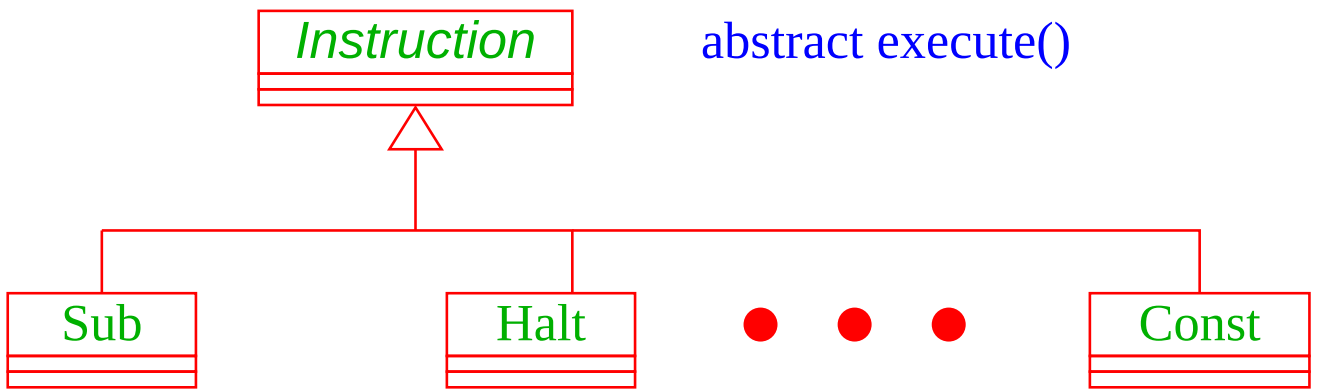
- In der Klasse `Halt` wird die Objekt-Methode `halted()` neu definiert.
- Achtung bei `Sub` mit der Reihenfolge der Argumente!

... die Funktion `main()` einer Klasse `Jvm`:

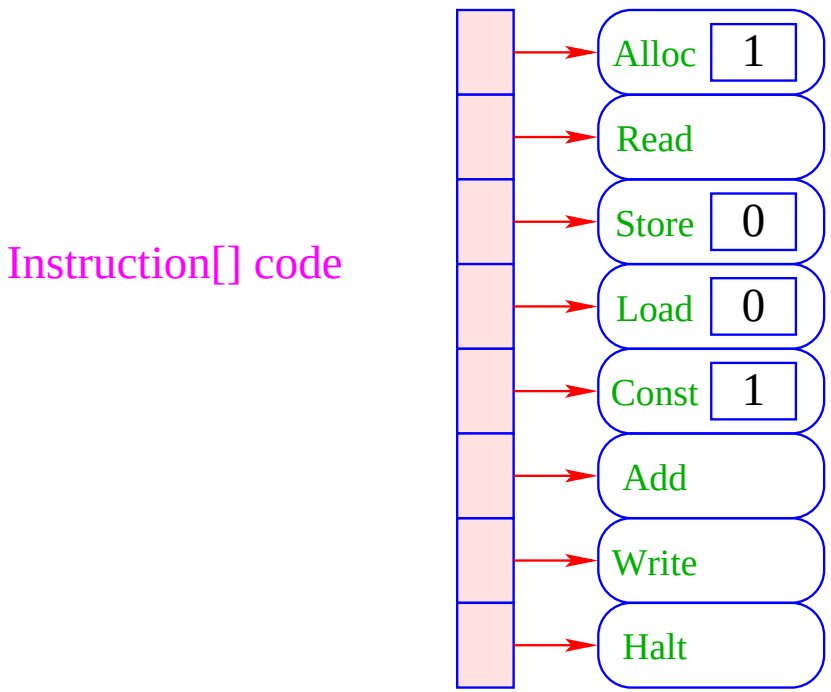
```
public static void main(String[] args) {
   Instruction[] code = getCode();
   Instruction ir = code[0];
   while(!ir.halted())
      ir = code[ir.execute()];
   }
```

- Für einen vernünftigen Interpreter müssen wir natürlich auch in der Lage sein, ein JVM-Programm einzulesen, d.h. eine Funktion `getCode()` zu implementieren...

Die abstrakte Klasse `Instruction`:



- Jede Unterklasse von `Instruction` verfügt über ihre eigene Methode `execute()`.
- In dem Feld `Instruction[] code` liegen Objekte aus solchen Unterklassen.

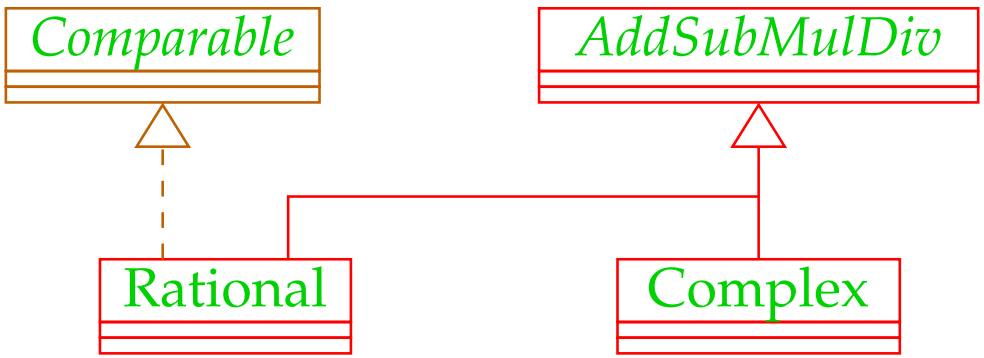


- Die Interpreter-Schleife ruft eine Methode `execute()` für die Elemente dieses Felds auf.
- Welche konkrete Methode dabei jeweils aufgerufen wird, hängt von der konkreten Klasse des jeweiligen Objekts ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

- Die Interpreter-Schleife ruft eine Methode `execute()` für die Elemente dieses Felds auf.
- Welche konkrete Methode dabei jeweils aufgerufen wird, hängt von der konkreten Klasse des jeweiligen Objekts ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

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

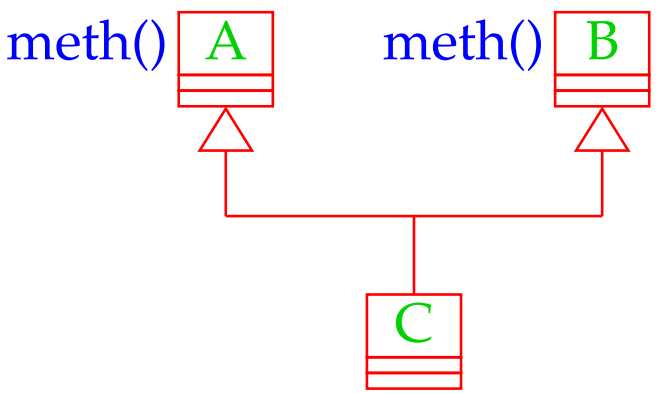
Beispiel:



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

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

- Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:
  - Auf welche Klasse bezieht sich `super` ?
  - Welche Objekt-Methode `meth()` ist gemeint, wenn wenn mehrere Oberklassen `meth()` implementieren ?



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

- 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:

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

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

Beispiel (Forts.):

```
public class Rational extends AddSubMulDiv
                      implements Comparable {
private int zaehler, nenner;
public int compareTo(Object cmp) {
   Rational fraction = (Rational) cmp;
   long left = zaehler * fraction.nenner;
   long right = nenner * fraction.zaehler;
   if (left == right) return 0;
   else if (left < right) return -1;
   else return 1;
   } // end of compareTo
   ...
}    // end of class Rational
```

- `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 Objekt-Methoden 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 :-(

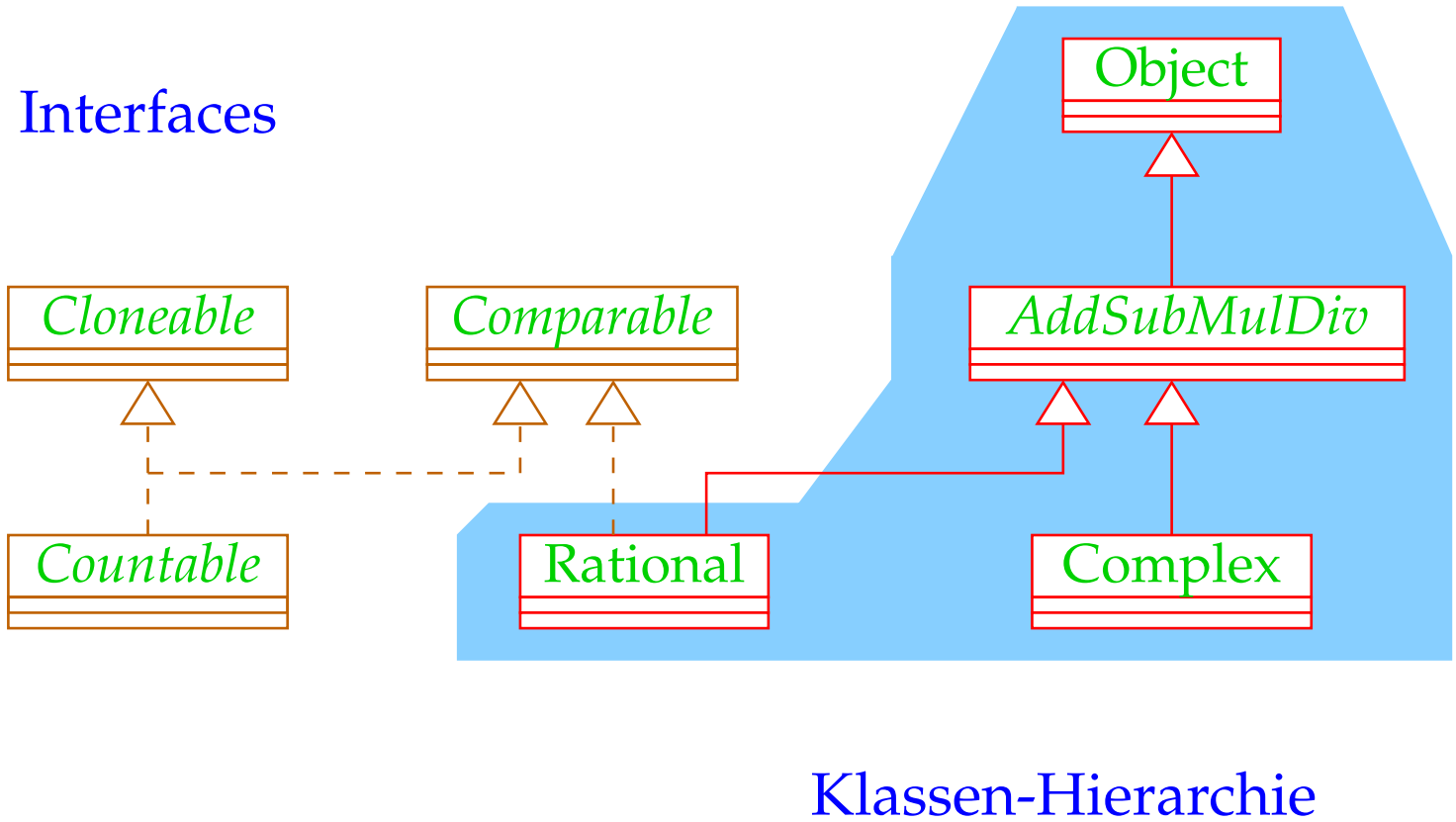
- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten auch umdefinieren...
- (kommen Konstanten gleichen Namens in verschiedenen implementierten Interfaces vor, gibt's einen Laufzeit-Fehler...)

Beispiel (Forts.):

```
public interface Countable extends Comparable, Cloneable {
   Countable next();
   Countable prev();
   int number();
   }
```

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

Übersicht:


Interfaces
Object
Cloneable
Comparable
AddSubMulDiv
Countable
Rational
Complex
Klassen-Hierarchie

14 Polymorphie

Problem:

- Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur grade nochmal definieren :-(

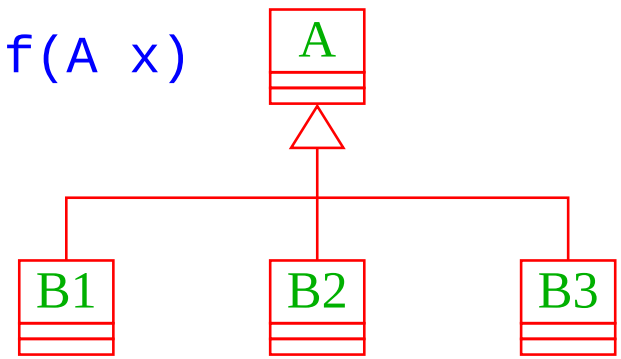
14.1 Unterklassen-Polymorphie

Idee:

- Eine Operation meth `(` A `x)` lässt sich auch mit einem Objekt aus einer Unterklasse von A aufrufen !!!
- Kennen wir eine gemeinsame Oberklasse Base für alle möglichen aktuellen Parameter unserer Operation, dann definieren wir meth einfach für Base ...
- Eine Funktion, die für mehrere Argument-Typen definiert ist, heißt auch polymorph.

Statt:

... besser:



Fakt:

- Die Klasse `Object` ist eine gemeinsame Oberklasse für alle Klassen.
- Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.
- Einige nützliche Methoden der Klasse `Object` :
  - `String toString()` liefert (irgendeine) Darstellung als `String`;
  - `boolean equals(Object obj)` testet auf Objekt-Identität oder Referenz-Gleichheit:

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

...

- `int hashCode()` liefert eine eindeutige Nummer für das Objekt.
- ... viele weitere geheimnisvolle Methoden, die u.a. mit ↑paralleler Programm-Ausführung zu tun haben :-)

Achtung:

`Object`-Methoden können aber (und sollten evt.:-) in Unterklassen durch geeignetere Methoden überschrieben werden.

Beispiel:

```
public class Poly {
   public String toString() { return "Hello"; }
}
public class PolyTest {
   public static String addWorld(Object x) {
      return x.toString()+" World!";
   }
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(addWorld(x)+"\n");
   }
}
```

... liefert:

```
Hello World!
```

- Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- Die Klasse `Poly` ist eine Unterklasse von `Object`.
- Einer Variable der Klasse A kann ein Objekt jeder Unterklasse von A zugewiesen werden.
- Darum kann `x` das neue `Poly`-Objekt aufnehmen :-)

Bemerkung:

- Die Klasse `Poly` enthält keinen explizit definierten Konstruktor.
- Eine Klasse A, die keinen anderen Konstruktor besitzt, enthält implizit den trivialen Konstruktor `public A () {}`.

Achtung:

```
public class Poly {
   public String greeting() {
      return "Hello";
   }
}
public class PolyTest {
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(x.greeting()+" World!\n");
   }
}
```

... liefert ...

... einen Compiler-Fehler:

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                                   ^
1 error
```

- Die Variable `x` ist als `Object` deklariert.
- Der Compiler weiss nicht, ob der aktuelle Wert von `x` ein Objekt aus einer Unterklasse ist, in welcher die Objekt-Methode `greeting()` definiert ist.
- Darum lehnt er dieses Programm ab.

Ausweg:

- Benutze einen expliziten cast in die entsprechende Unterklasse!

```
public class Poly {
   public String greeting() { return "Hello"; }
}
public class PolyTest {
   public void main(String[] args) {
      Object x = new Poly();
      if (x instanceof Poly)
         System.out.print(((Poly) x).greeting()+" World!\n");
      else
         System.out.print("Sorry: no cast possible!\n");
   }
}
```

Fazit:

- Eine Variable `x` einer Klasse A kann Objekte `b` aus sämtlichen Unterklassen B von A aufnehmen.
- Durch diese Zuweisung vergisst Java die Zugehörigkeit zu B, da Java alle Werte von `x` als Objekte der Klasse `A` behandelt.
- Mit dem Ausdruck `x instanceof` B können wir zur Laufzeit die Klassenzugehörigkeit von `x` testen ;-)
- Sind wir uns sicher, dass `x` aus der Klasse B ist, können wir in diesen Typ casten.
- Ist der aktuelle Wert der Variablen `x` bei der Überprüfung tatsächlich ein Objekt (einer Unterklasse) der Klasse B, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine ↑Exception ausgelöst :-)

Beispiel: Unsere Listen

```
public class List {
   public Object info;
   public List next;
   public List(Object x, List l) {
      info=x; next=l;
   }
   public void insert(Object x) {
      next = new List(x,next);
   }
   public void delete() {
      if (next!=null) next=next.next;
   }
   ...
```

```
    public String toString() {
       String result = "["+info;
       for (List t=next; t!=null; t=t.next)
          result=result+", "+t.info;
       return result+"]";
    }
    ...
} // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `int`.
- Die `toString()`-Methode ruft implizit die (stets vorhandene) `toString()`-Methode für die Listen-Elemente auf.

... aber Achtung:

```
...
Poly x = new Poly();
List list = new List (x);
x = list.info;
System.out.print(x+"\n");
...
```

liefert ...

... einen Compiler-Fehler, da der Variablen `x` nur Objekte einer Unterklasse von `Poly` zugewiesen werden dürfen.

Stattdessen müssen wir schreiben:

```
...
Poly x = new Poly();
List list = new List (x);
x = (Poly) list.info;
System.out.print(x+"\n");
...
```

Das ist hässlich !!! Geht das nicht besser ???

14.2 Generische Klassen

Idee:

- Seit Version 1.5 verfügt Java über generische Klassen ...
- Anstatt das Attribut `info` als `Object` zu deklarieren, geben wir der Klasse einen Typ-Parameter `T` für `info` mit !!!
- Bei Anlegen eines Objekts der Klasse `List` bestimmen wir, welchen Typ `T` und damit `info` haben soll ...

Beispiel: Unsere Listen

```
public class List<T> {
   public T info;
   public List<T> next;
   public List (T x, List<T> l) {
      info=x; next=l;
   }
   public void insert(t x) {
      next = new List<T> (x,next);
   }
   public void delete() {
      if (next!=null) next=next.next;
   }
   ...
```

```
    public static void main (String [] args) {
       List<Poly> list = new List<Poly> (new Poly(),null);
       System.out.print (list.info.greeting()+"\n");
    }
} // end of class List
```

```
   public static void main (String [] args) {
      List<Poly> list = new List<Poly> (new Poly(),null);
      System.out.print (list.info.greeting()+"\n");
   }
} // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typ-Parameter `T` = `Poly`.
- Deshalb ist `list.info` vom Typ `Poly` :-)
- Folglich ruft `list.info.greeting()` die entsprechende Methode der Klasse `Poly` auf :-))