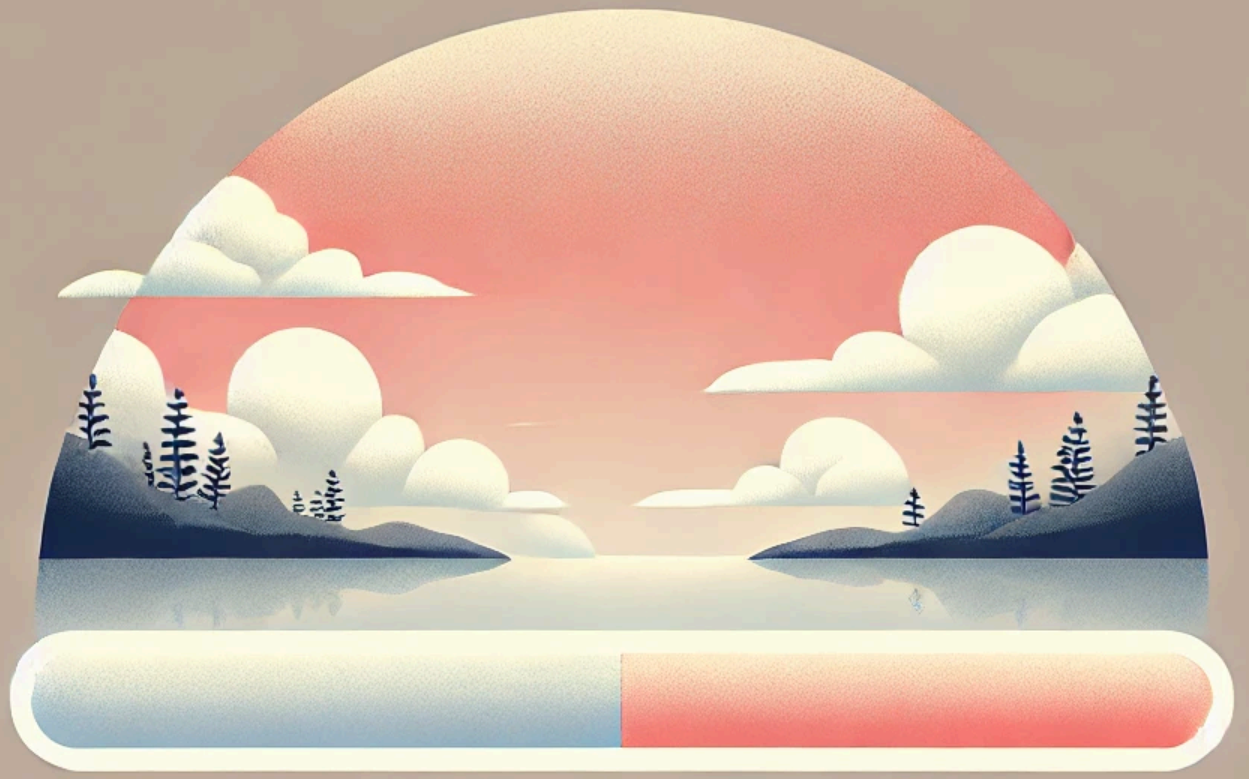
Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 08)

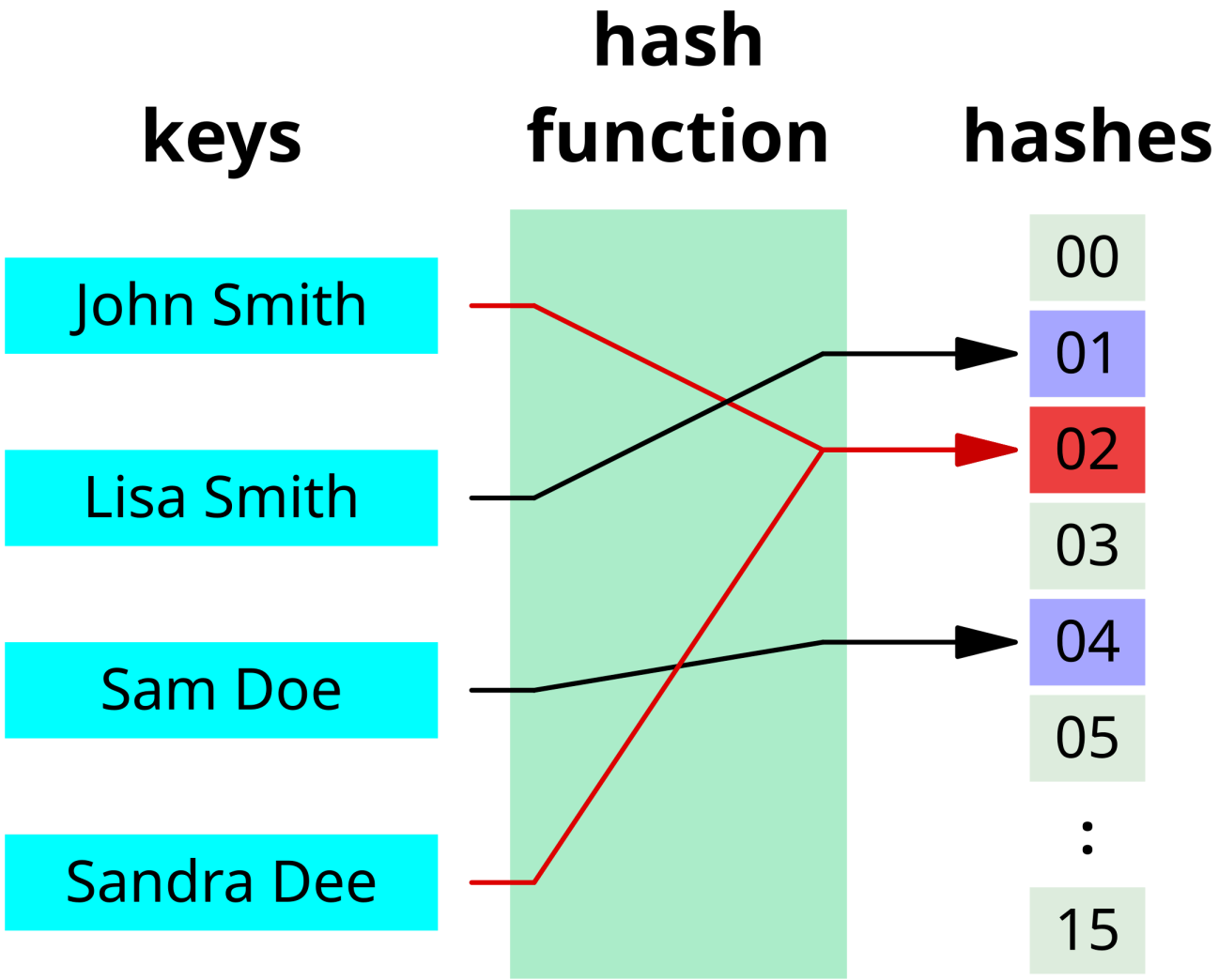


50%

HALFWAY THERE

Lernziele: Hash Map

- **Verstehen**, wie eine **Hash Map** funktioniert und wann sie verwendet wird.
- Eine **Hash Map in Java erstellen**, Daten hinzufügen und abrufen.
- Hash Maps als **Instanzvariable** in Klassen nutzen.
- **Schlüssel und Werte** einer Hash Map mit einer **for-each-Schleife** durchgehen.


keys
hash function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

(`wikipedia`)

Hashing: Show CLI `md5`

Einführung in die Hash Map

- Eine **HashMap** ist eine Datenstruktur, die Daten als **Schlüssel-Wert-Paare** speichert.
  - Jeder **Schlüssel** ist eindeutig und wird verwendet, um den zugehörigen **Wert** zu finden.
- **Schneller Zugriff**: Werte können sehr schnell über ihren Schlüssel abgerufen werden, da die HashMap interne **Hash-Funktionen** verwendet.
- In Java verwenden wir die Methoden `put()` zum **Hinzufügen** eines Schlüssel-Wert-Paares und `get()` zum **Abrufen** eines Werts anhand des Schlüssels.

Beispiel:

```
// Erstellen einer HashMap, die Postleitzahlen mit Städten verknüpft
HashMap<String, String> postalCodes = new HashMap<>();

// Hinzufügen von Schlüssel-Wert-Paaren
postalCodes.put("00710", "Helsinki");
postalCodes.put("10001", "New York");

// Abrufen eines Werts anhand des Schlüssels
System.out.println(postalCodes.get("00710"));  // Ausgabe: Helsinki
System.out.println(postalCodes.get("10001"));  // Ausgabe: New York
```

Hash Map: Typen für Schlüssel und Werte

- In Java ist eine HashMap eine generische Klasse, die zwei Typ-Parameter benötigt:
  - **Schlüsseltyp**: Der Datentyp der Schlüssel, z.B. `String`, `Integer`.
  - **Wertetyp**: Der Datentyp der Werte, z.B. `String`, `Integer`, benutzerdefinierte Objekte.
- Dies ermöglicht Flexibilität, um verschiedene Arten von Daten zu speichern und zu verknüpfen.

Beispiel:

- Erstellen wir eine HashMap, die `String`-Schlüssel (Städtenamen) mit `Integer`-Werten (Bevölkerungszahlen) verknüpft.

```
// Erstellen einer HashMap mit String-Schlüsseln und Integer-Werten
HashMap<String, Integer> population = new HashMap<>();

// Hinzufügen von Daten
population.put("Helsinki", 631695);
population.put("Berlin", 3644826);

// Abrufen von Daten
System.out.println("Einwohnerzahl von Helsinki: " + population.get("Helsinki"));
System.out.println("Einwohnerzahl von Berlin: " + population.get("Berlin"));
```

Einfache Anwendung: Spitznamen

- **Anwendungsbeispiel**: Wir möchten die Spitznamen von Personen speichern.
- Wir können eine HashMap verwenden, um die vollständigen Namen (als Schlüssel) mit ihren Spitznamen (als Werte) zu verknüpfen.

Beispielcode:

```
// Erstellen einer HashMap für Spitznamen
HashMap<String, String> nicknames = new HashMap<>();

// Hinzufügen von vollständigen Namen und ihren Spitznamen
nicknames.put("matthew", "matt");
nicknames.put("michael", "mix");
nicknames.put("arthur", "artie");

// Abrufen eines Spitznamens anhand des vollständigen Namens
System.out.println(nicknames.get("matthew"));  // Ausgabe: matt
System.out.println(nicknames.get("arthur"));   // Ausgabe: artie
```

- **Vorteil**: Die HashMap ermöglicht schnellen Zugriff, selbst bei großen Datenmengen.

Hash Map: Maximale Werte pro Schlüssel

- **Wichtig zu wissen**: In einer HashMap ist jeder Schlüssel **eindeutig** und kann **nur mit einem Wert** verknüpft sein.
- Wenn wir einem existierenden Schlüssel einen neuen Wert zuweisen, wird der alte Wert **überschrieben**.
- Dies bedeutet, dass die HashMap **nicht** mehrere Werte für denselben Schlüssel speichern kann.

Beispielcode:

```
// Erstellen einer HashMap
HashMap<String, String> numbers = new HashMap<>();

// Hinzufügen eines Werts zum Schlüssel "Uno"
numbers.put("Uno", "One");

// Überschreiben des Werts für denselben Schlüssel
numbers.put("Uno", "Ein");

// Abrufen des Werts für "Uno"
System.out.println(numbers.get("Uno"));  // Ausgabe: Ein
```

- **Hinweis**: Wenn wir mehrere Werte für einen Schlüssel speichern möchten, müssen wir andere Strukturen verwenden, z.B. eine Liste als Wert.

Beispiel: Eine Hash Map mit Objekten

- **Flexibilität**: Eine HashMap kann nicht nur primitive Datentypen, sondern auch **Objekte** als Schlüssel oder Werte verwenden.
- Dies ermöglicht das Speichern komplexerer Datenstrukturen.

Beispielcode:

```
// Angenommen, wir haben eine Klasse Book
public class Book {
    private String name;
    private int published;
    private String content;

    public Book(String name, int published, String content) {
        this.name = name;
        this.published = published;
        this.content = content;
    }

    public String getName() {
        return this.name;
    }

    // Weitere Methoden...
}

// Erstellen einer HashMap mit String-Schlüsseln und Book-Objekten als Werte
```

Hash Map und Effizienz

- **HashMaps** bieten eine **hohe Effizienz** beim Speichern und Abrufen von Daten basierend auf Schlüsseln.
- Intern verwenden HashMaps eine **Hash-Funktion**, um die Position eines Eintrags direkt zu berechnen.
- Dies ermöglicht eine **konstante Zeitkomplexität O(1)** für Einfüge- und Abrufoperationen.
- **Vergleich**: In einer **ArrayList** oder einfachen Liste müssen Elemente sequenziell durchsucht werden (**O(n)**).

Beispiel:

```java
// Durchsuchen einer Liste mit 10 Millionen Elementen
ArrayList<Book> bookList = new ArrayList<>();
// Hinzufügen von 10 Millionen Büchern...

// Lineare Suche
Book foundBook = getBookFromList("Sense and Sensibility");

// Methode zum linearen Durchsuchen der Liste
public Book getBookFromList(String title) {
    for (Book book : bookList) {
        if (book.getName().equals(title)) {
            return book;
        }
    }
    return null;
}
```

- **Mit einer HashMap** können wir das Buch in konstanter Zeit finden:

```
// Verwenden einer HashMap für schnellen Zugriff
HashMap<String, Book> bookMap = new HashMap<>();
// Hinzufügen von Büchern zur HashMap...

Book foundBook = bookMap.get("Sense and Sensibility");
```

Durchlaufen der Schlüssel einer Hash Map

- Es kann nützlich sein, **alle Schlüssel** in einer HashMap zu durchlaufen, z.B. um alle Einträge anzuzeigen.
- Die Methode `keySet()` gibt ein **Set** aller Schlüssel in der HashMap zurück.

Beispielcode:

```
// Angenommen, wir haben eine HashMap namens 'library'
HashMap<String, Book> library = new HashMap<>();

// Hinzufügen von Büchern
library.put("Pride and Prejudice", new Book("Pride and Prejudice", 1813, "..."))
library.put("1984", new Book("1984", 1949, "..."));

// Durchlaufen aller Schlüssel
for (String key : library.keySet()) {
    System.out.println("Buchtitel: " + key);
}
```

Durchlaufen der Werte einer Hash Map

- Neben den Schlüsseln kann es sinnvoll sein, **alle Werte** in einer HashMap zu durchlaufen.
- Die Methode `values()` gibt eine **Collection** aller Werte zurück.

Beispielcode:

```
// Durchlaufen aller Werte
for (Book book : library.values()) {
    System.out.println("Buchdetails: " + book);
}
```

- **Vorteil**: Zugriff auf alle gespeicherten Objekte und deren Methoden.

Hash Maps mit primitiven Typen

- **Primitive Typen** (z.B. `int`, `double`, `char`) können nicht direkt als Typparameter für generische Klassen verwendet werden.
- Stattdessen verwenden wir die entsprechenden **Wrapper-Klassen** (z.B. `Integer`, `Double`, `Character`).
- **Auto-Boxing** ermöglicht die automatische Konvertierung zwischen primitiven Typen und Wrapper-Klassen.

Beispielcode:

```
// Erstellen einer HashMap mit Integer-Schlüsseln
HashMap<Integer, String> map = new HashMap<>();

// Auto-Boxing von int zu Integer
map.put(1, "First Entry");
map.put(2, "Second Entry");

// Abrufen von Werten
System.out.println(map.get(1));  // Ausgabe: First Entry
```

Autoboxing in Java

- **Autoboxing** ist ein Mechanismus in Java, der die automatische Konvertierung von **primitiven Datentypen** zu ihren entsprechenden **Wrapper-Klassen** ermöglicht.
- **Primitive Datentypen** und ihre Wrapper-Klassen:
  - `int` ↔ `Integer`
  - `double` ↔ `Double`
  - `char` ↔ `Character`
  - `boolean` ↔ `Boolean`
- **Anwendung**: Erleichtert die Verwendung von primitiven Typen in Datenstrukturen, die Objekte erwarten.

Beispiel: Autoboxing

- **Autoboxing** in Aktion:

```
// Beispiel mit einer HashMap
HashMap<Integer, String> map = new HashMap<>();

int key = 5;
String value = "Five";

// Beim Aufruf von put() wird 'key' automatisch in 'Integer' umgewandelt
map.put(key, value);

// Abrufen des Werts
String retrievedValue = map.get(key);

System.out.println("Schlüssel: " + key + ", Wert: " + retrievedValue);  // Ausgabe: Schlüssel: 5, Wert: Five
```

- **Weitere Beispiele**:

```
ArrayList<Double> numbers = new ArrayList<>();

// Autoboxing von double zu Double
numbers.add(3.14);
numbers.add(2.718);

// Intern wird '3.14' in 'Double.valueOf(3.14)' umgewandelt
```

Unboxing in Java

- **Unboxing** ist der automatische Prozess, bei dem ein Objekt einer Wrapper-Klasse in den entsprechenden **primitiven Datentyp** umgewandelt wird.
- Dies geschieht, wenn ein Objektwert in einem Kontext verwendet wird, der einen primitiven Wert erwartet.

Beispielcode:

```
Integer wrapperValue = Integer.valueOf(20);

// Unboxing: 'wrapperValue' wird automatisch zu 'int' konvertiert
int primitiveValue = wrapperValue;

// Verwendung in einem arithmetischen Ausdruck
int sum = wrapperValue + 10;

System.out.println("Summe: " + sum);  // Ausgabe: Summe: 30
```

Vorteile von Autoboxing

- **Einfacherer Code**: Weniger explizite Konvertierungen notwendig.
- **Verbesserte Lesbarkeit**: Klarerer und sauberer Code.
- **Konsistente Verwendung von Generics**: Erleichtert die Arbeit mit generischen Klassen.

Beispiel ohne Autoboxing:

```
ArrayList<Integer> numbers = new ArrayList<>();

// Vor Java 5 musste man manuell boxen
numbers.add(Integer.valueOf(42));
```

Beispiel mit Autoboxing:

```
// Ab Java 5
numbers.add(42);  // Autoboxing wandelt '42' in 'Integer.valueOf(42)' um
```

Hash Maps und `null`-Werte

- **Standardverhalten**: Bei Abfrage eines nicht existierenden Schlüssels gibt `get()` **null** zurück.
- **Vorsicht**: `null` kann zu `NullPointerException` führen, wenn nicht korrekt behandelt.
- **Lösungen**:
  - Verwenden von `getOrDefault()`:

```
System.out.println(map.getOrDefault("Unknown Key", "Standardwert"));
```

  - Überprüfen mit `containsKey()`:

```
if (map.containsKey("SomeKey")) {
    // Schlüssel existiert
} else {
    // Schlüssel existiert nicht
}
```

- **Vorteil**: Bessere Kontrolle über Standardwerte und Fehlervermeidung.

Vergleich von Suchalgorithmen

Aufbau des Experiments

- **Ziel**: Effizienz verschiedener Suchalgorithmen vergleichen.
- **Vorgehensweise**:
  - Generieren eines **Arrays** mit **10.000.000** zufälligen Ganzzahlen.
  - Implementierung von drei Suchmethoden:
    a. **Lineare Suche**
    b. **Binäre Suche** (nach Sortierung)
    c. **HashMap-Suche**
- **Parameter**:
  - Array-Größe als Kommandozeilen-Argument.
- **Durchführung**:
  - Suchen nach derselben Zahl mit unterschiedlicher Häufigkeit:
    - 1 Suche
    - 100 Suchen
    - 10.000 Suchen

Präsentation der Suchalgorithmen

- **Detaillierte Betrachtung**:
  - **Lineare Suche**: Funktionsweise und Effizienz bei großen Datenmengen.
  - **Binäre Suche**: Voraussetzungen (sortiertes Array) und Algorithmus.
  - **HashMap-Suche**: Nutzung von HashMaps für schnellen Zugriff.
- **Vergleichskriterien**:
  - Zeitkomplexität.
  - Gesamtzeitaufwand bei verschiedenen Suchhäufigkeiten.
- **Zielsetzung**:
  - Verständnis der **Effizienzunterschiede**.
  - Anwendung der **Prinzipien der Programmierung** wie Effizienz und Algorithmendesign.

Code-Highlights der Suchalgorithmen

Lineare Suche

- **Beschreibung**: Durchläuft das Array sequenziell.
- **Zeitkomplexität**: O(n)

```
public boolean linearSearch(int[] array, int target) {
    for (int num : array) {
        if (num == target) {
            return true;
        }
    }
    return false;
}
```

Binäre Suche

- **Beschreibung**: Funktioniert auf sortierten Arrays, teilt den Suchbereich.
- **Zeitkomplexität**: O(log n)

```
Arrays.sort(array);  // Sortieren des Arrays (einmalig)

public boolean binarySearch(int[] array, int target) {
    int index = Arrays.binarySearch(array, target);
    return index >= 0;
}
```

HashMap-Suche

- **Beschreibung**: Verwendet eine HashMap für schnellen Zugriff.
- **Zeitkomplexität**: O(1) im Durchschnitt

```
HashMap<Integer, Boolean> map = new HashMap<>();
for (int num : array) {
    map.put(num, true);
}

public boolean hashMapSearch(HashMap<Integer, Boolean> map, int target) {
    return map.containsKey(target);
}
```

Experimentelle Ergebnisse: Lineare Suche

- **Testergebnisse**:

Anzahl der Suchen	Gesamtzeitaufwand
1 Suche	4 ms
100 Suchen	188 ms
10.000 Suchen	17.903 ms

- **Analyse**:
  - Lineare Zunahme der Suchzeit.
  - Ungeeignet für große Datenmengen oder häufige Suchen.

Experimentelle Ergebnisse: Binäre Suche

- **Testergebnisse**:

Vorgang	Zeitaufwand
Sortierzeit	880 ms
1 Suche	0 ms
100 Suchen	0 ms
10.000 Suchen	2 ms
1.000.000 Suchen	26 ms

- **Analyse**:
  - Einmalige Sortierzeit erforderlich.
  - Sehr effizient bei wiederholten Suchen.
  - Zeitaufwand steigt logarithmisch.

Experimentelle Ergebnisse: HashMap-Suche

- **Testergebnisse**:

Vorgang	Zeitaufwand
Aufbau der HashMap	N/A (Teil der Initialisierung)
1 Suche	0 ms
100 Suchen	0 ms
10.000 Suchen	2 ms
1.000.000 Suchen	7 ms

- **Analyse**:
  - Konstant schnelle Suchzeiten.
  - Keine Sortierzeit erforderlich.
  - Sehr effizient bei großen Datenmengen und häufigen Suchen.
  - Aber?

Zusammenfassung der Ergebnisse

Methode	Sortierzeit	1 Suche	100 Suchen	10.000 Suchen	1.000.000 Suchen
Lineare Suche	N/A	4 ms	188 ms	17.903 ms	N/A
Binäre Suche	880 ms	0 ms	0 ms	2 ms	26 ms
HashMap-Suche	N/A	0 ms	0 ms	2 ms	7 ms

- **Schlussfolgerung**:
  - **Lineare Suche** ist ineffizient für große Datenmengen.
  - **Binäre Suche** ist effizient nach einmaliger Sortierung.
  - **HashMap-Suche** bietet konstant schnelle Zugriffszeiten ohne Sortierung.

Wichtige Erkenntnisse

- **Effizienz** ist ein zentrales Prinzip der Programmierung.
- Die Wahl des richtigen Suchalgorithmus hängt von den Anforderungen ab.
- **HashMaps** eignen sich hervorragend für schnelle Suchen in großen Datenmengen.
- **Binäre Suche** ist effizient, erfordert aber ein sortiertes Array.
- **Lineare Suche** sollte vermieden werden, wenn schnelle Zugriffszeiten benötigt werden.

Mathematisch(er)e Einführung in Hashing

- **Hashing** ist eine Technik, die einen Eingabewert in einen **Hash-Wert** umwandelt.
- Der Hash-Wert dient als Index zur schnellen Speicherung und zum effizienten Abrufen von Daten.
- Hashing verwendet eine **Hash-Funktion** $h(x)$, die einen Schlüssel x in eine Position in einem Array oder einer Tabelle transformiert.

Hash-Funktion: Mathematische Definition

- Eine **Hash-Funktion** ist eine mathematische Funktion $h : X \to \{0, 1, \ldots, m-1\}$.
  - X ist die Menge der möglichen Schlüssel.
  - m ist die Größe des Arrays oder der Hash-Tabelle.
- **Ziel**: Die Hash-Werte sollten **gleichmäßig** und **zufällig** verteilt sein, um Kollisionen zu minimieren.

Beispiel für eine einfache Hash-Funktion

- **Modulare Hash-Funktion**:

$$h(x) = (x \cdot a + b) \mod m$$

  - x ist der Schlüssel (Frage: was, wenn x eine Datei oder ein Objekt ist?)
  - a, b sind Konstanten.
  - m ist die Größe der Hash-Tabelle.

- **Beispiel**:

  - $x = 42, a = 7, b = 3, m = 10$
  - Berechnung:

$$h(42) = (42 \cdot 7 + 3) \mod 10 = 297 \mod 10 = 7$$

Eigenschaften einer guten Hash-Funktion

- **Deterministisch**: Gleicher Eingabewert ergibt gleichen Hash-Wert.
- **Effizient**: Schnelle Berechnung des Hash-Werts.
- **Kollisionsresistent**: Minimiert die Wahrscheinlichkeit von Kollisionen.

Kollisionen

- Eine **Kollision** tritt auf, wenn $h(x_1) = h(x_2)$ für $x_1 \neq x_2$.
- Aufgrund begrenzter Größe der Hash-Tabelle sind Kollisionen unvermeidbar.
- **Auswirkungen**: Können die Effizienz beeinträchtigen, müssen behandelt werden.

Kollisionen bei modularen Hash-Funktionen

Was ist eine Kollision?

- **Kollision**: Zwei unterschiedliche Eingaben x_1 und x_2 erzeugen denselben Hash-Wert:

$$h(x_1) = h(x_2)$$

- Gegebene Hash-Funktion:

$$h(x) = (x \cdot a + b) \mod m$$

- Bedingung für Kollision:

$$(x_1 - x_2) \cdot a \equiv 0 \mod m$$

Beispiel einer Kollision

Parameter der Hash-Funktion

- $a = 2, b = 3, m = 10$
- Eingaben:
  - $x_1 = 1$
  - $x_2 = 6$

Berechnung

1. Für $x_1 = 1$:

$$h(1) = (2 \cdot 1 + 3) \mod 10 = 5 \mod 10 = 5$$

2. Für $x_2 = 6$:

$$h(6) = (2 \cdot 6 + 3) \mod 10 = 15 \mod 10 = 5$$

Graphische Darstellung der Kollision

Hash-Werte für $h(x) = (2x + 3) \mod 10$

x	$h(x)$
0	3
1	5
2	7
3	9
4	1
5	3
6	5
7	7
8	9
9	1

- das ist **keine** gute Hash-Funktion (üblicherweise ist m prim)
- die Funktion ist nicht surjektiv
- a und c sind nicht teilerfremd (coprime)

Kollisionen

- Kollisionen sind unvermeidbar, wenn:
  - Der Wertebereich der Eingaben x größer als der Modulus m ist.
  - $(x_1 - x_2) \cdot a$ ein Vielfaches von m ist.
- Modularer Hash-Algorithmus sollte **Parameter** a, b, m sorgfältig wählen, um Kollisionen zu minimieren.

Behandlung von Kollisionen

- **Chaining**: Verwendung von Listen an jedem Array-Index.
- **Open Addressing**: Finden einer alternativen Speicherposition.

Chaining Beispiel: (Pseudo-Code)

```
function hash_insert(key, value):
    index = h(key)
    bucket = array[index]
    bucket.append((key, value))
```

Anwendung: Hash Map

- **HashMaps** verwenden Hash-Funktionen, um Schlüssel auf Indexe abzubilden.
- **Speicherung**: Schlüssel-Wert-Paare werden in Buckets gespeichert.
- **Abruf**: Hash-Funktion berechnet Index, schneller Zugriff auf Wert.

Zusammenfassung: Hashing

- **Hashing** ermöglicht effizientes Speichern und Abrufen von Daten.
- **Gute Hash-Funktionen** sind entscheidend für die Effizienz.
- **Kollisionen** müssen effektiv behandelt werden.

Zusammenfassung: Hash Maps

- **HashMaps** speichern Schlüssel-Wert-Paare für schnellen Zugriff.
- **Eindeutige Schlüssel**: Jeder Schlüssel ist einzigartig, überschreiben möglich.
- **Durchlaufen**: Schlüssel und Werte können iteriert werden.
- **Effizienz**: Ideal für große Datenmengen mit häufigem Zugriff.

Methoden Überschreiben und Überladen in Java

Überschreiben (Overriding)

- **Definition**: Eine Methode der **Unterklasse** ersetzt die Methode der **Oberklasse** mit **gleicher Signatur**.
- **Zweck**: Anpassung oder Erweiterung der Funktionalität der Oberklasse.

Eigenschaften:

- Gleicher Methodenname
- Gleiche Parameterliste
- Gleicher Rückgabetyp oder ein Untertyp (kovarianter Rückgabetyp)
- Sichtbarkeit darf nicht eingeschränkt werden
- Wird in der Regel mit `@Override` annotiert

Beispiel für Überschreiben

```java
class Oberklasse {
    public void zeigeNachricht() {
        System.out.println("Nachricht aus der Oberklasse");
    }
}

class Unterklasse extends Oberklasse {
    @Override
    public void zeigeNachricht() {
        System.out.println("Nachricht aus der Unterklasse");
    }
}

public class Main {
    public static void main(String[] args) {
        Oberklasse obj = new Unterklasse();
        obj.zeigeNachricht(); // Ausgabe: Nachricht aus der Unterklasse
    }
}
```

Erklärung:

- `Unterklasse` überschreibt die Methode `zeigeNachricht()` der `Oberklasse`.
- Beim Aufruf von `zeigeNachricht()` wird die Version der `Unterklasse` verwendet.

Überladen (Overloading)

- **Definition**: Mehrere Methoden im **selben Kontext** mit **gleichem Namen**, aber **unterschiedlichen Parametern**.
- **Zweck**: Bereitstellung verschiedener Implementierungen für unterschiedliche Eingaben.

Eigenschaften:

- Gleicher Methodenname
- Unterschiedliche Parameterliste (Anzahl oder Typ)
- Rückgabetyp kann variieren
- Keine Vererbung erforderlich

Beispiel für Überladen

```
class Rechner {
    public int addiere(int a, int b) {
        return a + b;
    }

    public double addiere(double a, double b) {
        return a + b;
    }

    public int addiere(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Rechner rechner = new Rechner();
        System.out.println(rechner.addiere(2, 3));          // Ausgabe: 5
        System.out.println(rechner.addiere(2.5, 3.5));      // Ausgabe: 6.0
        System.out.println(rechner.addiere(1, 2, 3));       // Ausgabe: 6
    }
}
```

Erklärung:

- Die Methode `addiere` ist dreimal überladen mit unterschiedlichen Parameterlisten.
- Der passende Methodenaufruf wird basierend auf den Argumenten ausgewählt.

1

Vergleich: Überschreiben vs. Überladen

Merkmal	Überschreiben	Überladen
Ort	Zwischen Ober- und Unterklasse	Innerhalb derselben Klasse
Parameterliste	Muss identisch sein	Muss unterschiedlich sein
Rückgabetyp	Muss kompatibel sein (kovariant)	Kann beliebig sein
Beziehung	Nutzt Vererbung und Polymorphismus	Keine Vererbung notwendig
Annotation	Verwendung von `@Override` empfohlen	Keine Annotation notwendig

Hinweise

- **Überschreiben** erfordert Vererbung und ermöglicht Polymorphismus.
- **Überladen** erhöht die Flexibilität von Methoden innerhalb einer Klasse.
- Verwenden Sie `@Override`, um unbeabsichtigte Fehler beim Überschreiben zu vermeiden.
- Beim **Überladen** spielt der Rückgabetyp keine Rolle bei der Methodenauflösung.

Lernziele ("Vergleichen")

- **Wiederholung** des Vergleichs der Objektsgleichheit mit `equals`.
- **Verständnis** der Funktion der `hashCode`-Methode.
- **Erlernen** der "annähernden" Gleichheit von Objekten.
- **Nutzung** vorgefertigter Werkzeuge zur Erstellung von `equals` und `hashCode`.

Methode zum Testen auf Gleichheit - `equals`

- Die `equals`-Methode überprüft, ob zwei Objekte **inhaltlich gleich** sind.
- Standardmäßig wird die **Referenzgleichheit** überprüft.

Beispiel ohne Überschreibung:

```
Book book1 = new Book("Title", 2000, "...");
Book book2 = new Book("Title", 2000, "...");

System.out.println(book1.equals(book2)); // Ausgabe: false
```

- Um den **Inhaltsvergleich** zu ermöglichen, muss `equals` überschrieben werden.

Überschreiben der `equals`-Methode

- **Anpassung** der `equals`-Methode ermöglicht inhaltlichen Vergleich.

Beispiel:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Book)) return false;
    Book other = (Book) obj;
    return this.name.equals(other.name) && this.published == other.published;
}
```

- **Vergleich** der relevanten Attribute.

Beispiel für den `equals`-Vergleich

Nach Überschreiben von `equals`:

```
Book book1 = new Book("Title", 2000, "...");
Book book2 = new Book("Title", 2000, "...");

System.out.println(book1.equals(book2)); // Ausgabe: true
```

- **Ergebnis**: Objekte mit gleichem Inhalt werden als gleich angesehen.

`hashCode`-Methode

- **Zweck**: Liefert einen numerischen Hash-Wert für ein Objekt.
- **Wichtig** für Hash-basierte Datenstrukturen wie **HashMap**.
- **Regel (WICHTIG)**
  Wenn `equals` zwei Objekte als gleich ansieht, müssen sie denselben `hashCode` haben.

Konsistenz von `hashCode` und `equals`

Java-Anforderung:

- **Wenn `equals` zwei Objekte als gleich definiert**, müssen ihre **`hashCode`-Werte ebenfalls gleich sein**.
- Gilt besonders für:
  - `HashSet`
  - `HashMap`
  - Andere hash-basierte Datenstrukturen.

Warum?

- Konsistenz stellt sicher, dass Objekte korrekt in Hash-Datenstrukturen verarbeitet werden.

Beispiel: Klasse `Book`

```java
import java.util.Objects;

public class Book {
    private String name;
    private int published;

    public Book(String name, int published) {
        this.name = name;
        this.published = published;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Book book = (Book) obj;
        return published == book.published &&
               Objects.equals(name, book.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, published);
    }
```

Erklärung: Objects.hash()

Was ist `Objects.hash()`?

- Eine **statische Methode** aus der Klasse `java.util.Objects`.
- Erzeugt einen **Hashcode** basierend auf den übergebenen Objekten.

Warum wird es verwendet?

- Zum Implementieren von `hashCode()`, um konsistente und effiziente Hashcodes zu erzeugen.
- **Konsistenz**: Wenn zwei Objekte laut `equals()` gleich sind, müssen ihre Hashcodes gleich sein.

Erklärung: Objects.hash()

Wie funktioniert `Objects.hash()`?

- Verwendet eine Kombination der `hashCode()`-Methoden der übergebenen Objekte.
- **Details:**
  - Null-Werte werden sicher behandelt: `Objects.hash(null)` gibt 0 zurück.
  - Liefert einen **kombinierten Hashcode**, der die Eingaben eindeutig repräsentiert.

Im Beispiel:

```
@Override
public int hashCode() {
    return Objects.hash(name, published);
}
```

- Nimmt `name` und `published` als Eingaben.
- **Vorteil**: Reduziert Boilerplate-Code und sorgt für robusten Hashcode.

Wichtig: Der generierte Hashcode sollte in derselben Laufzeitumgebung stabil sein, kann aber zwischen unterschiedlichen JVM-Implementierungen variieren.

Verwendung und Konsistenztest

```java
Book book1 = new Book("Effective Java", 2018);
Book book2 = new Book("Effective Java", 2018);

// Konsistenz von equals und hashCode
System.out.println(book1.equals(book2)); // true
System.out.println(book1.hashCode() == book2.hashCode()); // true

// Verwendung in HashSet
Set<Book> books = new HashSet<>();
books.add(book1);
books.add(book2);

System.out.println(books.size()); // 1
System.out.println(books); // [Book{name='Effective Java', published=2018}]
```

Konsistenz:

- `book1.equals(book2)` gibt `true` zurück.
- Ihre `hashCode`-Werte sind gleich.
- `HashSet` erkennt, dass die Objekte gleich sind, und speichert nur eines.

Nochmal: Wichtige Punkte

Warum `Objects.hash` verwenden?

- Fasst mehrere Felder zusammen, um den `hashCode` zu berechnen.
- Kürzer und weniger fehleranfällig als manuelle Berechnung.

Beispiel:

```
@Override
public int hashCode() {
    return Objects.hash(name, published);
}
```

Java garantiert:

- Bei konsistentem `hashCode` und `equals` verhalten sich Hash-basierte Strukturen korrekt.

Beispiel: Nicht-konsistente Implementierung

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Book book = (Book) obj;
    return Objects.equals(name, book.name); // Vergleicht nur 'name'
}

@Override
public int hashCode() {
    return published; // Hashcode basiert nur auf 'published'
}
```

Konsequenzen:

- `equals()` vergleicht nur das Feld `name`.
- `hashCode()` basiert nur auf `published`.
- Das führt zu **Inkonsistenz**.

Beispielcode

```
Book book1 = new Book("Java Basics", 2023);
Book book2 = new Book("Java Basics", 2024);

System.out.println("book1.equals(book2): " + book1.equals(book2));

HashSet<Book> books = new HashSet<>();
books.add(book1);
books.add(book2);

System.out.println("HashSet enthält: " + books);
```

Ausgabe

```
book1.equals(book2): true
HashSet enthält: [Book{name='Java Basics', published=2023},
                  Book{name='Java Basics', published=2024}]
```

Warum?

- Laut `equals()` sind `book1` und `book2` gleich.
- Aber ihre `hashCode` -Werte unterscheiden sich:
  - `book1.hashCode() = 2023`
  - `book2.hashCode() = 2024`
- **HashSet** behandelt sie daher als unterschiedliche Objekte.

Korrekte Implementierung

Konsistenter `hashCode`:

```java
@Override
public int hashCode() {
    return Objects.hash(name); // Konsistenz mit equals
}
```

Erwartete Ausgabe nach der Korrektur

```
book1.equals(book2): true
HashSet enthält: [Book{name='Java Basics', published=2023}]
```

Warum funktioniert es jetzt?

- Beide Objekte haben denselben `hashCode`, da er auf `name` basiert.
- **HashSet** erkennt sie als gleich und speichert nur eins.

Regeln:

1. Wenn zwei Objekte laut `equals()` gleich sind, müssen ihre Hashcodes gleich sein.
2. Verwenden Sie `Objects.hash(...)`, um konsistente Hashcodes einfach zu implementieren.
3. Testen Sie immer `equals` und `hashCode` zusammen in Szenarien mit Hash-basierten Collections.

```
Objects.hash(name, published);
```

Damit bleibt Ihr Code robust und leicht wartbar!

Noch mehr Beispiele

Beispiel: Verwendung von `hashCode`

```
HashMap<Book, String> borrowers = new HashMap<>();
Book book = new Book("Title", 2000, "...");

borrowers.put(book, "Alice");
System.out.println(borrowers.get(new Book("Title", 2000, "..."))); // Ausgabe: A
```

- **Ohne korrektes `equals` und `hashCode`** würde `get` null zurückgeben.

Vergleich von Objekten in Listen

- **ArrayList** verwendet `equals` für Methoden wie `contains`.

Beispiel:

```
ArrayList<Book> books = new ArrayList<>();
books.add(new Book("Title", 2000, "..."));

System.out.println(books.contains(new Book("Title", 2000, "..."))); // Ausgabe:
```

- **Wichtig**: `equals` muss korrekt überschrieben sein.

Verwendung von `hashCode` in Hash-basierten Strukturen

- **HashMap** nutzt `hashCode`, um den Speicherort zu bestimmen.
- **Effizienz**: Korrekte Implementierung verbessert Leistung und Korrektheit.

Zusammenfassung

- `equals` ermöglicht inhaltlichen Vergleich von Objekten.
- `hashCode` ist entscheidend für die Funktion von Hash-basierten Strukturen.
- **Konsistente Implementierung** beider Methoden ist unerlässlich.
- **Prinzipien** wie **Kapselung** und **Abstraktion** werden angewendet.

Prinzipien der Programmierung

Hier:

- **Effizienz**: Ressourcen optimal nutzen.
- **Abstraktion**: Komplexität reduzieren durch Fokus auf das Wesentliche.
- **Modularität**: Programme in unabhängige Module aufteilen.
- **Kapselung**: Daten und Methoden innerhalb von Klassen schützen.
- **Konsistenz**: Einheitliche Logik und Konzepte im gesamten Code.

Nicht so sehr:

- **Kohäsion**: Klassen sollten eine klar definierte Aufgabe erfüllen.

Modularität

- **Definition**: Aufteilung von Programmen in **unabhängige**, **austauschbare** Module.
- **Vorteile**:
  - **Wiederverwendbarkeit**: Module können in verschiedenen Projekten genutzt werden.
  - **Wartbarkeit**: Einfacheres Debuggen und Aktualisieren.
  - **Abstraktion**: Versteckt interne Implementierungsdetails.

Beispiel:

```
public class HashMap<K, V> {
    // Interna sind gekapselt
}
```

- **HashMap** ist ein Modul, das eine klare Schnittstelle bietet.

Effizienz: Die Rolle von `hashCode` bei der Suche

- **Effizienzsteigerung** durch korrekt implementiertes `hashCode`.
- **Schneller Zugriff** in Hash-basierten Strukturen.
- **Ressourcenschonung**: Reduziert Rechenzeit und Speicherbedarf.

Kapselung: Sicherstellung der Datenintegrität

- **Kapselung** schützt Daten vor unbefugtem Zugriff.
- **`equals` und `hashCode`** kontrollieren den Vergleich von Objekten.
- **Vorteil**: Verhindert unerwünschte Seiteneffekte.

Beitrag zur Kapselung	Erklärung
Vergleich auf Abstraktionsebene	`equals` ermöglicht Vergleich ohne Zugriff auf interne Felder.
Verstecken der internen Struktur	`hashCode` generiert Werte aus Feldern, ohne deren Details offenzulegen.
Flexibilität bei Änderungen	Änderungen an internen Feldern erfordern nur Anpassung in `equals` / `hashCode`.
Zusammenarbeit mit Datenstrukturen	Hash-basierte Datenstrukturen verwenden die öffentliche API (`equals` , `hashCode`).

Kohärenz (hier nur als Ergänzung)

- **Kohärenz**: Logische Zusammengehörigkeit von Komponenten.
- **Problem bei Nicht-Kohärenz**:

```
public class Rechteck {
    private int breite;
    private int hoehe;
    private int flaeche;

    public void setBreite(int breite) {
        this.breite = breite;
    }

    public void setHoehe(int hoehe) {
        this.hoehe = hoehe;
    }

    public int getFlaeche() {
        return flaeche; // Nicht aktualisiert
    }
}
```

- **Lösung**: Fläche bei Änderung von Breite oder Höhe aktualisieren.

Konsistenz

Konsistenz von `equals` und `hashCode`

- **Wichtig:** `equals` und `hashCode` müssen **konsistent** implementiert werden.
- **Regeln:**
  i. Wenn zwei Objekte laut `equals()` gleich sind, müssen sie denselben `hashCode` haben.
  ii. Wenn zwei Objekte ungleich sind, dürfen sie unterschiedliche `hashCode`-Werte haben (nicht zwingend notwendig, aber empfohlen).
- **Warum?**
  - Sicherstellung korrekten Verhaltens in Hash-basierten Collections wie `HashSet`, `HashMap`, etc.
  - Vermeidung von Fehlern oder unerwartetem Verhalten.

Fazit: Immer `equals` und `hashCode` zusammen überprüfen und anpassen! **Konsistenz!**

Kohäsion (hier nur als Ergänzung)

- **Kohäsion**: Klasse erfüllt eine **spezifische Aufgabe**.
- **Gute Kohäsion**:

```java
public class Buch {
    private String titel;
    private String autor;

    public String getAutor() {
        return autor;
    }
}
```

- **Schlechte Kohäsion**:

```java
public class Buch {
    // Unpassende Methode
    public void erzeugeBibliotheksBenutzer() {
        // ...
    }
}
```

Zusammenfassung

Hier Fokus auf:

- **Modularität**: Erhöht Wiederverwendbarkeit und Wartbarkeit.
- **Effizienz**: Optimierung durch geeignete Datenstrukturen und Algorithmen.
- **Kapselung**: Schützt Daten und fördert kontrollierten Zugriff.
- **Abstraktion**: Komplexität reduzieren durch Fokus auf das Wesentliche.
- **Konsistenz**: Einheitliche Anwendung von Methoden im gesamten Code.

Nicht so sehr:

- **Kohärenz und Kohäsion**: Stellen logische Konsistenz sicher.

Lernziele (Gruppierung von Daten mit HashMaps)

- **Verwendung einer Liste als Wert** in einer HashMap.
- **Datenkategorisierung** mit einer HashMap.

HashMap mit einem Wert pro Schlüssel

```
HashMap<String, String> phoneNumbers = new HashMap<>();
phoneNumbers.put("Pekka", "040-12348765");
System.out.println("Pekkas Nummer: " + phoneNumbers.get("Pekka"));

phoneNumbers.put("Pekka", "09-111333");
System.out.println("Pekkas Nummer: " + phoneNumbers.get("Pekka"));
```

Pekkas Nummer: 040-12348765

Pekkas Nummer: 09-111333

- **Problem**: Alter Wert wird überschrieben.

HashMap mit mehreren Werten pro Schlüssel

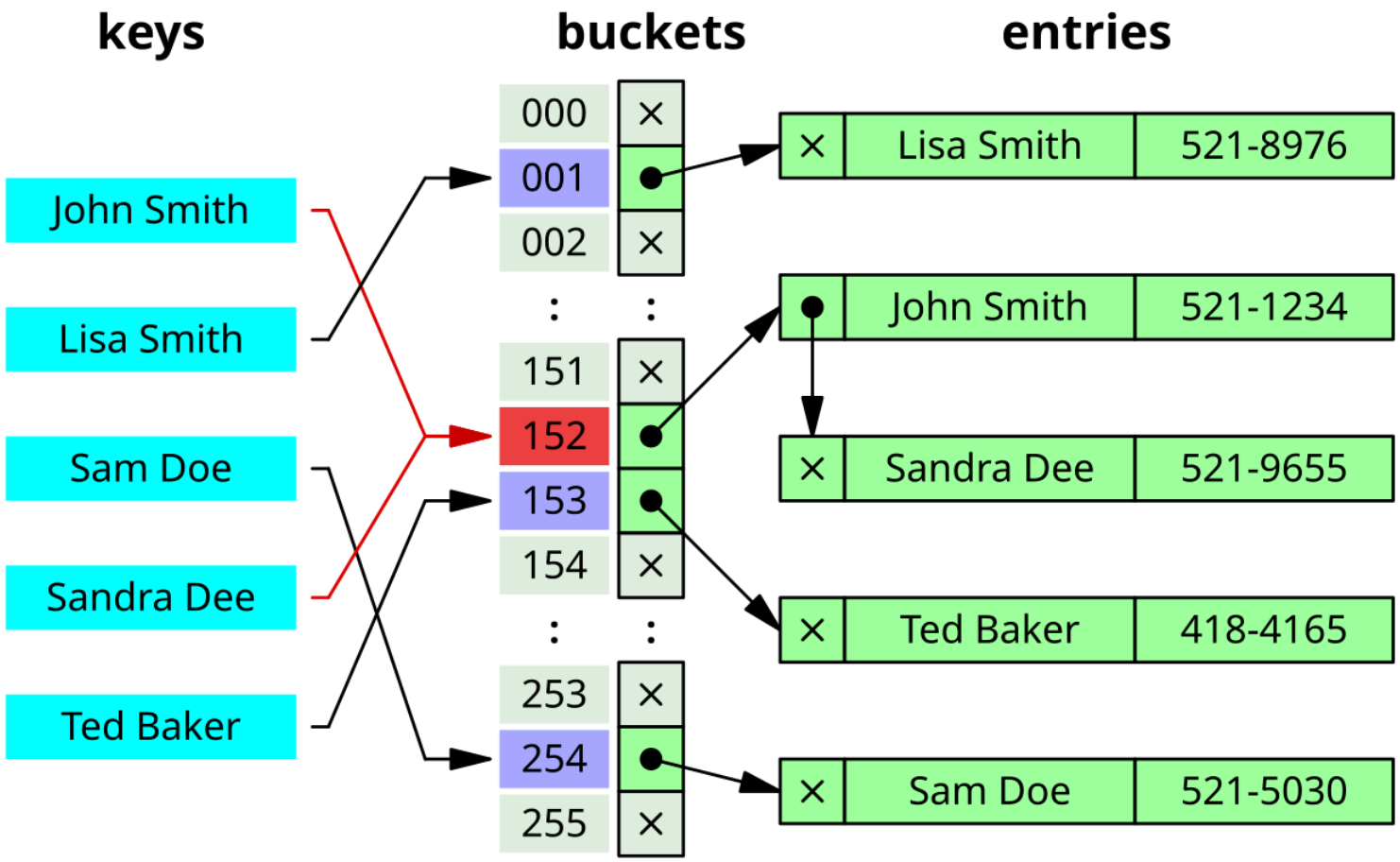
- **Lösung**: Verwenden einer **Liste** als Wert.

```
HashMap<String, ArrayList<String>> phoneNumbers = new HashMap<>();
phoneNumbers.put("Pekka", new ArrayList<>());

phoneNumbers.get("Pekka").add("040-12348765");
phoneNumbers.get("Pekka").add("09-111333");

System.out.println("Pekkas Nummern: " + phoneNumbers.get("Pekka"));
```

Pekkas Nummern: [040-12348765, 09-111333]


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

(`wikipedia`)

HashMap mit Listen als Werte

- **Deklaration**:

```
HashMap<String, ArrayList<String>> phoneNumbers = new HashMap<>();
```

- **Schlüssel**: `String` (z.B. Name)
- **Wert**: `ArrayList<String>` (Liste von Telefonnummern)
- **Vorteil**: Speicherung mehrerer Werte pro Schlüssel.

Beispiel: TaskTracker

- **Ziel**: Nachverfolgung abgeschlossener Übungen pro Benutzer.

```
public class TaskTracker {
    private HashMap<String, ArrayList<Integer>> completedExercises;

    public TaskTracker() {
        this.completedExercises = new HashMap<>();
    }

    public void add(String user, int exercise) {
        this.completedExercises.putIfAbsent(user, new ArrayList<>());
        this.completedExercises.get(user).add(exercise);
    }

    public void print() {
        for (String name : completedExercises.keySet()) {
            System.out.println(name + ": " + completedExercises.get(name));
        }
    }
}
```

Verwendung des TaskTracker

```
TaskTracker tracker = new TaskTracker();
tracker.add("Ada", 3);
tracker.add("Ada", 4);
tracker.add("Pekka", 4);
tracker.add("Matti", 1);
tracker.print();
```

Matti: [1]

Pekka: [4]

Ada: [3, 4]

Überprüfen, ob ein Element in einer ArrayList ist

Beispiel: TaskTracker

Wir wollen prüfen, ob **Ada Task 4** abgeschlossen hat.

Code zur Überprüfung

```
public boolean hasCompletedTask(String user, int exercise) {
    // Überprüfen, ob der Nutzer existiert und die Aufgabe in der Liste ist
    return this.completedExercises.containsKey(user) &&
           this.completedExercises.get(user).contains(exercise);
}
```

Erklärung

Was macht diese Zeile?

```
this.completedExercises.get(user).contains(exercise);
```

1. **`this.completedExercises.get(user)`**:
   - Holt die **ArrayList** der abgeschlossenen Aufgaben für den Nutzer `user`.
   - Rückgabe: Eine Liste der Aufgaben, z. B. `[3, 4]`.
2. **`.contains(exercise)`**:
   - Durchsucht die **ArrayList**, ob sie den Wert `exercise` enthält.
   - Rückgabe: `true`, wenn die Aufgabe vorhanden ist, sonst `false`.

Laufzeit

- `get(user)`:
  - **O(1)** (Konstante Zeit) bei einer **HashMap**.
- `.contains(exercise)`:
  - **O(n)** (Lineare Zeit), da die ArrayList sequenziell durchsucht wird.
  - `n` = Anzahl der Aufgaben des Nutzers.

Umgang mit Hash-Kollisionen

- **Kollisionen** sind unvermeidbar, aber handhabbar.
- **Verkettete Listen** oder **Open Addressing** sind gängige Lösungen.
- **Hashing** bleibt trotz Kollisionen eine effiziente Methode.

Zusammenfassung

- **HashMaps** sind leistungsfähige Datenstrukturen für schnellen Datenzugriff.
- **Flexibilität** durch Verwendung von Listen als Werte.
- **Prinzipien der Programmierung** wie Effizienz und Modularität werden angewendet.

Java und Modulare Hash-Funktionen

- **Hinweis**: Java verwendet **nicht explizit** die modulare Hash-Funktion:

$$h(x) = (x \cdot a + b) \mod m$$

- **Stattdessen**:
  - Spezifische `hashCode()` -Implementierungen pro Klasse.
  - Ziel: Gleichmäßige Verteilung der Hashcodes.

Beispiel: `String.hashCode()`

- **Formel**:

$$\text{hash} = s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \cdots + s[n-1]$$

  - (s[i]): Zeichen an Position (i)
  - (n): Länge des Strings
- **Merkmale**:
  - Verwendet die Primzahl **31**.
  - Berücksichtigt die **Reihenfolge** der Zeichen.

Beispiel: `List.hashCode()`

- **Implementierung**:

```
public int hashCode() {
    int hashCode = 1;
    for (E e : this)
        hashCode = 31 * hashCode + (e == null ? 0 : e.hashCode());
    return hashCode;
}
```

- **Merkmale**:
  - Iteriert über alle Elemente der Liste.
  - Nutzt rekursiv die `hashCode()`-Methoden der Elemente.
  - Multipliziert kumulativen Hashcode mit **31**.

Zusammenfassung

- **Java verwendet nicht direkt modulare Hash-Funktion**.
- **Spezielle `hashCode()`-Methoden** pro Klasse verbessern die Verteilung.
- **Beispiele gezeigt**:
  - `String.hashCode()`: Polynomische Berechnung mit Primzahl 31.
  - `List.hashCode()`: Kombination der Hashcodes der Elemente.
- **Ziel**: Effiziente und kollisionsarme Hashing-Mechanismen.

Ende Teil 08