Prozedurale Programmierung
Objektorientierte Programmierung
Funktionale Programmierung

Charakter der prozeduralen Programmierung

über destruktive Zuweisungen in Prozeduren änderbarer Programmzustand,
Kontollstrukturen basierend auf strukturierter Programmierung,
kaum Programmierung im Groben, höchstens (generische) Module,
weder Prozeduren noch Objekte als Daten

- → überschaubare, anfängerfreundliche Menge sprachlicher Ausdrucksmittel
- → viel Kontrolle über Details
- → spezielle Hardware ansprechbar
- → schon kleine Programme wirken komplex
- → niedrige Abstraktionsgrade, häufig λ-Abstraktion, manchmal nominale Abstraktion
- → Basis für formale Korrektheitsbeweise bei λ-Abstraktion
- → Schleifen häufig, Rekursion selten
- → entweder nur dynamisch oder weitgehend statisch typisiert (explizite Typspezifikationen)
- → unkontrollierbare Kommunikation über Variablen und Aliase ist problematisch

Prozedurale Programmierung in Java

```
public class ProceduralCourse {
    private final static int MAX = 1000;
    private static Student[] studs = new Student[MAX];
    ...
    private static void readStudents(InputStream stream) { ... }
    private static void printStatus() { ... }
    public static void main(String[] args) throws IOException {
        readStudents(new FileInputStream(args[0]));
        printStatus();
    }
}
class Student {
    public int regNo, curr;
    public String name;
}
```

→ statische Variablen als globale Variablen
→ statische Methoden als Prozeduren
→ Objekte von Hilfsklassen ohne Methoden als Records
→ direkte Zugriffe auf public Objektvariablen von außen
→ Prozeduren private oder public (Export aus Modul)

→ prozedural gesehen: ganz normal bzw. typisch
→ aus Blickwinkel anderer Paradigmen: „sehr schmutzig“

Einsatzgebiete prozeduraler Programmierung

hardwarenahe Programmierung — statisch typisiert

Echtzeitprogrammierung — statisch typisiert

Scripting — dynamisch typisiert

flexible Software-Architekturen (z. B. Micro-Services) — statisch oder dynamisch typisiert

Hobby-Programmierung und Anwendungsprogrammierung — eher dynamisch typisiert

→ sehr viele Leute beherrschen die prozedurale Programmierung
→ die meisten jemals geschriebenen Programme sind prozedural
→ prozedurale Programmierung nach wie vor aktuell
→ heute oft objektorientierte Sprachen für prozedurale Programmierung eingesetzt
z. B. Vererbung als Konzept zur Strukturierung von Prozedurbibliotheken
→ Trend zurück zu Wurzeln z. B. „objektorientierte Empfehlungen“ ignorieren

Charakter der objektorientierten Programmierung

professionelle Werkzeugkette für Entwicklung und Wartung großer, langlebiger Software

nominale Abstraktionen auf hohem Niveau

Programmierung im Groben (Faktorisierung, Abstraktionshierchien, ...) im Mittelpunkt

Zusammenarbeit professioneller Entwickler_innen notwendig

→ komplexes Gefüge an Denkmustern
→ sehr teuer, aber auch für sehr komplexe Systeme erfolgversprechend
→ erfordert vollen Einsatz und viel Wissen („ein bisschen objektorientiert“ ist sinnlos)
→ ungeeignet für kleine Projekte und sehr komplexe Algorithmen
→ überfordert unerfahrene Programmierer_innen

Objekt

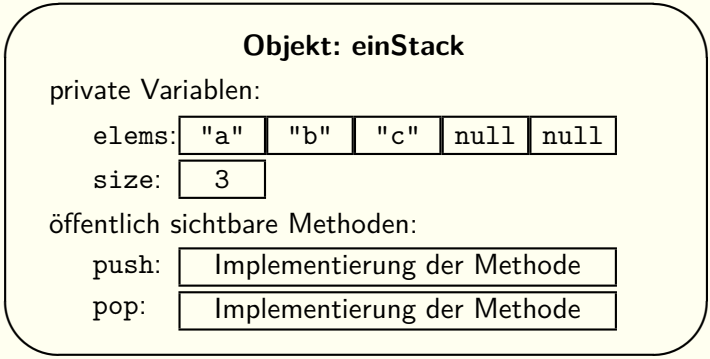
Objekt kapselt Variablen und Methoden zu Einheit, regelt Sichtbarkeit

Identität Adresse

Zustand Variableninhalte
(gleich $\neq$ identisch)

Verhalten Methodenbeschreib.
(Verhalten $\neq$ Implementierung)

Schnittstellen Verwendbarkeit



Softwareobjekt simuliert und abstrahiert reales Objekt („abstakter Datentyp“)

Klasse

→ eigentliche Klasse von nicht-statischen Inhalten gebildet
→ jede Java-Klasse ist auch Modul (Übersetzungseinheit)
→ gleiche Klasse ⇒ gleiches Verhalten, gleiche Schnittstellen
→ jedes Objekt ist Instanz *einer* Klasse, *mehrerer* Typen
→ Java-Interface ist Spezialform von Klasse

```
public class Stack {
    private String[] elems;
    private int size = 0;
    public Stack(int sz) { // new stack of size sz
        elems = new String[sz];
    }
    // push pushes elem onto stack if not yet full
    public void push(String elem) {
        if (size < elems.length) elems[size++] = elem;
    }
    // pop returns element taken from stack, null if empty
    public String pop() {
        if (size > 0) return elems[--size]; else return null;
    }
}
```

Polymorphismus

- Polymorphismus
  - universeller Polymorphismus
    - Generizität
    - Untertypbeziehungen !!!
  - Ad-hoc-Polymorphismus
    - Überladen
    - Typumwandlungen

jeder Referenzvariable (und jeder Ausdruck) hat gleichzeitig mehrere Typen:

deklarierter Typ	in Deklaration angegebener Typ für statische Typprüfung, Überladen, ...
dynamischer Typ	Klasse des in der Variable gerade enthaltenen Objekts (zur Laufzeit) für dynamisches Binden, `getClass()`, `instanceof`, ...
statischer Typ	speziellster vom Compiler ermittelter Typ von Qualität des Compilers abhängig, nur für Optimierungen

Untertypen und Vererbung

Untertypen wenn Ersetzbarkeit — B ist Untertyp von A wenn jede Instanz von B verwendet werden kann, wo eine Instanz von A erwartet wird

→ ermöglicht Austausch von Programmteilen, Untertypen essenziell für Wartung

→ strukturelle Ersetzbarkeit automatisch prüfbar, nominale Ersetzbarkeit nicht automatisch

Verbung übernimmt Code aus Oberklasse

→ erspart etwas Schreibarbeit, garantiert keine Ersetzbarkeit

praktischer Ansatz: Untertypen und Vererbung zu kombinieren versucht

→ Vererbung nur wenn strukturelle Ersetzbarkeit erfüllt

→ keine Garantie für weitere Bedingungen (Bedeutungen von Namen und Kommentaren)

Terminologie: „Untertypen“ wenn alle Bedingungen für Ersetzbarkeit erfüllt (nominal), „Vererbung“ wenn nur strukturelle Ersetzbarkeit erfüllt

→ für Compiler kein Unterschied (Bedeutungen von Namen und Kommentaren ignoriert)

→ Programmierer_innen müssen Ersetzbarkeit für inhaltliche Bedeutungen sicherstellen

Erfolgsfaktoren in der objektorientierten Programmierung

Faktorisierung = Modularisierung = Zerlegung in Modularisierungseinheiten (Objekte)

- → sehr viele Möglichkeiten unterschiedlicher Qualität zur Faktorisierung
- → erleichtert gezielten Einsatz von Erfahrung
- → ermöglicht Wiederverwendung durch Ersetzbarkeit
- → Refaktorisierung (Änderung der Faktorisierung) durch viele Werkzeuge unterstützt
- → überfordert Anfänger_innen

zyklische Entwicklungsprozesse

- → Analyse, Design, Implementierung, Verifikation überlappend, nicht hintereinander
- → frühes Feedback, Anforderungen änderbar, aber Fortschritt schwerer planbar

Einsatz von Erfahrung

- → sowohl persönliche Erfahrungen als auch Erfahrungen der Gemeinschaft
- → gezielter Einsatz unverzichtbar

Verantwortlichkeiten einer Klasse

definiert durch drei w-Ausdrücke, „ich“ ist ein Objekt der betrachteten Klasse:

was ich weiß	Objektzustand
was ich mache	Objektverhalten
wen ich kenne	sichtbare andere Objekte oder Klassen

wer Klasse entwickelt ist zuständig für Änderungen in den Verantwortlichkeiten der Klasse

→ Werkzeug um klar zu regeln, wer im Team wofür zuständig ist

Klassenzusammenhalt

Klassenzusammenhalt
= Grad des Zusammenhangs zwischen den Verantwortlichkeiten der Klasse
→ Beziehungen innerhalb eines Objekts der Klasse, nicht nach außen

hoch wenn
(1) Variablen und Methoden gut zusammenarbeiten prüfen durch versuchsweises Entfernen
(2) und durch Namen gut beschrieben prüfen durch versuchsweises Umbenennen

Klassenzusammenhalt soll hoch sein
→ Hinweise auf stabile Faktorisierung = Refaktorisierung wahrscheinlich unnötig (gut)
→ Refaktorisierung bei gleichbleibender Qualität eher schwierig (schlecht), aber kaum nötig

„Zusammenhalt“ lässt sich auf andere Arten von Modularisierungseinheiten erweitern

Objektkopplung

Objektkopplung = Abhängigkeit der Objekte voneinander

→ Beziehungen zu anderen Objekten (der gleichen oder anderer Klassen)

Stärke der Objektkopplung hängt ab von
(1) Anzahl der nach außen sichtbare Methoden und Variablen
(2) Anzahl der Methodenaufrufe und Variablenzugriffe in anderen Objekten
(3) Anzahl der Parameter dieser Methoden

Objektkopplung soll schwach sein

→ Hinweis auf gute Kapselung ($\neq$ stabile Faktorisierung)

→ weniger unnötige Beeinflussungen bei Änderungen (Refaktorisierung einfacher)

wenn auf Klassenzusammenhalt und Objektkopplung geachtet wird, führen wenige Refaktorisierungsschritte wahrscheinlich zu stabiler Faktorisierung und guter Kapselung

Wirkungsweise der objektorientierten Programmierung

Objekte als Daten — sehr mächtiges Werkzeug, ohne Maßnahmen undurchschaubar

- → Objekte, Variablen haben nominale abstrakte Datentypen, abstrakt verständlich
- → dynamisches Binden erzwingt abstraktes Verständnis (Kontrollfluss nicht nachvollziehbar)
- → örtlich eingegrenzte Kommunikation über Variablen möglich (innerhalb einer Klasse)
- → offensiver Umgang mit Aliasen (Identität als klares Unterscheidungsmerkmal)

Untertypbeziehungen — notwendigerweise auf zumindest nominaler Basis

- → ermöglichen komplexe Beziehungen zwischen Typen mit garantierter Ersetzbarkeit
- → ermöglichen einfache Wartung und Wiederverwendung in großem Stil
- → erfordern Programmierdisziplin und viel Wissen über Ersetzbarkeit
  da falsch angenommene Untertypbeziehung (Vererbung) schwere Fehler verursacht
- → verlangen gute Dokumentation verwendeter Abstraktionen (z. B. Design-by-Contract)

Einsatz von Erfahrung — persönlich und gemeinschaftlich

- → diverse Entwurfsrichtlinien, die weit über die syntaktische Ebene hinausgehen

Charakter der funktionalen Programmierung

Funktionen als Daten verwendbar,
Seiteneffekte verboten (heute) oder unerwünscht (früher),
Programmierung im Groben gut unterstützt (nominale Abstraktion eher unwichtig)

- → ohne Seiteneffekte keine Kommunikation über gemeinsame Variablen
- → Aliase harmlos, Original und Kopie nicht unterscheidbar (*referenzielle Transparenz*)
- → „sauber“: aufgesammeltes Wissen geht nie verloren
- → Funktion höherer Ordnung = funktionale Form, kann jede Kontrollstruktur ersetzen
- → bei hohen Abstraktionsgraden eher λ-Abstraktion oder strukturelle Abstraktion
- → ausschließlich Rekursion statt Schleifen
- → heute meist vollständig statisch typisiert (Typinferenz), ältere Sprachen dynamisch
- → Lazy-Evaluation einfach

Einfache funktionale Programmierung in Java (1)

```
public class FunctionalCourse {
   public static void main(String[] args) throws IOException {
      if (args.length != 4)  throw ...;
      System.out.println(status(
         scan(new PointsF(2), args[3],
              scan(new PointsF(1), args[2],
                   scan(new PointsF(0), args[1],
                        scan(new StudsF(), args[0], null))))));
   }
   private static Points scan(Func f, String file, Points p) throws ... {
      return f.apply(new Scanner(new FileInputStream(file)), p);
   }
   private static String status(Points p) { ... }
}
interface Func { Points apply(Scanner in, Points p); }
```

Einfache funktionale Programmierung in Java (2)

```
class PointsF implements Func {
   final int part;
   public PointsF(int part) { this.part = part; }
   public Points apply(Scanner in, Points p) { ... }
   private Points modify(Points p, int regNo, int pnt) { ... } ...
}
class StudsF implements Func {
   public Points apply(Scanner in, Points p) { ... } ...
}
class Stud {
   public final int regNo, curr;
   public final String name;
   public Stud(int regNo, int curr, String name) { ... }
}
...
```

→ alle Variablen `final` oder wie `final` verwendet
→ Objekte von Klassen ohne Methoden als Records
→ statische Methoden und Objekte als Funktionen
→ Objektvariablen aus Umgebung, Hilfsfunktionen

Paradigmenvergleich

	funktional	prozedural	objektorientiert
Hauptziel	Programmiereffizienz	gute Kontrolle	langfristige Wartung
wichtigste Daten	Funktionen	Zahlen, Arrays	Objekte
Programmieren im	Feinen	Feinen	Groben
Abstraktionsform	strukturell, λ, (nominal)	λ, (nominal)	nominal
Abstraktionslevel	niedrig bis hoch	niedrig	hoch
Seiteneffekte	verboten	wichtig	wichtig
Umgang m. Aliasen	referentielle Transp.	problematisch	Identität
Erlernbarkeit	mittelmäßig	einfach	schwer
Fehleranfälligkeit	niedrig	hoch	mittel
Typisierung (meist)	statisch (Typinferenz)	statisch, dynam.	stark
besondere Eignung	komplexe Algorithmen	hardwarenahe Pr.	große Projekte
kaum geeignet für	hardwarenahe Pr.	große Projekte	kleine Pr., kompl. Alg.