# Ausnahmebehandlung in Java

```
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help {
        if (helpNeeded())
            throw new Help();
    }
}
...
    try { ... }
    catch (Help e) { ... }
    catch (Exception e) { ... }
    finally { ... }
```

# Einsatz von Ausnahmebehandlungen

- Ursachen unvorhergesehener Programmabbrüche finden
  (kaum vermeidbar, außer durch Wiederaufsetzen)
- kontrolliertes Wiederaufsetzen nach Fehlern
  (in der Praxis notwendig, sinnvolles Aufsetzen schwierig)
- vorzeitiger Ausstieg aus Sprachkonstrukten
  (fehleranfällig, vor allem wenn nicht lokal; vermeidbar)
- Rückgabe alternativer Ergebniswerte
  (oft Hinweis auf schlechte Programmstruktur; vermeidbar)

# Einsatzbeispiele für Ausnahmen (1)

Ohne Ausnahmebehandlung:

```
while (x != null)
    x = x.getNext();
```

Mit Ausnahmebehandlung:

```
try {
    while (true)
        x = x.getNext();
}
catch (NullPointerException e) {}
```

fehleranfällig, da Ausnahme auch in `getNext` auslösbar

# Einsatzbeispiele für Ausnahmen (2)

trickreiche Verwendung von Ausnahmen:

```
                                    try { throw x }
if (x instanceof T1) {...}          catch (T1 x) {...}
else if (x instanceof T2 {...}      catch (T2 x) {...}
...                                 ...
else if (x instanceof Tn {...}      catch (Tn x) {...}
else {...}                          catch (Exception x) {...}
```

keine nicht-lokalen Ausnahmen (daher weniger fehleranfällig)

aber beide Varianten schwer wartbar

# Einsatzbeispiele für Ausnahmen (3)

Ohne Ausnahmebehandlung:

```
public static String addA (String x, String y) {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else return "Error";
}
```

Mit Ausnahmebehandlung:

```
public static String addB (String x, String y)
        throws NoNumberString {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else throw new NoNumberString();
}
```

sinnvoller Einsatz, da Fehlerabfragen vermieden werden

# Iterator (Cursor)

Zweck: sequentieller Zugriff auf Elemente eines Aggregats

Anwendungsgebiete:

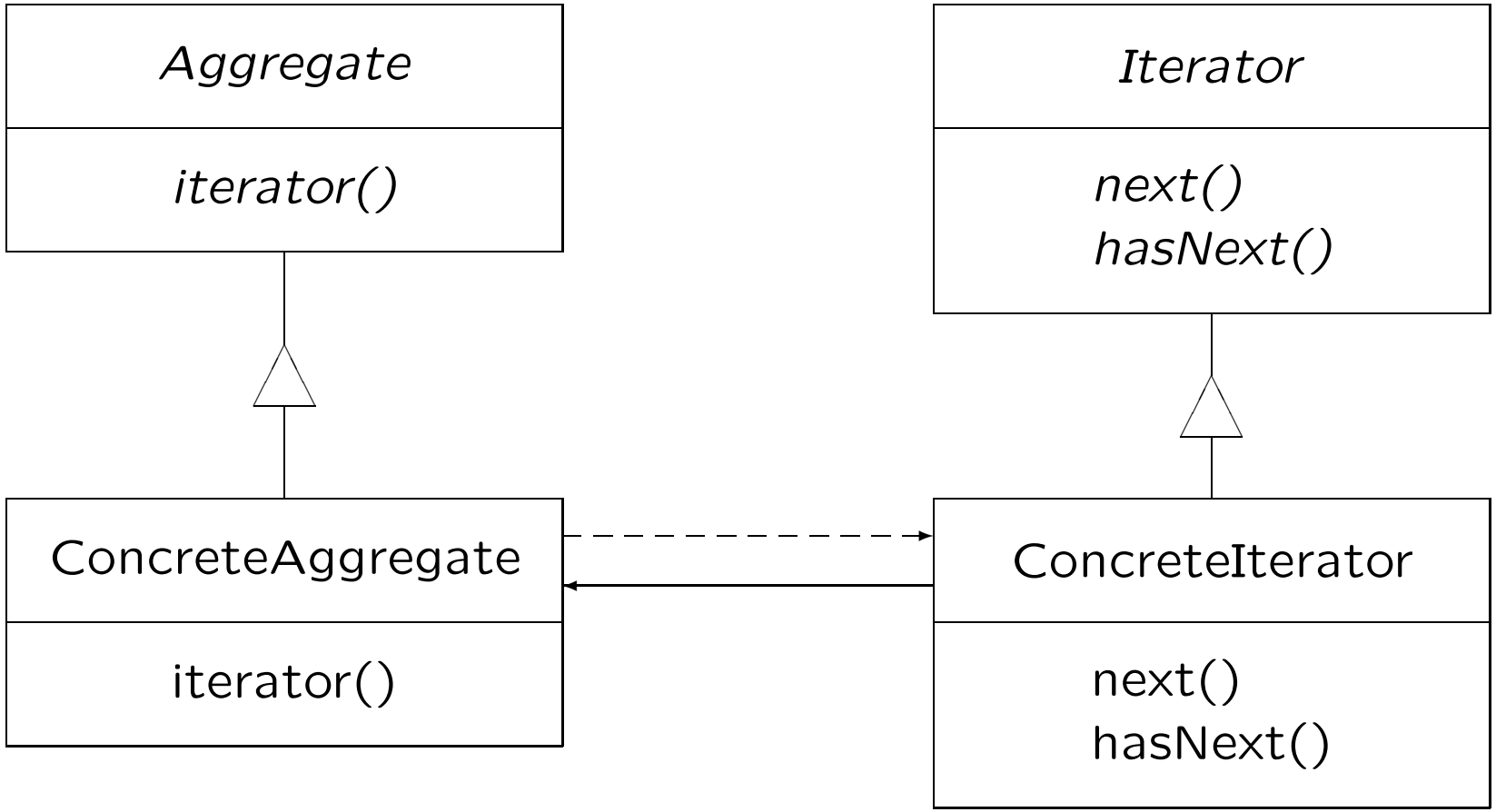
- Zugriff auf Aggregatinhalt

  innere Darstellung bleibt gekapselt

- mehrere Abarbeitungen des Aggregatinhalts

- einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen (polymorphe Iterationen)

# Iterator: Struktur



# Iterator: Eigenschaften

- unterstützen unterschiedliche Arten der Abarbeitung von Aggregaten (mehrere Iteratorklassen pro Aggregatklasse)
- vereinfachen Schnittstelle von „Aggregate“
- mehrere gleichzeitige Abarbeitungen möglich

# Iterator: Implementierungshinweise

- externe Iteratoren flexibler, aber weniger einfach

  extern: Anwender holt nächstes Element (siehe Beispiele)

  intern: Iterator wendet Operation auf alle Elemente an

- interne Iteratoren besser wenn Beziehungen zwischen Elementen bei Abarbeitung zu berücksichtigen sind

- Algorithmus zum Durchwandern des Aggregats in Aggregat oder Iterator definiert

- Aggregatänderungen während der Abarbeitung berücksichtigen (robuster Iterator)

- auch auf leeren Aggregaten brauchbar

# Factory Method (Virtual Constructor)

Zweck: Definition einer Schnittstelle für Objekterzeugung

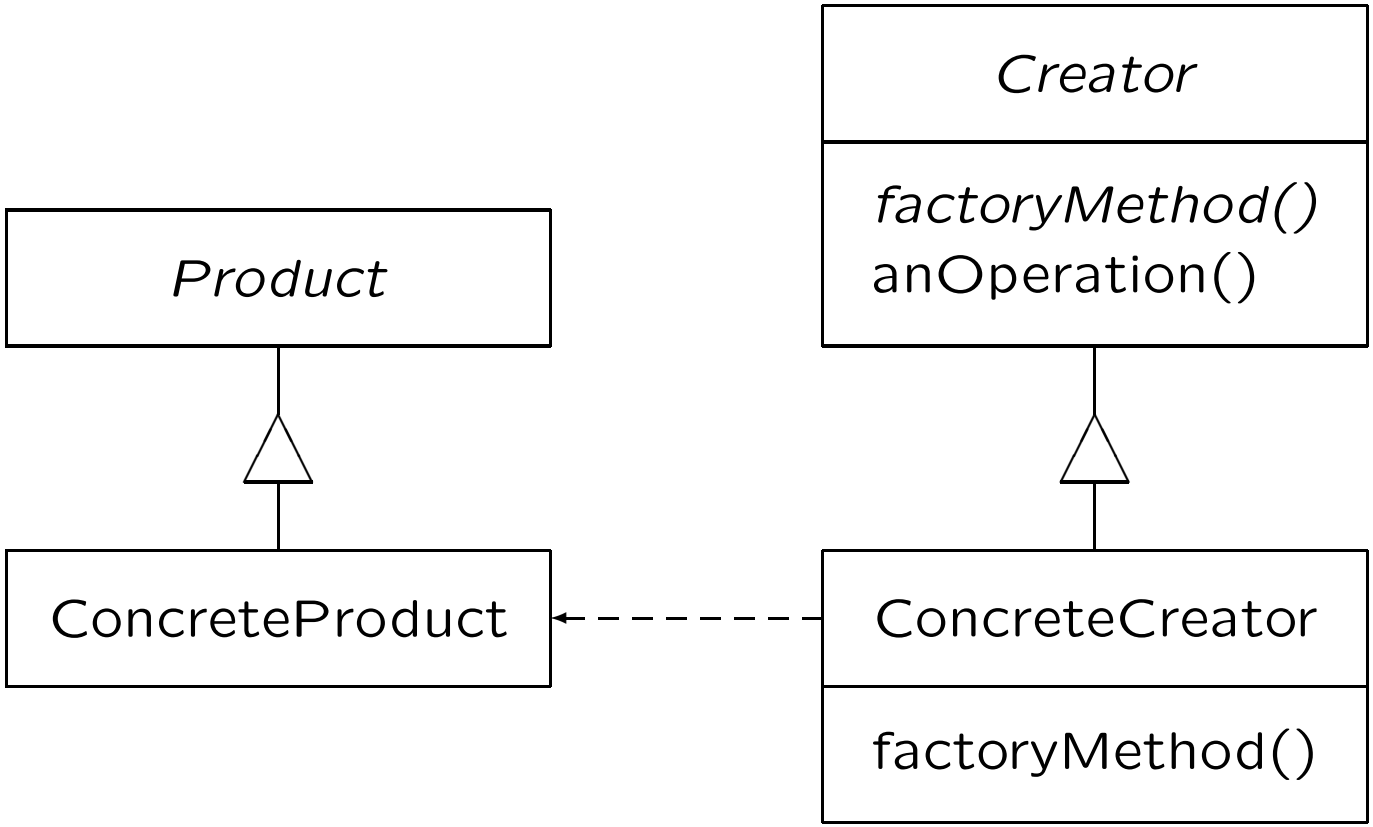
Anwendungsgebiete:

- Klasse neuer Objekte bei Objekterzeugung unbekannt
- Unterklassen sollen Klasse neuer Objekte bestimmen
- Klassen delegieren Verantwortlichkeiten an Unterklassen (Wissen um Unterklasse soll lokal bleiben)

# Factory Method: Beispiel 1

```
abstract class Document { ... }
class Text extends Document { ... }
...   // classes Picture, Video, ...
abstract class DocCreator {
    abstract Document create();
}
class TextCreator extends DocCreator {
    Document create() { return new Text(); }
}
...   // classes PictureCreator, VideoCreator, ...
class NewDocManager {
    private DocCreator c = ...;
    public void set (DocCreator c) { this.c = c; }
    public Document newDoc() { return c.create(); }
}
```

# Factory Method: Struktur



# Factory Method: Eigenschaften

- Anknüpfungspunkte (hooks) für Unterklassen $\Rightarrow$ flexibel

  Entwicklung von Unterklassen vereinfacht

- verknüpfen parallele Klassenhierarchien
  (Creator-Hierachie mit Product-Hierarchie)

  Beispiel: `generiereFutter` vom Typ `Futter` in `Tier` (abstr.)
  erzeugt in `Rind` neue Instanz von `Gras`
  und in `Tiger` neue Instanz von `Fleisch`

- oft große Anzahl an Unterklassen nötig

# Factory Method: Beispiel 2

Anwendung einer Factory Method für lazy initialization

```
abstract class Creator {
    private Product product = null;
    protected abstract Product createProduct();
    public Product getProduct() {
        if (product == null)
            product = createProduct();
        return product;
    }
}
```

ConcreteProduct kann in Java nicht als Typparameter angegeben werden (da nach `new` kein Typparameter erlaubt)

# Prototype

Zweck: Prototyp-Objekt spezifiziert Art eines neuen Objekts

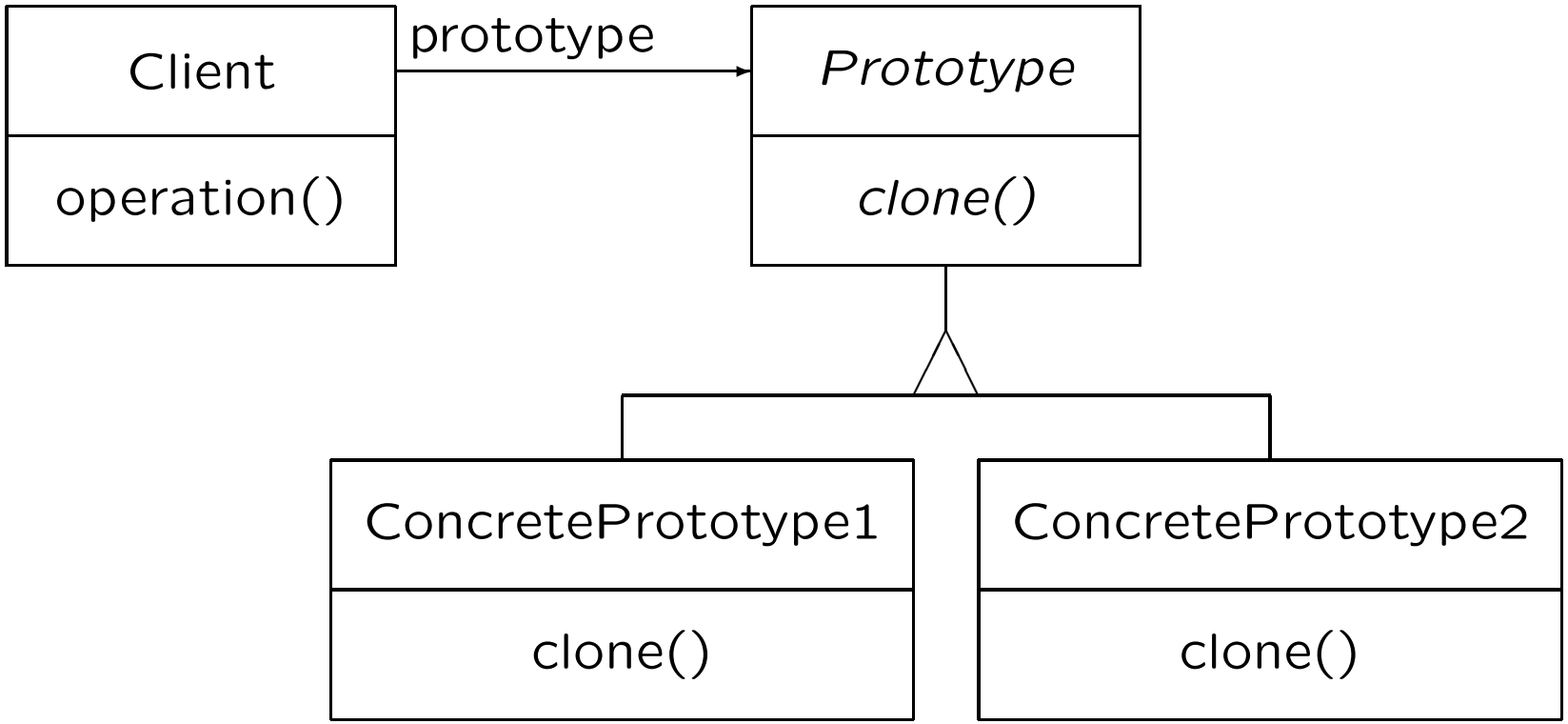
Objekterzeugung durch Kopieren des Prototyps

Anwendungsgebiete:

- Klasse des neuen Objekts erst zur Laufzeit bekannt
- Creator-Klassenhierarchie parallel zu Product-Klassenhierarchie (also Factory Method) soll vermieden werden
- jede Instanz hat einen von wenigen Zuständen

  $\Rightarrow$ Kopieren des Prototyps im gewünschten Zustand einfacher als Konstruktoraufruf

# Prototype: Struktur



# Prototype: Eigenschaften

- verstecken Product-Klassen vor Anwendern

  geänderte Product-Klassen beeinflussen Anwender nicht

- Prototypmenge dynamisch änderbar (Klassenstrukt. nicht)

- Prototypen dynamisch änderbar (Klassen nicht)

  in hochdynamischen Systemen: Verhalten durch Objektkomposition statt Klassendefinition festlegbar

- vermeiden große Anzahl an Unterklassen

- erlauben dynamische Konfiguration von Programmen auch in Sprachen wie C++

# Prototype: Implementierungshinweise

- `clone` in Java für flache Kopien in `Object` vordefiniert
  (wenn `Cloneable` implementiert)
- Erzeugen tiefer Kopien schwierig — zyklische Strukturen
- Prototyp-Manager zur Verwaltung der Prototypen
- `clone` hat keine (geeigneten) Parameter
  daher oft Initialisierungsmethoden nötig
- von dynamischen Sprachen direkt unterstützt