

## 14 Abstrakte Klassen, finale Klassen, Interfaces

- ▶ Eine **abstrakte** Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- ▶ Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls **abstrakt**.
- ▶ Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden.
- ▶ Mit abstrakten Klassen können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

## Auswertung von Ausdrücken

```
1 public abstract class Expression {
2     private int value;
3     private boolean evaluated = false;
4     public int getValue() {
5         if (!evaluated) {
6             value = evaluate();
7             evaluated = true;
8         }
9         return value;
10    }
11    abstract protected int evaluate();
12 } // end of class Expression
```

- ▶ Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- ▶ Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` — immer mit einer anderen Implementierung.

## 14 Abstrakte Klassen, finale Klassen, Interfaces

- ▶ Eine **abstrakte** Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- ▶ Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls **abstrakt**.
- ▶ Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden.
- ▶ Mit abstrakten Klassen können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

## Abstrakte Methoden und Klassen

- ▶ Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- ▶ Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- ▶ Für die abstrakte Methode muss der vollständige Kopf angegeben werden — inklusive den Parametertypen und den (möglicherweise) geworfenen Exceptions.
- ▶ Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

## Auswertung von Ausdrücken

```
1 public abstract class Expression {
2     private int value;
3     private boolean evaluated = false;
4     public int getValue() {
5         if (!evaluated) {
6             value = evaluate();
7             evaluated = true;
8         }
9         return value;
10    }
11    abstract protected int evaluate();
12 } // end of class Expression
```

- ▶ Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- ▶ Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` — immer mit einer anderen Implementierung.

## Beispiel

- ▶ Die Methode `evaluate()` soll den Ausdruck auswerten.
- ▶ Die Methode `getValue()` speichert das Ergebnis in dem Attribut `value` ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

### Beispiel für einen Ausdruck:

```
1 public final class Const extends Expression {
2     private int n;
3     public Const(int x) { n = x; }
4     protected int evaluate() {
5         return n;
6     } // end of evaluate()
7 } // end of class Const
```

## Abstrakte Methoden und Klassen

- ▶ Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- ▶ Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- ▶ Für die abstrakte Methode muss der vollständige Kopf angegeben werden — inklusive den Parametertypen und den (möglicherweise) geworfenen Exceptions.
- ▶ Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

## Das Schlüsselwort `final`

- ▶ Der Ausdruck `Const` benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- ▶ Die Klasse ist als `final` deklariert.
- ▶ Zu als `final` deklarierten Klassen dürfen keine Unterklassen deklariert werden!!!
- ▶ Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als `final` deklariert werden. . .
- ▶ Statt ganzer Klassen können auch einzelne Variablen oder Methoden als `final` deklariert werden.
- ▶ Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- ▶ Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden ⇒ **Konstanten**.

## Beispiel

- ▶ Die Methode `evaluate()` soll den Ausdruck auswerten.
- ▶ Die Methode `getValue()` speichert das Ergebnis in dem Attribut `value` ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

### Beispiel für einen Ausdruck:

```
1 public final class Const extends Expression {
2     private int n;
3     public Const(int x) { n = x; }
4     protected int evaluate() {
5         return n;
6     } // end of evaluate()
7 } // end of class Const
```

```
1 public final class Add extends Expression {
2     private Expression left, right;
3     public Add(Expression l, Expression r) {
4         left = l; right = r;
5     }
6     protected int evaluate() {
7         return left.getValue() + right.getValue();
8     } // end of evaluate()
9 } // end of class Add
10 public final class Neg extends Expression {
11     private Expression arg;
12     public Neg(Expression a) { arg = a; }
13     protected int evaluate() { return -arg.getValue(); }
14 } // end of class Neg
```

- ▶ Der Ausdruck **Const** benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- ▶ Die Klasse ist als **final** deklariert.
- ▶ Zu als **final** deklarierten Klassen dürfen keine Unterklassen deklariert werden!!!
- ▶ Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als final deklariert werden. . .
- ▶ Statt ganzer Klassen können auch einzelne Variablen oder Methoden als **final** deklariert werden.
- ▶ Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- ▶ Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden ⇒ **Konstanten**.

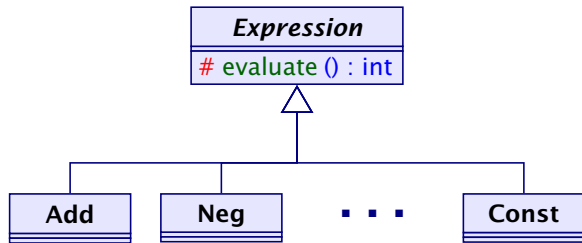
## main()

```
1 public static void main(String[] args) {
2     Expression e = new Add (
3         new Neg (new Const(8)),
4         new Const(16));
5     System.out.println(e.getValue());
6 }
```

- ▶ Die Methode `getValue()` ruft eine Methode `evaluate()` sukzessive für jeden Teilausdruck von `e` auf.
- ▶ Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- ▶ Das nennt man auch **dynamische Bindung**.

## Andere Ausdrücke

```
1 public final class Add extends Expression {
2     private Expression left, right;
3     public Add(Expression l, Expression r) {
4         left = l; right = r;
5     }
6     protected int evaluate() {
7         return left.getValue() + right.getValue();
8     } // end of evaluate()
9 } // end of class Add
10 public final class Neg extends Expression {
11     private Expression arg;
12     public Neg(Expression a) { arg = a; }
13     protected int evaluate() { return -arg.getValue(); }
14 } // end of class Neg
```



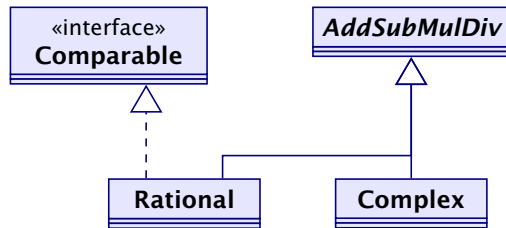
Leider (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren. . .

```
1 public static void main(String[] args) {
2     Expression e = new Add (
3         new Neg (new Const(8)),
4         new Const(16));
5     System.out.println(e.getValue());
6 }
```

- ▶ Die Methode `getValue()` ruft eine Methode `evaluate()` sukzessive für jeden Teilausdruck von `e` auf.
- ▶ Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- ▶ Das nennt man auch **dynamische Bindung**.



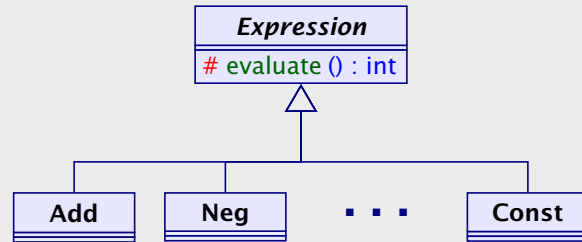
## Beispiel



**AddSubMulDiv** = Objekte mit Operationen `add()`, `sub()`, `mul()`, und `div()`

**Comparable** = Objekte, die eine `compareTo()`-Operation besitzen.

## Klassenhierarchie

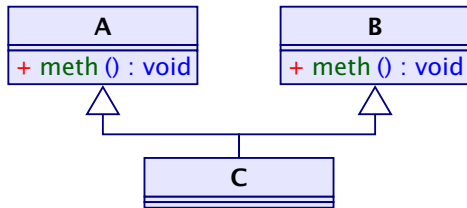


**Leider** (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren. . .

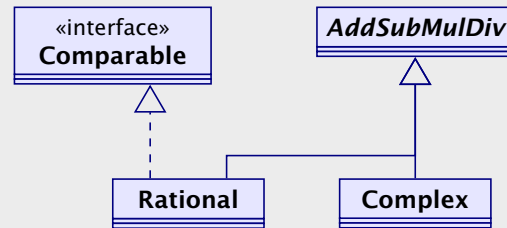
# Mehrfachvererbung

Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:

- ▶ Auf welche Klasse bezieht sich `super`?
- ▶ Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren?



# Beispiel

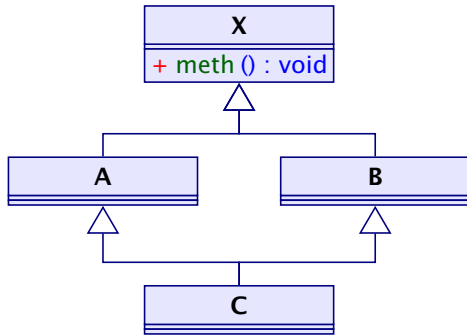


`AddSubMulDiv` = Objekte mit Operationen `add()`, `sub()`, `mul()`, und `div()`

`Comparable` = Objekte, die eine `compareTo()`-Operation besitzen.

## Mehrfachvererbung

- ▶ Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren? Insbesondere

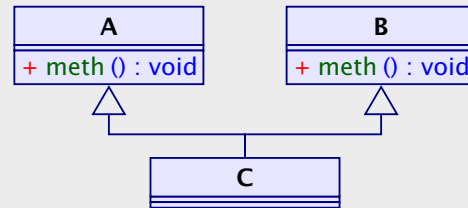


**deadly diamond of death**

## Mehrfachvererbung

Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:

- ▶ Auf welche Klasse bezieht sich `super`?
- ▶ Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren?



# Interfaces

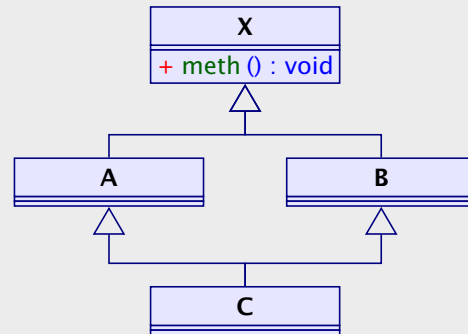
- ▶ Kein Problem entsteht, wenn die Objekt-Methode `meth()` in allen Oberklassen abstrakt ist,
- ▶ oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt.

Ein **Interface** kann aufgefasst werden als eine abstrakte Klasse, wobei:

- ▶ alle Objekt-Methoden abstrakt sind;
- ▶ es keine Klassen-Methoden gibt;
- ▶ alle Variablen **Konstanten** sind.

# Mehrfachvererbung

- ▶ Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren? Insbesondere



**deadly diamond of death**

## Beispiel

```
1 public interface Comparable {  
2     int compareTo(Object x);  
3 }
```

- ▶ **Object** ist die gemeinsame Oberklasse aller Klassen.
- ▶ Methoden in Interfaces sind automatisch Objektmethoden und **public**.
- ▶ Es muss eine **Obermenge** der in Implementierungen geworfenen Exceptions angegeben werden.
- ▶ Evt. vorkommende Konstanten sind automatisch **public static**.

## Interfaces

- ▶ Kein Problem entsteht, wenn die Objekt-Methode **meth()** in allen Oberklassen abstrakt ist,
- ▶ oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt.

Ein **Interface** kann aufgefasst werden als eine abstrakte Klasse, wobei:

- ▶ alle Objekt-Methoden abstrakt sind;
- ▶ es keine Klassen-Methoden gibt;
- ▶ alle Variablen **Konstanten** sind.

## Beispiel

```
1 public class Rational extends AddSubMulDiv
2     implements Comparable {
3     private int zaehler, nenner;
4     public int compareTo(Object cmp) {
5         Rational fraction = (Rational) cmp;
6         long left = (long)zaehler * (long)fraction.nenner;
7         long right = (long)nenner * (long)fraction.zaehler;
8         return left == right ? 0:
9             left < right ? -1:
10                1;
11     } // end of compareTo
12     ...
13 } // end of class Rational
```

## Beispiel

```
1 public interface Comparable {
2     int compareTo(Object x);
3 }
```

- ▶ **Object** ist die gemeinsame Oberklasse aller Klassen.
- ▶ Methoden in Interfaces sind automatisch Objektmethoden und **public**.
- ▶ Es muss eine **Obermenge** der in Implementierungen geworfenen Exceptions angegeben werden.
- ▶ Evt. vorkommende Konstanten sind automatisch **public static**.

## Erläuterungen

- ▶ `class A extends B implements B1, B2, ..., Bk {...}` gibt an, dass die Klasse `A` als Oberklasse `B` hat und zusätzlich die Interfaces `B1, B2, ..., Bk` unterstützt, d.h. passende Objektmethoden zur Verfügung stellt.
- ▶ `Java` gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- ▶ Die Konstanten des Interface können in implementierenden Klassen **direkt** benutzt werden.
- ▶ Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- ▶ Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- ▶ Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

## Beispiel

```
1 public class Rational extends AddSubMulDiv
2     implements Comparable {
3     private int zaehler, nenner;
4     public int compareTo(Object cmp) {
5         Rational fraction = (Rational) cmp;
6         long left = (long)zaehler * (long)fraction.nenner;
7         long right = (long)nenner * (long)fraction.zaehler;
8         return left == right ? 0:
9             left < right ? -1:
10                1;
11     } // end of compareTo
12     ...
13 } // end of class Rational
```

## Erläuterungen

- ▶ Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- ▶ Erweiternde Interfaces können Konstanten umdefinieren...
- ▶ Kommt eine Konstante gleichen Namens `const` in verschiedenen implementierten Interfaces `A` und `B` vor, kann man sie durch `A.const` und `B.const` unterscheiden.

### Beispiel:

```
1 public interface Countable extends Comparable, Cloneable {  
2     Countable next();  
3     Countable prev();  
4     int number();  
5 }
```

## Erläuterungen

- ▶ `class A extends B implements B1, B2, ..., Bk {...}` gibt an, dass die Klasse `A` als Oberklasse `B` hat und zusätzlich die Interfaces `B1, B2, ..., Bk` unterstützt, d.h. passende Objektmethoden zur Verfügung stellt.
- ▶ `Java` gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- ▶ Die Konstanten des Interface können in implementierenden Klassen **direkt** benutzt werden.
- ▶ Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- ▶ Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- ▶ Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

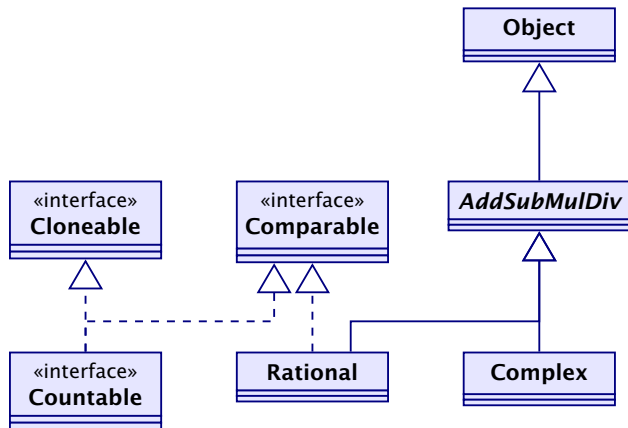


- ▶ Das Interface `Countable` umfasst die (beide vordefinierten) Interfaces `Comparable` und `Cloneable`.
- ▶ Das vordefinierte Interface `Cloneable` verlangt eine Objektmethode `public Object clone()` die eine Kopie des Objekts anlegt.
- ▶ Eine Klasse, die `Countable` implementiert, muss über die Objektmethoden `compareTo()`, `clone()`, `next()`, `prev()` und `number()` verfügen.

- ▶ Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- ▶ Erweiternde Interfaces können Konstanten umdefinieren...
- ▶ Kommt eine Konstante gleichen Namens `const` in verschiedenen implementierten Interfaces A und B vor, kann man sie durch `A.const` und `B.const` unterscheiden.

### Beispiel:

```
1 public interface Countable extends Comparable, Cloneable {  
2     Countable next();  
3     Countable prev();  
4     int number();  
5 }
```



- ▶ Das Interface `Countable` umfasst die (beide vordefinierten) Interfaces `Comparable` und `Cloneable`.
- ▶ Das vordefinierte Interface `Cloneable` verlangt eine Objektmethode `public Object clone()` die eine Kopie des Objekts anlegt.
- ▶ Eine Klasse, die `Countable` implementiert, muss über die Objektmethoden `compareTo()`, `clone()`, `next()`, `prev()` und `number()` verfügen.