Kapitel 4. Fortgeschrittene OOP

Ausnahmen (Exceptions)
Nebenläufige Prozesse (Threads)
Reflektion (Reflection)
Generizität (Genericity)
Zusammenfassung & Ausblick

Exceptions: Objekte für Ausnahmen

Wozu braucht man Exceptions

Was sind Exceptions

Arten von Exceptions

Umgang mit Exceptions

Beispiele

Wofür braucht man „Exceptions“?

- Aufgabe:
  - Fehlerbehandlung
- Beispiel:
  - zu öffnende Datei existiert nicht
  - der Verweis auf das Dateiobjekt ist **`null`**
  - das Laufzeitsystem ist abgestürzt
- Herkömmliche „Lösung“: Ergebniscodes
  - 0 = Datei existiert nicht
  - -1 = Datei ist nicht lesbar
  - -2 = Datei ist nicht schreibbar
  - -3 = Datei ist gesperrt
  - ...
  - -99 = Plattenplatz erschöpft
- Problem von Ergebniscodes
  - unzuverlässig
    - wird einfach vergessen
  - unübersichtlich
    - Vermischung von Fehlerbehandlung und „normalem“ Ablauf
  - unwartbar
    - neue Fehlermöglichkeiten erfordern Programm-Änderungen an allen Stellen wo sie auftreten können
- Idee (in Java):
  - Fehlerbehandlung wird (wo sinnvoll) vom Compiler erzwungen
  - Trennung von Fehlerbehandlung und „normalem“ Ablauf
  - Abfangen allgemeiner Fehler fängt auch alle ihre aktuellen und zukünftigen Spezialfälle mit ab

Was sind Exceptions?

- „Exception“ (in Java)
  - Objekt, das einen Fehler („Ausnahme“) beschreibt
- Prinzip
  1. Methoden, die auf das Auftreten von Fehlern einers bestimmten Fehlertyps vorbereitet sind, geben dies explizit an mit
     - ```
       try {...Aktion...}
       catch (...Fehlertyp...) {...Fehlerbehandlung...}
       finally {...Was immer ausgeführt werden muss...};
       ```
  2. Methoden, die Fehler feststellen, erzeugen ein Exception-Objekt des passenden Typs
     - ```
       throw new ExceptionClass(...);
       ```
  3. Laufzeitsystem bricht die Ausführung aller Methoden ab, die die Exception nicht behandeln können (kein `catch` für den Exceptiontyp haben)
     - Dabei werden aber `finally`-Blöcke noch ausgeführt
  4. Fortsetzung der Ausführung bei Methode mit „passendem“ `catch`

Der Rahmen try-catch-finally

```
try {
      // Normalanweisungen
}
catch ( Exception1 e ) {
      // Ausnahmebehandlung1
}
catch ( Exception2 e ) {
      // Ausnahmebehandlung2
}
... beliebig viele catch-Blöcke ...
finally {
      // Aufräumarbeiten
}
```

Enthält (sinnvollerweise) Anweisungen der Form

```
if ( Ausnahmebedingung )
   throw new SomeException();
```

Dieser Block wird ausgeführt wenn `SomeException` ein Untertyp von `Exception2` ist.

Dieser Block wird immer ausgeführt, bevor der try-catch-finally-Rahmen auf irgendeine Art verlassen wird.

Sowohl bei normaler Beendigung des try-Blocks, als auch bei vorzeigtiger Beendigung durch eine Exception.

Exception-Beispiel: Lesen aus URLs

Was wir zur Fehlerbehandlung drumherum zusätzlich tun **müssen**!

Was wir eigentlich tun **wollen**!

```
try {
        URL yahoo = new URL("http://www.yahoo.com/");
        InputStream in = yahoo.openStream();
        ...
        in.read();
        ...;
        in.close();

} catch (MalformedURLException mue) {
        System.out.println("MalformedURLException: " + mue );

} catch (IOException ioe) {
        System.out.println("IOException: " + ioe );
};
```

Reihenfolge wichtig:

- Speziellere Exception vor allgemeinerer!

Denn

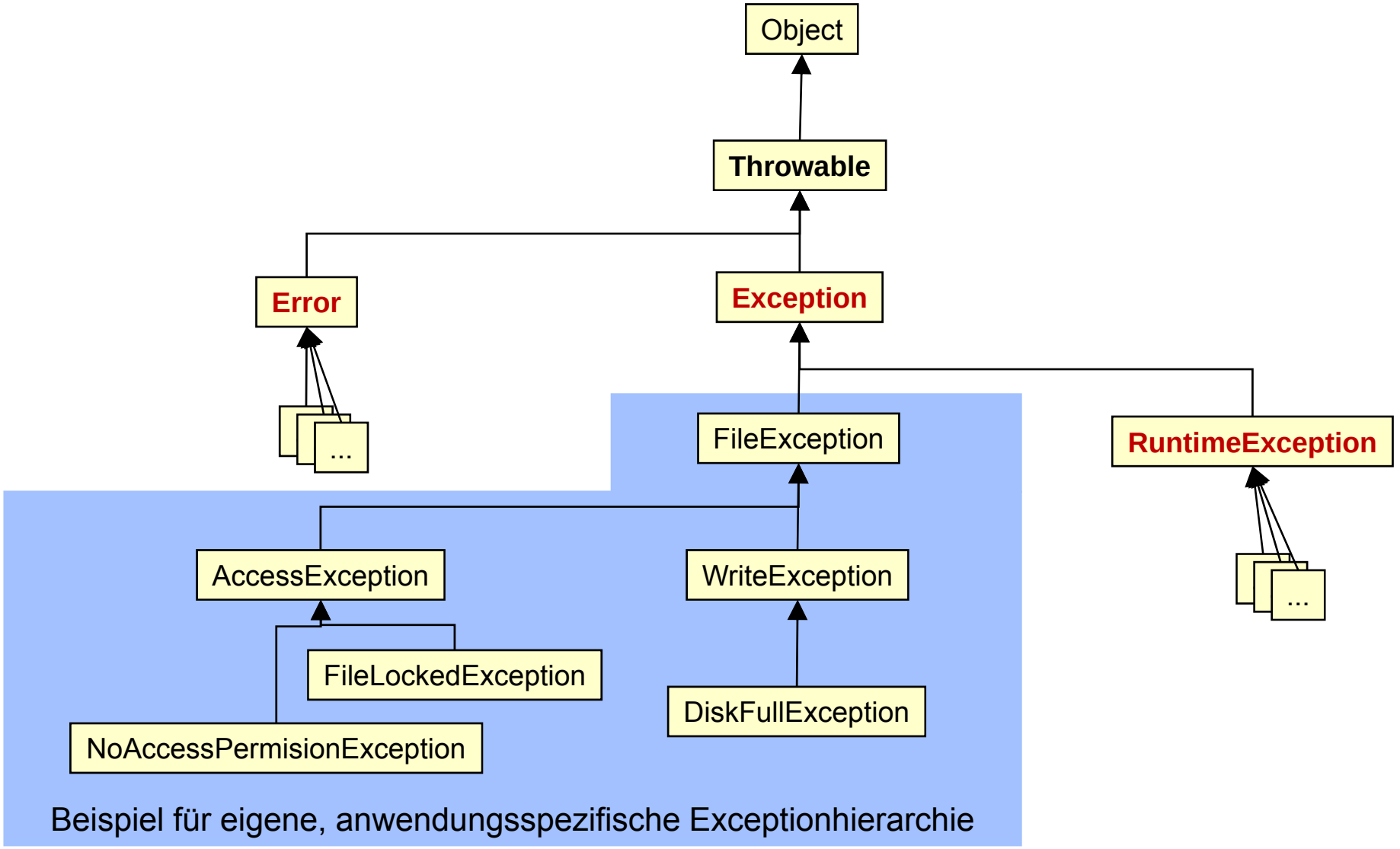
- Nur der erste passende catch-Block wird ausgeführt.
- Danach geht die Ausführung bei einem vorhandenen finally weiter.
- Danach hinter der gesamten try-catch-finally-Anweisung.

Erzeugtes Exception-Objekt kann in die Fehlerbehandlung einbezogen werden

Arten von Ausnahmen

- Errors
  - Fehler des Laufzeitsystems
  - Sollen nicht abgefangen werden
- Runtime Exceptions
  - Typische Programmierfehler, die überall auftreten können
    - Null-Pointer-Zugriff
    - Bereichsüberschreitung bei Arrayzugriffen
  - Müssen nicht abgefangen werden
- Checked Exceptions
  - Typische Fehlerfälle der versuchten Aktion, die einen bestimmten Zustand der Anwendung signalisieren
    - Datei existiert nicht / ist nicht lesbar
  - Müssen abgefangen oder deklariert werden ➔ das wird vom Compiler überprüft!

Die Exception-Hierarchie



Umgang mit Checked Exceptions

- Erste Variante: Exception an Aufrufer durchreichen

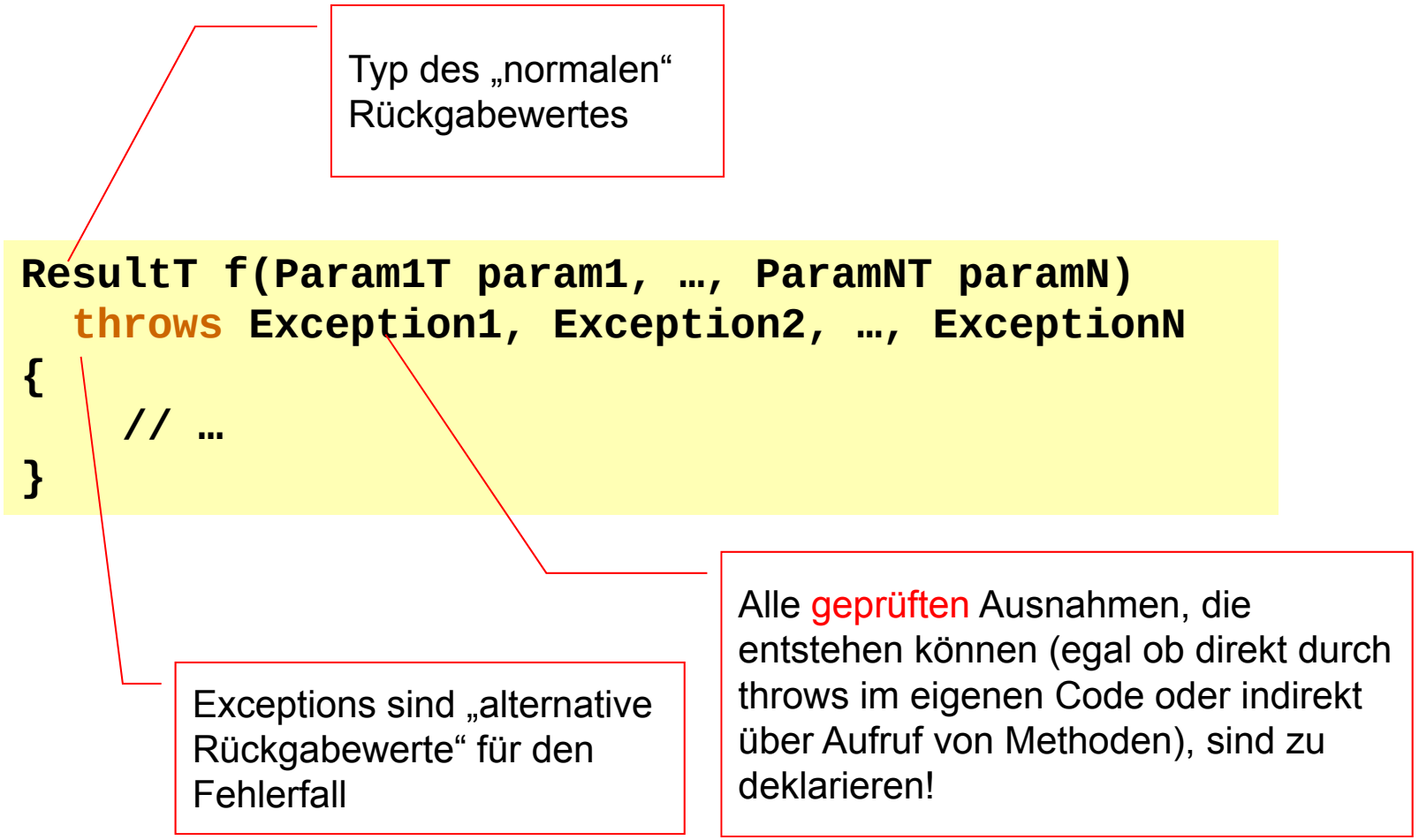
```
public int count(Stream s) throws java.io.IOException {
   int count = 0;
   while (s.read()!= Stream.END)
     count++;
   return count;
}
```

- Zweite Variante: Exception selbst behandeln

```
public int count(Stream s) {
   int count = 0;
   try {
       while (s.read()!= Stream.END)
         count++;
   } catch (java.io.IOException e) { count = 0 };
   return count;
}
```

- Alles andere wird vom Compiler abgelehnt
  - Compiler nutzt dazu die Information aus der throws-Klausel aufgerufener Methoden

Deklaration von Ausnahmen mit throws



Objekte für Ausnahmen (Beispiel 1)

```
// Eigene Exception-Typ-Deklaration:

public class NatException extends Exception {
    public int value;
    public NatException(int v) {
       value = v;

    }
}
```

```
// Beispiel für Aufruf mit Fehlerbehandlung:
try {
    fac = factorial(n);
}
catch (NatException ne) {
    System.err.println("NatException"
      + "in call of factorial() for input "
      + ne.value);
}
```

```
// Fehlerentdeckung und -signalisierung
// durch Erzeugung einer Exception:

public class Test {

    /** Factorial function
     *  @param n: n >= 0
     *  @return n factorial.
     *  @exception NatException n < 0.
     */
    public static int factorial(int n)
           throws NatException {
        // Exception?
        if ( n < 0 ) throw new NatException(n);
        // Recursion
        return (n==0 ? 1 : n * factorial(n-1));
    }
}
```

Test auf Ausnahmefall

Objekte für Ausnahmen (Beispiel 2)

```
/**
 * Klasse für Ausnahmen bei
 * Division durch 0.
 */
class DivisionByZeroException
extends Exception {}
```

Test auf Ausnahmefall

```
public class Complex {
    private double re;  //Realteil
    private double im;  //Imaginärteil

    /** Komplexe Division. Rückgabewert
     *  ist der Quotient von this und d.
     *  @param d Divisor: d != 0.
     *  @exception DivisionByZeroException d == 0.
     */
    Complex divide(Complex d)
       throws DivisionByZeroException {

       if (d.re==0.0 && d.im==0.0)
          throw new DivisionByZeroException();
       //Berechne Reziprokwert von d
       Complex r = new Complex(
          d.re/(d.re*d.re+d.im*d.im),
          -d.im/(d.re*d.re+d.im*d.im) );
       return multComplex(r);

    }
}
```

Ungeprüfte Ausnahmen

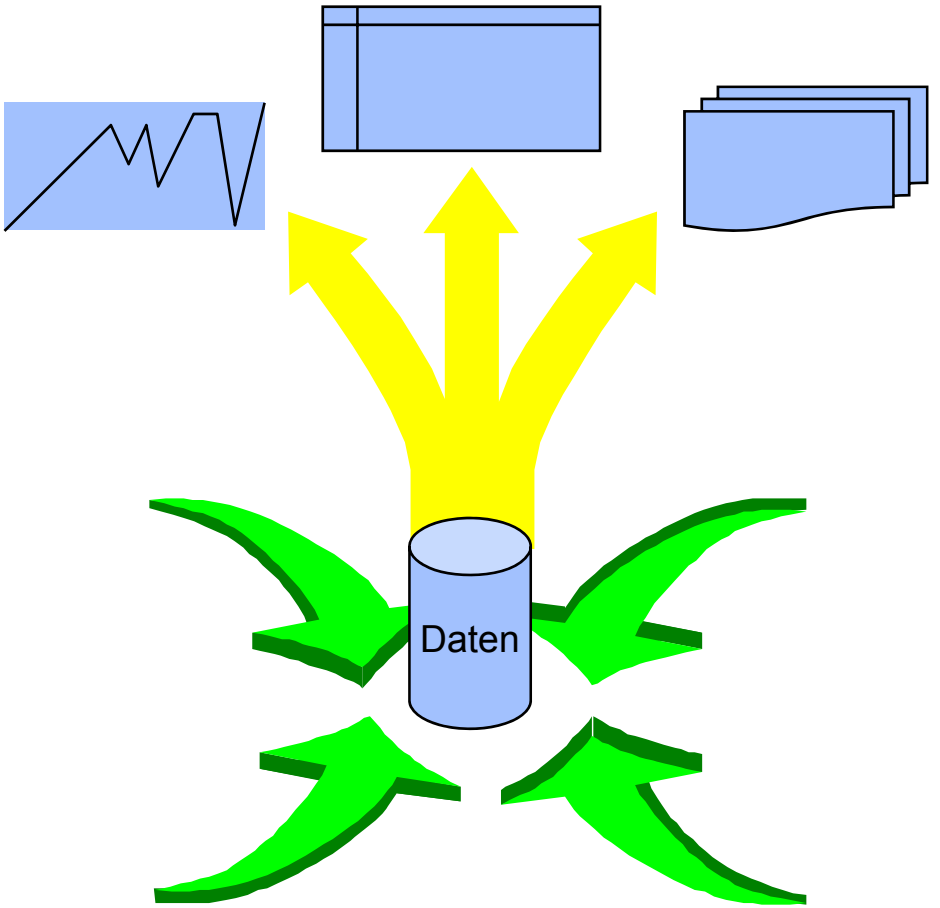
- Ungeprüfte Ausnahmetypen sind alle Klassen, die von **`Error`** oder **`RuntimeException`** erben
  - Zum Beispiel
    - **`NullPointerException`**
    - **`IndexOutOfBoundsException`**
    - **`OutOfMemoryError`**
    - **`StackOverflowError`**
- Ungeprüfte Ausnahmetypen müssen in Methodenköpfen nicht deklariert werden

Nebenläufige Prozesse (‚threads‘)

Wofür braucht man Threads?
Was sind Threads?
Erzeugung von Threads
Synchronisation
Beispiele
Weitere Operationen auf Threads
Zustände von Threads

Wofür braucht man „Threads“?

- Aufgabe
  - ineinander verzahnte Abläufe
- Beispiel
  - aus verschiedenen Quellen „gefütterter“ Datenbestand soll immer aktuell angezeigt werden
- Problem
  - herkömmliche Programme sind sequentiell
  - verzahnte Abläufe sind es nicht
    - unvorhersehbare Reihenfolge von Teilabläufen
- Idee
  - Jeden sequentiellen Teilablauf getrennt programmieren
  - System garantiert Synchronisation des Zugriffs auf gemeinsame Datenbestände und gerechte Zeitzuteilung



Wofür braucht man „Threads“? (Fortsetzung)

- Steigerung der Programmeffizienz durch parallele Programmausführung
- Threads können auf Multi-Core-Prozessoren parallel ausgeführt werden
  - Multi-Core Prozessoren de facto „Standard“
    - Schon jetzt Dual-Core bzw. Quad-Core „normal“
    - In den nächsten Jahren Verdoppelung der Anzahl der Cores alle18 Monate erwartet
  - Bemerkung:
    - Taktfrequenz kaum noch steigerbar
    - Signalausbreitung mit Lichtgeschwindigkeit bei 3 GHz Taktfrequenz:
      - Pro Takt 10cm!

Was sind Threads?

- „Thread“ ≈ „Lightweight Process“
  - ➔ sequentieller Kontrollfluss (Thread = Faden)
- Ein Prozeß kann mehrere Threads enthalten, die ...
  - aktiviert und deaktiviert werden können
  - im aktiven Zustand zueinander „parallel“ arbeiten
  - auf gemeinsame Resourcen zugreifen
    - Synchronisation erforderlich!
- Präemptives multithreading
  - Zuteilung und Entzug von Rechenzeit an aktive threads geschieht automatisch
  - Prioritätsgesteuert
- Vorteile
  - effizienter als Prozeßwechsel auf Betriebssystemebene
  - für Entwickler leichter zu handhaben (platformunabhängig)
  - aber: platformabhängiges Verhalten (auf jeder Platform einzeln testen)!

Thread-Definition als Unterklasse von „Thread“

- Thread-Definition
  - Erzeugung einer Unterklasse von Thread
  - Implementierung der run()-Methode

```
class MyThread extends Thread {
  public void run() { ... }
}
```

- Thread-Erzeugung
  - Konstruktor-Aufruf

```
/* Erzeugen 2-er MyThreads: */
   Thread t1 = new MyThread();
   Thread t2 = new MyThread();
```

- Thread-Start
  - Aufruf der start()-Methode (aus Klasse „Thread“ geerbt)

```
/* Starten der neuen threads: */
   t1.start();
   t2.start();
```

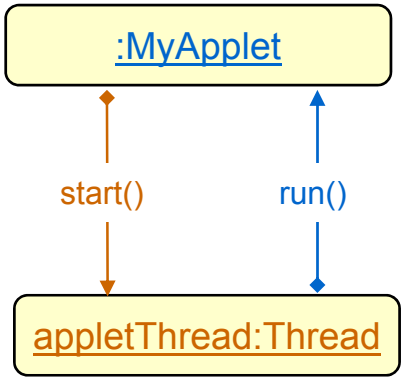
Thread-Definition durch Implementierung von „Runnable“

- Thread-Definition
  - Eigene Klasse implementiert Methode **`run()`** des Interface **`Runnable`**
  - Sie kann von beliebiger Oberklasse ≠ Thread erben!

```
class MyApplet extends Applet implements Runnable {
  Thread appletThread;
  public void run() {
    ...
  }
```

- Thread-Erzeugung
  - Übergeben einer **`Runnable`**-Instanz an **`Thread`**-Konstruktor
  - Aufruf der start()-Methode des Threads

```
 public void start() {              // Start des Applets
    appletThread = new Thread(this);
    appletThread.start();
  }
}
```



  - start()-Methode des Threads ruft die run()-Methode der **`Runnable`**-Instanz auf

Parametrisierung von Threads

- Beobachtung
  - run() und start() sind parameterlos
  - allgemeine Schnittstelle
- Frage
  - Woher weiß der Thread, was genau er tun soll?
- Antwort
  - Parameter werden im Konstruktor übergeben und in Instanzvariablen gespeichert

```
class MyThread extends Thread {
  public void run() { ... }
  // ... siehe unten ...
}
```

```
/* Erzeugen 2-er MyThreads: */
   Thread t1 = new MyThread();
   Thread t2 = new MyThread();

/* Starten der neuen threads: */
   t1.start();
   t2.start();
```

```
/* Instanzvariablen für Parameter: */
  String param1;
  int    param2;

/* Konstruktor speichert Parameter: */
  public MyThread(String p1, int p2){
    param1 = p1;
    param2 = p2;          ..., ...);
  }                       ..., ...);
```

Parametrisierung von Threads

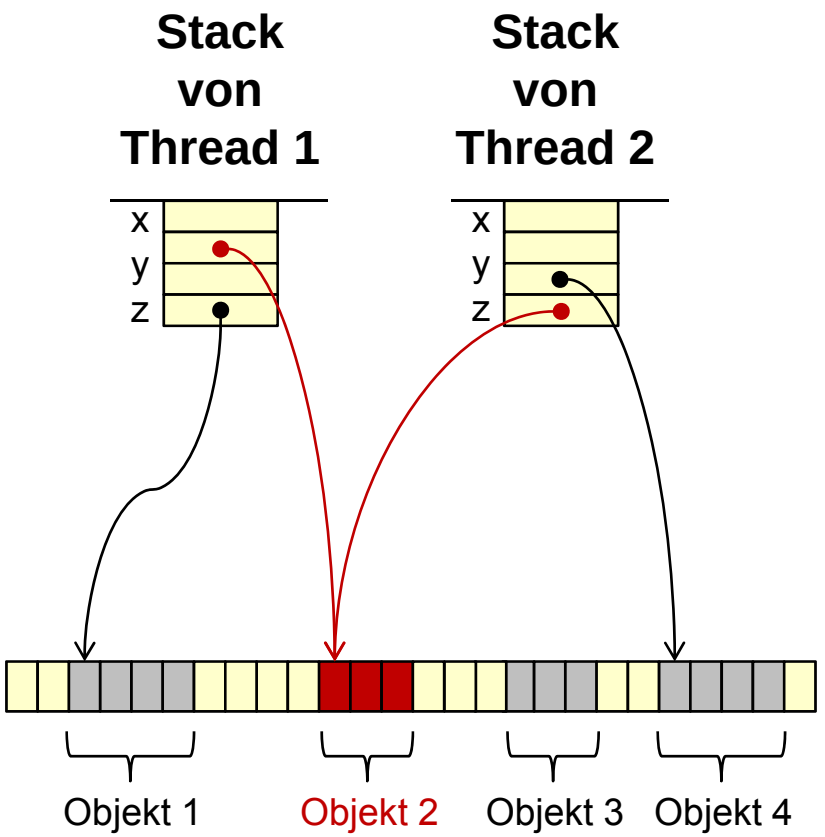
- Beobachtung
  - run() und start() sind parameterlos
  - allgemeine Schnittstelle
- Frage
  - Woher weiß der Thread, was genau er tun soll?
- Antwort
  - Parameter werden in Instanzvariablen gespeichert
  - Ihre Werte werden im Konstruktor übergeben

```
/* Erzeugen 2-er MyThreads: */
   Thread t1 = new MyThread();
   Thread t2 = new MyThread();

/* Starten der neuen threads: */
   t1.start();
   t2.start();
```

```
class MyThread extends Thread {
  public void run() { ... }

  /* Instanzvariablen für Parameter: */
  String param1;
  int    param2;

  /* Konstruktor speichert Parameter: */
  public MyThread(String p1, int p2){
    param1 = p1;
    param2 = p2;
  }

}
```

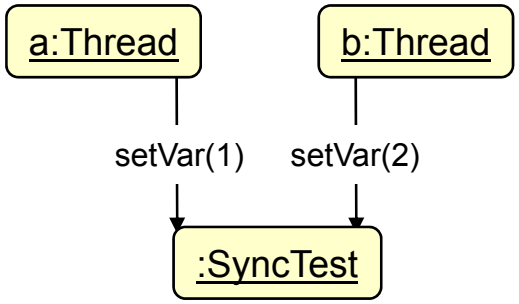
Threads brauchen Synchronisation

- Jeder Thread hat seinen eigenen Stack
  - Zugriffe auf lokale Variablen und Parameter brauchen keine Synchronisation
- Alle Threads teilen sich den Heap
  - Objektzugriffe brauchen dringend Synchronisation!



Thread-Synchronisation (1)

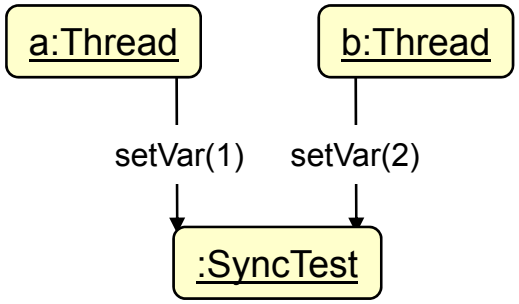
```
class SyncTest {
   int var;
   ...
   public  synchronized setVar(int newValue) {
     var = newValue
}
```



- Schlüsselwort **synchronized**
  - Garantiert, dass die Ausführung der synchronisierten Methode nicht von anderen synchronisierten Methoden der gleichen Klasse ununterbrochen wird.
  - Alle nicht als **synchronized** deklarierten Methoden können von verschiedenen Threads gleichzeitig (d.h. ineinander verzahnt) ausgeführt werden! → Auch gleichzeitig zu synchronisierten Methoden
- Wann nutzen?
  - Nebenläufige Zugriffe auf verschiedene Inhalte sind nicht schlimm
  - … sondern sogar erwünscht (→ Effizienz)
  - … sofern diese Inhalte nicht bestimmten Konsistenzbedingungen unterliegen

Thread-Synchronisation (2)

```
class SyncTest {
   int var;
   ...
   public  synchronized setVar(int newValue) {
     var = newValue
}
```



- Klasse ist “thread-safe”
  - **Naive Definition**: Es findet kein gleichzeitiger Zugriff von zwei Threads auf die gleiche Instanz der Klasse statt
    - Ausnahme: Mehrfache lesende Zugriffe
  - **Mittel**: Alle Methoden der Klasse, die Instanzvariablen verändern sind `synchronized`
  - **Implikation**: Klassen mit öffentlichen (`public`) Variablen sind nie „threadsafe“!
    - Man kann Felder nicht als als `synchronized` deklarieren, sondern nur Methoden!
  - **Korrekte Definition**: Alle Änderungen an Feldern, die zueinander konsistent sein müssen, finden atomar (unterbrechungsfrei) statt
    - Wie kann man das garantieren?

Thread-Synchronisation (3)

- Problem
  - Wie garantiert man thread-sicheres Verhalten von Anweisungsfolgen, für die es keine eigenen Methoden gibt?
- Idee: Synchronization einzelner Blöcke
  - `synchronized` -Statement enthält einen Block dessen Ausführung nicht unterbrochen werden darf
  - Explizite Angabe des Objektes das als Semaphor genutzt wird

Muss zu einem Objekt auswertbar sein.
Dieses Objekt wird als Semaphor genutzt.

```
synchronized( expr ) {
    // Block dessen Anweisungsfolge nicht unterbrechbar ist
}
```

Ein nicht thread-sicheres Program

```
package threads;

public class Counter implements Runnable {
    private int c = 0;

    public void run() {
        int delta = 5;
        int expected = c+delta;

        for (int i=0; i<delta; i++) {
            c++; threadMessage("Counter = " + c);
        }

        int real = c;
        if ( !(real==expected) )
            threadMessage(
              "Expected counter value to be "
              + expected + " but it is " + real);
    }

    public static void main(String[] args) {
        Runnable cnt = new Counter();

        Thread t1 = new Thread(cnt);
        Thread t2 = new Thread(cnt);

        t1.start();
        t2.start();
    }

    // ... siehe rechts oben ...
```

```
    // ... siehe linke Seite ...

    private void threadMessage(String message) {
        String tname = Thread.currentThread().getName();
        System.out.format("%s: %s%n", tname, message);
    }
}
```

Name des aktuellen Threads und Nachricht ausgeben.

Das Feld c delta mal um 1 erhöhen

Das Feld c delta mal um 1 erhöhen und das erwartete Ergebnis (expected) mit dem tatsächlichen (real) vergleichen.

Zwei Threads starten die auf dem gleichen Counter arbeiten.

Ein nicht thread-sicheres Program - mit nicht nachvollziehbaren Ergebnissen

```
package threads;

public class Counter implements Runnable {
    private int c = 0;

    public void run() {
        int delta = 5;
        int expected = c+delta;

        for (int i=0; i<delta; i++) {
            c++; threadMessage("Counter = " + c);
        }

        int real = c;
        if ( !(real==expected) )
            threadMessage(
              "Expected counter value to be "
              + expected + " but it is " + real);
    }

    public static void main(String[] args) {
        Runnable cnt = new Counter();

        Thread t1 = new Thread(cnt);
        Thread t2 = new Thread(cnt);

        t1.start();
        t2.start();
    }

    // ... siehe rechts oben ...
```

```
    // ... siehe linke Seite ...

    private void threadMessage(String message) {
        String tname = Thread.currentThread().getName();
        System.out.format("%s: %s%n", tname, message);
    }
}
```

```
Thread-0: Counter = 1
Thread-0: Counter = 3
Thread-0: Counter = 4
```

```
Thread-0: Counter = 1
Thread-0: Counter = 3
Thread-0: Counter = 4
```

```
Thread-0: Counter = 1
Thread-1: Counter = 2
Thread-1: Counter = 3
Thread-1: Counter = 5
Thread-1: Counter = 6
Thread-1: Counter = 7
Thread-1: Expected counter value to be 6 but it is 7
Thread-0: Counter = 4
Thread-0: Counter = 8
Thread-0: Counter = 9
Thread-0: Counter = 10
Thread-0: Expected counter value to be 5 but it is 10
```

Die Erhöhung des Counters um 5 ist nicht atomar!

Ein fast thread-sicheres Program

```
package threads;

public class Counter implements Runnable {
    private int c = 0;

    synchronized
    private int incrementBy(int delta) {
       int expected=c+delta;

       for (int i=0; i<delta; i++) {
          c++; threadMessage("Counter = " + c);
       }
       return expected;
    }

    public void run() {
      int expected = incrementBy(5);




      if (! ( c == expected))
          threadMessage("Expected ... "
          + expected + " but it is " + real);
    }

    public static void main(String[] args) {
       // ... wie vorhin ...
    }
```

```
    // ... siehe linke Seite ...

    private void threadMessage(String message) {
       String tname = Thread.currentThread().getName();
       System.out.format("%s: %s%n", tname, message);
    }
}
```

Name des aktuellen Threads und Nachricht ausgeben.

Unterbrechungsfrei das Feld c delta mal um 1 erhöhen und gleichzeitig den erwarteten Ergebniswert berechnen

Das Feld c 5 mal um 1 erhöhen und das erwartete Ergebnis (expected) mit dem tatsächlichen (real) vergleichen mit Hilfe der Methode incrementBy().

Zwei Threads starten die auf dem gleichen Counter arbeiten.

Ein fast thread-sicheres Program

```
package threads;

public class Counter implements Runnable {
    private int c = 0;

    synchronized
    private int incrementBy(int delta) {
       int expected=c+delta;

       for (int i=0; i<delta; i++) {
          c++; threadMessage("Counter = " + c);
       }
       return expected;
    }

    public void run() {
      int expected = incrementBy(5);




      if (! ( c == expected))
          threadMessage("Expected ... "
          + expected + " but it is " + real);
    }

    public static void main(String[] args) {
       // ... wie vorhin ...
    }
```

```
    // ... siehe linke Seite ...

    private void threadMessage(String message) {
       String tname = Thread.currentThread().getName();
       System.out.format("%s: %s%n", tname, message);
    }
}
```

```
Thread-0: Counter = 1
Thread-0: Counter = 2
Thread-0: Counter = 3
Thread-0: Counter = 4
Thread-0: Counter = 5
```

```
Thread-0: Counter = 1
Thread-0: Counter = 2
Thread-0: Counter = 3
Thread-0: Counter = 4
Thread-0: Counter = 5
```

```
Thread-0: Counter = 2
Thread-0: Counter = 3
Thread-0: Counter = 4
Thread-0: Counter = 5
Thread-1: Counter = 6
Thread-1: Counter = 7
Thread-1: Counter = 8
Thread-1: Counter = 9
Thread-1: Counter = 10
Thread-0: Expected counter value to be 5 but it is 6
```

Die Erhöhung des Counters ist atomar, aber der Test ob der Wert des Counters der Erwartete ist, ist nicht atomar!

Ein thread-sicheres Program

```
package threads;

public class Counter implements Runnable {
    private int c = 0;

    synchronized
    private int incrementBy(int delta) {
       int expected=c+delta;

       for (int i=0; i<delta; i++) {
          c++; threadMessage("Counter = " + c);
       }
       return expected;
    }

    public void run() {
      int expected;
      int real;

      synchronized (this) {
         expected=incrementBy(5);
         real = c;
      }

      if (! (real==expected))
          threadMessage("Expected ... "
          + expected + " but it is " + real);
    }

    public static void main(String[] args) {
       // ... wie vorhin ...
    }
```

```
    // ... siehe linke Seite ...

    private void threadMessage(String message) {
       String tname = Thread.currentThread().getName();
       System.out.format("%s: %s%n", tname, message);
    }
}
```

Name des aktuellen Threads und Nachricht ausgeben.

Unterbrechungsfrei das Feld c delta mal um 1 erhöhen und gleichzeitig den erwarteten Ergebniswert berechnen

Test auf erwartetes Ergebnis ist Teil eines synchronisierten Blocks.

Die Ausgabe muss nicht synchronisiert werden, da sie sich nur auf lokale Variablen bezieht!

Zwei Threads starten die auf dem gleichen Counter arbeiten.

Ein thread-sicheres Program - mit immer dem gleichen Ergebnis

```
package threads;

public class Counter implements Runnable {
    private int c = 0;

    synchronized
    private int incrementBy(int delta) {
       int expected=c+delta;

       for (int i=0; i<delta; i++) {
          c++; threadMessage("Counter = " + c);
       }
       return expected;
    }

    public void run() {
      int expected;
      int real;

      synchronized (this) {
         expected=incrementBy(5);
         real = c;
      }

      if (! (real==expected))
          threadMessage("Expected ... "
          + expected + " but it is " + real);
    }

    public static void main(String[] args) {
       // ... wie vorhin ...
    }
```

```
    // ... siehe linke Seite ...

    private void threadMessage(String message) {
       String tname = Thread.currentThread().getName();
       System.out.format("%s: %s%n", tname, message);
    }
}
```

```
Thread-0: Counter = 1
Thread-0: Counter = 2
Thread-0: Counter = 3
Thread-0: Counter = 4
Thread-0: Counter = 5
Thread-1: Counter = 6
Thread-1: Counter = 7
Thread-1: Counter = 8
Thread-1: Counter = 9
Thread-1: Counter = 10
```

Thread Steuerung

- Die Synchronisation mittels **`synchronized`**-Methoden und -Blöcken reicht manchmal nicht.
  - Daher gibt es weitere Möglichkeiten, das Zusammenspiel von Threads zu steuern
- Das Zentrale Konzept ist der Erwerb und die Freigabe von „Sperren“ (locks) auf Objekten die als „Monitor“ genutzt werden
  - Siehe Vorlesung „Technische Informatik“ oder Google und Wikipedia zum Thema Monitore und „gegenseitiger Ausschluss“ (mutual exclusion)
- Nachfolgend nur die entsprechenden Methoden von Threads, ohne Erläuterung der Hintergründe

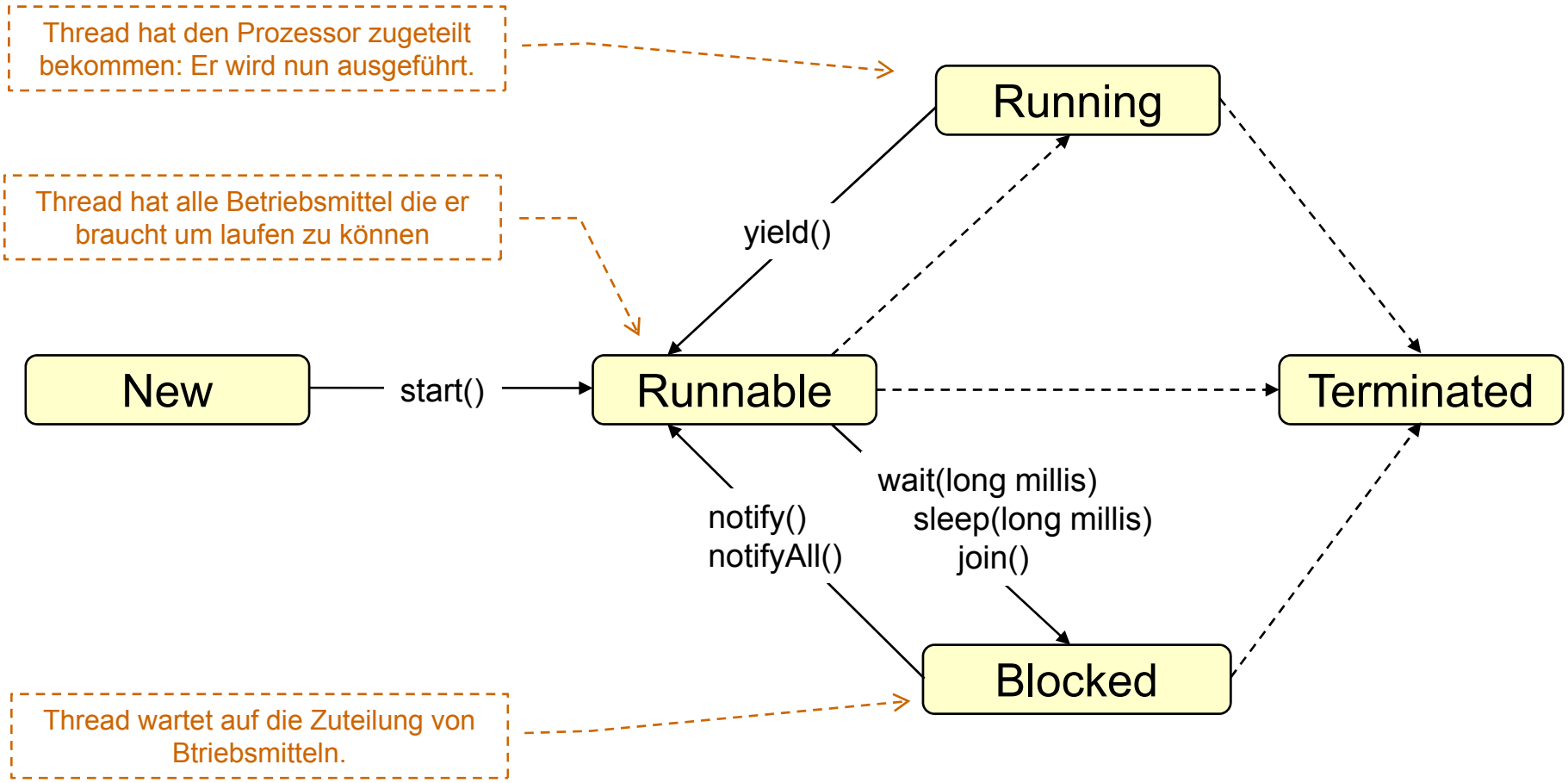
Thread Steuerung in Java (1)

- Thread.sleep(millis)
  - Aktiven Thread *n* Millisekunden „schlafen“ lassen.
  - In dieser Zeit verbraucht der Thread keine Ressourcen!
  - Der Thread gibt dabei seine „Sperren“ (locks) nicht auf!
- Thread.yield()
  - Aktiven Thread anhalten (mit Freigabe seiner Sperren) um auch andere Threads dran kommen zu lassen
- *t*.join(*n*)
  - Aufrufenden Thread (mit Freigabe seiner Sperren) *n* Millisekunden auf das Ende von Thread *t* warten lassen. Danach weitermachen, falls *t* nicht beendet wurde.
  - Ohne Parameter: Beliebig lange auf Ende von *t* warten.

Thread Steuerung in Java (2)

- object.wait(n)
  - Der aufrufende Thread gibt seine Sperre auf object auf!
  - Der aufrufende Thread wartet anschliessend *n* Millisekunden auf die Freigabe der Sperre von *object* durch andere Threads.
  - Wenn die Zeit verstrichen ist oder object freigegeben wurde, wird der Aufrufer „geweckt“, d.h. wieder in die Warteliste für object aufgenommen.
  - Parameterlose Variante: Beliebig lange warten.
- object.notify()
  - Der aufrufende Thread gibt seine Sperre auf object auf!
  - Einer der via wait() auf object wartenden threads wird „geweckt“.

Thread-Steuerung und Zustände von Threads



- Zustandsübergänge für die es keine expliziten Methoden gibt (gestrichelt dargestellt) erfolgen automatisch

Weitere Themen / Problemfelder

- Verhungern (Starvation)
  - Thread wartet unendlich, obwohl er dran kommen könnte
  - Herausforderung: „Faires“ Verteilungsverfahren, das Verhungern verhindert
    - Aufgabe der Implementierer des Java Laufzeitsystems
- Verklemmung (Deadlock)
  - Unendliches gegenseitiges Warten, da jeder der beteiligten Threads ein paar Ressourcen festhält die ein anderer braucht
  - Selbst drauf achten, dass man Ressourcen in der „richtigen“ Reihenfolge belegt und wieder freigibt!

Weitere Infos

- Java Tutorial
  - http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html
- „Java als erste Programmiersprache“, Kapitel 18

Reflektion

Reflektion allgemein

Reflektion in Java

Reflektive Architektur von Java

Die Klassen Class und Object

Beispiele

Nutzen und Risiken

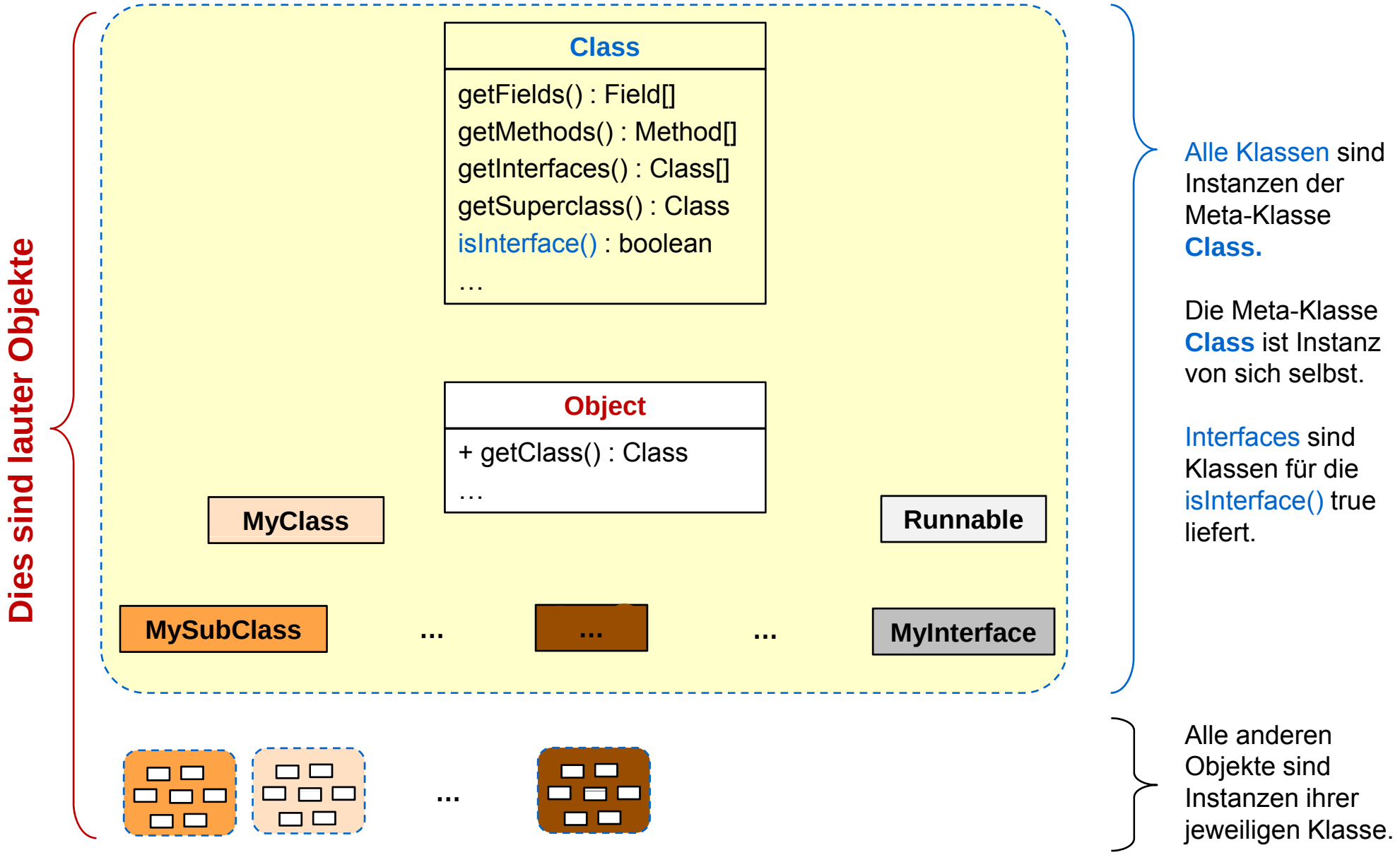
Reflektion (reflection)

- Reflektive Sprache
  - Definition: Die Elemente der Sprache sind in der Sprache selbst darstellbar
  - Nutzen: Sie können somit zur Laufzeit mit den in der Sprache verfügbaren Mitteln analysiert und bearbeitet werden
- Reflektive objektorientierte Sprache
  - „Alles ist ein Objekt!“ – Auch die Sprachkonzepte!!!
    - Typen, Klassen, Methoden, Felder, …
    - Ausdrücke, Anweisungen
  - Beispiele konsequent reflektiver OO-Sprachen
    - Smalltalk, Self, …

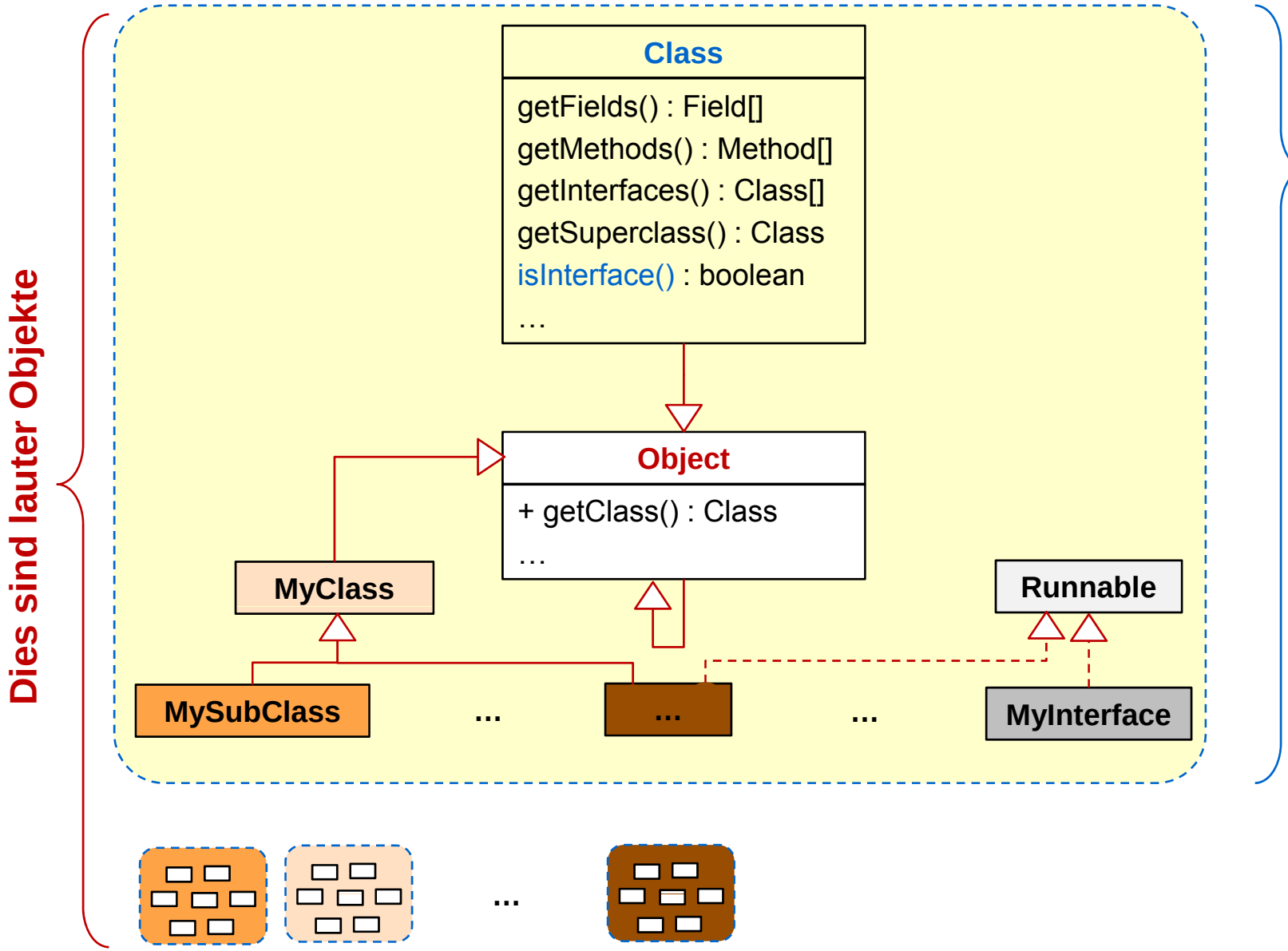
Java als teil-reflektive Sprache

- Klassen sind Objekte
  - Sie sind Instanzen der Klasse **`Class`**
- Jedes Objekt kennt sein Klassenobjekt
  - getClass() wird von **`Object`** an alle Klassen vererbt
- Methoden und Felder können als Objekte dargestellt werden
  - Dazu bietet **`Class`** passende Methoden
  - Reflektiv dargestellte Methoden und Felder können zur Laufzeit analysiert werden (Introspektion / introspection)
  - Aufrufe an sie können dynamisch erstellt werden
  - Es ist aber nicht möglich, sie zu verändern → Sicherheit!

Reflektive Architektur von Java (1)



Reflektive Architektur von Java (2)

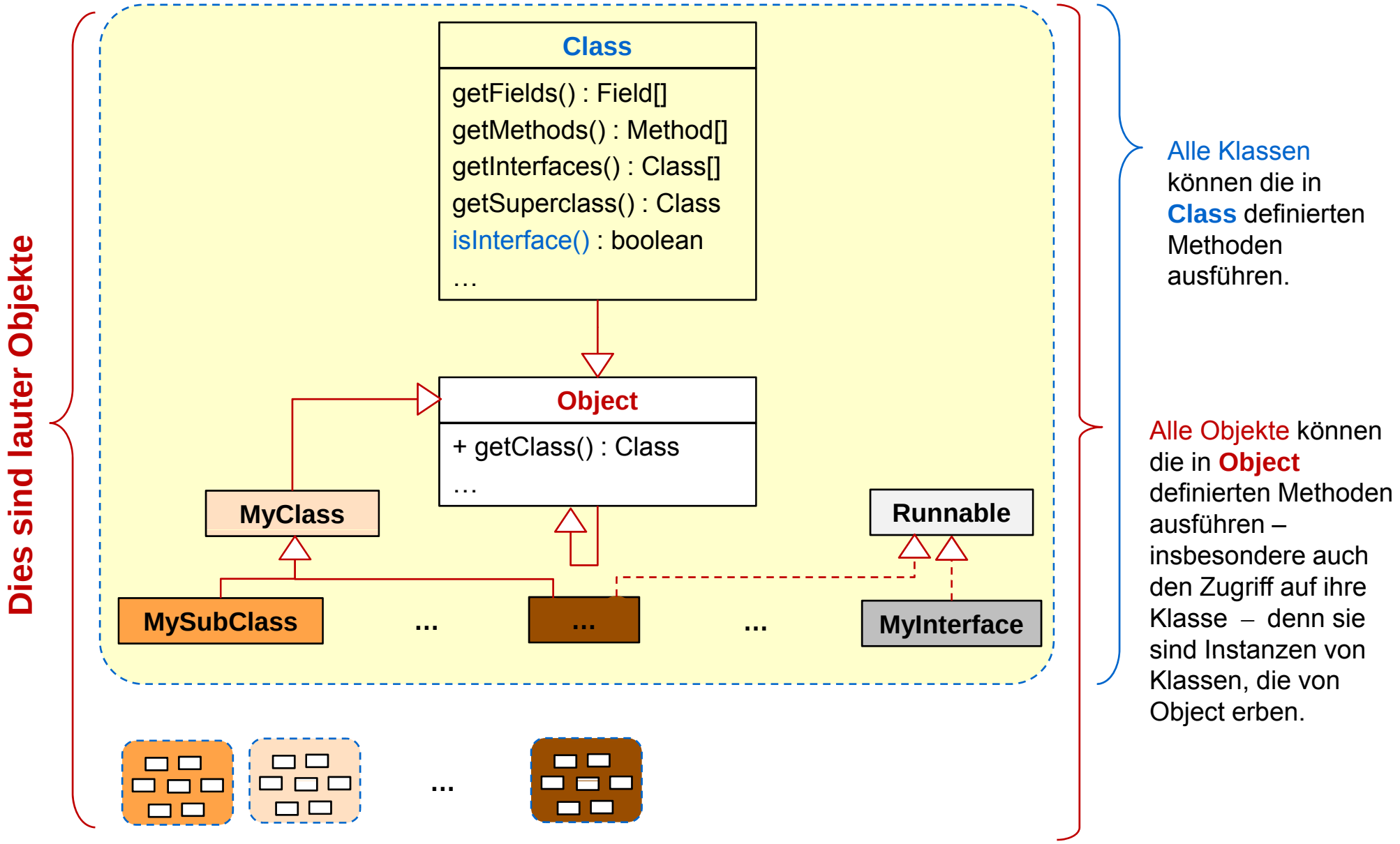


Alle Klassen erben von **Object** – direkt oder indirekt.

Die Klasse **Object** erbt von sich selbst.

Interfaces erben nicht von **Object**

Reflektive Architektur von Java (3)



Die Klasse „Class“

- Bietet Zugriff auf
  - Klassen via ihren Namen
  - die Mitglieder von Klassen
    - Felder, Methoden, Klassen
  - die Obertypen von Klassen
    - Oberklasse und implementierte Interfaces
- Ermöglicht die Erzeugung von Instanzen
  - newInstance()
- Und noch vieles mehr
  - Siehe java.lang.Class

Class
forName(String name) : Class getField(name:String) : Field getFields() : Field[] getMethod(name:String, Class[] paramTypes) : Method getMethods() : Method[] getInterfaces() : Class[] getSuperclass() : Class getDeclared…() : … // Fields, Methods, Constructors, Classes newInstance() : Object …

- Weitere Analysemethoden
  - java.lang.reflect.**Field**, java.lang.reflect.**Method**, …
  - … incl. Umgehung von Zugriffsschutz für spezielle Anwendungen

Die Klasse „Object“

- Bietet Zugriff auf die Klasse eines Objektes
  - getClass()
- Ermöglicht Objektvergleich
  - equals()
  - hashCode()
- Ermöglicht Stringdarstellung von Objekten
  - toString()
- Ermöglicht Objekte als Monitore zu nutzen
  - notify(), notifyAll(), wait(), …

Object

+ getClass() : Class
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
+ notify() // und weitere Varianten
+ wait() // und weitere Varianten
# clone() : Object 	

- Ermöglicht Kopieren von Objekten
  - clone()
  - Diese Methode ist „protected“
  - Damit man sie aufrufen kann muss das Interface Clonable implementiert werden.

Nutzungs-Beispiele

- Finde Klasse eines Objektes heraus

```
Object o;

System.out.println("Dies Objekt hat den Typ " +
                   o.getClass().getName());
```

```
Ausgabe für o == "abc":
"Objekt hat Typ java.lang.String"
```

- Finde heraus ob der Typ eines Objektes ein Subtyp des Typs eines Anderen ist

```
Object o1, o2;
Class c1 = o1.getClass();
Class c2 = o2.getClass();

if (c1.isAssignableFrom(c2))
     System.out.println("Is assignable");
else System.out.println("Is not assignable");
```

```
Ausgabe für o1 == "abc" und o2 == "cde":
 „Is assignable"
```

Reflektion über Reflektion

Vorteile / Anwendungen

- Erweiterbarkeit
  - Anwendung kann unbekannte Klassen dynamisch über ihren Namen instanzieren
- Entwicklungsumgebungen
  - Anzeige der Mitglieder von Klassen, Typinformation, …
  - Debugger brauchen Zugriff auf private Mitglieder
- Testwerkzeuge
  - Analyse was getestet werden soll

Nachteile / Risiken

- Schlechte Performanz
  - Standardoptimierungen greifen nicht für reflektiven Code
- Sicherheitsrisiken
  - Reflektiver Code läuft nur wenn der Ausführende die nötigen Rechte hat
- Zugriffsschutz-Umgehung
  - zerstört Kapselung
- Kein statische Typprüfung
  - Laufzeitfehler / Exceptions

Weitere Informationen

- Java Tutorial
  - http://java.sun.com/docs/books/tutorial/reflect/index.html

universitätbonn

Generische Datentypen und Generische Programmierung

→Wird im nächsten Kapitel zusammen mit Verwendungsbeispielen aus den Java-Standardbibliotheken besprochen

Zusammenfassung und Ausblick

Lebenszyklus von Objekten

- Typdefinition → Interfaces
- Typimplementierung → Klassen

Statisch (Programm)

- Objekt-Erzeugung → Instanzen
- Objekt-Aktivität → Nachrichten
- Objekt-Tod → Garbage Collection

Laufzeit

Rückblick: Verschiedene Arten von Hierarchien

- Spezialisierungs-Hierarchien
  - "ist Spezialfall von"
  - Konzeptuelle Beziehung aus Sicht des Anwendungsfeldes (Endandwender)
- Subtyp-Hierarchien
  - "ist einsetzbar für"
  - Austauschbarkeit / Verwendbarkeit für den Benutzer einer Klasse
- Vererbungs-Hierarchien
  - "erbt Code von"
  - inkrementelle Code-Wiederverwendung für den Autor einer Klasse

Gute Modellierung erfordert diese drei Dinge auseinanderzuhalten (die Unterschiede zu verstehen) bzw. sie zur Deckung zu bringen (so zu modellieren, dass die Hierarchien möglichst deckungsgleich sind).

Objekt-orientierte Programmierung = Wartbarkeit + Wiederverwendbarkeit

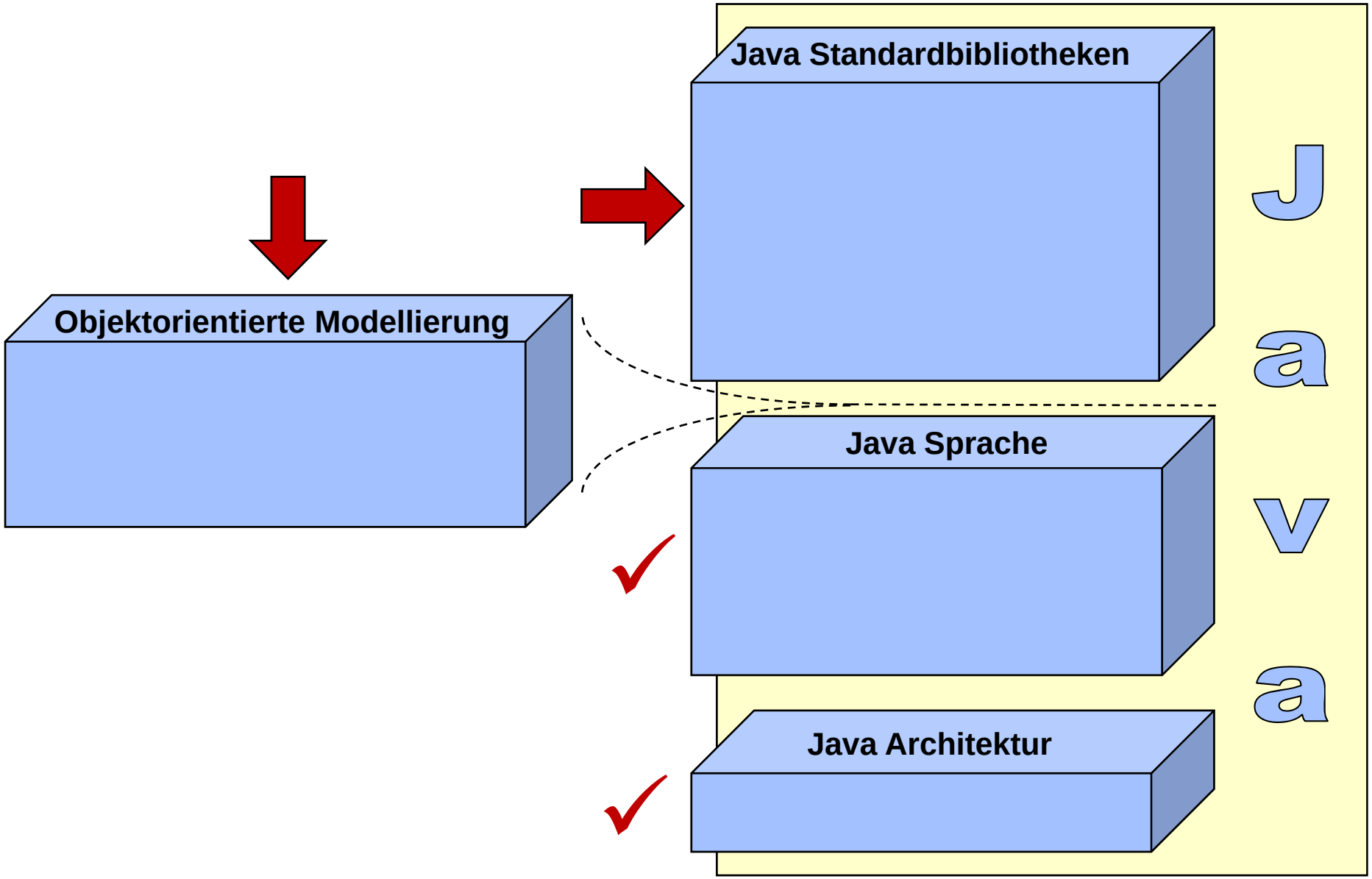
- Kapselung ➡ Änderbarkeit
  - Änderung der Implementierung einer Klasse erfordert keine Änderung anderer Klassen
- Polymorphismus und Dynamisches Binden ➡ Erweiterbarkeit
  - Hinzufügen neuer Klassen erfordert keine Änderung von Benutzer-Klassen
- Vererbung ➡ Wiederverwendbarkeit
  - Wiederverwendung von "geerbtem" Code in Unterklassen

➔ Weniger Implementierungs- und Wartungsaufwand

➔ Kürzere Entwicklungszeiten

Da die Wartung bis zu 80% der Kosten einer Software verschlingt,
gibt es zur objekt-orientierten Programmierung keine Alternativen
wenn termin- und kostengerechte Softwareentwicklung angestrebt ist!

Rückblick und Ausblick



Generics in Java

- Since Java 5.0
- Changes to the JDK classes:
  - Collections and Iterators are generic
- No changes to the Class File Formats or the JVM
  - The compiler checks type parameter and removes them
  - Parameterized types will not be macro expanded (unlike C++-templates!)
  - Some limitations (arrays, instanceof) because of limited runtime support*

* **Gilad Bracha**, „Generics in the Java Programming Language“ java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf