {
    private class Link {
        Link next;
        Object value;
    }
    private Link first;
}
```

Detta förklarar varför vi inte kunde binda `ElementType` till **`int`** förut – eftersom en **`int`** inte är ett `Object`.

...och med explicita övre gränser

- Under huven kompileras...

```
public class List<E extends Shape> {
    private class Link {
        Link next;
        E value;
    }
    private Link first;
}
```

- ...ned till...

```
public class List {
    private class Link {
        Link next;
        Shape value;
    }
    private Link first;
}
```

UU

En ”svag” implementation

- Under huven kompileras...

```
List<Integer> list1 = new List<Integer>();
int v1 = 7;
bool v3 = false;

list1.add(v1); // ok
list1.add(v3); // does not compile
```

- ...ned till...

```
List list1 = new List();
int v1 = 7;
bool v3 = false;

list1.add((Integer) v1); // ok
list1.add((Integer) v3); // does not compile
```

En ”svag” implementation

- Under huven kompileras...

```
List<Integer> list1 = new List<Integer>();
int v1 = 7;
bool v3 = false;

list1.add(v1); // ok
```

- ...ned till...

Detta är fortfarande typsäkert eftersom all interaktion med listan skyddas av (`Integer`)-omvandlingar!

```
List list1 = new List();
int v1 = 7;
bool v3 = false;

list1.add((Integer) v1); // ok
list1.add((Integer) v3); // does not compile
```

UU

Man kan binda typparametrar ”överallt”

```
public class StringList extends List<String> {}

public class Foo {
    public List<Boolean> getBar() { ... }
}
```

UU

Titta också på...

- Screencasten om parametrisk polymorfism
- Titta på hur parametrisk polymorfism används i Javas standardbibliotek
- Notera användanden som t.ex.

  `Comparable<K>`

  `Class<K>`

  etc.

- Varför det kan vara problematiskt att ge inre klasser (i motsats till nästlande klasser) typparametrar istället för att bara använda den omslutande klassens typparametrar

Undantagshantering

Undantagshantering

- När något går fel i Java *genereras* och *kastas* ett *undantag*
- Ett undantagsobjekt beskriver

  Typen av fel

  Var i programmet felet uppstod

  Vägen programmet tagit för att komma till felet i form av en stacktrace
- När ett undantag kastas avbryts exekveringen

  Kontrollen flyttas till *närmast omslutande undantagshanterare för denna typ av fel*

  Om det inte finns en sådan termineras programmet
- Untantagshanterare implementeras med hjälp av den s.k. *try-catch-satsen*

Att generera och kasta ett undantag

- En mängd fördefinierade undantag finns i Javas standardbibliotek:

  RuntimeException, IllegalArgumentException, ArrayIndexOutOfBoundsException, ArithmeticException, ClassCastException, ...

- Dessa är vanliga klasser – att generera ett undantag = instantiering

  new RuntimeException();

  new IllegalArgumentException(); etc.

- Att kasta ett undantag görs med nyckelordet **throw** – normalt bakas generering och kastande av undantag ihop:

  throw new RuntimeException();

- ...men även throw x; om x är en variabel som innehåller ett untantag

Att definiera egna undantag

- Lämpligt att göra i klasser för att förenkla felhantering

  Koden nedan definierar ett nytt undantag genom att ärva av undantagsklassen `Exception` som sedan kan kastas som vanligt

```
/**
 * @author Tobias Wrigstad (tobias.wrigstad@it.uu.se)
 * @date 2013-10-20
 */
public class DuplicateElementException extends Exception {
    public DuplicateElementException(String msg) { super(msg); }
}
```

```
throw new DuplicateElementException("Element "
                + e.toString() + " already in the set");
```

Felhanterare: try–catch–(finally)

- Följande kod installerar en felhanterare för all kod som körs i ... – även sådan som är i anropade metoder

```
try {
    ... // code that could fail
} catch(ExceptionType1 e) {
    // code to handle this type of failure
} catch(ExceptionType2 e) {
    // code to handle this type of failure
} finally {
    // code to always run -- fail or nofail
}
```

UU

Vad skrivs ut när `example()` körs?

```
void bar() { int stupid = 1/0; }

void foo() { bar(); System.out.println("!!!"); }

void example() {
    try {
        foo();
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } catch(Exception e) {
        System.err.println("Do something else");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

!!! *skrivs aldrig ut!*

UU

Vad skrivs ut när `example()` körs?

```
void bar() { int stupid = 1/0; }

void foo() { bar(); }

void example() {
    try {
        foo();
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } catch(Exception e) {
        System.err.println("Do something else");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

Varför går följande kod inte att kompilera?

```
void bar() { int stupid = 1/0; }

void foo() { bar(); }

void example() {
    try {
        foo();
    } catch(Exception e) {
        System.err.println("Do something else");
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

UU

Varför går följande kod inte att kompilera?

```
Test.java:15: exception java.lang.ArithmeticException has already been caught
    } catch(ArithmeticException e) {
            ^
```

```
void example() {
    try {
        foo();
    } catch(Exception e) {
        System.err.println("Do something else");
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

UU

Titta också på...

- Screencasten om undantagshantering och programmet ExceptionDemo i kursrepot
- Vilka typer av fördefinierade undantag som finns i Javas standardbibliotek
- Skillnaden mellan kontrollerade och okontrollerade undantag (tas bl.a. upp i screencasten ovan; på eng. checked/unchecked exceptions)
- Varför det är problematiskt att använda inre klasser (i motsats till nästlande klasser) för att definiera undantag