Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 12)

Typparameter

(Generics in Java)

Lernziele

- Sie wissen, was mit dem Konzept eines generischen Typparameters gemeint ist.
- Sie sind mit bereits existierenden Java-Klassen vertraut, die generische Typparameter verwenden.
- Sie können eigene Klassen erstellen, die generische Typparameter nutzen.

Motivation: Warum Generics?

- Flexible, wiederverwendbare Datenstrukturen
- Typsicherheit zur Kompilierzeit
- Vermeidung von Casts
- Klare Code-Struktur

Beispiel: Ohne Generics

```
import java.util.ArrayList;

ArrayList list = new ArrayList();
list.add("Hallo");
list.add(42);

for (Object obj : list) {
    String str = (String) obj;  // Cast nötig
    System.out.println(str);
}
```

- Problem: **Fehleranfällig** (z. B. beim Casten).
- Laufzeitfehler, wenn falscher Typ.

Beispiel: Mit Generics

```java
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<>();
list.add("Hallo");
// list.add(42);  // Compiler-Fehler: Nur Strings erlaubt

for (String str : list) {
    System.out.println(str);  // Kein Cast nötig
}
```

- Vorteil: Typsicherheit durch den Compiler.

Syntax von Generics

Allgemeine Syntax

```java
class KlasseName<T> {
    private T objekt;

    public KlasseName(T objekt) {
        this.objekt = objekt;
    }

    public T getObjekt() {
        return objekt;
    }
}
```

- `T` ist ein **Typ-Parameter**, z. B. `String` oder `Integer`.

Generische Klasse: Beispiel

```java
public class Box<T> {
    private T inhalt;

    public void setInhalt(T inhalt) {
        this.inhalt = inhalt;
    }

    public T getInhalt() {
        return inhalt;
    }
}

Box<String> stringBox = new Box<>();
stringBox.setInhalt("Hallo Welt");
System.out.println(stringBox.getInhalt());
```

- Vorteil: **Flexibilität** für beliebige Typen.

Verwendung von Generics

1. **In Collections**:
   - `ArrayList<E>`: Generische Liste.
   - `HashMap<K, V>`: Generische Map.
2. **In Methoden**:
   - ```
     public static <T> void printArray(T[] array) {
         for (T element : array) {
             System.out.println(element);
         }
     }
     ```
3. **Mit eigenen Klassen**:
   - Klassen mit Typ-Parametern.
     ```

Bounded Generics

```
public class Bounded<T extends Number> {
    private T zahl;

    public Bounded(T zahl) {
        this.zahl = zahl;
    }

    public double quadrat() {
        return zahl.doubleValue() * zahl.doubleValue();
    }
}

Bounded<Integer> obj = new Bounded<>(5);
System.out.println(obj.quadrat());  // Ausgabe: 25.0
```

- Einschränkung auf Typen: `T extends Number`.

Einführung: Locker-Klasse

```java
public class Locker<T> {
    private T element;

    public void setValue(T element) {
        this.element = element;
    }

    public T getValue() {
        return element;
    }
}
```

Verwendung von Locker

```
Locker<String> stringLocker = new Locker<>();
stringLocker.setValue(":)");
System.out.println(stringLocker.getValue());
```

Ergebnis:

```
:)
```

```
Locker<Integer> integerLocker = new Locker<>();
integerLocker.setValue(5);
System.out.println(integerLocker.getValue());
```

Ergebnis:

```
5
```

Mehrere Typparameter

```java
public class Pair<T, K> {
    private T first;
    private K second;

    public void setValues(T first, K second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return this.first;
    }

    public K getSecond() {
        return this.second;
    }
}
```

Vorhandene Java-Klassen

- `ArrayList<T>` für Listen
- `HashMap<K, V>` für Schlüssel-Wert-Paare
- `Comparable<T>` für vergleichbare Objekte

```
List<String> strings = new ArrayList<>();
Map<String, String> keyValuePairs = new HashMap<>();
```

Einfacher Exkurs: Prinzipien der Programmierung

- Abstraktion: Generics trennen die konkreten Datentypen von der allgemeinen Funktionsweise.
- Modularisierung: Eine generische Klasse kann an vielen Stellen wiederverwendet werden.
- Wartbarkeit: Änderungen in einer generischen Klasse können global positive Effekte haben.

Type Erasure in Java

- **Type Erasure** entfernt generische Typen zur Laufzeit.
- Generics existieren nur zur **Compile-Zeit**.
- Zur **Laufzeit** wird der generische Typ auf einen "Erasure" reduziert:
  - **Unbeschränkte Typen** (z. B. `<T>`) → `Object`
  - **Beschränkte Typen** (z. B. `<T extends Number>`) → `Number`

Beispiel: Generics mit Type Erasure

Code zur Compile-Zeit:

```java
public class Locker<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}

Locker<String> locker1 = new Locker<>();
Locker<Integer> locker2 = new Locker<>();
```

Beispiel: Generics mit Type Erasure

Laufzeit (nach Type Erasure):

```java
public class Locker {
    private Object item;

    public void setItem(Object item) {
        this.item = item;
    }

    public Object getItem() {
        return item;
    }
}
```

Auswirkungen von Type Erasure

- **Kein Unterschied zur Laufzeit:**

```
System.out.println(locker1.getClass() == locker2.getClass()); // true
```

- **Typ-Sicherheit nur zur Compile-Zeit:**

```
locker1.setItem("Text");
locker2.setItem(42); // Fehler wird zur Laufzeit ignoriert.
```

- **Gefährliche Casts:**

```
Locker rawLocker = locker1; // Raw-Typ
rawLocker.setItem(42); // Typprüfung wird umgangen
String item = locker1.getItem(); // ClassCastException
```

Gefährlicher Cast mit Raw-Typen

Problem:

```java
Locker<String> locker1 = new Locker<>();
Locker rawLocker = locker1; // Raw-Typ
rawLocker.setItem(42); // Typprüfung wird umgangen
String item = locker1.getItem(); // ClassCastException
```

Ablauf:

1. **Raw-Typ ignoriert Generics:**
    - `rawLocker` verliert die Typinformation `<String>`.
    - Compiler kann nicht mehr sicherstellen, dass der richtige Typ verwendet wird.
2. **Einfügen eines falschen Typs:**
    - `rawLocker.setItem(42)` fügt einen `Integer` statt eines `String` ein.
3. **Abrufen des falschen Typs:**
    - `locker1.getItem()` erwartet einen `String`, erhält aber einen `Integer`.
    - **ClassCastException**: `Integer` kann nicht in `String` konvertiert werden.

Warum ist das gefährlich?

1. **Typprüfung wird umgangen:**
   - Raw-Typ deaktiviert die Generics-Sicherheit des Compilers.
2. **Laufzeitfehler:**
   - Fehler wie `ClassCastException` treten zur Laufzeit auf, nicht zur Compile-Zeit.
3. **Unvorhersehbares Verhalten:**
   - Andere Teile des Codes verlassen sich auf die generischen Typen.
   - Fehler können schwer zu finden und zu beheben sein.

Beispiel:

```
Locker rawLocker = locker1;
rawLocker.setItem(42); // Compiler erlaubt dies
String item = locker1.getItem(); // ClassCastException
```

Lösung: Raw-Typen vermeiden

1. **Verwende immer generische Typen:**

```java
Locker<String> locker1 = new Locker<>();
Locker<String> safeLocker = locker1; // Typ bleibt erhalten
```

2. **Compiler verhindert falsche Typen:**

```java
safeLocker.setItem(42); // Compiler-Fehler: Typ stimmt nicht
```

3. **Keine Laufzeitfehler:**

   - Generics garantieren, dass nur der richtige Typ verwendet wird.

Never use raw types !

- **Raw-Typen niemals verwenden:**
  - Deaktivieren Typprüfung.
  - Führen zu Laufzeitfehlern wie `ClassCastException`.
- **Generics immer nutzen:**
  - Compiler sorgt für Typ-Sicherheit.
  - Fehler werden bereits zur Compile-Zeit erkannt.
- **Sicherer Code:**
  - Generics machen den Code robuster und leichter wartbar.

Warum Type Erasure in Java?

1. **Abwärtskompatibilität:**
    - Code vor Java 5 sollte weiterhin funktionieren.
    - Generics sollten ältere Bibliotheken unterstützen.
2. **Performance:**
    - Keine zusätzliche Laufzeit-Typprüfung.
    - Weniger Overhead für die JVM.
3. **Einfaches JVM-Design:**
    - JVM behandelt generische Klassen genauso wie nicht-generische Klassen.

Moderne Sprachen mit Generics zur Laufzeit

1. **C#:**
   - **Reified Generics** behalten Typinformationen bei. (reificare - zu einem Ding machen)

```
List<int> numbers = new List<int>();
Console.WriteLine(numbers.GetType()); // List`1[System.Int32]
```

2. **Kotlin:**
   - Begrenzt durch die JVM, aber `inline`-Funktionen können Typinformationen bewahren.
3. **Swift:**
   - Generics bleiben zur Laufzeit erhalten.

```
var numbers: [Int] = [1, 2, 3]
print(type(of: numbers)) // Array<Int>
```

4. **Rust:**
   - Generics werden während der Kompilierung konkretisiert (Monomorphisierung, kein Type Erasure, hervorragende Performance kombiniert mit Typ-Sicherheit zur Compile-Zeit)

Vorteile von Reified Generics

1. **Typ-Sicherheit zur Laufzeit:**
   - Falsche Typen verursachen keine ClassCastException.
2. **Reflexion:**
   - Generische Typinformationen bleiben für Laufzeit-Operationen verfügbar.
3. **Flexibilität:**
   - Einfachere Implementierung von Methoden mit generischen Typen.

Type Erasure vs. Moderne Generics

- **Java:**
  - Type Erasure für Kompatibilität und Leistung.
  - Einschränkungen bei Typ-Sicherheit zur Laufzeit.
- **Moderne Sprachen (z. B. C#, Swift):**
  - Reified Generics für mehr Sicherheit und Flexibilität.
  - Komplexere Laufzeitumgebung erforderlich.
- **Abwägung:**
  - Java bleibt bei Type Erasure, um Stabilität und Einfachheit zu gewährleisten.
- Java führt Generics nur zur Kompilierzeit ein.
- Während der Laufzeit werden die Typparameter „verwischt“ (erased).

Quiz / IntelliJ

Generische Interfaces

```
public interface List<T> {
    void add(T value);
    T get(int index);
    T remove(int index);
}
```

- Ähnlich wie bei Klassen
- Typparameter wird in der Interface-Deklaration festgelegt

Implementierung eines generischen Interfaces

```java
public class GeneralList<T> implements List<T> {
    ArrayList<T> values = new ArrayList<>();

    public void add(T value) {
        this.values.add(value);
    }

    public T get(int index) {
        return this.values.get(index);
    }

    public T remove(int index) {
        return this.values.remove(index);
    }
}
```

Zusatzbeispiel: GeneralList in Aktion

```java
public static void main(String[] args) {
    GeneralList<String> gl = new GeneralList<>();
    gl.add("Hello");
    gl.add("World");

    System.out.println(gl.get(0)); // Output: Hello
    System.out.println(gl.get(1)); // Output: World

    gl.remove(0);
    System.out.println(gl.get(0)); // Output: World
}
```

Grenzen von Generics

- **Keine primitiven Typen:**
  - Generics unterstützen nur Objekttypen, keine primitiven Typen wie `int`, `double`, etc.
  - Lösung: Verwende Wrapper-Klassen (z. B. `Integer`, `Double`).

Mehr zu Generics

- **Wildcards (`? extends T`, `? super T`):**
  - `? extends T`:
    - Erlaubt das Lesen von Objekten, die **Subtypen von `T`** sind (also T oder spezieller).
    - Schreiboperationen sind eingeschränkt (außer `null`).

```java
public static void printList(List<? extends Number> list) {
    for (Number num : list) {
        System.out.println(num);
    }
    // list.add(10); // Compiler-Fehler
}
```

  - `? super T`:
    - Die Liste akzeptiert Typen, die **Supertypen** von T sind (also T oder Typen, die allgemeiner als T sind).
    - Erlaubt das Hinzufügen von Objekten, die **Subtypen von `T`** sind (also T oder spezieller).
    - =Lesen ist eingeschränkt, weil der genaue Typ der Elemente nicht bekannt ist; der Compiler gibt sie als Object zurück.

```java
public static void addNumbers(List<? super Integer> list) {
    list.add(10); // Erlaubt
    // Integer num = list.get(0); // Compiler-Fehler
}
```

Mehr zu Generics

- **Komplexe Anwendungsfälle:**
  - PECS-Prinzip:
    - **Producer Extends, Consumer Super**
    - **Producer Extends:** Verwende `? extends T` für **Lesen**.

```java
List<? extends Number> numbers = List.of(1, 2, 3.5);
for (Number n : numbers) {
    System.out.println(n);
}
```

    - **Consumer Super:** Verwende `? super T` für **Schreiben**.

```java
List<? super Integer> integers = new ArrayList<>();
integers.add(42);
```

Erklärung: `? extends T` (Obergrenze) und `? super T`

- **`? extends T`: Obergrenze - Bedeutung:**
  - Erlaubt Subtypen von `T`.
  - Sicher für **Lesen**, eingeschränkt für **Schreiben**.
- **Warum eingeschränkt für Schreiben?**
  - Der Compiler kann nicht garantieren, welcher Subtyp in der Liste erlaubt ist.

```
List<? extends Number> numbers = new ArrayList<>(List.of(1, 2.5));  // Typinferenz und Least Upper Bound (LUB) -> Numbers
numbers.add(3); // Compiler-Fehler: Ist "3" ein Integer oder Double?
```

  - Nur `null` ist erlaubt, da es keinen konkreten Typ erfordert.
- **Beispiel:**

```
public static void printList(List<? extends Number> list) {
    for (Number num : list) {
        System.out.println(num);
    }
    // list.add(10); // Compiler-Fehler
}
```

- **Typischer Anwendungsfall:** Wenn du nur Daten **lesen** möchtest.

Erklärung: `? extends T` und `? super T` (Untergrenze)

- **`? super T`: Untergrenze - Bedeutung:**
  - Erlaubt Supertypen von `T`.
  - Sicher für **Schreiben**, eingeschränkt für **Lesen**.
- **Warum eingeschränkt für Lesen?**
  - Der Compiler weiß nur, dass die Liste einen Supertyp von `T` enthält, aber nicht den genauen Typ.

```
List<? super Integer> list = new ArrayList<Object>();
list.add(42); // Sicher
// Integer num = list.get(0); // Compiler-Fehler: Typ ist unbekannt, könnte Object sein
Object num = list.get(0); // sicher
```

  - Gelesene Elemente werden als `Object` behandelt.
- **Beispiel:**

```
public static void addNumbers(List<? super Integer> list) {
    list.add(42); // Sicher
    // Integer num = list.get(0); // Compiler-Fehler
}
```

- **Typischer Anwendungsfall:** Wenn du nur Daten **schreiben** möchtest.

Vergleich: `? extends T` vs. `? super T`

Eigenschaft	`? extends T` (Obergrenze)	`? super T` (Untergrenze)
Lesen	Sicher als `T`	Nur als `Object`
Schreiben	Nicht erlaubt	Sicher als `T`
Beispiel-Szenario	Lesen von Daten	Hinzufügen von Daten
Eingeschränkt, weil...	Compiler kennt genauen Typ der Subtypen nicht	Compiler kennt nur den Supertyp

PECS-Prinzip

- **Producer Extends:**
  - Verwende `? extends T`, wenn du Daten **lesen** möchtest.
  - Beispiel: `List<? extends Number>` zum Iterieren über Zahlen.
- **Consumer Super:**
  - Verwende `? super T`, wenn du Daten **schreiben** möchtest.
  - Beispiel: `List<? super Integer>` zum Hinzufügen von Integer-Werten.
- Das PECS-Prinzip hilft dir, die richtige Wildcard basierend auf der Verwendung auszuwählen:
  - **Producer:** `extends`
  - **Consumer:** `super`

Zusammenfassung

- Generics sorgen für typsichere, wiederverwendbare Datenstrukturen.
- Das Prinzip: `<T>` bei Klassen und Interfaces, um Datentypen zur Laufzeit zuzuweisen.
- Java nutzt Type Erasure – zur Laufzeit verschwinden Typinformationen.
- Mit Generics vermeidet man Casting und verbessert die Codequalität.

Was ist Casting?

- **Definition:**
  - Casting ist das **Explizite Umwandeln** eines Objekts oder Wertes von einem Typ in einen anderen.
- **Zwei Arten von Casting:**
  1. **Primitive Typen:** Umwandlung zwischen primitiven Datentypen.
     ```
     double d = 3.14;
     int i = (int) d; // Ergebnis: 3
     ```
  2. **Referenztypen:** Umwandlung zwischen kompatiblen Objekttypen in einer Vererbungsbeziehung.
     ```
     Animal animal = new Dog();
     Dog dog = (Dog) animal; // Downcasting
     ```
- **Erforderlich bei:**
  - **Arbeiten mit allgemeinen Typen** wie `Object` oder rohen Typen (`Raw Types`).
  - **Vererbungsstrukturen**, wenn polymorphe Objekte verwendet werden.

Warum sollte Casting gut überlegt sein?

- **Verlust der Typ-Sicherheit:**
  - Casting kann zu **ClassCastException** führen, wenn der Typ zur Laufzeit nicht passt.

```
Object obj = "Text";
Integer num = (Integer) obj; // ClassCastException
```

- **Unübersichtlicher Code:**
  - Häufiges Casting macht den Code schwerer lesbar und anfälliger für Fehler.
- **Schlechtes Design:**
  - Casting kann ein Hinweis darauf sein, dass Typen oder Generics nicht optimal genutzt werden.
- **Beispiel für ein Problem:**

```
List list = new ArrayList();
list.add("Text");
Integer num = (Integer) list.get(0); // Fehler bei Laufzeit
```

Wie kann man Casting vermeiden?

1. **Verwende Generics:**
   - Generics eliminieren viele Notwendigkeiten für Casting.

```
// Ohne Generics (schlecht):
List list = new ArrayList();
list.add("Text");
String str = (String) list.get(0); // Casting erforderlich

// Mit Generics (besser):
List<String> list = new ArrayList<>();
list.add("Text");
String str = list.get(0); // Kein Casting erforderlich
```

2. **Nutze Polymorphismus:**
   - Lass Objekte ihre spezifischen Methoden selbst aufrufen, ohne Downcasting.

```
Animal animal = new Dog();
animal.makeSound(); // Kein Casting erforderlich
```

Wie kann man Casting vermeiden?

3. **Type-Safe APIs erstellen:**
   - Entwerfe Klassen und Methoden so, dass sie spezifische Typen verwenden.

```
public class Box<T> {
    private T item;

    public void setItem(T item) { this.item = item; }
    public T getItem() { return item; }
}

Box<String> box = new Box<>();
box.setItem("Text");
String str = box.getItem(); // Kein Casting erforderlich
```

- flexibel und typsicher: der generische Typparameter ermöglicht es, die Klasse flexibel und sicher für verschiedene Typen zu verwenden.

- **Casting ist nützlich, aber riskant:**
    - Es ist notwendig in bestimmten Kontexten (z. B. Reflexion, Legacy-Code).
    - Kann aber Laufzeitfehler verursachen und Typ-Sicherheit verlieren.
- **Vermeide Casting, wenn möglich:**
    - Nutze Generics, Polymorphismus und Type-Safe APIs.
- **Behalte Lesbarkeit und Wartbarkeit im Fokus:**
    - Code ohne Casting ist in der Regel sicherer, klarer und weniger fehleranfällig.

Generics in anderen Sprachen

Was sind Generics?

- Generics ermöglichen die Erstellung von **typsicheren und wiederverwendbaren Codebausteinen**.
- Unterschiedliche Sprachen implementieren Generics auf verschiedene Weise.

Sprachen mit Generics (Beispiele):

1. **C++** (Templates)
2. **Python** (Typannotationen und Generics seit PEP 484/585)
3. **Java** (Type Erasure)
4. **Kotlin** (Reified Generics in Inline-Funktionen)

Generics in C++ (Templates)

Eigenschaften:

- **Compile-Zeit-Generics:** Generics werden während der Kompilierung konkretisiert (Monomorphisierung).
- **Flexibilität:** Unterstützt sowohl primitive als auch benutzerdefinierte Typen.

```cpp
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(5, 10) << std::endl;   // Integer
    std::cout << add(3.5, 2.5) << std::endl; // Double
}
```

Vorteile:

- Keine Typbeschränkung (funktioniert mit allen Typen).
- Effizient: Generierter Code ist auf den Typ spezialisiert.

Nachteile:

- Kein Laufzeit-Typ: Code-Explosion bei vielen Typen.

Generics in Python

Eigenschaften:

- **Dynamisch:** Typprüfung erfolgt zur Laufzeit.
- **Typannotationen:** Seit PEP 484/585 können Generics für Typhinweise verwendet werden.

```
from typing import List, TypeVar

T = TypeVar('T')

def add_elements(elements: List[T]) -> T:
    return sum(elements)

print(add_elements([1, 2, 3]))       # Integer-Liste
print(add_elements([1.1, 2.2, 3.3])) # Float-Liste
```

Vorteile:

- Flexibilität: Keine strikte Typbindung.
- Leicht lesbarer Code.

Nachteile:

- Keine echte Typprüfung: Typen sind nur Hinweise. (mypy is your friend!)
- Potenziell langsamer als kompilierte Sprachen.

Ein Python Beispiel

```python
from typing import Dict

def get_employee_name(employee_id: int, employees: Dict[int, str]) -> str:
    return employees.get(employee_id, "Unknown")

# Example dictionary of employee names
employee_data = {
    101: "Alice",
    102: "Bob",
    103: "Charlie",
}

# Correct usage
print(get_employee_name(101, employee_data))  # Output: "Alice"

# Problematic usage (no runtime error, but incorrect logic)
print(get_employee_name("102", employee_data))  # Output: "Unknown"
```

Generics in Kotlin

Eigenschaften:

- **Reified Generics:** In Inline-Funktionen können Typinformationen zur Laufzeit erhalten bleiben.
- Unterstützt auch Java-ähnliche Generics (Type Erasure).

```
inline fun <reified T> printType(value: T) {
    println("Der Typ ist: ${T::class}")
}

fun main() {
    printType("Hello")
    printType(42)
}
```

Vorteile:

- Reified Generics für Laufzeit-Typinformationen.
- Moderne Syntax, leicht verständlich.

Nachteile:

- Einschränkungen bei Type Erasure außerhalb von Inline-Funktionen.

Vergleich: Generics in verschiedenen Sprachen

Sprache	Typprüfung	Laufzeit-Typen	Primitive Typen	Besonderheit
C++	Compile-Zeit	Nein	Ja	Templates, sehr flexibel
Python	Laufzeit	Ja	Ja	Typen als Hinweise, dynamisch
Java	Compile-Zeit	Nein (Type Erasure)	Nein	Abwärtskompatibel
Kotlin	Compile-Zeit + Reified	Teilweise	Nein	Reified Generics in Inline-Funktionen

Vergleich von Generics: C++ vs. Java

Generics in C++ (Monomorphisierung)

- **C++-Templates:** Der Compiler erzeugt für jeden verwendeten Typ eine separate Version der generischen Funktion oder Klasse.
- **Effizient:** Jede Typ-Version ist optimiert und direkt im Maschinencode vorhanden.

Beispiel in C++

```cpp
#include <iostream>
#include <string>

template <typename T>
void printValue(T value) {
    std::cout << value << std::endl;
}

int main() {
    printValue(1);          // Generiert Code für int
    printValue(3.14);       // Generiert Code für double
    printValue("Hello");    // Generiert Code für const char*
    printValue(std::string("World")); // Generiert Code für std::string
}
```

- Der Compiler erstellt:
  - `void printValue(int value)`
  - `void printValue(double value)`
  - `void printValue(const char* value)`
  - `void printValue(std::string value)`

Generics in Java (Type Erasure)

Generics in Java

- **Type Erasure:** Der generische Typ existiert nur zur Compile-Zeit und wird zur Laufzeit auf `Object` reduziert (oder auf den Bound des generischen Typs).
- **Effizient bei Speicherplatz:** Ein einziger Code für alle Typen.

Beispiel in Java

```java
public class Main {
    public static <T> void printValue(T value) {
        System.out.println(value);
    }

    public static void main(String[] args) {
        printValue(1);          // Integer
        printValue(3.14);       // Double
        printValue("Hello");    // String
        printValue(new String("World")); // String
    }
}
```

- Während der Kompilierung prüft der Compiler die Typen.
- Zur Laufzeit existiert nur:

```java
public static void printValue(Object value) {
    System.out.println(value);
}
```

Vergleich: Monomorphisierung vs. Type Erasure

Eigenschaft	C++ (Monomorphisierung)	Java (Type Erasure)
Effizienz	Typenspezifischer, optimierter Code	Einheitlicher generischer Code
Code-Duplizierung	Ja, separate Funktionen für jeden Typ	Nein, einheitlicher Code
Primitive Typen	Unterstützt	Nur über Wrapper-Klassen
Typprüfung	Compile-Zeit	Compile-Zeit
Reflexion von Typen	Möglich	Generische Typen zur Laufzeit nicht verfügbar

Beispiel für Reflexion (reflection)

Reflexion ist die Fähigkeit eines Programms, zur Laufzeit Informationen über seine Struktur (Klassen, Methoden, Felder) zu analysieren und dynamisch darauf zuzugreifen.

```java
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        // Klasse laden
        Class<?> cls = Class.forName("java.util.ArrayList");

        // Alle Methoden anzeigen
        Method[] methods = cls.getMethods();
        for (Method method : methods) {
            System.out.println(method.getName());
        }

        // Dynamische Instanzierung und Methodenaufruf
        Object instance = cls.getDeclaredConstructor().newInstance();
        Method addMethod = cls.getMethod("add", Object.class);
        addMethod.invoke(instance, "Hello, Reflection!");

        System.out.println(instance); // Ausgabe: [Hello, Reflection!]
    }
}
```

ArrayList und Hashtabelle

Lernziele

- Sie wissen, wie Sie eine generische Liste mit variabler Größe implementieren.
- Sie kennen eine mögliche Methode, wie die Java ArrayList implementiert sein könnte.
- Sie wissen, wie man eine Hashtabelle implementiert.
- Sie kennen eine mögliche Methode, wie Java HashMap intern funktionieren könnte.

Rückblick: Arrays

```
int[] numbers = new int[3];
numbers[0] = 2;
numbers[2] = 5;

System.out.println(numbers[0]);
System.out.println(numbers[2]);
```

Output:

```
2
5
```

- **Feste Größe**: Einmal angelegt, kann die Länge nicht mehr geändert werden.
- Zugriff via **Index**: `numbers[i]`.

Zusatzfolie: Prinzipien der Programmierung

- **Modularität**: Wir kapseln Logik (z.B. Array-Verwaltung) in Klassen.
- **Abstraktion**: Nutzer*innen der Klasse müssen interne Details nicht kennen.
- **Wiederverwendung**: Unsere ArrayList-Implementierung kann überall genutzt werden.

Warum ArrayList?

- **Variable Größe**: Wird bei Bedarf vergrößert.
- **Erleichterung**: Keine Sorgen, ob das Array „voll“ ist.
- **Typ-Parameter**: Beliebige Datentypen verwendbar.
- **Methoden**: z. B. `add`, `get`, `remove`, `contains`.

Eigene Liste: Start

```java
public class List<Type> {
    private Type[] values;
    private int firstFreeIndex;

    public List() {
        this.values = (Type[]) new Object[10];
        this.firstFreeIndex = 0;
    }
}
```

- Internes **Array** fester Größe (10).
- `firstFreeIndex` als Zähler für die erste freie Position.

Problem mit `Object`

```
public class ObjectList {
    private Object[] values;
    private int size;

    public ObjectList() {
        this.values = new Object[10];
        this.size = 0;
    }

    public void add(Object value) {
        this.values[size] = value;
        size++;
    }

    public Object get(int index) {
        return values[index];
    }
}

public class Main {
    public static void main(String[] args) {
        ObjectList list = new ObjectList();
        list.add("Text");
        list.add(42);

        // Fehler: Falsches Casting zur Laufzeit
        String value = (String) list.get(1); // ClassCastException
    }
}
```

Problem:

- **Keine Typprüfung:** Unterschiedliche Typen können eingefügt werden.
- **Laufzeitfehler:** ClassCastException bei falschem Cast.

Lösung mit Generics

```java
public class GenericList<Type> {
    private Type[] values;
    private int size;

    @SuppressWarnings("unchecked")
    public GenericList() {
        this.values = (Type[]) new Object[10];
        this.size = 0;
    }

    public void add(Type value) {
        this.values[size] = value;
        size++;
    }

    public Type get(int index) {
        return values[index];
    }
}

public class Main {
    public static void main(String[] args) {
        GenericList<String> list = new GenericList<>();
        list.add("Text");
        // list.add(42); // Compiler-Fehler

        String value = list.get(0); // Kein Casting erforderlich
        System.out.println(value); // Ausgabe: Text
    }
}
```

Vorteile:

1. **Typ-Sicherheit:** Compiler verhindert falsche Typen.
2. **Kein Casting:** Abruf von Werten ohne explizites Casting.
3. **Lesbarkeit:** Klarerer und wartbarer Code.

Vergleich: `Object` vs. Generics

Eigenschaft	`Object`	Generics
Typprüfung	Keine (Laufzeitfehler möglich)	Compile-Zeit (sicher)
Laufzeit-Casting	Erforderlich	Nicht erforderlich
Lesbarkeit	Weniger klar	Klare Typzuordnung
Fehlertoleranz	Fehler erst zur Laufzeit	Fehler bei der Kompilierung

- Generics sind sicherer und helfen, typische Fehler zu vermeiden.
- Verwende Generics, wenn der Typ vorhersehbar ist.

Methode: `add`

```java
public void add(Type value) {
    if (this.firstFreeIndex == this.values.length) {
        grow(); // Arraygröße bei Bedarf vergrößern
    }
    this.values[this.firstFreeIndex] = value;
    this.firstFreeIndex++;
}
```

- **Check**: Ist das Array voll? Dann `grow()`.
- **Einfügen**: Wert an `firstFreeIndex`.

Methode: `grow`

```
private void grow() {
    int newSize = this.values.length + this.values.length / 2;
    Type[] newValues = (Type[]) new Object[newSize];
    for (int i = 0; i < this.values.length; i++) {
        newValues[i] = this.values[i];
    }
    this.values = newValues;
}
```

- Neue Länge = ca. 1,5-fache Größe.
- Kopieren aller Werte ins neue Array.

Effizienz beim Wachsen

- **Kopieren**: Bei sehr großen Arrays kann das Kopieren zeitaufwändig sein.
- **Amortized Cost**: Trotzdem bleibt das Einfügen in ArrayList durchschnittlich effizient.
- **Alternativen**: LinkedList, andere Strategien zum Wachsen.

Methoden: `contains` und `remove`

```
public boolean contains(Type value) {
    for (int i = 0; i < this.firstFreeIndex; i++) {
        if (this.values[i].equals(value)) {
            return true;
        }
    }
    return false;
}
```

```
public void remove(Type value) {
    int index = indexOfValue(value);
    if (index < 0) return;
    moveToTheLeft(index);
    this.firstFreeIndex--;
}
```

- **contains**: Durchsucht nur gefüllte Einträge.
- **remove**: Findet Index, verschiebt Elemente links.

HashMap: Idee

- **Schlüssel-Wert-Paare** (Key-Value).
- **Hash-Funktion**: Berechnet Index im internen Array.
- **Kollisionen**: Mehrere Schlüssel können zum selben Index gehören -> Liste am Index.

```java
// usage (roughly)
HashMap<String, String> map = new HashMap<>();
map.add("Hello", "World");
System.out.println(map.get("Hello"));
```

Schlüssel-Wert-Paar (Pair)

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    // getters und setters
}
```

- Generische Typen `K`, `V`.
- `key` und `value` beliebigen Datentyps.
- Grundlage für Einträge in der HashMap.

Aufbau einer HashMap

```
public class HashMap<K, V> {
    private List<Pair<K, V>>[] values;
    private int numNonEmptyLists;

    public HashMap() {
        this.values = new List[32];
        this.numNonEmptyLists = 0;
    }
}
```

- Intern: Array von Listen (`List<Pair<K, V>>[]`).
- Jeder Index speichert eine Liste von Schlüssel-Wert-Paaren.

Suchen eines Werts: `get(K key)`

```
public V get(K key) {
    int hashValue = Math.abs(key.hashCode() % this.values.length);
    if (this.values[hashValue] == null) {
        return null;
    }
    List<Pair<K, V>> bucket = this.values[hashValue];
    for (int i = 0; i < bucket.size(); i++) {
        if (bucket.value(i).getKey().equals(key)) {
            return bucket.value(i).getValue();
        }
    }
    return null;
}
```

- **Hash berechnen**.
- **Liste am Index durchsuchen**.
- Key matcht -> Value zurückgeben, sonst `null`.

Werte hinzufügen: `add(K key, V value)`

```
public void add(K key, V value) {
    List<Pair<K, V>> bucket = getListBasedOnKey(key);
    int index = getIndexOfKey(bucket, key);

    if (index < 0) {
        bucket.add(new Pair<>(key, value));
        this.numNonEmptyLists++;
    } else {
        bucket.value(index).setValue(value);
    }

    if (1.0 * this.numNonEmptyLists / this.values.length > 0.75) {
        grow();
    }
}
```

- **Berechnen**: Hash-Wert -> entsprechender Bucket (Liste).
- **Existiert Key schon?** -> Value überschreiben, sonst neuen Pair anfügen.
- **Wachstum**: Wenn 75% der Plätze gefüllt sind, `grow()`.

Hilfsmethode: getListBasedOnKey

```java
private List<Pair<K, V>> getListBasedOnKey(K key) {
    int hashValue = Math.abs(key.hashCode() % this.values.length);
    if (this.values[hashValue] == null) {
        this.values[hashValue] = new List<>();
    }
    return this.values[hashValue];
}
```

- **Zweck**
  - Ermittelt (per `hashCode() % length`) den Index im Bucket-Array.
  - Legt eine neue Liste an, wenn am Index noch keine existiert.
  - Gibt die vorhandene bzw. neu angelegte Bucket-Liste zurück.

Hilfsmethode: getIndexOfKey

```java
private int getIndexOfKey(List<Pair<K, V>> myList, K key) {
    for (int i = 0; i < myList.size(); i++) {
        if (myList.value(i).getKey().equals(key)) {
            return i;
        }
    }
    return -1;
}
```

- **Zweck**
  - Durchsucht die (Key, Value)-Paare in der übergebenen Bucket-Liste.
  - Vergleicht die Schlüssel per `equals`.
  - Gibt den Index des passenden Schlüssels zurück oder `-1`, wenn er nicht gefunden wird.

Einfacher Exkurs: Kollisionsaufteilung

- Wenn mehrere Keys denselben Hash-Wert haben, landen sie in derselben Liste (Kollision).
- HashMap verteilt Keys dadurch auf viele Listen (sogenannte Buckets).
- Ideal: wenige Elemente pro Liste -> schnelle Suche.
- Analyse von Algorithmen

Fortgeschritten: Performance und Wachstum

- **Rehashing**: Beim `grow()` müssen alle Pair-Elemente neu in das vergrößerte Array eingefügt werden.
- **Amortisiert O(1)**: Im Durchschnitt bleibt das Einfügen in HashMaps effizient.
- **Worst Case**: Schlechte Hash-Funktion -> Alle Daten in einer Liste -> O(n) Suche.

```
private void grow() {
    List<Pair<K, V>>[] newArray = new List[this.values.length * 2];
    // Kopiere die alten Werte -> neu verteilen
    this.values = newArray;
}
```

`grow` Methode

- Copy-Funktion:

```java
private void copy(List<Pair<K, V>>[] newOne, int fromIdx) {
    for (int i = 0; i < this.values[fromIdx].size(); i++) {
        Pair<K, V> value = this.values[fromIdx].value(i);

        int hashValue = Math.abs(value.getKey().hashCode() % newOne.length);
        if(newOne[hashValue] == null) {
            newOne[hashValue] = new List<>();
        }

        newOne[hashValue].add(value);
    }
}
```

- Diese Funktion kopiert alle (Key, Value)-Paare aus einer bestimmten Bucket-Liste des alten Bucket-Arrays (`values[fromIdx]`) in die entsprechende Bucket-Liste des neuen Bucket-Arrays.

`grow` Methode

- Wir wollen beim Überschreiten eines bestimmten Füllgrads das interne Array vergrößern.
- Dabei wird jedes Bucket (jede Liste) in das neue, größere Array übertragen.
- Wir verwenden die `copy`-Methode, um das Kopieren zu erledigen.

Schritt-für-Schritt

```
private void grow() {
    // 1. Neues Array anlegen (doppelt so groß)
    List<Pair<K, V>>[] newArray = new List[this.values.length * 2];

    // 2. Alte Buckets kopieren
    for (int i = 0; i < this.values.length; i++) {
        if (this.values[i] != null) {
            copy(newArray, i);
        }
    }

    // 3. Verweis auf neues Array setzen
    this.values = newArray;
}
```

Erklärung: Neuer Index

- `int hashValue = Math.abs(value.getKey().hashCode() % newArray.length);`
- Da `newArray` größer ist, verteilen sich die Einträge typischerweise anders.
- „Rehashing“ führt zu einer meist besseren Verteilung, weniger Kollisionen und damit schnellerem Zugriff.

Vergrößern einer HashMap – Beispiel Rehashing

Ausgangssituation

- Array-Länge = 5
- Schlüssel A: `hashCode() = 5`
- Schlüssel B: `hashCode() = 10`
- Modulo 5:
  - A: `5 % 5 = 0`
  - B: `10 % 5 = 0`
- **Beide** Schlüssel landen im **Bucket 0**

Nach Vergrößerung

- Array-Länge = 10
- Neuer Modulo:
  - A: `5 % 10 = 5` (Bucket 5)
  - B: `10 % 10 = 0` (Bucket 0)
- Die zwei Schlüssel sind nun in **unterschiedlichen** Buckets.

Warum vergrößern?

- Bessere **Performance**: Je größer das Array relativ zur Anzahl der Einträge, desto geringer das Risiko langer Listen in einem Bucket.
- **Load Factor**: Ein typischer Wert ist 0.75 – wenn 75% der Bucketplätze belegt sind, wird vergrößert.

`grow`-Methode (Zusammenfassung)

- Die `grow`-Methode erzeugt ein größeres Array und kopiert alle Einträge mithilfe der `copy`-Methode.
- Durch das Rehashing verteilen sich die Schlüssel neu.
- So bleibt die HashMap auch bei wachsenden Datenmengen effizient.

Entfernen in HashMap: `remove(K key)`

```
public V remove(K key) {
    List<Pair<K, V>> bucket = getListBasedOnKey(key);
    if (bucket.size() == 0) {
        return null;
    }

    int idx = getIndexOfKey(bucket, key);
    if (idx < 0) {
        return null;
    }

    Pair<K, V> pair = bucket.value(idx);
    bucket.remove(pair);
    return pair.getValue();
}
```

- **Key im Bucket suchen**, Pair entfernen.
- Gibt den Wert zurück, sonst `null`.
- Beachte: Unnötige leere Listen können erzeugt werden.

Vergleich: Liste vs. HashMap

```java
long start = System.nanoTime();
// enthält "Foo"?
myList.contains("Foo");
long end = System.nanoTime();
```

- Liste: In Worst-Case O(n) (lineares Durchsuchen).
- HashMap: In der Praxis (mit guter Hash-Funktion) O(1) amortisiert.
- Richtig implementiert sind HashMaps sehr schnell bei Suchanfragen.

Zusammenfassung

- Diese **selbstgeschriebene** HashMap dient nur zu **Demonstrationszwecken**.
- Sie zeigt **beispielhaft** die Funktionsweise einer HashMap:
  - Array von **Buckets** (Listen)
  - Hash-Wert-Berechnung für Schlüssel
  - **Kollisionen** durch mehrere Schlüssel im selben Bucket
  - Dynamisches **Vergrößern** (Rehashing)

Was zeichnet diese Beispiel-Implementierung aus?

1. **Listen statt ArrayList**
   - Statt der Standard-Java-Klassen nutzen wir eine eigene Listenklasse als Bucket.
2. **Hilfsmethoden**
   - getListBasedOnKey: erzeugt bei Bedarf einen Bucket.
   - getIndexOfKey: ermittelt den Index eines Schlüssels in der Bucket-Liste.
3. **add und remove**
   - `add` überschreibt existierende Schlüssel oder hängt neue Einträge an.
   - `remove` kann leere Buckets erzeugen, wenn der Schlüssel nicht existiert.
4. **Wachstumsmechanismus**
   - Bei Überschreiten eines definierten Füllgrads (z. B. 75 %) vergrößern wir das Array und verteilen alle Einträge neu (Rehashing).

Abgrenzung zur Java HashMap

- **Kein Ersatz** für `java.util.HashMap`
- **Vereinfachte** Struktur ohne viele Optimierungen
- Dient dem **Verständnis** wichtiger Prinzipien:
    - **Key-Value**-Speicherung
    - **Hashing & Kollisionen**
    - **Rehashing & Load-Factor**

Fazit

- **HashMap**-ähnlicher Code hilft, die **Idee** dahinter zu verstehen.
- Die gezeigte Version ist **nicht** identisch mit der offiziellen Java-Implementierung.
- Erkenntnis:
    - Schlüssel eindeutig,
    - Werte werden bei Kollision in Listen verwaltet,
    - Vergrößern des Arrays, wenn viele Einträge vorhanden sind.

Zufälligkeit (Randomness) in Java

Lernziele

- Zufallszahlen generieren und verstehen, wann sie benötigt werden
- Die Java-Klasse `Random` verwenden, um verschiedene Zufallszahlen zu erzeugen

Motivation: Warum Zufallszahlen?

- Computerspiele (z. B. Würfeln, Lotterien, Gegnerbewegungen)
- Verschlüsselung und Sicherheit (unvorhersehbare Schlüssel)
- Maschinelles Lernen (z. B. zufällige Initialisierung, Daten-Mischung)
- Simulationen (z. B. Monte-Carlo-Methoden)

Zufallszahlen: Die `Random`-Klasse

```java
import java.util.Random;

public class Raffle {
    public static void main(String[] args) {
        Random ladyLuck = new Random(); // create Random object

        for (int i = 0; i < 10; i++) {
            int randomNumber = ladyLuck.nextInt(10);
            System.out.println(randomNumber);
        }
    }
}
```

Zusatzfolie (einfach): Spannweite bei nextInt

Wie funktioniert `nextInt(10)`?

- Gibt eine Zahl im Bereich **[0..9]** zurück.
- Intern wendet Java eine Pseudozufallsfunktion an, die basierend auf einem internen Zustand neue Werte berechnet.

Zufallszahlen für Intervalle

```
Random random = new Random();
// Temperatur zwischen [-30..50]
int temperature = random.nextInt(81) - 30;
System.out.println(temperature);
```

- **0 bis 80**: `nextInt(81)`
- **-30 verschieben**: Von -30 bis +50

Zufallswerte als double

```
double p = random.nextDouble();
// liefert [0..1)

if (p <= 0.1) {
    System.out.println("It rains");
} else if (p <= 0.4) {
    System.out.println("It snows");
} else {
    System.out.println("The sun shines");
}
```

- `nextDouble()`: Liefert eine Zufallszahl zwischen [0..1).
- Häufig für **Wahrscheinlichkeiten** oder Simulationen genutzt.

Zusatzfolie (anspruchsvoll): Normalverteilung

```
double gaussVal = random.nextGaussian();
// Mittelwert = 0, Standardabweichung = 1
```

- `nextGaussian()` liefert Werte aus einer **Normalverteilung** (Gaußsche Glockenkurve).
- nützlich für realitätsnähere Simulationen (Größen, Gewichte, Messfehler etc.)

Beispiel: Würfel

```java
public class Die {
    private Random random;
    private int numberOfFaces;

    public Die(int numberOfFaces) {
        this.random = new Random();
        this.numberOfFaces = numberOfFaces;
    }

    public int throwDie() {
        // Werte: 1..numberOfFaces
        return 1 + this.random.nextInt(this.numberOfFaces);
    }
}
```

- Konstruktor: Anzahl der Würfelseiten festlegen.
- Methode `throwDie()`: zufällige Zahl zwischen 1 und numberOfFaces.

Pseudo vs. echte Zufälligkeit

- **Pseudozufall**: Rechner generiert eine reproduzierbare Sequenz basierend auf einem Startwert (Seed).

```
Random random = new Random(System.currentTimeMillis());
```

oder auch

```
Random random = new Random();
random.setSeed(System.currentTimeMillis());
```

- **Echte Zufälligkeit**: Wird z. B. aus physikalischen Quellen ermittelt (Strahlung, Rauschen, Lavalampe).
- Für viele Anwendungen (z. B. Spiele) reichen Pseudozufallszahlen vollkommen aus.

Komplexität von Pseudozufallszahlengeneratoren

- **Pseudozufallszahlengeneratoren (PRNGs):**
    - Algorithmen, die Sequenzen von Zahlen erzeugen, die „zufällig“ erscheinen.
    - **Deterministisch:** PRNGs verwenden einen Startwert (Seed), um eine reproduzierbare Sequenz zu generieren.
- **Komplexität:**
    - **Periodenlänge:** Wie viele Zahlen können erzeugt werden, bevor sich die Sequenz wiederholt?
    - **Verteilungsqualität:** Wie gleichmäßig sind die Zahlen über den Wertebereich verteilt?
    - **Effizienz:** Wie schnell können Zahlen erzeugt werden?

Beispiele für PRNGs:

- **Linear Congruential Generator (LCG):** Einfach, aber oft begrenzte Periodenlänge.
- **Mersenne Twister:** Längere Perioden und gute Verteilung, aber nicht kryptografisch sicher.

Beispiel: Linear Congruential Generator (LCG)

Java-Implementierung eines einfachen PRNGs

```java
public class SimplePRNG {
    private long seed;
    private static final long MULTIPLIER = 1664525;
    private static final long INCREMENT = 1013904223;
    private static final long MODULUS = (long) Math.pow(2, 32);

    public SimplePRNG(long seed) {
        this.seed = seed;
    }

    public int nextInt() {
        seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
        return (int) (seed & 0x7FFFFFFF); // Nur positive Zahlen
    }

    public static void main(String[] args) {
        SimplePRNG prng = new SimplePRNG(12345);
        for (int i = 0; i < 10; i++) {
            System.out.println(prng.nextInt());
        }
    }
}
```

Eigenschaften:

- **Periodenlänge:** Abhängig von den Parametern, maximal 2^{32}.
- **Einschränkungen:** Schlechte Parameterwahl kann zu Mustern und Wiederholungen führen.

Historische Probleme mit schlechten Generatoren

Fallstudie: PRNG in C (rand)

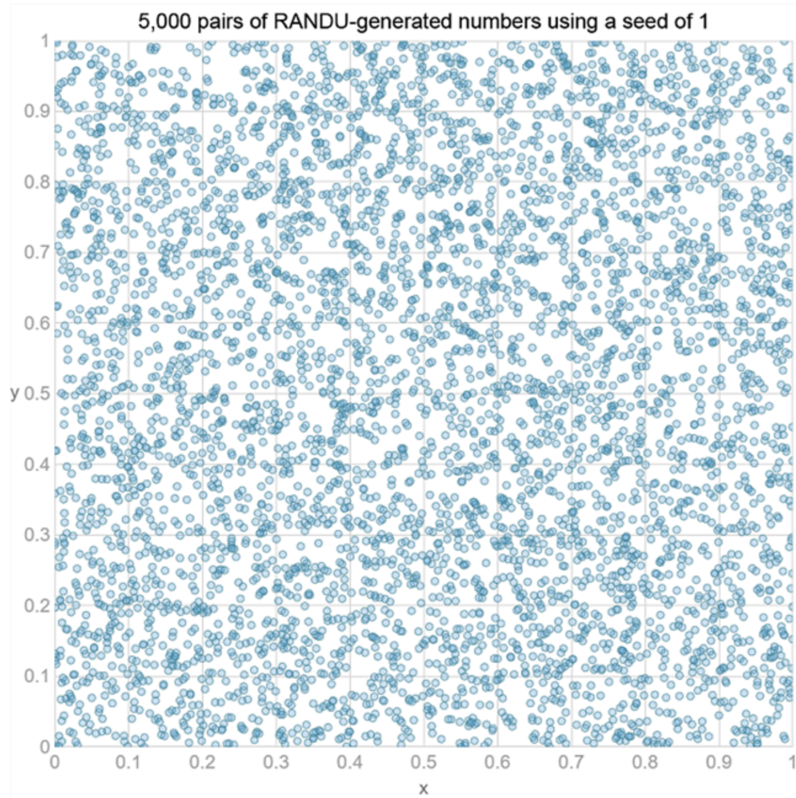
- Die `rand()` **-Funktion in der Standardbibliothek von C** verwendet oft einen schwachen PRNG.
  - Begrenzte Periodenlänge.
  - Schlechte Verteilungsqualität.
  - Beispiele für Fehler:
    - **1990er Jahre:** Vorhersagbare Zufallszahlen führten zu Sicherheitslücken in Anwendungen wie Netzwerksimulationen.

Fallstudie: Glücksspiel-Fehler

- **2003:** Online-Pokerplattform entdeckte, dass ihr PRNG vorhersehbar war.
  - Spieler konnten zukünftige Karten vorhersagen.
  - Ursache: Verwendung eines schwachen PRNGs mit kurzem Seed.

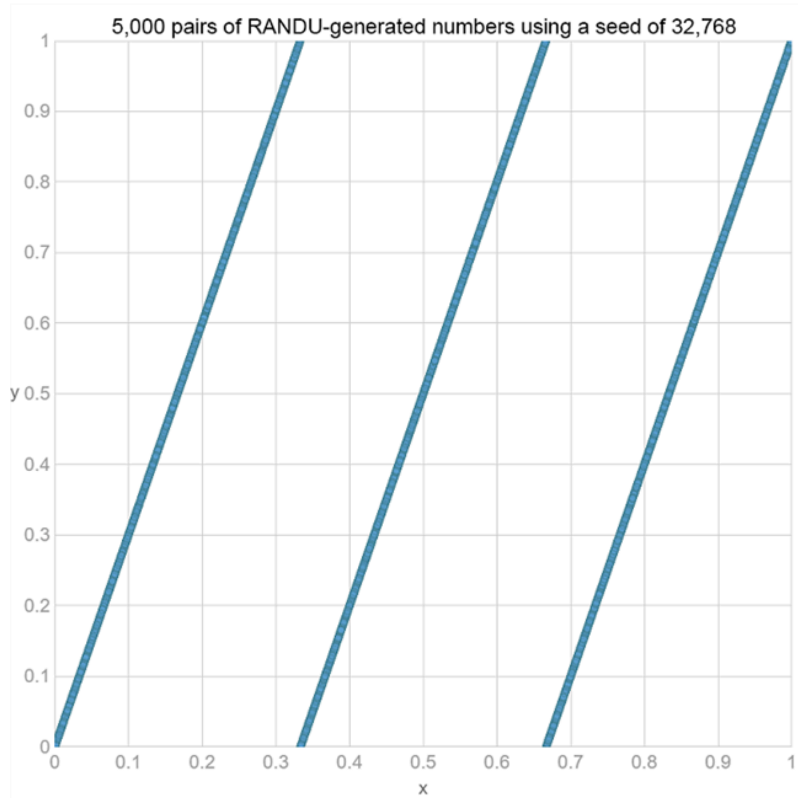
Konsequenzen:

- Finanzielle Verluste.
- Vertrauensverlust.



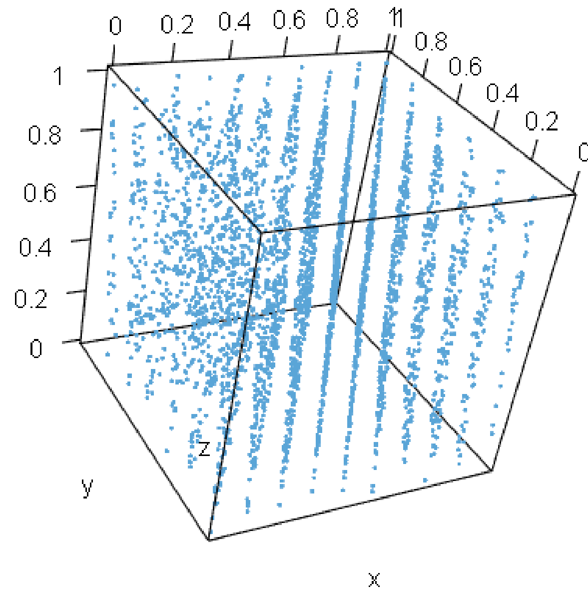
RANDU

linear congruential generator" (LCG) developed by IBM in the 1950's.



RANDU

linear congruential generator" (LCG) developed by IBM in the 1950's.



RANDU

linear congruential generator" (LCG) developed by IBM in the 1950's.

Bedeutung guter Zufallszahlengeneratoren

- **Anwendungen:**
  - **Simulationen:** Physik, Wettervorhersage, Monte-Carlo-Methoden.
  - **Spiele:** Sicherstellen, dass Ergebnisse wirklich zufällig erscheinen.
  - **Kryptografie:** Erfordern kryptografisch sichere Generatoren.
- **Forschungshintergrund:**
  - Zahlreiche Studien und Algorithmen, um bessere Zufallszahlen zu erzeugen.
  - Beispiele: **Mersenne Twister, xorshift, SecureRandom**.
- **Kryptografisch sichere PRNGs:**
  - Erfüllen zusätzliche Sicherheitsanforderungen.
  - Beispiele: `SecureRandom` **in Java**, **Fortuna**.

Zusammenfassung

- **Random**-Klasse: Erzeugt int- und double-Zufallswerte.
- **Intervalle verschieben**: nextInt + Addition/Subtraktion.
- **Prozent/Wahrscheinlichkeiten**: `nextDouble()`
- **Normalverteilung**: `nextGaussian()`
- **Pseudozufälligkeit**: oft ausreichend, aber nicht wirklich „echt“.

Multidimensionale Daten

Lernziele

- Verstehen, wie man **multidimensionale** Daten darstellt
- Erstellen und Verwenden **mehrdimensionaler Arrays**

Mehrdimensionale Arrays in Java

```
int rows = 2;
int columns = 3;
int[][] twoDimensionalArray = new int[rows][columns];
```

- **`int[][]`** für zweidimensionale Arrays
- `twoDimensionalArray.length` = Anzahl der Zeilen
- `twoDimensionalArray[row].length` = Spalten in dieser Zeile

Durchlaufen mit for-Schleifen

```
for (int row = 0; row < twoDimensionalArray.length; row++) {
    for (int column = 0; column < twoDimensionalArray[row].length; column++) {
        int value = twoDimensionalArray[row][column];
        System.out.println(row + ", " + column + ": " + value);
    }
}
```

- **Verschachtelte Schleifen**: Erst Zeile wählen, dann Spalte
- Grundprinzip bei jeder Dimensionserhöhung: weitere Schleife

Initialwerte und Zuweisung

```
twoDimensionalArray[0][1] = 4;
twoDimensionalArray[1][1] = 1;
twoDimensionalArray[1][0] = 8;
```

- Standardwert für `int`-Felder: **0**
- Änderungen einzelner Elemente wie gewohnt möglich

Anwendungsideen

- **Koordinaten**: (x, y) in einem Raster, z. B. 2D-Karten oder Bilder
- **Tabellen/Matrizen**: z. B. Pixelwerte in der Bildverarbeitung
- **Spiele**: Brettspiele wie Schach oder Sudoku
- **Mehrdimensionale Arrays**: lassen sich auch in 3D oder 4D anlegen, z. B. `int[][][]`.
- **Graphentheorie**: Adjazenzmatrix

Magische Quadrate

- **Magisch**: Summe jeder Zeile, Spalte, Diagonale ist gleich
- **Beispiel** (3×3):

```
8  1  6
3  5  7
4  9  2
```

- Aufgaben:
  - Summen der Zeilen, Spalten, Diagonalen berechnen
  - Algorithmus (Siamese-Methode) zum Erstellen eines magischen Quadrats (ungerade Größe)

Siamese-Methode

1. Platziere **1** in der **mittleren Spalte** der **obersten** Zeile.
2. Gehe **eine Zeile hoch** und **eine Spalte nach rechts**, platziere **2**.
3. Falls man raus „wandert“, auf die gegenüberliegende Seite springen.
4. Wenn der Platz schon belegt ist, gehe stattdessen **einen Schritt nach unten**.

Ergebnis: ein magisches Quadrat für ungerade Seitenlänge.

Tome II. *of the Kingdom of* SIAM. 229

towards the right: that is to ſay, that from the upper tranſverſe they deſcend immediately to that below.

3. When they have placed a number in the laſt caſe of a tranſverſe, the following is put in the firſt caſe of the tranſverſe immediately ſuperior, that is to ſay, that from the laſt upright, they return immediately to the firſt upright on the left.

4. In every other occurrence, after having placed a number, they place the following in the caſes which follow diametrically or ſlantingly from the bottom to the top, and from the left to the right, until they come to one of the caſes of the upper tranſverſe, or of the laſt upright to the right.

5. When they find the way ſtopp'd by any caſe already filled with any number, then they take the caſe immediately under that which they have filled, and they continue it as before, diametrically from the bottom to the top, and from the left to the right.

Theſe few Rules, eaſie to retain, are ſufficient to range all the unequal ſquares in general. An example renders them more intelligible.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

This ſquare is eſſentially different from that of *Agrippa*; and the method of *Bachet* is not eaſily accommodated thereto; and on the contrary, the *Indian* method may eaſily give the ſquares of *Agrippa*, by changing it in ſomething.

1. They place the unite in the Caſe, which is immediately under that of the Center, and they purſue it diametrically from top to bottom, and from the left to the right.

2. From the loweſt caſe of an upright, they paſs to the higheſt caſe of the upright which follows on the right; and from the laſt caſe of a Tranſverſe they return to the left to the firſt caſe of the Tranſverſe immediately inferior.

3. When the way is interrupted, they re-aſſume two caſes underneath that which they filled; and if there remains no caſe underneath, or that there remains but one, the firſt caſe of the upright is thought to return in order after the laſt, as if it was indeed underneath the loweſt.

Quelle:

- Bild aus Wikipedia
- Dokument: *Simon de la Loubère's 1693 "A new historical relation of the kingdom of Siam."*

Vergleich mit C und Fortran: row-major vs. column-major

Row-major (z. B. C, C++):

- Mehrdimensionale Arrays werden **zeilenweise** im Speicher abgelegt.
- Beispiel in C:

```c
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

  - Hier liegen die Daten sequentiell als [1,2,3,4,5,6] im Arbeitsspeicher (row-major).

Column-major (z. B. Fortran):

- Mehrdimensionale Arrays werden **spaltenweise** im Speicher abgelegt.
- Beispiel in (pseudo)Fortran-Syntax:

```fortran
integer :: matrix(2,3)
matrix = reshape( [1,2,3,4,5,6], shape(matrix) )
```

  - Hier liegen die Daten sequentiell als [1,4,2,5,3,6] (column-major).

Warum (ir)relevant für Java?

- Java speichert mehrdimensionale Arrays als **„Array von Arrays“**.
- **Kein zusammenhängender Block** im Speicher wie in C/Fortran.
- **Row-major vs. column-major** spielt in Java daher **kaum eine Rolle**, weil jedes `array[row]` eine **eigene Referenz** ist.
- Für Performance-Optimierungen in Java, wenn man z. B. großen Wert auf Cache-Lokalität legt, kann ein row-major-ähnlicher Aufbau aber **indirekt hilfreich** sein (indem man bspw. auf Anordnung der Daten achtet).

Fazit:

- In Sprachen mit direktem Speicherzugriff (C/Fortran) sehr wichtig, wie die Daten hintereinander im RAM liegen.
- In Java eher **abstrakt**: man arbeitet mit Referenzen und hat keinen direkten Einfluss auf das genaue Layout.
- Trotzdem kann man sich beim Durchlaufen (Reihen zuerst, dann Spalten) an row-major anlehnen, damit Zugriffe im Cache effizienter sind.

Zusammenfassung

- **Zweidimensionale Arrays** = Array von Arrays
- Zugriff mit `array[row][column]`
- **Verschachtelte Schleifen** zum Durchlaufen
- Einsatz in Tabellen, Matrizen, Rasterdaten
- „Magische Quadrate“ als Beispiel

Fortgeschrittene Java-Konzepte

- **Generics**: Ermöglichen typsichere, wiederverwendbare Datenstrukturen
- **Zufall**: Mit `Random` (Pseudozufall) und normalverteilten Zufallswerten
- **Mehrdimensionale Daten**: 2D-/3D-Arrays für komplexe Strukturen

Verbindung zu weiteren Themen (beispielhaft)

- **Softwaretechnik**:
  - Entwurf von **großen Projekten**
  - Nutzung komplexer Datenstrukturen
  - Wichtigkeit von Wartbarkeit, Erweiterbarkeit, Lesbarkeit
- **Analyse von Algorithmen**:
  - Komplexität von Listen, Maps, 2D-Operationen
- **Kryptographie**:
  - Verwendung von Zufallszahlen (z. B. sichere Schlüssel)
  - Mathematische Fundamente für sichere Übertragungen

Brücke zur Mathematik und Betriebssystemen

- **Mathe**:
  - Zahlentheorie (für Verschlüsselung)
  - Wahrscheinlichkeitsrechnung (für Monte-Carlo-Methoden, Randomness)
- **Betriebssysteme**:
  - Speicherverwaltung (Arrays, Referenzen)
  - Parallelisierung (z. B. Thread-sichere Nutzung von Collections)

Prinzipien der Programmierung

- **Abstraktion**: Generics und mehrdimensionale Strukturen verbergen interne Details
- **Modularisierung**: Klare Aufteilung in Klassen, Methoden, Pakete
- **Wartbarkeit**: Wiederverwendbarkeit durch saubere Entkopplung (z. B. generische Klassen)

Am Ende – und zugleich ein Anfang

- Diese Themen zeigen, wie **vielfältig** die Java-Welt ist.
- Wir stehen an einem Punkt, an dem Sie
  - fortgeschrittene Konzepte verstehen,
  - und zugleich den **Einstieg** in noch größere Felder finden.
- **Motivation**:
  - Neue Projekte? Eigene Libraries schreiben?
  - Vertiefung in Algorithmen & Datenstrukturen?
  - Sichere Programmierung (Krypto)?
  - Machine Learning, Big Data?

Ausblick

- **Software-Engineering**
  - Architektur großer Systeme, Design Patterns
- **Algorithmen & Datenstrukturen**
  - Performance-Optimierung, Spezialstrukturen
- **Verteilte Systeme & paralleles Programmieren**
  - Threading, Nebenläufigkeit
- **Mathematische Methoden**
  - Statistische Modelle, komplexe Zufallsprozesse
- **Fazit**:
  - Java ist ein mächtiges Tool,
  - Ihr Lernen geht hier aber erst richtig los!

Ende (Teil 12)