Review

Es wird einen Review am 15.2.2011 geben.

Letzte Vorlesung

- **JButtonDemo**
  - neue Funktionalität durch Komposition
  - Illustration innerer Klassen
  - Callbacks und Event Handling in Fenstersystemen
- **MyComponent**
  - neue Funktionalität durch Vererbung und Überschreibung (inheritance and overriding of methods)
  - Illustration von Graphik in Fenstersystemen

SUBTYPEN UND VERERBUNG

Subtypen

- **wenn B ein Subtyp von A ist, dann dürfen Objekte von Typ B überall dor verwendet werden, wo Objekte von Typ A verwendet werden dürfen**
- **Subtyprelationen entstehen in Java durch Vererbung**

```
class A { ... }

class B extends A { ... }

class Main {

    void f(A a) { ... }

    void g() {
        f(new A()); // deklarierter Typ
        f(new B()); // Aufruf mit Subtyp: zulässig
    }
}
```

Wichtige Konzepte

- **Schnittstellenvererbung**
  - class B implements A { ... }
  - B verhält sich wie A
  - keine Implementierung übernommen
- **Implementierungsvererbung**
  - class B extends A { ... }
  - Implementierung wird von der Superklasse übernommen
  - Implementierungen können überschrieben werden
  - Implementierung wird durch dynamische Bindung ausgewählt
  - in Java: nur eine Superklasse
- **in beiden Fällen: Subtypenbeziehung**

Implementierung ohne Subtypen?

- Vererbung von Implementierung bringt zwigend Subtypenbeziehung in Java
- Was tun wenn wir das nicht wollen?

Aggregation / Delegation

- **Klasse B soll Funtionalität von Klasse A erhalten, ohne ein Subtyp zu sein.**

```
class Array<E> {
    Array(Iterator<E> c) {
      ...
    }

    int size() {
        ...
    }

    E at(int i) {
        ...
    }

    void put(int i,E e) {
        ...
    }
}
```

```
class ROArray<E> {
    private Array<E> a;

    ROArray(Iterator<E> c) {
      a = new ROArray<E>(c);
    }

    int size() {
        return a.size();
    }

    E at(int i) {
        return a.at(i);
    }
    // (no put method)
}
```

Andere Eigenschaften

- **Vererbung darf keine Zyklen enthalten**
  - ~~class B extends A~~
  - ~~class C extends B~~
  - ~~class A extends C~~
- **Felder**
  - in Java: wenn B Subtyp von A ist, dann ist B[] ein Subtyp von A[]
  - Feldtypen sind kovariant
  - kann zu Fehlern führen
  - dies gilt nicht für Array<B> und Array<A>

Typfehler bei Feldern durch Kovarianz

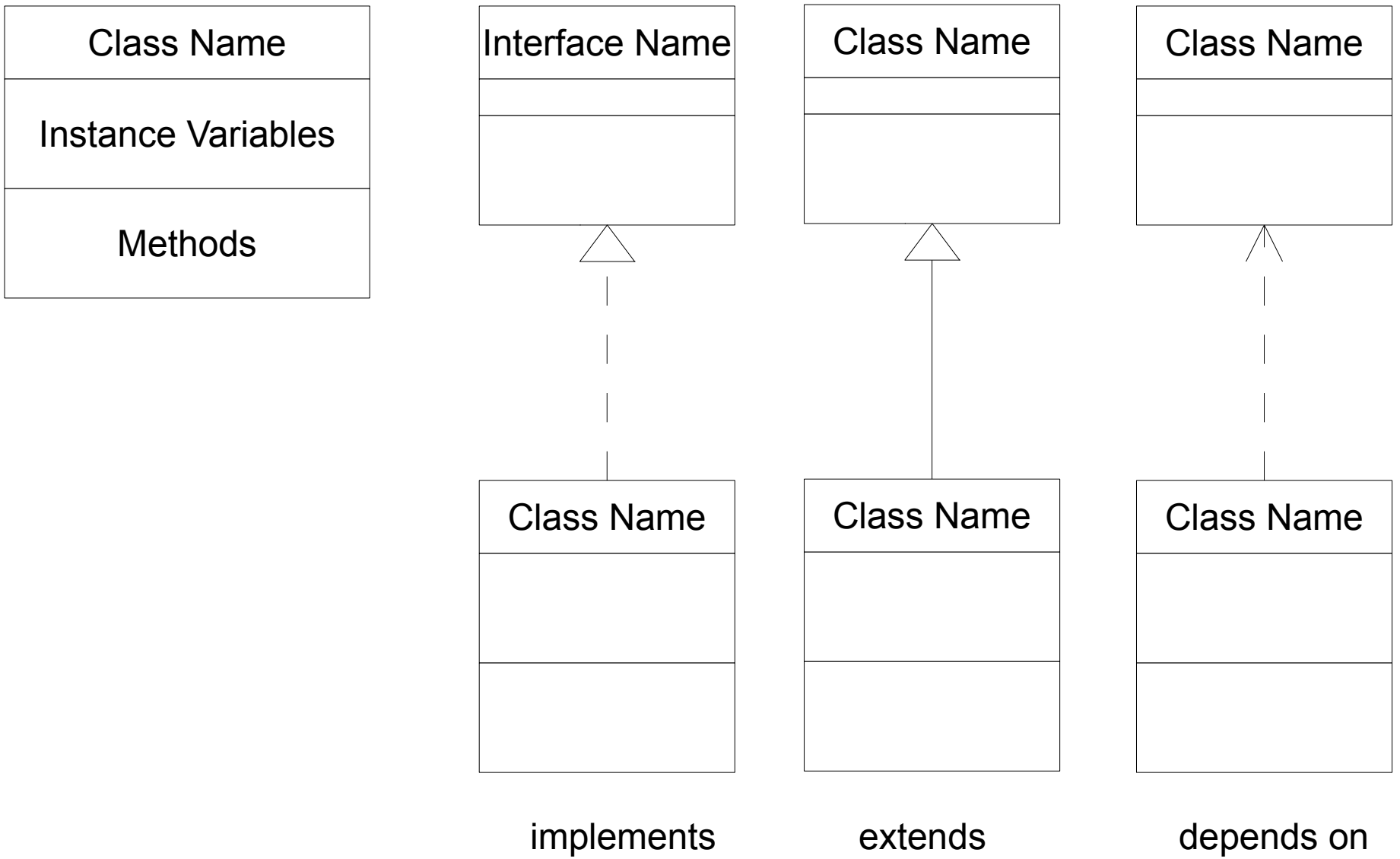
```
class A {}
class B {}

class Cov {
    public static void main(String[] args) {
        A[] as = new A[10];
        Object[] os = as;
        os[0] = new B();
    }
}
```

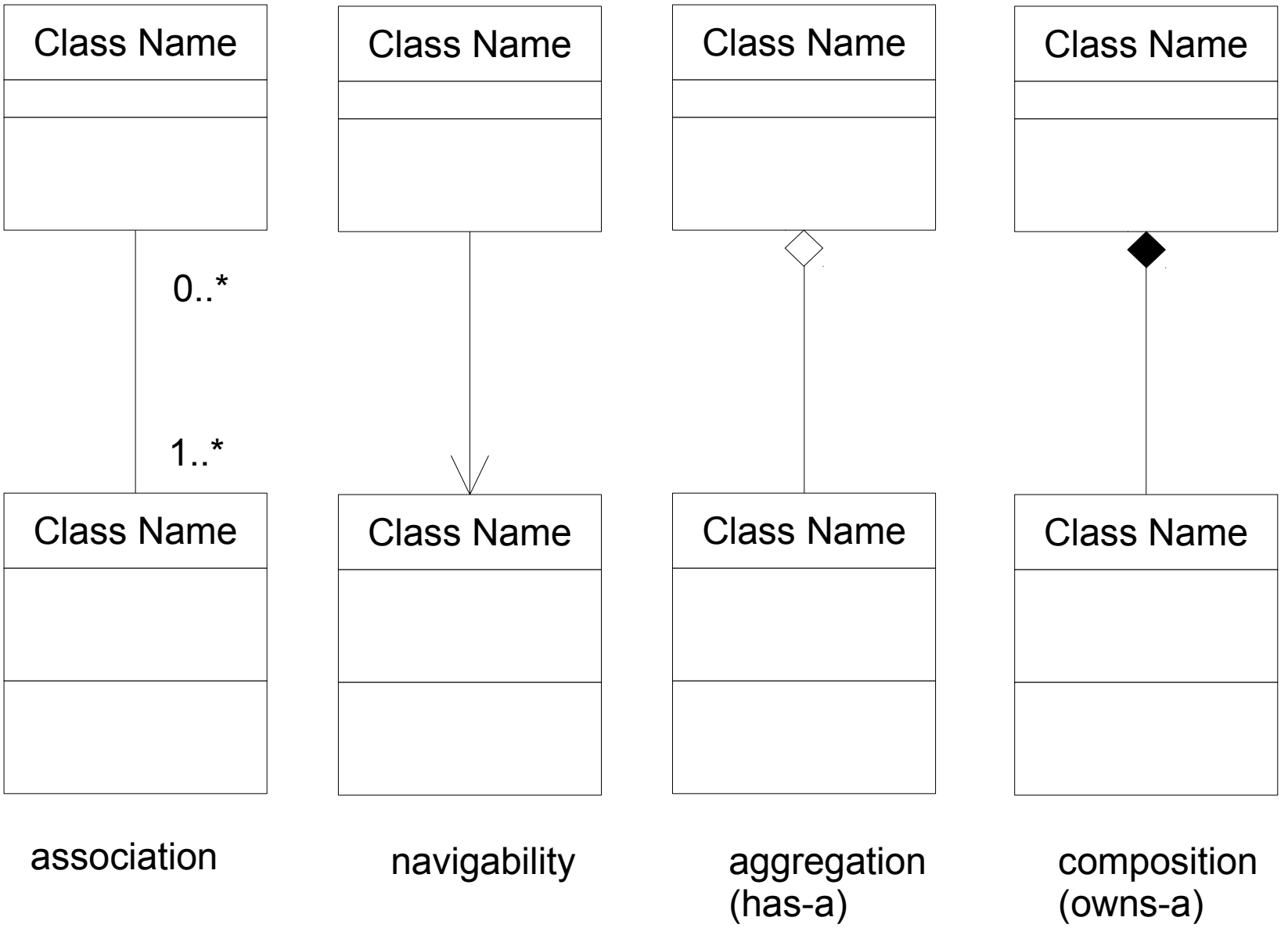
```
$ javac Cov.java
$ java Cov
Exception in thread "main"
java.lang.ArrayStoreException: B
   at Cov.main(Cov.java:8)
$
```

UML

Abhängigkeiten von Klassen



Abhängigkeiten von Instanzen



EXCEPTIONS

```
class Convert {
    static int parseInt(String s) {
        int result = 0;
        for(int i=0;i<s.length();i++) {
            int digit = (int)s.charAt(i)-(int)'0';
            if(digit<0||digit>9) return -99999;
            result = 10*result + digit;
        }
        return result;
    }
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        for(;;) {
            String line = in.readLine();
            int input = parseInt(line);
            if(input==-99999) {
                System.out.println(“try again”);
                continue;
            }
            int output = (input-32)*5/9;
            System.out.println(""+input+"F = "+output+"C");
        }
    }
```

```
class Convert {
    static int parseInt(String s) throws NumberFormatException {
        int result = 0;
        for(int i=0;i<s.length();i++) {
            int digit = (int)s.charAt(i)-(int)'0';
            if(digit<0||digit>9) throw new NumberFormatException();
            result = 10*result + digit;
        }
        return result;
    }
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        for(;;) {
            String line = in.readLine();
            int input;
            try { input = parseInt(line); }
            catch(NumberFormatException e) {
                System.out.println("try again");
                continue;
            }
            int output = (input-32)*5/9;
            System.out.println(""+input+"F = "+output+"C");
        }
    }
}
```

Java exceptions

- **Eigenschaften**
  - exceptions sind alternative Rückkehr aus Funktion
  - hauptsächlich für Fehlerbehandlung benutzt
- **throwing**
  - **throw x** — where x is some object
  - x must be an instance of a class that derives from Exception
  - often: **throw new SomeException(information)**
  - beendet Funktionsausführung sofort
- **catching**
  - **try { ... } catch(ClassName variable) { ... } finally { ... }**
- **Deklaration**
  - exceptions müssen i.A. deklariert werden

alles zusammen...

```
class MyException extends Exception {
}

void f() throws MyException {
    ...
    throw new MyException();
    ...
}

void g() {
    try {
        ... f() ...
    } catch(MyException e) {
        // aufgerufen wenn MyException auftritt
    } finally {
        // immer aufgerufen
    }
}
```

propagieren lassen

```
void f() throws MyException {
    ...
    throw new MyException();
    ...
}

void g() throws MyException {
    ... f() ...
}
```

Java exceptions

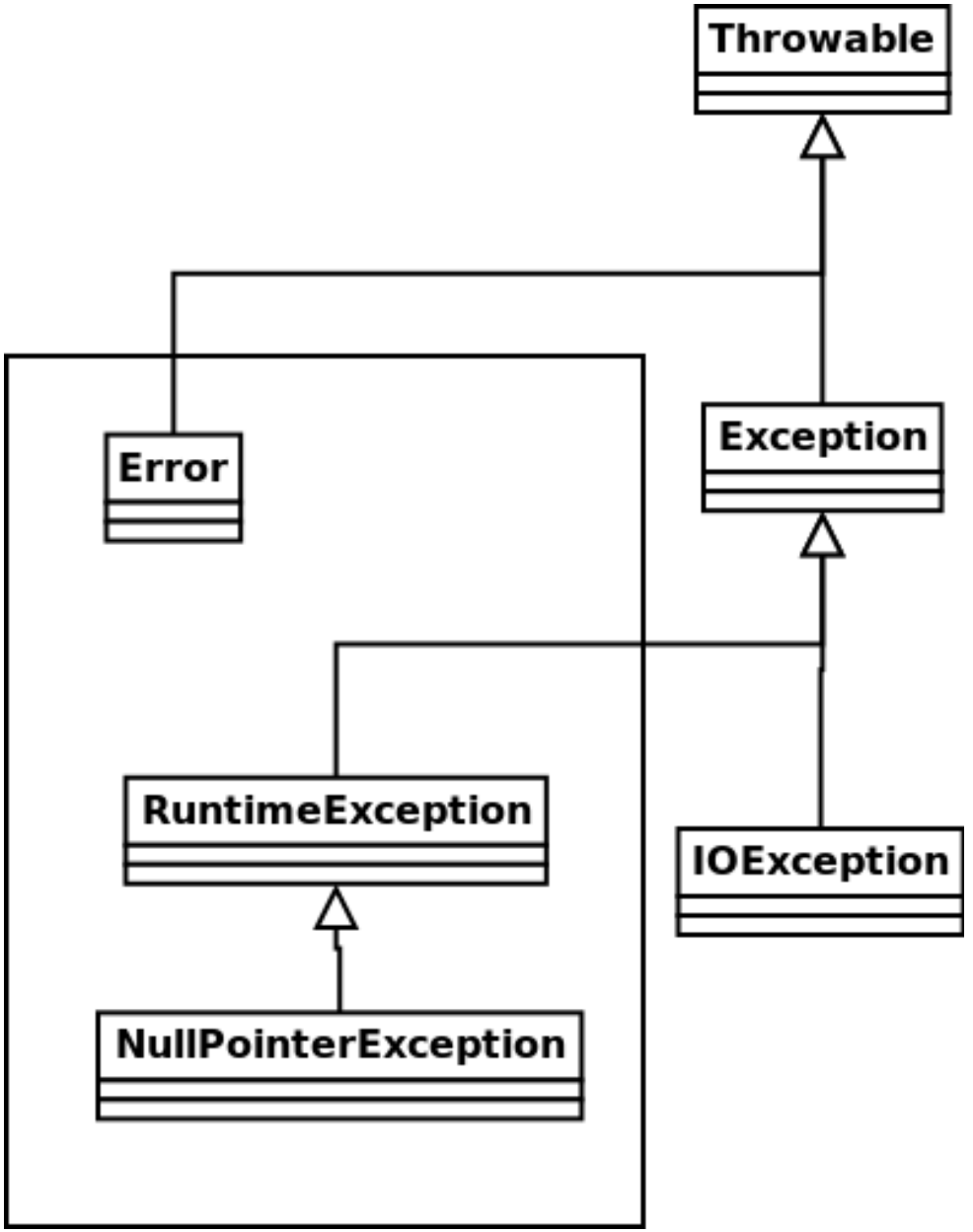
```
class Throw {
    static void f(int i) throws Exception {
        System.out.println("> "+i);
        if(i==0) throw new Exception();
        f(i-1);
        System.out.println("< "+i);
    }
    public static void main(String[] args)
                                throws Exception {
        f(5);
    }
}
```

```
$ java Throw
> 5
> 4
> 3
> 2
> 1
> 0
Exception in thread "main"
java.lang.Exception
    at Throw.f(Throw.java:4)
    at Throw.f(Throw.java:5)
    at Throw.f(Throw.java:5)
    at Throw.f(Throw.java:5)
    at Throw.f(Throw.java:5)
    at Throw.f(Throw.java:5)
    at Throw.main(Throw.java:9)
$
```

- **Beachte**
  - sofortige Beendigung
  - keine Rückkehr
  - “Backtrace” auf Konsole

Java exceptions

- Java exceptions sind alle Subklassen von Throwable
- Java exceptions müssen deklariert werden, bis auf Subklassen von Error und RuntimeException
- Error ist für schwere, non-recoverable Fehler
- Hierarchie wird in catch benutzt.
- Exception-Deklarationen sind kovariant (vgl. Felder)



RuntimeExceptions

- **die folgenden können auftreten, auch wenn nicht deklariert**
  - NullPointerException
  - NumberFormatException
  - ClassCastException
  - IndexOutOfBoundsException

  - aller anderen Subklassen von RuntimeException

Fehlerbehandlung

- behandle Fehler dort, wo es Sinn macht
- lasse andere Fehler einfach durch
- “räume auf” im “finally” Block

JAVA I/O

Stream I/O Klassen

- **InputStream**
  - FilterInputStream
    - BufferedInputStream
    - ...
  - ByteArrayInputStream
  - FileInputStream
  - ...
- **OutputStream**
  - FilterOutputStream
    - BufferedOutputStream
    - ...
  - ByteArrayOutputStream
  - FileOutputStream
  - ...

stream

- **stream = Datenströme**
- **Eingabe / Ausgabe**
  - Eingabeströme lesen Daten — read
  - Ausgabeströme schreiben Daten — write
- **Eingabeströme ähnliche wie Iteratoren**
  - “gib mir nächsten Buchstaben/Zeile/...” — read
  - “am Ende angekommen” — EOF, end-of-file
  - blockieren — Daten sind nicht immer gleich verfügbar

Streams

- **InputStream**
  - read() — ein byte Daten oder -1, kann **blockieren**
  - available() — Anzahl der verfügbaren Bytes
  - close() — beenden der Interaktion mit dem Strom
- **OutputStream**
  - write() — schreibe ein byte an Daten, kann **blockieren**
  - flush() — schreibe **gepufferte** Daten
  - close() — beende Interaktion mit dem Strom

vordefinierte Ströme

- **System.in**
  - InputStream, der Benutzereingaben an das Programm liefert
- **System.out**
  - OutputStream, der Buchstaben dem Benutzer zeigt
- **System.err**
  - OutputStream, der Buchstaben dem Benutzer zeigt; für Fehlermeldungen

Readers / Writers

- **externe Daten können als...**
  - int (Zahlen) verarbeitet werden (InputStream / OutputStream)
  - char (Buchstaben) verarbeitet werden (Reader / Writer)
- **streams können in Reader/Writer umgewandelt werden**
  - reader = new InputStreamReader(inputStream)
  - writer = new OutputStreamWrite(outputStream)

Reader / Writer

- **Reader**
  - close()
  - read() → int, read(char[] buf)
  - ready()
- **Writer**
  - close()
  - write(int), write(char[])
  - ready()

BufferedReader / BufferedWriter

- **BufferedReader**
  - Reader + readLine
  - fordert mehr Daten an, selbst wenn noch nicht benötigt
  - größere Effizienz
  - **new BufferedReader(someReader)**
- **BufferedWriter**
  - Writer + newLine()
  - Daten werden nicht unb. sofort übermittelt
  - werden in größere Pakete vereinigt um Effizienz zu erhöhen
  - **new BufferedWriter(someWriter)**

...StreamReader

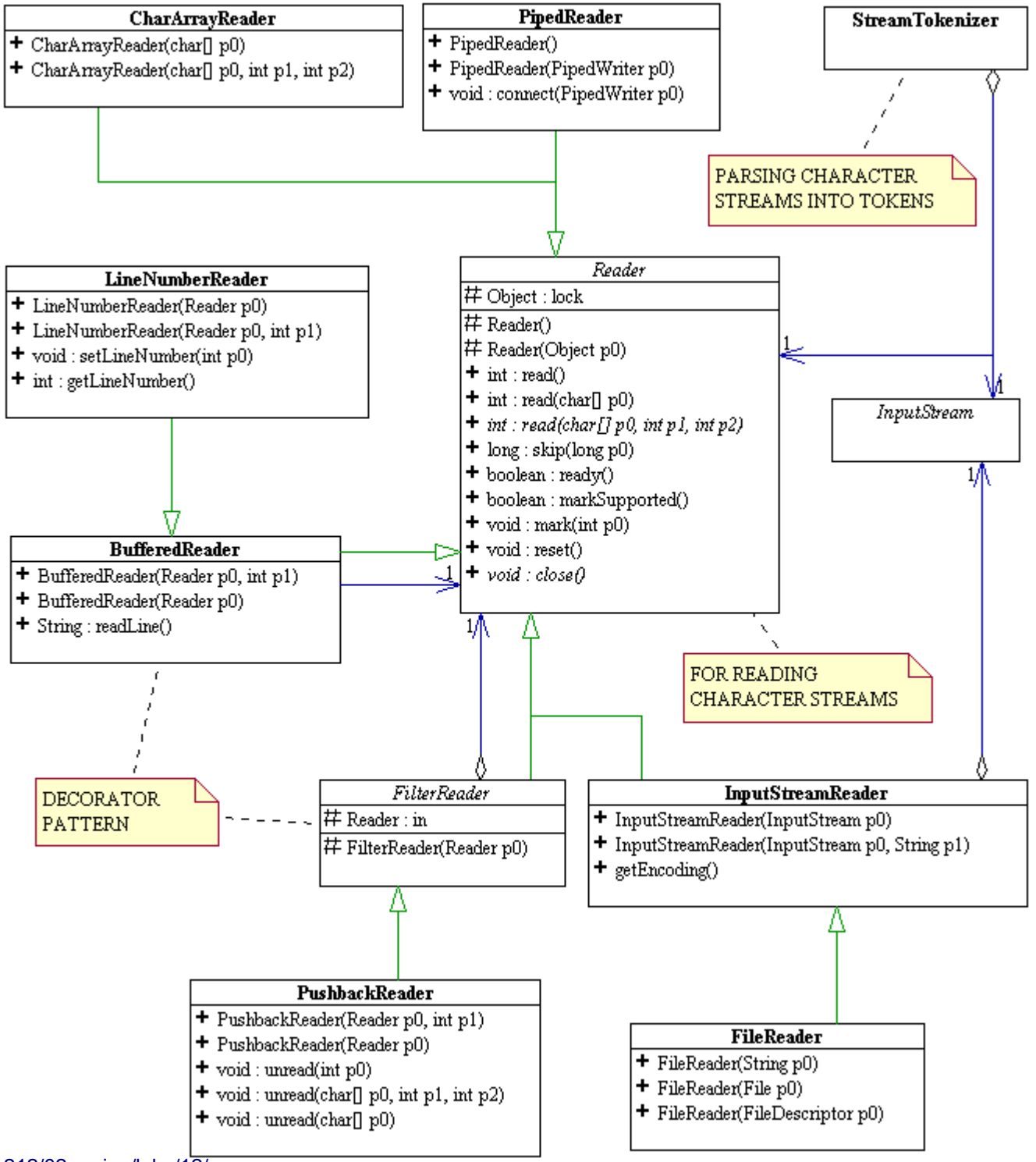
- **InputStreamReader, OutputStreamWriter**
  - müssen wissen, wie man zwischen bytes und Buchstaben konvertiert
- **Konstruktoren**
  - InputStreamReader(InputStream in,Charset cs)
  - InputStreamReader(InputStream in,CharsetEncoder enc)
  - OutputStreamWriter(OutputStream out,Charset cs)
  - OutputStreamWriter(OutputStream out,CharsetEncoder enc)
- **Charset, CharsetEncoder**
  - Abstraktionen, die byte ↔ char Konvertierungen beschreiben
  - UTF-8, Unicode, ISO-8859-15, ..

FileReader / FileWriter

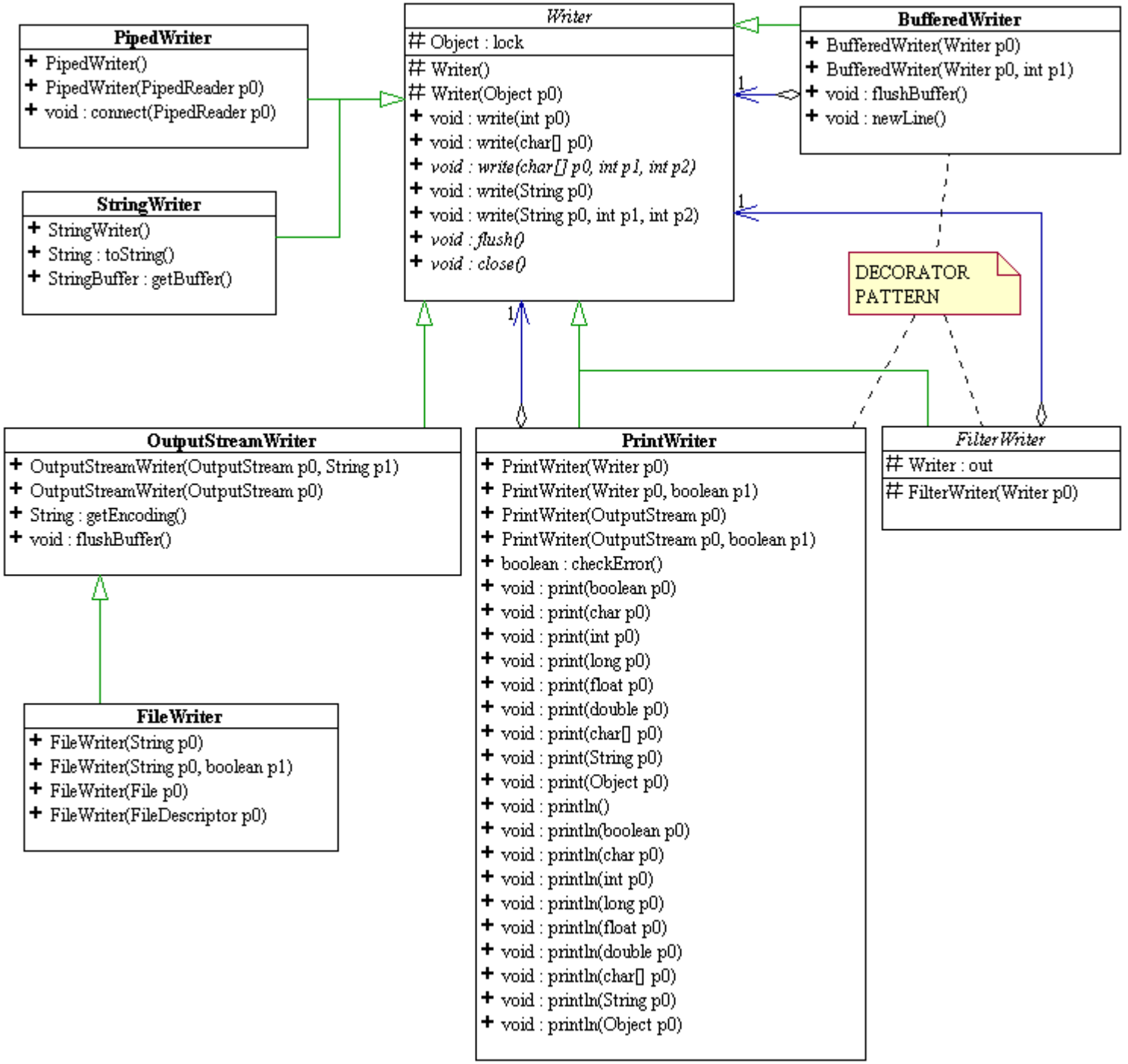
- **FileReader / FileWriter**
  - wie InputStreamReader / OutputStreamWriter
  - spezielle Konstruktoren für Dateien
- **Konstruktoren**
  - FileReader(String filename)
  - FileReader(File file)
  - FileWriter(String filename)
  - FileWriter(File file)

I/O Exceptions

- viele Java I/O Operationen können IOException auslösen
- File... Operationen können FileNotFoundException auslösen
- diese müssen deklariert und/oder behandelt werden



http://www.clear.rice.edu/comp212/02-spring/labs/12/



http://www.clear.rice.edu/comp212/02-spring/labs/12/

ELEMENTARE DATENTYPEN

KONVERTIERUNG

elementare Typen

- **zur Erinnerung...**
- **zwei Arten von Werten**
  - Werte in elementare Typen, wie **int**, **float**, ...
  - Objektreferenzen
- **Repräsentation**
  - Objektreferenzen: Speicheradressen
  - Werte elementarer Typen: Zahlen im Speicher
- **Syntax**
  - in Java: Objektsyntax nicht möglich mit elementaren Typen

Subtypbeziehungen?

- **int und float haben gemeinsame Operationen: +, *, ...**
- **bestehen Subtypenbeziehungen?**

- **Fragen:**
  - sind die Operationen von einem eine Untermenge des anderen?
  - sind die Werte des einen eine Untermenge des anderen?

Subtypbeziehungen?

- **Operationen**
  - int hat alle(?) Operationen, wie float: Arithmetik usw.
  - float hat einige Operationen mehr: sin, cos, ...
  - daher: int Subtyp von float???
- **Werte**
  - float hat: 10e23, int hat nicht so große Werte
  - ABER int hat: 2^30-7 präzise, float hat nicht diesen Wert
  - daher: keine Subtypbeziehung

Subtypbeziehungen

- **Alle Klassen sind Untertypen von Object**
- **Daher ist immer zulässig:**
  - Object object = new SomeClass();
- **elementare Datentypen sind nicht Untertypen von Object**
  - Object object = 3; /* ??? */

Experiment

```
$ cat Test.java
class Test {
    public static void main(String[] args) {
        Object obj = 3;
        System.out.println(""+obj.getClass());
    }
}
$ javac Test.java
$ java Test
class java.lang.Integer
$
```

Automatische Typkonvertierung

- **automatische Konvertierung**
  - int → java.lang.Integer (“Boxing”)
  - java.lang.Integer → int (“Unboxing”)
- **andere solche Konvertierungen**
  - byte ↔ Byte, short ↔ Short, int ↔ Integer, long ↔ Long
  - float ↔ Float, double ↔ Double
  - char ↔ Character
  - boolean ↔ Boolean

Andere Typkonvertierungen

```
$ cat Test.java
class Test {
    public static void main(String[] args) {
        int i = 0;
        float x = 0;
        i = x;
        x = i;
    }
}
$
```

```
$ javac Test.java
Test.java:5: possible loss of precision
found   : float
required: int
        i = x;
            ^
1 error
$
```

Typkonvertierungen für Werte

- **widening conversions**
  - short → int, int → long usw.
  - kein Informationsverlust
  - automatisch
- **narrowing conversions**
  - long → int, int → short
  - Informationsverlust
  - nicht automatisch
  - eingebaute Konvertierungsfunktionen (cast-Notation)
- **integer/FP conversions**
  - integer → FP automatisch
  - FP → integer nicht automatisch
  - cast-Notation für eingebaute Konvertierungsfunktionen

Typkonvertierung für Referenzen

- **Konvertierung nach Superklasse**
  - upcast
  - automatisch oder explizit mit cast-Notation
  - immer zulässig
- **Konvertierung nach Subklasse**
  - downcast
  - nur explizit mit cast-Notation
  - kann einen Fehler geben
- **diese Konvertierungen ändern nicht die Klasse des Objektes auf die die Referenz verweist**

Typkonvertierung für Referenzen

```
$ cat Test.java
class Test {
    class A { int x; }

    class B extends A { int y; }

    void f() {
        A a = new A();
        B b = new B();
        a = b;
        b = (B)a;
        b = a;
    }
}
$
```

```
$ javac Test.java
Test.java:11:
incompatible types
found   : Test.A
required: Test.B
        b = a;
            ^
1 error
$
```

dynamische Typfehler

```
class A { int x; }
class B extends A { int y; }
class C extends B { int z; }

class Test {
    public static void main(String[] args) {
        B b;
        b = new C();
        b = new B();
        b = (B)new A();
    }
}
```

```
$ javac Test.java
$ java Test
Exception in thread "main"
java.lang.ClassCastException: A cannot be cast to B
   at Test.main(Test.java:10)
$
```

instanceof

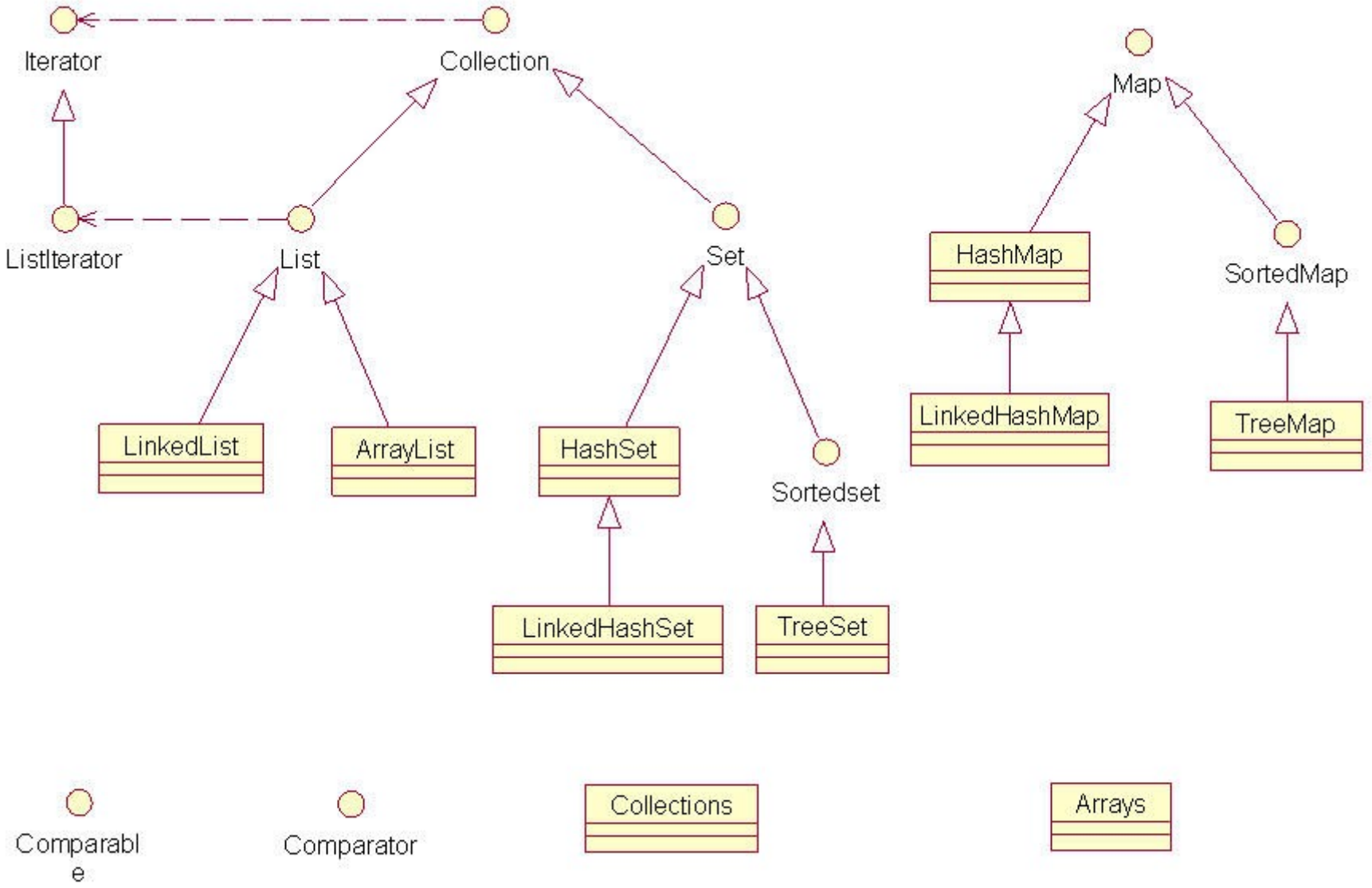
```
class A { int x; }
class B extends A { int y; }
class C extends B { int z; }

class Test {
    public static void main(String[] args) {
        B b;
        A x = new A();
        if(x instanceof B)
            b = (B)x;
        else
            b = null;
    }
}
```

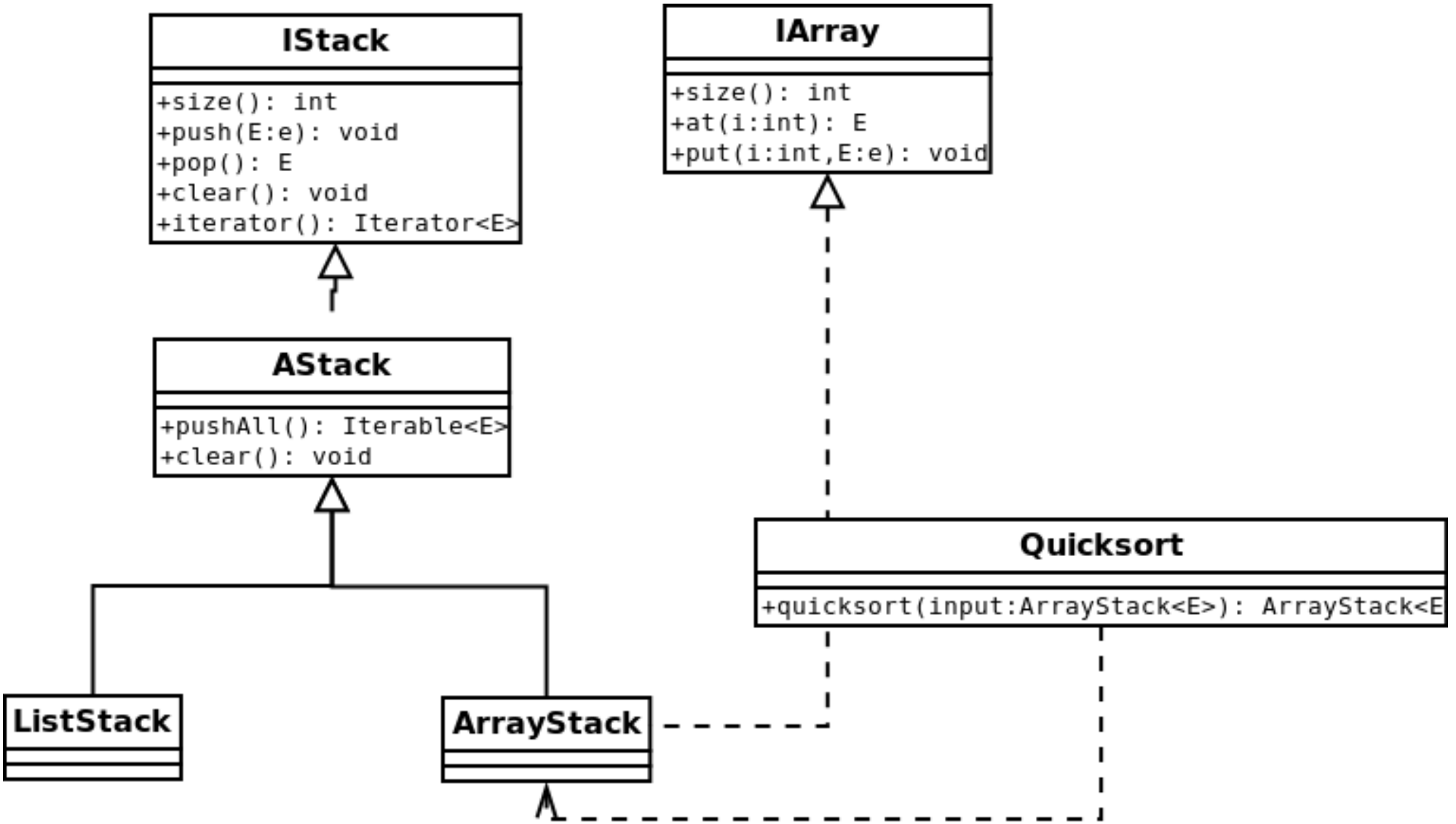
```
$ javac Test.java
$ java Test
$
```

PRAKTISCHES BEISPIEL

Java Collection Hierarchie



Unsere MiniCollection



Beachte

- Interface, Abstract Class, Implementation
- Parametrisierung
- UML diagram
- Comparable
- build files
- unit tests