C.6 Fundamentale Konzepte des objektorientierten Paradigmas

Eine Reihe programmiersprachlicher Konzepte, die unabhängig von den Ideen zur objektorientierten Programmierung entstanden sind, sind für das objektorientierte Paradigma von grundlegender Bedeutung:

- Abstraktion
  - Kapselung
  - Datenabstraktion in OOP
- Modularisierung
- Hierarchie

Weitere allgemeine programmiersprachliche Konzepte ergänzen viele objektorientierte Programmierumgebungen. Teilweise erhielten solche Konzepte auch erst durch die Verbindung mit dem objektorientierten Paradigma praktische Bedeutung.

- Typisierung
  - Typhierarchie
  - Polymorphismus
  - Generizität
- Nebenläufigkeit
- Persistenz

1 Abstraktion

Grundlegendes Konzept zur Lösung komplexer Probleme

- Für Zusammenhänge relevante Aspekte hervorheben
- Details vernachlässigen
  - Abstraktion hilft dem Software-Entwickler, sich auf das zu lösende Problem zu konzentrieren und sich den Blick auf die eigentlichen Probleme nicht durch Implementierungsdetails zu verstellen!

➔ Objektorientierung

- wichtig:
  - ➤ Signatur eines Objekts
  - ➤ Semantik eines Objekts

  } Sicht von aussen

  ➥ contract model: Sicht von aussen = Vertrag mit anderen Objekten

  - Signatur: Beschreibung der Objektschnittstellen — Methoden, Parameter, Ergebnisse
  - Semantik: Auswirkung von Methodenaufrufen

    Eine Klasse realisiert die Implementierung einer Abstraktion. Für den Anwender dieser Klasse — bzw. ihrer Instanzen — ist dabei lediglich der für ihn sichtbare Teil interessant.
    - Sichtbar sind zum einen die Schnittstellen (=Methoden, Parameter und Resulate), zum anderen die Auswirkungen von Operationen auf das zukünftige Verhalten des Objekts.
    - Änderungen des Objektzustands, die über die Schnittstellen nicht beobachtet werden können (z. B. ein Stack-Objekt fordert dynamisch Speicher nach, um weitere Daten speichern zu können und gibt ihn auch selbsttätig wieder frei, wenn er nicht mehr benötigt wird), sind für den Benutzer des Objekts irrelevant.

- unwichtig:
  - ➤ Implementierung eines Objekts
    - Instanzvariablen
    - Implementierung von Methoden

  - Die Implementierung eines Objekts sollte ausgewechselt werden können, solange sich die Schnittstelle und die Semantik der Operationen nicht ändert. Dafür ist es aber wichtig, daß der Benutzer keine Kenntnisse über Implementierungsdetails erhält, die für seine Anforderungen nicht erforderlich sind. Er könnte sonst Annahmen über das Verhalten des Objekts machen und sich auf diese Annahmen bei seiner Programmierung verlassen, obwohl solch ein Verhalten durch den Implementierer des Objekts nie zugesichert wurde.

- zuerst die Abstraktion beschreiben, dann Implementierung vornehmen
  - = zuerst Problemlösung spezifizieren, dann implementieren!

2 Kapselung

= Information Hiding
Verbergen der Implementierung einer Abstraktion vor den Benutzern der Abstraktion

- Gegenstück zu Abstraktion
  - ➤ Abstraktion stellt die äußeren Eigenschaften eines Objekts heraus
  - ➤ Kapselung verbirgt die Interna

- Grundvoraussetzung für Abstraktion

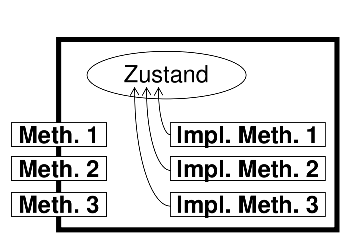
  B. Liskov *Damit Abstraktionen funktionieren, müssen Implementierungen gekapselt sein*

- Kapselung in der Objektorientierung
  - ◆ Darstellung des Objektzustands
  - ◆ Implementierung der Methoden

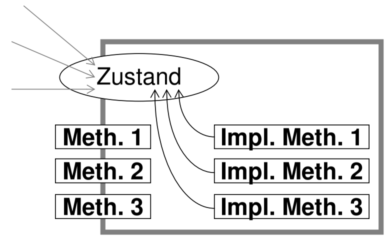
➔ Abstrakter Datentyp, Datenabstraktion

2 Datenabstraktion in OOP

➔ auf Objektzustand kann nur durch die Methoden des Objekts zugegriffen werden



Objekt als Implementierung eines ADT



Objekt ohne Datenabstraktion

- Viele Programmiersprachen erzwingen Datenabstraktion allerdings nicht (z. B. C++)
- Häufig ist ein Kompromiss zwischen klarer Strukturierung und Disziplin sowie Flexibilität und Effizienz erforderlich.
- Vor allem in der Systemprogrammierung ist Datenabstraktion in der Praxis nicht bis in die untersten Betriebssystemschichten durchzuhalten.
  - – Hinter einem Objekt kann sich "ein Stück Hardware" (z. B. ein Controller oder ein Prozessor) verbergen — Änderungen am Zustand solcher Objekte sind in manchen Fällen trotzdem von "außen" zu beobachten (z. B. weil nach einem Aufruf an ein Objekt, das den Treiber für eine serielle Schnittstelle realisiert, ein Signalpegel auf der Schnittstelle verändert wird und dadurch eine Telefonmodem-Verbindung abgebaut wird).

- Datenabstraktion in C++ und Java
  - ◆ Sichtbarkeitsbereiche (*Scope*-Regeln)
    - Kapselung durch *private*- und *protected*-Bereiche in Klassen
    - Objektschnittstellen im *public-Bereich* einer Klasse
    - *private* -Daten und -Methoden
      - – nur innerhalb von Methoden der gleichen Klasse sichtbar
    - *protected* -Daten und -Methoden (in Java: *private protected*)
      - – nur für Methoden der gleichen Klasse und in Methoden von Unterklassen sichtbar
    - *public* -Daten und -Methoden
      - – für alle Funktionen des Programms und für alle Methoden anderer Klassen sichtbar

        public -Methoden bilden die Schnittstelle der Klasse nach außen

        public -Daten erlauben eine Verletzung der Objektkapselung ➔ vermeiden!

3 Modularisierung

Aufteilung eines Programms in einzelne Komponenten kann seine Komplexität reduzieren

- ◆ Teilproblem überschaubarer
- ◆ Teilprobleme Entwicklergruppen zuzuordnen
- ◆ Module = separate Softwareentwicklungseinheiten

■ **vor allem: Aufteilung erzeugt Grenzen = Schnittstellen**
  - ➔ Schnittstellen müssen klar definiert werden
  - ➔ Schnittstellen müssen dokumentiert werden

■ viele Programmiersprachen unterscheiden zwischen Schnittstelle und Implementierung eines Moduls

➔ enge Beziehung zwischen Modularität und Kapselung

Die Unterstützung von Modularisierung in Programmiersprachen ist allerdings sehr unterschiedlich:

- In C oder C++ sind Module lediglich getrennt übersetzbare Dateien — Schnittstellen zwischen Modulen können über Include-Dateien konsistent gehalten werden. Die Sprachen sehen allerdings keinerlei Mechanismen vor, die eine konsistente Benutzung der Schnittstellen wirklich garantieren.
- Andere Sprachen (wie z. B. Modula) kontrollieren dagegen die Konsistenz der Modulschnittstellen direkt.
- In Java gibt es ein eigenes Modularisierungskonzept: *Packages*
  - Zusätzliche Schutzattribute: Klassen oder Methoden sind damit nur innerhalb des eigenen Paketes sichtbar
  - Pakete spannen einen eigenen Namensraum auf
    - ➔ keine Namenskonflikte zwischen unabhängig entwickelten Softwarekomponenten möglich
  - Das Laufzeitsystem garantiert die Einhaltung der Schutzattribute und die konsistente Nutzung der Modulschnittstellen

■ *Structured Design*: Gruppierung von Unterprogrammen

- Unterprogramme werden meist aufgrund ihrer Interaktionsbeziehungen auf Module verteilt.

■ OOD: Gruppierung von Objekten und Klassen

- Bei der Programmierung komplexer Softwaresysteme entsteht eine große Anzahl von Klassen
  - ➔ viele kleine Einheiten mit eigenen Schnittstellen
  - ➔ Beziehungen zwischen den Einheiten schwer überblickbar
- Aufgabe der Modularisierung im Rahmen von objektorientiertem Softwareentwurf ist damit, eine Gruppierung von Klassen auf Basis der Problemstruktur vorzunehmen. Diese kann sich von der Struktur, die durch die Interaktionsbeziehungen von Unterprogrammen entstehen würde stark unterscheiden!

4 Hierarchie

- ◆ Abstraktion & Kapselung helfen Details von Komponenten zu verbergen
- ◆ Modularisierung hilft zusammengehörende Abstraktionen zu bündeln

■ Überblick über große Problemstellungen immer noch schwierig
  - ➔ zu viele Abstraktionen
  - ➔ Hilfsmittel zur Organisation von Abstraktionen notwendig

■ Abstraktionen bilden oft Hierarchien
  - ➤ gemeinsame Eigenschaften → allgemeinere Abstraktionen
  - ➤ Unterschiede → Spezialisierungen
  - ➔ Hierarchie: Ordnung auf Abstraktionen

■ Hierarchie & Objektorientierung
  - ➤ Klassenstruktur: Vererbung → "**Art-von**"-Hierarchie (*kind of / is a*)
    - Vererbung
    - Delegation
  - ➤ Objektstruktur: Aggregation → “**Teil-von**"-Hierarchie (*part-of*)
    - Aggregation — ein Objekt ist aus mehreren Unterobjekten zusammengesetzt
      - Beispiel: Objekt Auto besteht aus Unterobjekten Motor, Getriebe, Lenkung, etc.

5 Typisierung

Typkonzept baut auf ADT-Theorie auf

Definitionen aus [Boo94]:

- Ein Typ ist eine genaue Charakterisierung der gemeinsamen Eigenschaften (bezüglich Struktur und Verhalten) einer Menge von Einheiten
- Typisierung ist das Erzwingen der Klasse eines Objekts, so dass Objekte unterschiedlicher Typen nicht oder zumindest nur in sehr eingeschränkter Weise untereinander ausgetauscht werden können

Bei Objekten wird der Typ durch die Signatur und die Semantik der Methodenaufrufe bestimmt.

Die meisten Programmiersprachen ermöglichen dem Programierer allerdings nur die Formulierung der syntaktischen Eigenschaften eines Typs.

– Bei vielen objektorientierten Sprachen (z. B. C++, aber nicht Java) erfolgt dies durch die Beschreibung einer Klassenschnittstelle.

➔ Die Beschreibung einer Klasse definiert gleichzeitig einen Typ.

Typisierung ermöglicht die Überprüfung von Ausdrücken auf Typ-Kompatibilität zur Übersetzungszeit

➔ Vermeidung von Fehlern

- Zuweisungen unterschiedlicher Typen oder die Übergabe falscher Typen als Parameter von Funktions- bzw. Methodenaufrufen kann zur Übersetzungszeit festgestellt werden. Auf diese Weise lassen sich viele Fehler von vorneherein vermeiden.

Strenge Typisierung
Konformität aller Typen in einem Ausdruck wird garantiert

➔ Statische Typisierung
Konformität wird komplett zur Übersetzungszeit überprüft

➤ Einschränkung der Flexibilität

- kompatible Typen

Häufig kann der Compiler nicht feststellen, dass zwei unterschiedliche Typen kompatibel sind und damit in einem Ausdruck (z. B. einer Zuweisung) problemlos zusammen verwendbar sind.

- dynamisches Binden

Wenn erst zur Laufzeit festgelegt werden soll, welches Objekt (oder auch welche Funktion in nicht-OOP) über eine Variable erreichbar ist, kann die Typ-Verträglichkeit nicht generell zur Übersetzungszeit überprüft werden.

➔ mehr Flexibilität durch Polymorphismus und Generizität

– Polymorphismus erlaubt neben der Verwendung des passenden Typs auch andere, konforme Typen.

– Generizität ermöglicht eine Parametrierung mit Typen. Dadurch können z. B. Klassen, die für verschiedene Typen einsetzbar wären, aufgrund strenger Typisierung aber nicht flexibel verwendet werden können, durch Typ-Parameter für die Verwendung in Zusammenhang mit einem Typ eingestellt werden.

6 Typhierarchie

Häufig 1:1-Relation zwischen Klassen und Typen — aber nicht notwendig

➤ mehrere Klassen können einen Typ implementieren

- Beispielsweise können unterschiedliche Klassen einen Datentyp "Stack" unterschiedlich implementieren (Daten werden in einem Feld oder in einer verketteten Liste verwaltet), aber die gleichen Methodenaufrufe (bezüglich Signatur und auch Semantik) anbieten.

➤ eine Klasse kann unterschiedliche Typen implementieren

- Eine Klasse kann beispielsweise besondere Methodenaufrufe zur Einstellung von Parametern oder Debug-Möglichkeiten besitzen, deren Verwendung in Teilen einer Anwendungsimplementierung aber nicht erlaubt sein darf (weil die Klasse möglicherweise später durch eine Implementierung ohne diese Methoden ersetzt werden soll). Wenn man in den Teilen der Anwendung einen Typ für die Klasse bekanntgibt, der die besonderen Methodenaufrufe nicht enthält, garantiert das Typsystem, dass sie auch nicht verwendet werden. In den Teilen der Anwendung, die die besonderen Methodenaufrufe benötigen, wird ein anderer Typ für die Klasse deklariert, der die Methoden enthält.

Hierarchie bei Klassen: Oberklasse ← Unterklasse

◆ Ziel: Wiederverwendung von Implementierung

◆ Unterklasse nicht notwenigerweise typ-konform zu Oberklasse

- In den meisten Programmiersprachen (auch in C++ und Eiffel, nicht aber in Java) ist es möglich, Unterklassen zu programmieren, die nicht typ-konform zur Oberklasse sind.
- Unter dem Gesichtspunkt, durch eine Vererbung bei Klassen Programmcode wiederzuverwenden ist dies eigentlich unproblematisch. Da viele Programmiersprachen aber durch Klassendeklarationen gleichzeitig Typdeklarationen vornehmen, ist es wichtig, unterscheiden zu können, wo durch Klassen-Vererbung ein konformer Untertyp entsteht und wo nicht.

Hierarchie bei Typen: Obertyp ← Untertyp

◆ Ziele: – Verhaltens-Vererbung
– Beschreibung **konformer** Typen (➔ Polymorphismus)

- Durch eine Typhierarchie soll ausschließlich festgelegt werden, welche Typen zu welchen anderen Typen konform sind.

Typvererbung (*Subtyping*) als Mechanismus zur Ableitung von Typen

- Analog zur Vererbung bei Klassen: der Untertyp erbt von einem Obertyp
- Unterschied zur Vererbung bei Klassen: der Untertyp ist auf jeden Fall konform zum Obertyp — d. h. er kann zusätzliche Methoden enthalten, enthält aber bezüglich Signatur und Semantik alle Methoden des Obertyps.
- Mehrfachvererbung bei Typen (ein Subtyp ist konform zu mehreren Obertypen) ist im Gegensatz zu Mehrfachvererbung bei Klassen absolut unproblematisch, da keine Implementierung verebt wird!

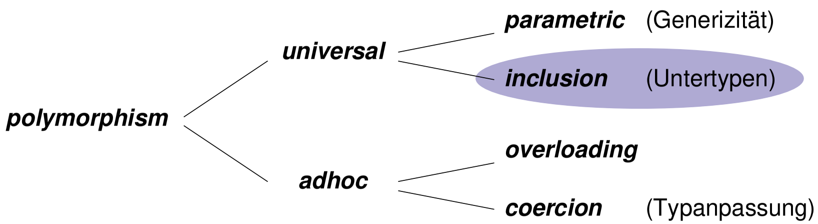
➔ Zusammenhänge zwischen Typen werden übersichtlich

– Identifikation konformer Typen

- Durch die Typhierarchie ist festgelegt, welche (Unter-)Typen statt eines Typs in einem Ausdruck verwendet werden dürfen.

7 Polymorphismus

- die Fähigkeit, verschiedene Formen anzunehmen [Mey88]
  - ➤ mehr als ein Typ für Werte oder Variablen
  - ➤ verschiedene Typen als Parameter zu Funktionen
  - ➤ verschiedene Typen als Operanden eines Operators
- Beispiel:
  +-Operator arbeitet mit int- und real-Werten
- Klassifikation [CW85]



universal polymorphism:

- funktioniert prinzipiell mit beliebig vielen Typen (alle Typen haben aber eine gemeinsame Grundstruktur — sind beispielsweise Subtypen eines gemeinsamen Obertyps, z. B. Zahlen in verschiedenen Darstellungen und Ausprägungen).

  parametric polymorphism:
  - Eine polymorphe Funktion besitzt einen expliziten Parameter, in dem der Typ des Parameters für jeden Aufruf mit übergeben wird (spezielle Ausprägung dieses Konzepts in Adas *generic functions* zu finden).

  ***inclusion polymorphism**:*
  - Bei Subtyping: ein Untertyp kann jederzeit statt des Typs verwendet werden.
    In objektorientierten Sprachen von großer Bedeutung: häufig mit Vererbung gekoppelt — wenn eine Unterklasse einen Untertyp implementiert, können Instanzen der Unterklasse statt Instanzen der Basisklasse verwendet werden.

ad hoc polymorphism:

- funktioniert nur auf einer endlichen Menge — möglicherweise völlig voneinander unabhängiger Typen.

  overloading:
  - Mehrere verschiedene Funktionen werden unter dem gleichen Namen definiert und es wird anhand des Aufruf-Kontexts (Anzahl und Typ der Aufrufparameter) entschieden, welche Implementierung aufgerufen wird. (Beispiele siehe C++)

  coercion:
  - Vor dem Aufruf der Funktion werden Parameter, deren Typ nicht mit der Funktionsdefinition übereinstimmt, automatisch in den passenden Typ konvertiert.
    (Beispiel für *Coercion*: in C wird bei der Addition eines int- und eines real-Wertes automatisch eine Konvertierung beider Operanden nach double durchgeführt).

8 Polymorphismus in C++

Einige Konzepte in C++ können zur Realisierung von Polymorphismus eingesetzt werden. Dabei sind drei Kategorien zu unterscheiden:

- *overloading polymorphism*
  - ➤ Function-name overloading
  - ➤ Operator overloading
    - Funktionen oder Operatoren eines Namens können für unterschiedliche Parameter- bzw. Operandentypen definiert werden.
- *inclusion polymorphism*
  - ➤ public Vererbung
    - Da in C++ Subklassen gleichzeitig Subtypen des Datentyps der Basisklasse bilden, sind sie typkonform zur Basisklasse. Instanzen der Subklasse können daher statt Instanzen der Basisklasse verwendet werden. In der Subklasse können dabei Methoden der Basisklasse anders implementiert sein.
- *coercion polymorphism*
  - ➤ Cast-Operatoren
    - Für die Verwendung eines "unpassenden" Typs als Parameter oder in einem Ausdruck können für Einzelfälle Regeln beschrieben werden, wie der unpassende Typ in einen passenden Typ konvertiert wird. Diese Konvertierung erfolgt dann automatisch.

8 Polymorphismus in C++ (2)

★ Inclusion-Polymorphism — *DER Polymorphismus in OOP*

Vererbung + Virtuelle Methoden + Objekt-Referenzen

- Eine Objektreferenz (Zeiger) hat einen Typ (= Klasse)
  - Instanzen dieser Klasse und ihrer Unterklassen können dem Zeiger zugewiesen werden
  - bei einem Methodenaufruf wird die tatsächliche Implementierung der Methode nicht von der Klasse des Zeigers, sondern von der Klasse des aktuellen Objekts bestimmt

➜ Man kann jederzeit einer Referenz irgendein Typ-konformes Objekt zuweisen und alles funktioniert

- man kann es als Parameter einer Methode übergeben
- der Programmierer der Methode musste den neuen Untertyp nicht kennen — so lange er konform zu dem Obertyp ist, den seine Methode erwartet

8 Polymorphismus in C++ (3)

★ Virtuelle Methoden & Inclusion Polymorphism — Beispiel:

```
class geo_obj {                       // general superclass
  public:
    virtual void draw();
};

class circle : public geo_obj {       // subclass
  public:
    void draw();
}

class square : public geo_obj {       // subclass
  public:
    void draw();
}

main () {
  geo_obj *ptr;
  ptr = new circle;
  ptr->draw();
...
```

- Der Aufruf **`ptr->draw()`** bewirkt den Aufruf der **`draw`**-Methode der Subklasse **`circle`**, obwohl **`ptr`** ein Zeiger auf ein Objekt der Klasse **`geo_obj`** ist!
- Wäre **`draw`** in **`geo_obj`** nicht als virtual deklariert worden, hätte der Aufruf **`ptr->draw()`** die Methode der Klasse **geo_obj** angesprochen, obwohl eine Instanz der Unterklasse **`circle`** an **`ptr`** gebunden war.
- Die **`draw`**-Methoden der Unterklasse und auch die **`draw`**-Methode der Oberklasse **`geo_obj`** müssen in obigem Beispiel natürlich noch außerhalb der Klassen definiert werden.
- Auf die Definition der **`draw`**-Methode der Oberklasse könnte auch verzichtet werden, **`geo_obj`** wäre dann eine abstrakte Klasse (siehe Abschnitt über abstrakte Klassen).
  - es könnten dann allerdings keine Instanzen von **`geo_obj`** erzeugt werden, da die Klasse nicht vollständig definiert ist
  - **`geo_obj`** wäre dann nur als Basisklasse im Rahmen von Vererbung zu gebrauchen — in obigem Beispiel durchaus sinnvoll, weil eine **`draw`**-Methode, die nicht näher beschriebene geometrische Objekte zeichnen kann, wohl ohnehin keinen Sinn macht.

Virtuelle Methoden einer Subklasse, die eine virtuelle Methode der Basisklasse neu implementieren, sind automatisch wieder virtuell (z. B. bei weiterer Vererbung).

➜ **`void draw();`** in der Klasse **`circle`** ist damit korrekt, es muß nicht **`virtual`** davor angegeben werden!

9 Typen und C++: Abstrakte Klassen

- Basisklasse deklariert Methoden und Parameter, definiert sie aber nicht
  - – *pure virtual function*
  - ➜ Basisklasse beschreibt einen Typ
- Subklassen definieren unterschiedliche Implementierungen der Methoden
  - ➜ jede Subklasse stellt eine Implementierung des Typs dar
- Basisklasse ist nicht instantiierbar
- Beispiel:

```
class geo_obj {                    // abstract class
  public:
    virtual void draw() = 0;      // pure virtual function
};
class circle : public geo_obj {   // subclass
  public:
    void draw() { ... }
}
```

- Die Klasse **geo_obj** beschreibt lediglich die Signatur von geometrischen Objekten, enthält aber keinerlei Implementierungen.
- Unterklassen von **geo_obj** stellen dann konkrete Implementierungen verschiedener geometrischer Objekte dar, die alle konform zu dem Datentyp **geo_obj** sind (und damit z. B. an einen Zeiger auf **geo_obj** zuweisbar sind).

- Grundsätzlich können auch einzelne (virtuelle) Methoden, die für alle Subklassen gleich sein sollen, in der abstrakten Basisklasse definiert werden und andere, die erst in den Subklassen implementiert werden sollen, als *pure virtual functions* angegeben werden. Die Basisklasse ist dann nach wie vor eine abstrakte Klasse, da sie nicht vollständig implementiert ist, sie entspricht aber nicht mehr einer reinen Typdefinition, sondern ist eine Mischung aus Typdefinition und -implementierung.
- Genauso können Subklassen von abstrakten Klassen auf die Definition einer *pure virtual function* verzichten (sie müssen deren Deklaration aber explizit wiederholen!). Solche Subklassen sind dann nach wie vor abstrakte Klassen und können nur im Rahmen von weiterer Vererbung eingesetzt werden.

10 Typen und Java: Interfaces

- 2 Möglichkeiten einen Typ zu deklarieren:
  - ◆ im Rahmen einer Klassendefinition
    - Java kennt auch abstrakte Klassen wie in C++
    - Nachteile der impliziten Typ-Deklarationen durch Klassen:
      - – Es ist nur einfache Vererbung möglich. Ein durch eine Klasse definierter Typ kann damit nur an von dieser Klasse abgeleitete Unterklassen vererbt werden.
      - – Es gibt keine Möglichkeiten, andere, unabhängig definierte, aber typ-kompatible Klassen zu solch einem Typ konform zu erklären.
    - ➜ Klassenvererbung führt automatisch zu Typvererbung
    - Es ist in Java nicht möglich, Subklassen zu bilden, die nicht typ-konform zur Oberklasse sind.
  - ◆ durch eine Interface-Deklaration
    - ➜ separate Typbeschreibung
    - Die Typ-Beschreibung erfolgt getrennt als `interface`
    - Danach kann im Rahmen einer Klassendefinition angegeben werden, daß die Klasse den Typ implementiert.
    - Der Compiler überprüft, ob in der Klassenimplementierung auch tatsächlich die Schnittstelle des Typs korrekt angegeben ist.
- Beispiel:

```
public interface Printable {
    public void Print();
}

public class MyPoint extends Object implements Printable {
    ....
    public void Print() {
        System.out.println("x="+x+" y="+y);
    }
}
```

10 Typen und Java: Interfaces (2)

- Vererbung & Mehrfachvererbung auf Interfaces
  - Während bei Klassenvererbung nur Einfachvererbung möglich ist, ist auf Typen auch Mehrfachvererbung zugelassen
    - d. h. ein Typ kann konform zu verschiedenen Obertypen sein
  - Da nur Schnittstellenkonformität deklariert wird, aber keine Implementierung übernommen wird, treten die Konfliktprobleme der Mehrfachverbung bei Klassen natürlich nicht auf.
- eine Klasse kann mehrere Typen implementieren
  - Bei einer Klassendefinition können mehrere, ggf. auch unterschiedliche Schnittstellen angegeben werden, die von der Klasse implementiert werden.
- Typkonformität ist transitiv
  - Wenn eine Klasse einen Typ implementiert, dann implementieren automatisch auch alle Subklassen davon diesen Typ.
- Exceptions sind auch Bestandteil der Typ-Schnittstelle
  - Im Gegensatz zu C++ können in Java auch die Exceptions, die eine Methode erzeugt in der Typ-Schnittstelle angegeben werden.
  - Ein Subtyp darf, um konform zu sein, nur die Exceptions des Basistyps (ggf. natürlich weniger Exceptions) bzw. konforme Exceptions erzeugen
    - Exceptions werden in Java durch Klassen repräsentiert. Eine konforme Exception ist damit eine Unterklasse bzw. eine zum Exception-Typ ebenfalls konforme Klasse.
- Beispiele:

```
interface Streamable extends FileIO, Printable {
    // additional Methods
    public void test() throws TestException;
}

class Test implements Streamable, Testinterface {
    ...
}
```

- Der Typ `Streamable` ist von den beiden Typen `FileIO` und `Printable` abgeleitet.
- Die Methode Test kann eine Exception `TestException` erzeugen — diese Tatsache ist Bestandteil des Typs `Streamable`.
- Die Klasse Test implementiert die beiden Typen `Streamable` und `Testinterface`

11 Generizität (*Genericity*)

- Möglichkeit, den Typ von programmiersprachlichen Einheiten durch Parameter festzulegen
  Beispiele:
  - generische Funktionsargumente (ein Funktionsparameter bestimmt den Typ anderer Parameter)
  - generische (parametrierbare) Klassen
- OOP: generische Klassen

  generische Klasse → Instantiierung (+Parametrierung) → tatsächliche Klasse
  tatsächliche Klasse → Instantiierung von Objekten
- Beispiel:
  ◆ allgemeine Klasse Stack
    – int-Stack
    – real-Stack
    – string-Stack
- Generizität kann weitgehend mit Vererbung nachgebildet werden [Mey86]
  Aber: eigentlich werden dadurch die Begriffswelten von Typ und Klasse (wie so oft) vermischt: Generizität ist ein Konzept, um mehr Flexibilität im Bereich Typisierung zu erreichen, während Vererbung Klassenhierarchien erzeugt!
- realisiert in Ada, Eiffel, C++ (ab. V 3.0, *Templates*) und Java (ab Java 5)

12 Generizität und C++: Templates

- Ziel: Klassendefinition ohne Festlegung auf bestimmte Typen
  - Vor allem wenn Klassen in Standardbibliotheken bereitgestellt werden sollen, gibt es viele Fälle, in denen von vorneherein nicht festgelegt werden kann, für welche Typen solch eine Klasse gebaut werden soll. Typische Beispiele sind Container-Klassen wie Listen, Vektoren oder assoziative Felder.
  - Grundsätzliche gibt es zwei Möglichkeiten, die Konstruktion solcher Klassen in einer Sprache zu unterstützen:
  - ◆ dynamische Typprüfung zur Laufzeit
    - Die betreffenden Typen werden zur Übersetzungszeit nicht festgelegt und durch dynamische Typprüfung zur Laufzeit wird eine konsistente Verwendung von Parametern und Instanzvariablen überprüft. Insbesondere in nicht-getypten Sprachen (wie Smalltalk) wird dieser Weg gewählt. Die Programmierung wird dadurch einfacher, vor allem sind alle Klassen sehr flexibel einsetzbar. Zur Laufzeit entsteht aber beträchtlicher Mehraufwand durch die dynamischen Typprüfungen.
  - ◆ statische Typprüfung zur Übersetzungszeit + parametrierbare Klassen
    - Die korrekte Verwendung von Typen bei Parameterübergabe und bei Zuweisungen wird zur Übersetzungszeit komplett überprüft. Um Klassen definieren zu können, ohne sich auf alle in der Klassenimplementierung verwendeten Typen von vorneherein festlegen zu müssen, werden parametrierbare Klassen eingeführt: Die Klassendefinition enthält Parameter, die in der Implementierung statt eines Typs eingesetzt werden können.

- Template = parametrierbare Klasse
- Beispiel:

```
template <class T> class stack {
  private:
    int index;
    T *array;
  public:
    void stack(int n)
              { index = 0; array = new T[n]; }
    void push(T elem)
              { array[index++] = elem; }
    T pop(void)
              { return(array[--index]); }
};
```

  - In der Klassendefinition können neben einem oder mehreren Typparametern auch andere Parameter auftreten. In dem Beispiel könnte man neben dem Typ **T** beispielsweise auch die Stackgröße als Template-Parameter angeben. Man könnte dann ein Objekt "Integer-Stack mit 10 Elementen" instantiieren. Dieses Objekt hätte aber einen anderen Typ als ein "Integer-Stack mit 20 Elementen". Wird die Größe dagegen im Konstruktor angegeben, so instantiiert man Objekte vom Typ "Integer-Stack" und übergibt dem Konstruktor die gewünschte Größe. Der Typ solcher Objekte (auch unterschiedlicher Größe) ist dann identisch.

13 Generizität und Java (ab Java 5)

- auf den ersten Blick ähnlich zu C++-Templates, aber viele Unterschiede im Detail
  - ➤ nur eine generische Klasse für alle Objekte
    - es wird keine parametrisierte Klasse erzeugt, von der Instanzen gebildet werden, sondern der Bytecode der Klasse liegt zur Laufzeit nur einmal vor
    - Type-Erasure: Typ-Variablen werden durch Object (bzw. bei Bounds durch den angegebenen Obertyp) ersetzt. Compiler fügt Casts in den Code ein, um zur Laufzeit Typ-Sicherheit zu gewährleisten.
  - ➤ Bounds erlauben die Einschränkung von Typ-Variablen auf bestimmte Subtypen
    - `T extends C & T1 & T2 ...`
      Typ T muss Untertyp der Klasse C bzw. der Interfaces T1 oder T2 ... sein
- Typ-Parametrierung auch auf Interfaces möglich
- Beispiel:

```
class AnimalStack <T extends Animal> {
    private int index;
    private T[] array;

    public void AnimalStack(int n)
                { index = 0; array = (T[])new Object[n];}
    public void push(T elem)
                { array[index++] = elem; }
    public T pop()
                { return(array[--index]); }
};
```

14 Nebenläufigkeit (Concurrency)

Nebenläufigkeit: mehrere Aktivitätsträger werden parallel von mehreren Prozessoren bearbeitet oder von einem Prozessor simuliert

- Nebenläufigkeit orthogonal zu Objektorientierung
  aber: Komplexität einer Problemlösung wird durch Nebenläufigkeit erhöht
  zusätzliche Probleme:
  - Koordinierung
    - gegenseitiger Ausschluß
  - Synchronisation
    - warten auf Ergebnisse anderer Aktivitätsträger
  - Verteilung
    - Verteilen der auszuführenden Objekte auf die verfügbaren Prozessoren

- Granularität: Nebenläufigkeit / Objekte (Kapseln)
  - ➤ Nebenläufigkeit feiner granular
    - ➔ NL auch objektintern
      - mehrere Aktivitätsträger gleichzeitig in einem Objekt
        → objektinterne Koordinierung erforderlich
  - ➤ Nebenläufigkeit grober granular
    - ➔ NL nur objektextern
      - immer nur maximal ein Aktivitätsträger zu jedem Zeitpunkt in einem Objekt
        → keine Koordinierungsprobleme innerhalb eines Objekts, Koordinierung kann auf die Objektinteraktionen beschränkt werden

- Integration der Kontrolle von Nebenläufigkeit in objektorientierte Sprachen
  - ➤ orthogonale Sprachen
    - Orthogonale Sprachen enthalten keine Sprachkonstrukte zur Kontrolle von Nebenläufigkeit. Zur Koordinierung muß der Programmierer sprach-externe Mechanismen wie z. B. Semaphore einsetzen.
  - ➤ nicht-orthogonale Sprachen
    - In nicht-orthogonalen Sprachen können Objekte gegen nebenläufige Ausführung geschützt werden. Dies wird entweder automatisch für alle Objekte vorgenommen
      ➔ **uniforme Sprachen**
      oder es kann durch Sprachkonstrukte vom Programmierer festgelegt werden, ob ein Objekt geschützt werden soll
      ➔ **nicht-uniforme Sprachen**.

15 Nebenläufigkeit und Java

- Thread-Konzept und Koordinierungsmechanismen sind in Java integriert
  ➥ nicht-orthogonale, nicht-uniforme Sprache bzgl. Nebenläufigkeit

- Erzeugung von Threads über Thread-Klassen
  - Die auszuführenden Methode wird in einer speziellen Klassen, die ein Interface `Runnable` implementiert als Methode `run` implementiert.
    - Thread erzeugen = "run-Methoden"-Objekt instantiieren, Thread-Objekt instantiieren und dem Konstruktor das "run-Methoden"-Objekt übergeben
    - Thread starten = Methode `start` an Thread-Objekt aufrufen,
      → Methode `run` wird nebenläufig ausgeführt
  - Es wird eine Subklasse von Thread definiert, die die `run`-Methode redefiniert. Eine Instanz der Subklasse wird erzeugt und an ihr die Methode `start` aufgerufen.
    - Thread erzeugen = Objekt der Subklasse instantiieren
    - Thread starten = Methode `start` an dem Objekt aufrufen
      → Methode `run` wird nebenläufig ausgeführt

- Beispiel

```
class MyClass implements Runnable {
        public void run()   {
                System.out.println("Hello\n");
        }
}

....
MyClass o1 = new MyClass();     // create object
Thread t1 = new Thread(o1);     // create thread to run in o1

t1.start();                     // start thread

Thread t2 = new Thread(o1);     // create second thread to run in o1

t2.start();                     // start second thread
```

15 Nebenläufigkeit und Java (2)

★ Koordinierungsmechanismen

- Monitore: exclusive Ausführung von Methoden eines Objekts
  - Methoden oder Code-Blöcke können `synchronized` deklariert werden
  - Alle synchronized-Methoden und Code-Blöcke eines Objekts (einer Instanz, nicht aller Instanzen einer Klassen gemeinsam!) bilden gemeinsam einen Monitor: d.h. es kann immer nur einThread zu einem Zeitpunkt in einer der Methoden bzw. Code-Blöcke aktiv sein.
  - Ein Objekt kann weitere Methoden besitzten, die nicht als synchronized deklariert wurden. Solche Methoden (z.B. Methoden, die nur auf dem Objektzustand lesen, aber keine kritischen, koordinierungsbedürftigen Operationen ausführen) können auch mehrfach parallel ausgeführt werden.

◆ Beispiel:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

◆ Conditions: gezieltes Freigeben des Monitors und Warten auf ein Ereignis

- Innerhalb eines Monitors kann "wait" aufgerufen werden
  ⇒ Monitor wird freigegeben und Thread legt sich schlafen.
- Jemand anderes kann innerhalb des Monitors "notify" oder "notifyAll" aufrufen
  ⇒ Einer der bzw. alle blockierten Threads werden aufgeweckt

15 Nebenläufigkeit und Java (3)

★ Koordinierungsmechanismen in Java 5

- mehrere APIs zur Unterstützung von Nebenläufigkeit
- Atomare Operationen
  - ➤ z.B. getAndSet oder compareAndSet auf Boolean, Integer oder Long
  - ➤ Paket java.util.concurrent.atomic
- Locks und Condition Variablen
  - ➤ orthogonale Implementierung zusätzlich zu synchronized/wait/notify
    - Vorteil: flexiblere Handhabung von Monitoren im Gegensatz zu den in die Sprache integrierten Mechanismen
    - Nachteil: expliziter Umgang mit Koordinierungsmechansmen ggf. fehlerträchtiger
  - ➤ Paket java.util.concurrent.locks
- Umfangreiches Paket java.util.concurrent
  - ➤ thread-sichere Container, Queues, Collections
  - ➤ Executor-Framework, Thread-Pools, Futures, Scheduling
  - ➤ Semaphore, Latches, Barrieren

16 Persistenz

★ Motivation für Persistenz **[ABC83]**

■ "aktive" Daten → Programmiersystem

- "Aktive" Daten, die *kurzfristig* im Speicher angelegt sind, werden mit den Hilfsmitteln des Programmiersystems (Sprache) bearbeitet.

■ "passive" Daten → DBMS oder Dateisystem

- "Passive" Daten, die die Programmausführung überleben sollen, werden einer Datenbank (DBMS) oder einem Dateisystem übergeben und mit dessen Funktionen verwaltet.

➔ 2 Sichten auf Daten

➔ Nachteile für den Anwender

➤ Konvertierung notwendig

- Programmieraufwand für die Konvertierung und den Transport zwischen passivem und aktivem Zustand (*scanf/printf*-Anweisungen)
(Atkinson schätzt i. a. ca. 30% des Codes!)

➤ Datenschutz des Programmiermodells geht verloren

- Schutzmechanismen des Programmiermodells (z. B. Typisierung) greifen nur, solange Daten aktiv sind - bei der ersten Wandlung kann alles verloren gehen.

★ allgemeine Definition:
Persistenz ist die Eigenschaft von Daten, das Ende einer Umgebung, in der sie entstanden oder benutzt wurden, zu überdauern

16 Persistenz (2)

★ **Persistenz in objektorientierten Systemen**

■ Objekte überleben das Ende der Umgebung (Aktivitätsträger, Anwendungsausführung), in der sie instantiiert oder benutzt wurden

■ In OO-Betriebssystemen mächtiger Basis-Mechanismus

➤ Datenspeicherung

➤ Datentransport

■ Beispiele

➤ Dateisysteme

- Eine traditionelle Datei ist ein persistentes Objekt mit Methoden wie *read*, *write* und *seek*.

➤ Datenbanksysteme

➤ persistente Kommunikationsobjekte

- Beispielsweise ein Puffer-Objekt

■ Eigenschaften des objektorientierten Programmiermodells gehen automatisch auf Mechanismen über, die damit erstellt wurden
Beispiele:

- Vererbung zur Realisierung unterschiedlicher Dateiobjekte
- Kontrolle von Nebenläufigkeit