Berner
Fachhochschule

Programmieren in Java

Stephan Fischli

Herbst 2021

Inhalt

- Einführung
- Datentypen
- Klassen und Objekte
- Enumerations
- Records
- Packages
- Vererbung
- Interfaces
- Innere Klassen
- Standardbibliothek
- Exception-Handling
- Collections
- Generics
- Ein-/Ausgabe
- Threading

Einführung

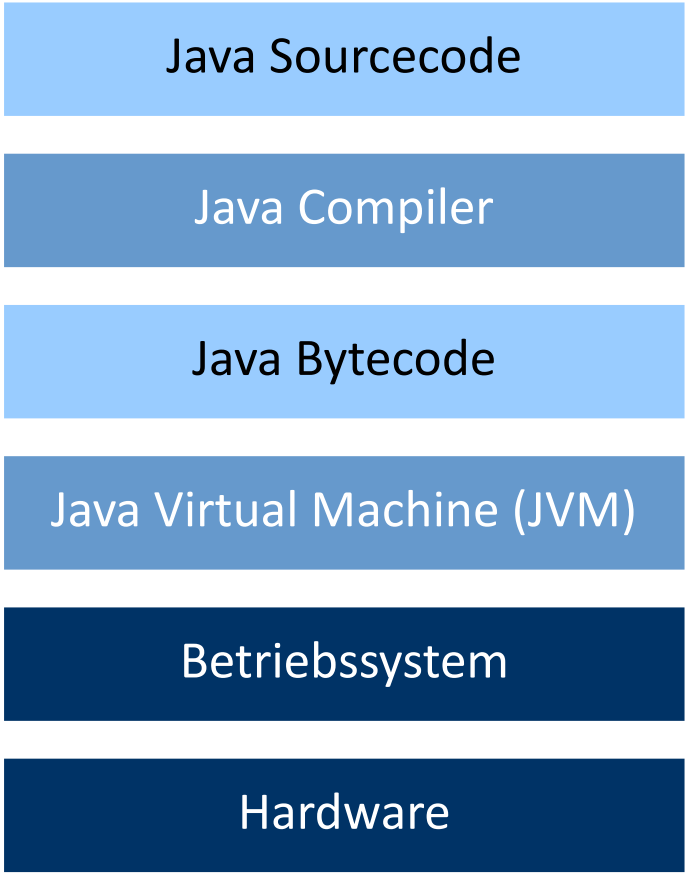
Warum Java?

- Einfachheit und Robustheit der Sprache
- Plattformunabhängigkeit (write once – run everwhere)
- Vielseitigkeit der Anwendungen
- Grosse Verbreitung

Java-Plattform

- Sprache
- Laufzeitumgebung
  Virtuelle Maschine (JVM), Garbage Collector, Hotspot-Compiler
- Standardbibliothek
  Collections, Ein-/Ausgabe, GUI, Datenbanken, Networking, ...
- Tools
  Compiler, Interpreter, Dokumentation, ...
- Editionen
  Standard (Java SE), Enterprise (Java EE), Micro (Java ME)

Java-Architektur



Entwicklungszyklus

- Implementierung in Datei Hello.java

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello " + (args.length > 0 ? args[0] : "world"));
    }
}
```

- Kompilation erzeugt Datei Hello.class

```
C:\>  javac Hello.java
```

- Ausführung der Klasse

```
C:\>  java Hello John
```

Geschichte von Java

Jahr	Version	Features
1995	JDK 1.0	Erste Version
1997	JDK 1.1	Innere Klassen, Reflection, Datenbanken
1998	J2SE 1.2	Collections, Swing
2000	J2SE 1.3	HotSpot-Compiler
2002	J2SE 1.4	Asserts, reguläre Ausdrücke, neue I/O-Bibliothek
2004	J2SE 5	Generics, Annotationen, Enums, Concurrency
2006	Java SE 6	Web Services
2011	Java SE 7	Try with Resources, Exception Multicatch
2014	Java SE 8	Lambda-Expressions, Streams
2016	Java SE 9	Modulsystem
2018	Java SE 10	Typinferenz lokaler Variablen
2019	Java SE 13	Switch-Ausdrücke, Mehrzeilen-Strings
2020	Java SE 14	Records, InstanceOf Pattern Matching
2020	Java SE 15	Sealed Classes
2021	Java SE 17	Switch Pattern Matching

Java-Community

- Java Community Process (JCP)
  Prozess zur Entwicklung neuer Java-Spezifikationen
  https://jcp.org/en/home/index
- Java Specification Request (JSR)
  Vorschläge für Erweiterungen der Java-Plattform
  https://jcp.org/en/jsr/overview
- OpenJDK
  Open-Source Entwicklung der Java-Plattform
  http://openjdk.java.net/

Literatur

- The Java Tutorial
  Raymond Gallardo et.al. (Addison-Wesley)
- Effective Java
  Joshua Bloch (CreateSpace)
- Einführung in Java
  Kai Günster (Rheinwerk)

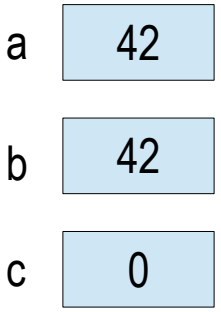
Datentypen

Primitive Typen

Primitive Typen

- repräsentieren Zahlen (byte, short, int, long, float, double), Zeichen (char) oder Wahrheitswerte (boolean)
- haben eine festgelegte Speichergrösse und Wertebereich
- können implizit oder explizit ineinander umgewandelt werden (Cast)
- erlauben bestimmte Operationen (Rechnen, Vergleiche, Bit-Manipulationen, Logische Verknüpfungen)
- haben Wertsemantik

```
int a = 42;
int b = a;
int c;
```

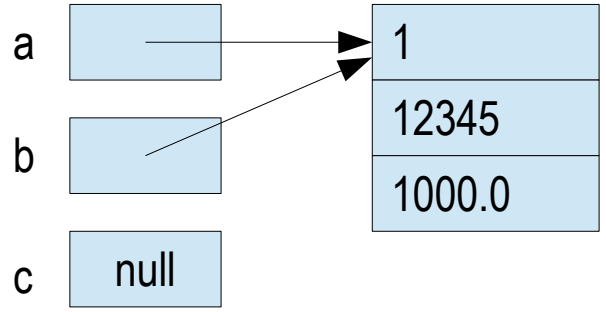


Objekttypen

Objekttypen

- repräsentieren Objekte und Arrays
- haben Referenzsemantik
- können null sein

```
Account a = new Account(1, "12345", 1000.0);
Account b = a;
Account c;
```



Referenzsemantik

- Nach Zuweisungen zeigen Referenzen auf dasselbe Objekt

```
a.balance += 500.0;
System.out.println(b.balance);
System.out.println(c.balance);
```

- Bei der Parameterübergabe werden Referenzen übergeben

```
void deposit(Account x, double amount) { x.balance += amount; }
deposit(a, 500.0);
System.out.println(a.balance);
```

- Bei Vergleichen werden Referenzen verglichen

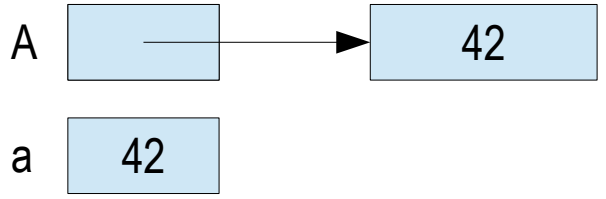
```
System.out.println(a == b);
System.out.println(a == c);
```

Wrapper-Klassen

Wrapper-Klassen (Byte, Short, Integer, Long, Float, Double, Character, Boolean)

- repräsentieren Objekte, die einen primitiven Typ enthalten
- haben zusätzliche Funktionalität (z.B. Parsen von Strings)
- können explizit oder implizit umgewandelt werden

```
Integer A = Integer.valueOf(42);
int a = A.intValue();

Integer A = 42;       // Autoboxing
int a = A;            // Unboxing
```

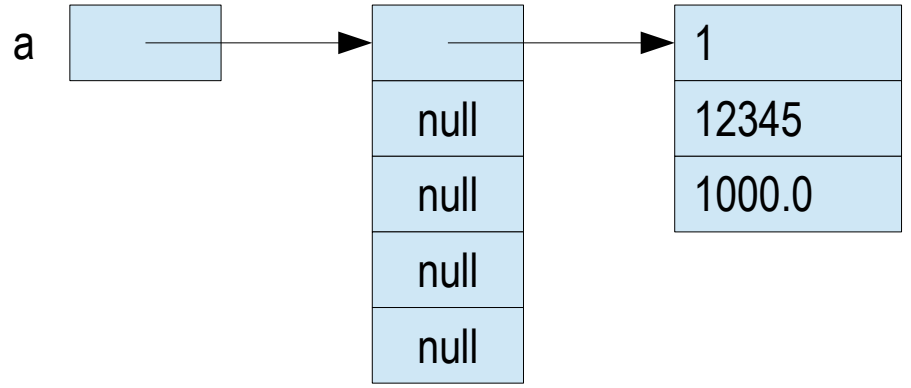


Arrays

Arrays

- enthalten eine feste Anzahl primitiver Werte oder Objekte eines Typs
- werden vor ihrer Verwendung deklariert, konstruiert und initialisiert
- kennen ihre Länge (Feld length)

```
Account[] a = new Account[5];
a[0] = new Account(1, "12345", 1000.0);
for (int i = 0; i < a.length; i++)
    System.out.println(a[i].getBalance());
```



Strings

Strings

- sind Zeichenketten, die als Objekte der Klasse String repräsentiert werden
- können nicht verändert werden (immutable)
- können mit dem Operator + konkateniert werden

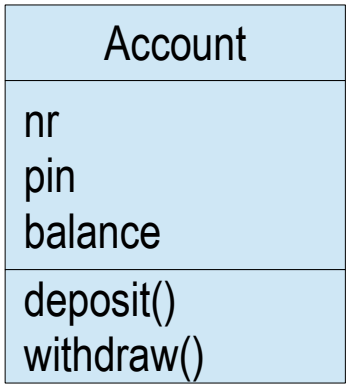
```
String copyright = "\u00a9 Copyright by ...";
if ("nobody".equals(myName)) ...
Sytem.out.println("Server startup time: " + time + " ms");
```

Klassen und Objekte

Definition einer Klasse

- Eine Klasse definiert einen Datentyp bestehend aus Feldern (Daten) und Methoden (Funktionen), welche auf die Felder zugreifen

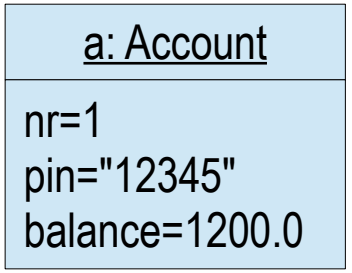
```
public class Account {
    // Felder
    int nr;
    String pin;
    double balance = 0.0;

    // Methoden
    void deposit(double amount) {
        balance += amount;
    }
    void withdraw(double amount) {
        balance -= amount;
    }
}
```



Erzeugen von Objekten

- Objekte einer Klasse werden mit dem new-Operator erzeugt
- Jedes Objekt erhält eine eigene Kopie der Felder
- Über die Objektreferenz kann auf die Felder und Methoden des Objekts zugegriffen werden

```
Account a = new Account();

a.nr = 1;
a.pin = "12345";
a.balance = 1000.0;

a.deposit(200.0);
```



Zugriffsrechte von Feldern und Methoden

- Private Felder sind nur innerhalb der Klasse zugreifbar (Daten-kapselung), ebenso private Methoden
- Public Felder und Methoden sind von überall aus zugreifbar

```
public class Account {
    private int nr;
    private String pin;
    private double balance = 0.0;

    public int getNr() { return nr; }                    // Selektoren
    public double getBalance() { return balance; }

    public void deposit(double amount) { ... }           // Modifikatoren
    public void withdraw(double amount) { ... }
}

Account a = new Account();
a.balance = 1000.0;                  // Fehler
```

Initialisieren von Objekten

- Konstruktoren werden implizit bei der Erzeugung eines Objekts aufgerufen und dienen der Initialisierung der Felder
- Wird kein Konstruktor definiert, so erzeugt der Compiler einen Defaultkonstruktor, der die Felder mit Defaultwerten initialisiert

```
public class Account {
    private int nr;
    private String pin;
    private double balance = 0.0;

    public Account(int number, String code) {          // Konstruktor
        nr = number;
        pin = code;
    }
    ...
}

Account a = new Account(1, "12345");
```

Überladen von Methoden

- Mehrere Methoden können denselben Namen haben, sofern sie sich in den Parametern unterscheiden
- Bei einem Aufruf entscheidet der Compiler anhand der Argumente, welche Methode ausgeführt wird (Argument Matching)

```
public class Account {
    ...
    public void withdraw(double amount) {
        balance -= amount;
    }
    public void withdraw(String code, double amount) {
        if (pin.equals(code)) withdraw(amount);
    }
}

Account a  = new Account();
a.withdraw("12345", 200.0);
```

Mehrfache Konstruktoren

- Auch Konstruktoren können überladen werden
- Mittels this() kann ein Konstruktor einen anderen Konstruktor aufrufen (muss das erste Statement sein)

```
public class Account {
    ...
    public Account(int number, String code, double amount) {
        nr = number; pin = code; balance = amount;
    }
    public Account(int number, String code) {
        this(number, code, 0.0);
    }
    ...
}

Account a = new Account(1, "12345", 1000.0);
```

this-Referenz

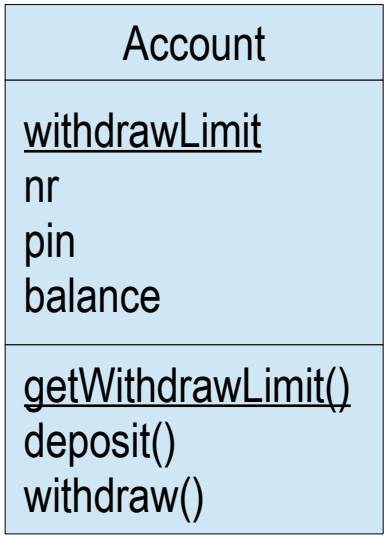
- Jeder Methode wird implizit die Referenz this auf das eigene Objekt übergeben
- this wird oft in Konstruktoren verwendet, um Namenskonflikte zwischen Feldern und Parametern aufzulösen

```
public class Account {
    private int nr;
    private String pin;
    private double balance = 0.0;

    public Account(int nr, String pin) {
        this.nr = nr;
        this.pin = pin;
    }
    ...
}
```

Klassenfelder und -methoden

- Klassenfelder existieren unabhängig von Objekten einmal pro Klasse und werden zur Speicherung globaler Daten verwendet
- Klassenmethoden werden nicht auf einem Objekt sondern auf der Klasse aufgerufen und können nur auf Klassenfelder zugreifen

```
public class Account {
    private static double withdrawLimit;
    ...
    public static int getWithdrawLimit() {
        return withdrawLimit;
    }
    public void withdraw(double amount) {
        if (amount <= withdrawLimit)
            balance -= amount;
    }
}

double limit = Account.getWithdrawLimit();
```



Konstante Felder

- Konstante Felder müssen bei ihrer Definition oder im Konstruktor initialisiert werden, danach dürfen sie nicht mehr verändert werden

```
public class Account {
    private static final double WITHDRAW_LIMIT = 5000.0;
    private final int nr;
    ...
    public void withdraw(double amount) {
        if (amount <= WITHDRAW_LIMIT)
            balance -= amount;
    }
}
```

Enumerations

Definition von Enums

Enum-Typen

- werden verwendet, um Konstanten zu definieren
- sind Klassen mit einer vordefinierten Menge von Objekten
- sind implizit von der Standardklasse Enum abgeleitet und erben allgemeine Methoden

```
public enum Month {
    JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}

for (Month month : Month.values())
    System.out.println(month.name());
Month month = Month.valueOf("AUG");
```

Verwendung von Enums

Enum-Werte

- werden mit dem Enum-Namen und dem Punktoperator referenziert
- können mit dem Vergleichsoperator verglichen und in switch-Statements verwendet werden

```
Month month = Month.AUG;
if (month == Month.JAN) ...
int days = switch (month) {
    case JAN -> 31;
    case FEB -> 28;
    ...
    case DEC -> 31;
}
```

Enums mit Feldern und Methoden

- Enums können auch Felder und Methoden haben
- Felder von Enums sind final und müssen mit einem privaten Konstruktor initialisiert werden

```
public enum Month {
    JAN(31), FEB(28), MAR(30), ..., DEC(31);
    private final int days;
    private Month(int days) { this.days = days; }
    public int days() { return days; }
}

Month month = Month.AUG;
System.out.println(month + " has " + month.days() + "days");
```

Records

Definition von Records

- Ein Record repräsentiert unveränderbare Objekte

```
public record Money(Currency currency, double amount) {}
```

- Der Java-Compiler erzeugt
  - private konstante Felder
  - public Getter-Methoden
  - einen Konstruktor zum Initialisieren aller Felder
  - passende equals-, hashCode- und toString-Methoden

Anpassung von Records

- Bei Bedarf kann der erzeugte Konstruktor angepasst werden

```
public record Money(Currency currency, double amount) {
    public Money {
        Objects.requireNonNull(currency);
    }
}
```

- Es dürfen weitere Konstruktoren mit verschiedenen Parametern sowie Klassenfelder und -methoden hinzugefügt werden

Packages

Definition eines Package

- Packages sind hierarchisch aufgebaute Sammlungen von Klassen und definieren Namensräume
- Um eine Klasse zu einem Package hinzuzufügen, wird am Anfang des Sourcefiles ein package-Statement verwendet
- Ohne package-Statement gehören die Klassen zum Default-Package ohne Namen

```
package org.bank.account;

public class Account {
    ...
}
```

Importieren eines Package

- Der vollständige Name einer Klasse setzt sich aus dem Package- und dem Klassennamen zusammen
- Mit einem import-Statement können Klassen mit dem Klassennamen allein verwendet werden
- Die Klassen aus dem Package java.lang werden defaultmässig importiert

```
package org.bank;
import org.bank.account.Account;
import org.bank.account.*;                    // alle Klassen

public class Bank {
    private Account[] accounts;
    public static void main(String[] args) { ... }
    ...
}
```

Klassenpfad

- Die Package-Hierarchien müssen sich in der Verzeichnisstruktur widerspiegeln
- Der Klassenpfad enthält die Root-Verzeichnisse der Package-Hierarchien und muss dem Compiler und Interpreter übergeben werden

- Directory-Struktur:

  C:\project\src\org\bank\Bank.java
  C:\project\src\org\bank\account\Account.java

- Kompilation:

  C:\project> javac -d bin src\org\bank\account\Account.java
  C:\project> javac **-cp bin** -d bin src\org\bank\Bank.java

- Ausführung:

  C:\project> java **-cp bin** org.bank.Bank

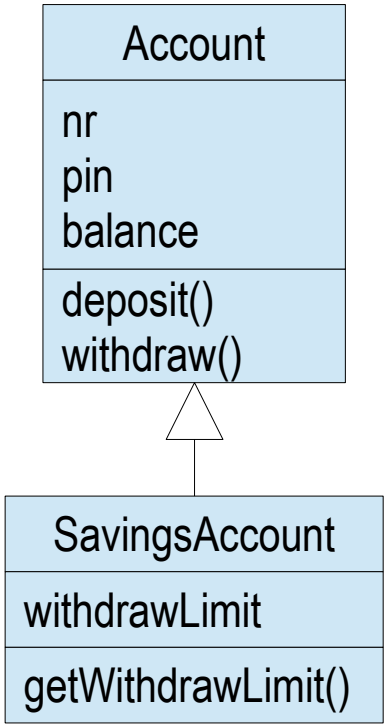
Vererbung

Ableiten einer Klasse

Eine Klasse kann von einer andern Klasse abgeleitet werden und

- erbt alle Felder und Methoden der Basisklasse
- kann neue Felder und Methoden hinzufügen (Erweiterung)
- definiert einen zur Basisklasse kompatiblen Datentyp

```
public class SavingsAccount extends Account {
    private double withdrawLimit;

    public double getWithdrawLimit() {
        return withdrawLimit;
    }
}
```



Konstruktorverkettung

- Konstruktoren werden nicht vererbt
- Mittels super() kann ein Konstruktor der abgeleiteten Klasse einen Konstruktor der Basisklasse aufrufen (muss das erste Statement sein)
- Andernfalls wird implizit der Defaultkonstruktor der Basisklasse aufgerufen

```
public class SavingsAccount extends Account {
    private double withdrawLimit;

    public SavingsAccount(int nr, String pin, double limit) {
        super(nr, pin);
        withdrawLimit = limit;
    }
    ...
}
```

Zugriffsrecht protected

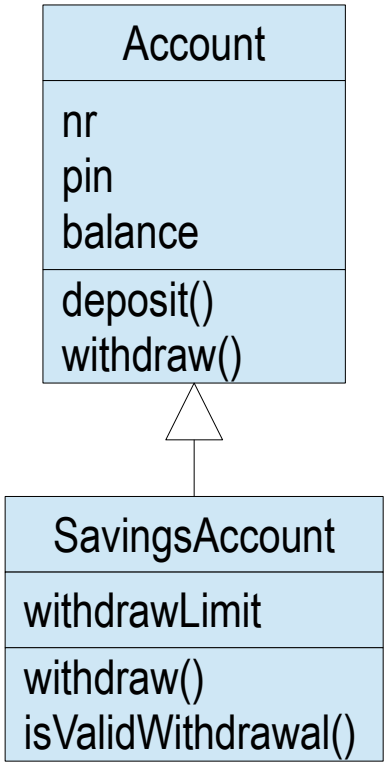
- Protected Felder und Methoden sind in der eigenen Klasse, in Klassen desselben Package und in abgeleiteten Klassen zugreifbar

```
public class Account {
    protected int nr;
    protected String pin;
    protected double balance;
    ...
}

public class SavingsAccount extends Account {
    ...
    private boolean isValidWithdrawal(double amount) {
        return amount <= withdrawLimit && amount <= balance;
    }
}
```

Überschreiben von Methoden

- Eine Methode der abgeleiteten Klasse überschreibt eine Methode der Basisklasse, wenn sie denselben Namen und dieselben Parameter hat
- Der Typ des Rückgabewerts beider Methoden muss übereinstimmen

```
public class SavingsAccount extends Account {
    private double withdrawLimit;
    …
    @Override
    public void withdraw(double amount) {
        if (isValidWithdrawal(amount))
            balance -= amount;
    }
}
```



Aufruf überschriebener Methoden

- Eine überschriebene Methode kann in der abgeleiteten Klasse weiterhin über die Referenz super aufgerufen werden

```
public class SavingsAccount extends Account {
    private double withdrawLimit;
    …
    @Override
    public void withdraw(double amount) {
        if (isValidWithdrawal(amount))
            super.withdraw(amount);
    }
}
```

Dynamische Bindung von Methoden

- Eine Referenz vom Typ einer Basisklasse kann auch auf ein Objekt einer abgeleiteten Klasse zeigen (Polymorphismus)
- Es können aber nur die Methoden des deklarierten Typs aufgerufen werden, sonst wird ein Cast benötigt
- Beim Aufruf einer überschriebenen Methode entscheidet der aktuelle Objekttyp, welche Methode ausgeführt wird

```
Account a;
a = new Account(...);
a.withdraw(200.0);                    // Methode von Account

a = new SavingsAccount(...);
double limit = a.getWithdrawLimit();  // Compiler-Fehler
a.withdraw(200.0);                    // Methode von SavingsAccount
```

Finale Methoden und Klassen

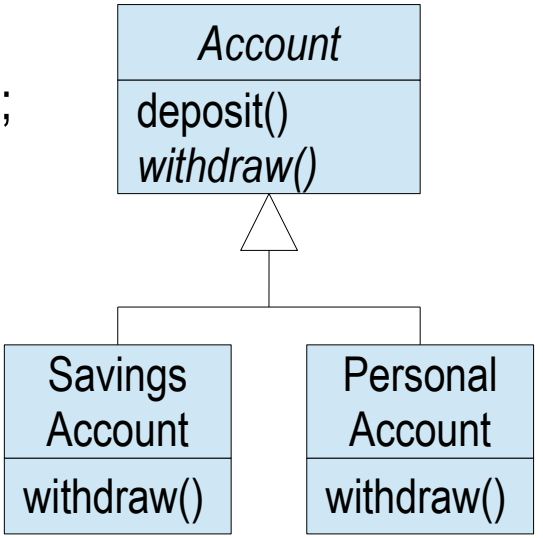
- Eine finale Methode kann in abgeleiteten Klassen nicht überschrieben werden
- Eine finale Klasse kann nicht abgeleitet werden

```
public class Account {
    ...
    public final String getNr() { return nr; }
    public final double getBalance() { return balance; }
}

public final class SavingsAccount {
    ...
}
```

Abstrakte Methoden und Klassen

- Abstrakte Methoden sind Methoden ohne Implementierung
- Hat eine Klasse eine abstrakte Methode, so ist die Klasse abstrakt
- Abstrakte Klassen können nicht instanziiert werden, dienen aber als Basisklassen anderer Klassen

```
public abstract class Account {
    public void deposit(double amount) { ... }
    public abstract void withdraw(double amount);
}
class SavingsAccount extends Account {
    public void withdraw(double amount) { ... }
}
class PersonalAccount extends Account {
    public void withdraw(double amount) { ... }
}
```



Interfaces

Definition eines Interface

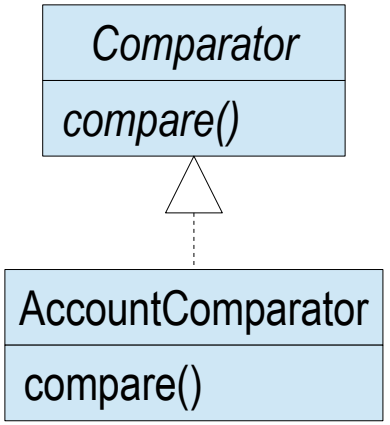
- Ein Interface definiert einen abstrakten Datentyp und beschreibt das Verhalten von Klassen
- Alle Methoden eines Interface sind implizit public und abstract

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

Implementieren eines Interface

- Eine Klasse implementiert ein Interface, indem sie die Methoden des Interface implementiert
- Eine Klasse kann mehrere Interfaces implementieren
- Eine Klasse ist typkompatibel zu den von ihr implementierten Interfaces

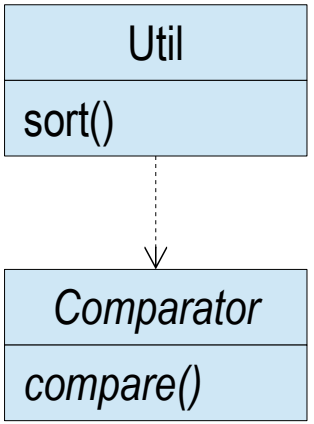
```
public class AccountComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Account a1 = (Account) o1; Account a2 = (Account) o2;
        return (int) (a1.getBalance() - a2.getBalance());
    }
}
```



Anwendung von Interfaces

- Interfaces werden oft als Parameter generischer Methoden verwendet
- Als Argumente können Objekte aller Klassen übergeben werden, die das Interface implementieren

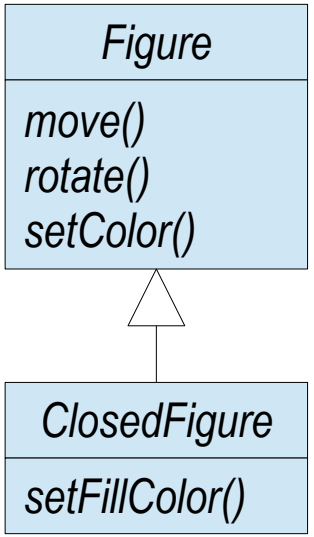
```
public class Util {
    public static void sort(Object x[], Comparator c) {
        for (int i = x.length - 1; i > 0; i--)
            for (int j = 0; j < i; j++)
                if (c.compare(x[j], x[j + 1]) > 0) {
                    Object t = x[j]; x[j] = x[j + 1]; x[j + 1] = t;
                }
    }
}

Account[] accounts = ...
Util.sort(accounts, new AccountComparator());
```



Vererbung von Interfaces

- Ein Interface kann von anderen Interfaces abgeleitet werden, wodurch es alle Methoden des Basis-Interface erbt

```
public interface Figure {
    void move(int dx, int dy);
    void rotate(int angle);
    void setColor(Color c);
}
interface ClosedFigure extends Figure {
    void setFillColor(Color c);
}
```



Statische und Default-Methoden

- Interfaces können statische Methoden haben, die wie Klassen-methoden nicht auf Objekten sondern auf dem Interface aufgerufen werden
- Default-Methoden ermöglichen es, neue Methoden zu Interfaces hinzuzufügen ohne die Rückwärtskompatibilität zu verletzen, indem sie eine Default-Implementierung haben

```
public interface Figure {
    static Color[] getColors() { ... }
    default void translate(int dx, int dy) { move(dx, dy); }
    void move(int dx, int dy);
    void rotate(int angle);
    void setColor(Color c);
}
```

Innere Klassen

Statische innere Klassen

- Statische innere Klassen sind Klassen, die innerhalb einer andern Klasse definiert werden und somit zu deren Namensraum gehören

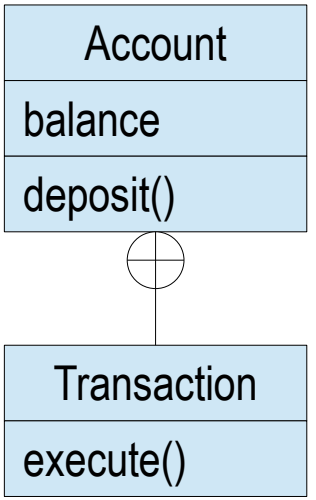
```
public class Account {
    private double balance;
    ...
    public static class BalanceComparator implements Comparator {
        public int compare(Object o1, Object o2) {
            Account a1 = (Account) o1; Account a2 = (Account) o2;
            return (int) (a1.balance - a2.balance);
        }
    }
}

Account[] accounts = ...
Util.sort(accounts, new Account.BalanceComparator());
```

Nicht-statische innere Klassen

Nicht-statische innere Klassen

- werden mit einem umgebenden Objekt instanziiert
- können auf die Felder und Methoden der umgebenden Klasse zugreifen

```
public class Account {
    private Transaction tx = new Transaction();
    private double balance;
    ...
    public void deposit(int amount) {
        tx.execute(Transaction.DEPOSIT, amount);
    }
    private class Transaction {
        static final int DEPOSIT = 1, WITHDRAWAL = -1;
        void execute(int type, double amount) { balance += type*amount; }
    }
}
```



Anonyme Klassen

Anonyme Klassen

- haben keinen Namen und werden gerade dort definiert, wo sie gebraucht werden
- können auf lokale Variablen zugreifen, die final sind

```
Account[] accounts = ...
final boolean ascending = ...

Util.sort(accounts, new Comparator() {
    public int compare(Object o1, Object o2) {
        Account a1 = (Account) o1; Account a2 = (Account) o2;
        return ascending ? (int) (a1.getBalance() - a2.getBalance()) :
                           (int) (a2.getBalance() - a1.getBalance());
    }
});
```

Standardbibliothek

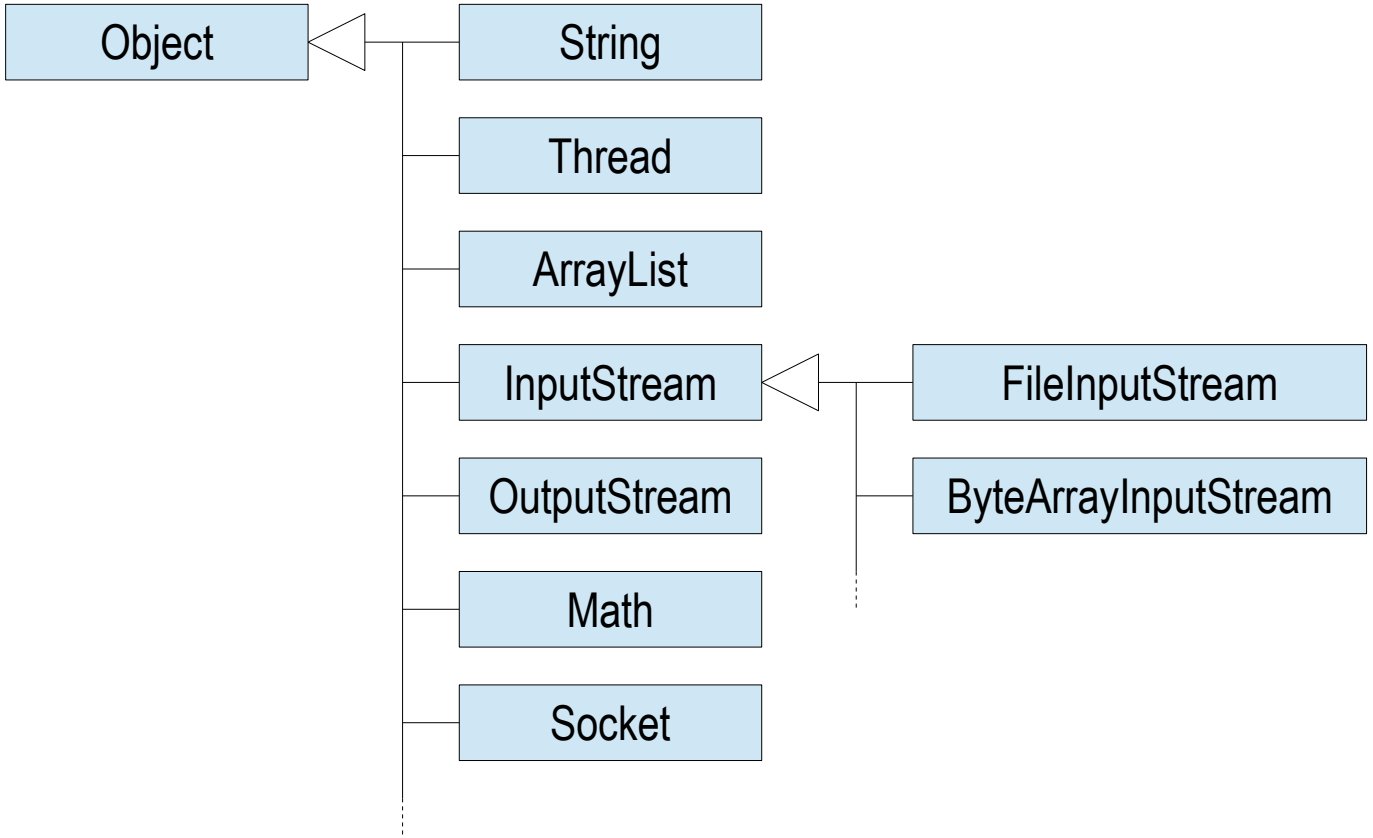
Packages

- Die Standard-Bibliothek umfasst über 4000 Klassen
- Kernklassen sind im Package java, speziellere im Package javax enthalten

Package	Inhalt
java.lang	Basisklassen (z.B. Object, String, Thread)
java.util	Hilfsklassen wie Collections
java.io, java.nio	Ein- und Ausgabe
java.math	Arithmetik langer Zahlen
java.net	Netzwerkkommunikation
java.security	Kryptographie
java.sql	Anbindung relationaler Datenbanken
java.text	Internationalisierung
java.time	Zeit- und Datumswerte
java.awt	Grafische Benutzeroberflächen

Klassenhierarchie

- In Java gibt es keine Mehrfachvererbung und alle Klassen sind implizit von der Klasse Object abgeleitet, die Klassenhierarchie ist somit ein Baum



Klasse Object

- Die Methoden der Klasse Object definieren ein gemeinsames Verhalten aller Objekte:
  - toString() erzeugt eine Stringdarstellung eines Objekts
  - equals() vergleicht zwei Objekte
  - hashCode() erzeugt einen Hashcode eines Objekts
  - clone() kopiert ein Objekt
  - finalize() wird aufgerufen, bevor ein Objekt zerstört wird
  - notify(), notifyAll() und wait() dienen der Thread-Synchronisation
  - getClass() gibt die Klasse eines Objekts zurück

Darstellung von Objekten als Strings

- Um eine Darstellung von Objekten als Strings zu erzeugen, kann die Methode toString() überschrieben werden
- Diese wird implizit von der Methode String.valueOf() und dem Konkatenationsoperator aufgerufen

```
public class Account {
    private int nr;
    private String pin;
    private double balance;
    ...
    @Override
    public String toString() {
        return "Account {nr=" + nr + ", balance=" + balance + "}";
    }
}

Account a = new Account(...);
System.out.println(a + " created");
```

Vergleichen von Objekten

- Um den Zustand von Objekten vergleichen zu können, muss die Methode equals() überschrieben werden
- Diese Methode wird u.a. beim Suchen von Objekten in Listen verwendet

```
public class Account {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Account)) return false;
        Account a = (Account) obj;
        return a.nr == nr;
    }
}

Account a1 = new Account(...);
Account a2 = new Account(...);
if (a1.equals(a2)) ...
```

Hashcode von Objekten

- Um Objekte effizienter vergleichen zu können, kann die Methode hashCode() überschrieben werden (wobei der Hashcode gleicher Objekte gleich sein muss)
- Diese Methode wird u.a. zum Auffinden von Objekten in Hashtabellen verwendet

```
public class Account {
    ...
    @Override
    public int hashCode() {
       return nr;
    }
}

Account a1 = new Account(...);
Account a2 = new Account(...);
if (a1.hashCode() == a2.hashCode())
    if (a1.equals(a2)) ...
```

Exception-Handling

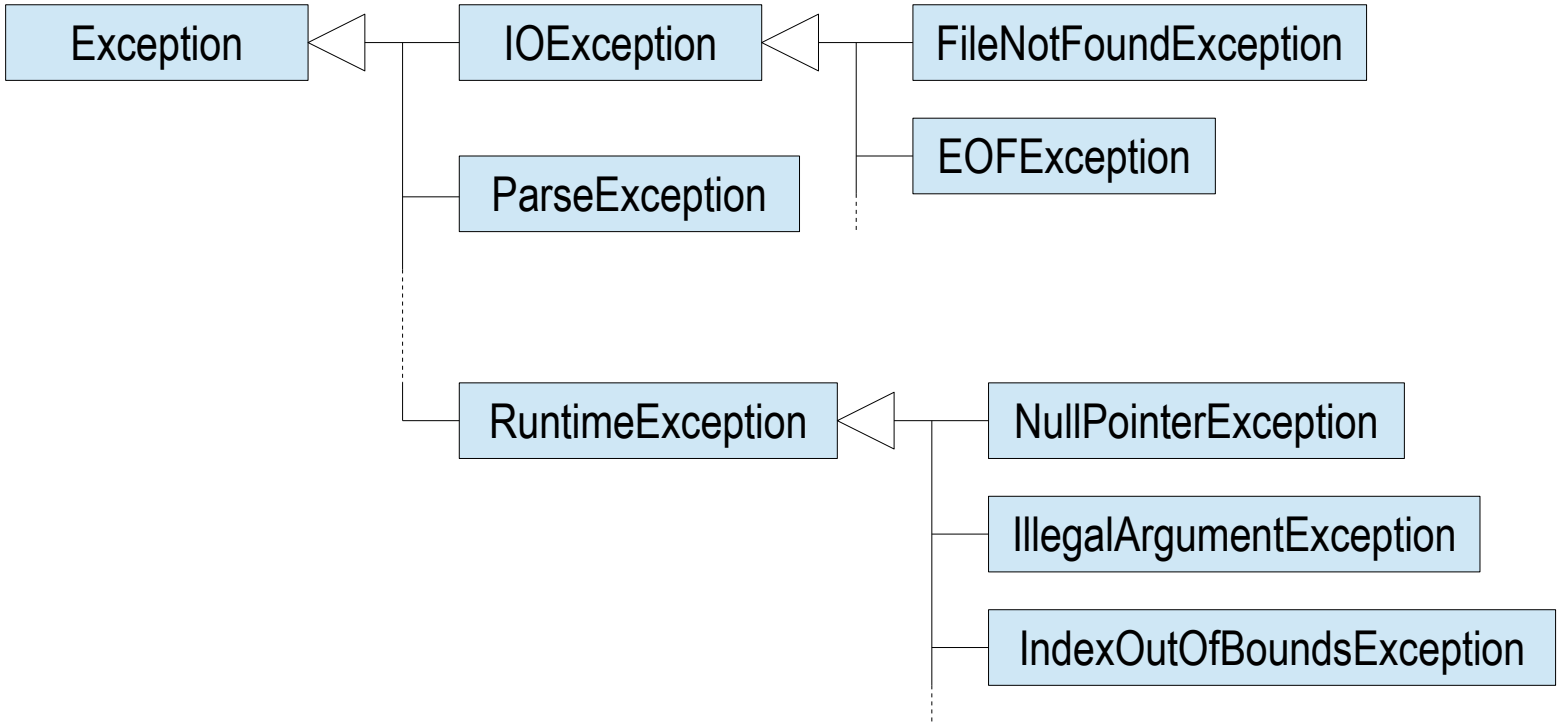
Einführung

Exceptions

- repräsentieren Fehler und ermöglichen, den normalen Programmablauf von Fehlersituationen zu trennen
- enthalten Informationen über den aufgetretenen Fehler
- werden geworfen und im Call Stack des Programms automatisch nach oben propagiert
- können irgendwo abgefangen und behandelt werden

Exception-Typen

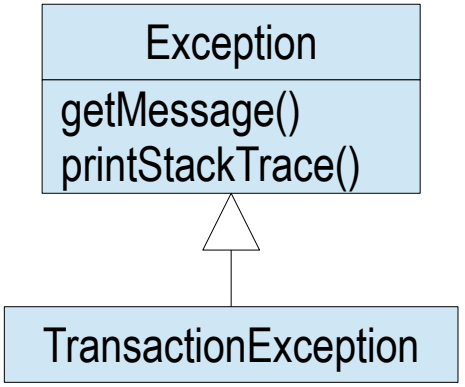
- Die Information über den aufgetretenen Fehler besteht primär aus dem Typ der geworfenen Exception
- Es gibt in der Java-Klassenbibliothek eine grosse Anzahl vordefinierter Exception-Klassen



Definieren einer Exception

- Exception-Klassen werden von der Standardklasse Exception abgeleitet und repräsentieren Fehlerkategorien
- Dem Basiskonstruktor kann eine Fehlermeldung übergeben und mit der Methode getMessage wieder abgefragt werden
- Detailinformationen über den aufgetretenen Fehler können in zusätzlichen Feldern enthalten sein

```
public class TransactionException extends Exception {
    public TransactionException(String message) {
        super(message);
    }
    ...
}
```



Werfen einer Exception

- Mittels throw kann in einer Methode eine Exception geworfen werden, wodurch die Methode unmittelbar verlassen wird
- Exceptions müssen in der throws-Klausel der Methode deklariert werden

```
public class Account {
    private int nr;
    private String pin;
    private double balance;
    ...
    public void withdraw(double amount) throws TransactionException {
        if (amount > balance)
            throw new TransactionException("Insufficient funds");
        balance -= amount;
    }
}
```

Abfangen einer Exception

- Eine Exception kann mit einem try-catch-Statement abgefangen und behandelt werden
- Tritt im try-Block eine Exception auf, so wird unmittelbar in den catch-Block verzweigt, andernfalls wird der catch-Block übersprungen

```
public class Bank {
    ...
    public void withdraw(int nr, String pin, double amount)
        try {
            Account account = findAccount(nr);
            account.checkPin(pin);
            account.withdraw(amount);
        }
        catch (TransactionException e) {
             System.err.println("Error: " + e.getMessage());
        }
    }
}
```

Abfangen verschiedener Exceptions

- Verschiedene Exceptions können mit mehreren catch-Blöcken unterschiedlich behandelt werden
- Es wird der erste catch-Block ausgeführt, dessen Parametertyp zur geworfenen Exception passt

```
public class Bank {
    ...
    public void withdraw(int nr, String pin, double amount)
        try {
            Account account = findAccount(nr);
            account.checkPin(pin);
            account.withdraw(amount);
        }
        catch (CredentialsException e) { ... }
        catch (TransactionException e) { ... }
        catch (Exception e) { ... }                    // fallback
    }
}
```

Weitergeben einer Exception

- Nicht abgefangene Exceptions werden im Callstack automatisch nach oben propagiert, müssen aber deklariert werden
- Wird eine Exception nirgends abgefangen, so wird der Stack Trace der Exception ausgegeben und das Programm terminiert

```
public class Bank {
    ...
    public void withdraw(int nr, String pin, double amount)
            throws CredentialsException, TransactionException {
        Account account = findAccount(nr);
        account.checkPin(pin);
        account.withdraw(amount);
    }
}
```

Finally-Block

- Oft müssen nach einer Fehlerbehandlung Aufräumarbeiten gemacht werden (z.B. Ressourcen freigeben)
- Das try-catch-Statement hat einen optionalen finally-Block, der in jedem Fall am Ende ausgeführt wird

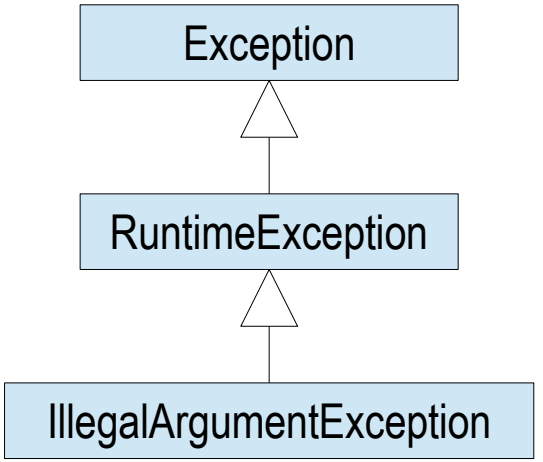
```
public class Bank {
    ...
    public void withdraw(int nr, String pin, double amount)
        try {
            Account account = findAccount(nr);
            account.checkPin(pin);
            account.withdraw(amount);
        }
        catch (CredentialsException e) { ... }
        catch (TransactionException e) { ... }
        finally { log("done"); }
    }
}
```

Unchecked Exceptions

Unchecked Exceptions

- repräsentieren unerwartete System- oder Programmierfehler
- sind von der Klasse RuntimeException abgeleitet
- müssen weder deklariert noch abgefangen werden

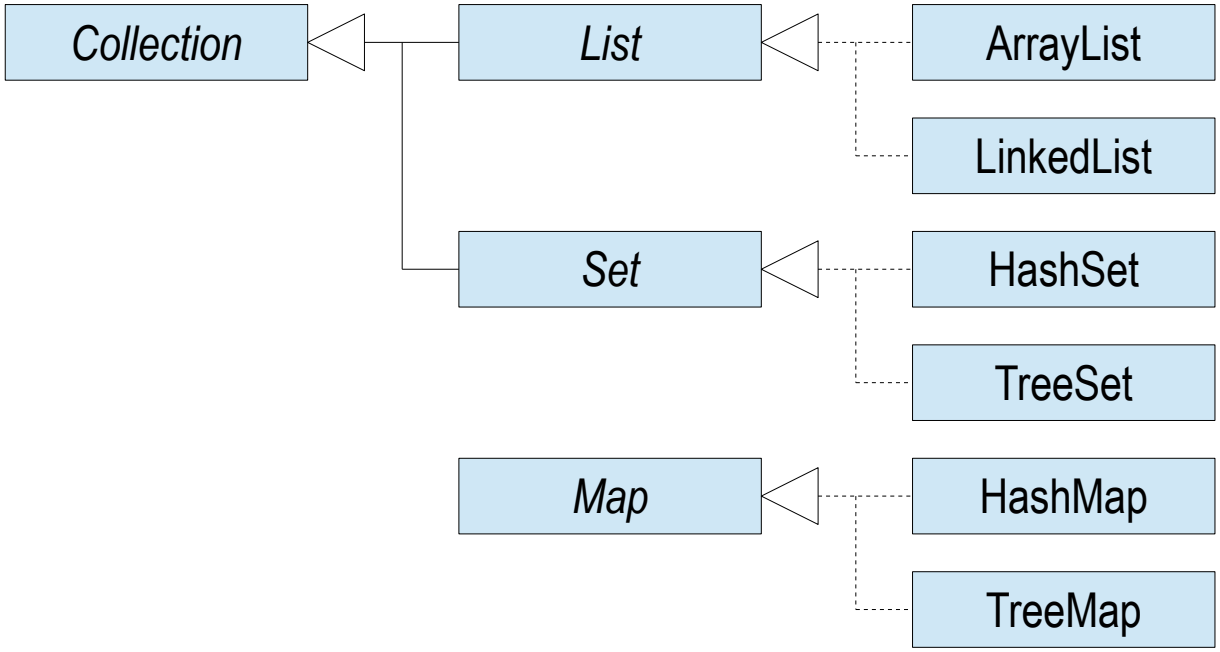
```
public class Account {
    private int nr;
    private String pin;
    private double balance;
    ...
    public Account(int nr, String pin) {
        if (pin == null)
            throw new IllegalArgumentException();
        ...
    }
}
```



Collections

Übersicht

- Collections erlauben mehrere Objekte zu verwalten
- Im Gegensatz zu Arrays haben Collections keine feste Grösse und bieten zahlreiche Methoden zur Manipulation ihrer Elemente
- Das Collection-Framework besteht aus mehreren Interfaces und verschiedenen Implementierungen



Listen

- Elemente bleiben in der Reihenfolge, mit der sie hinzugefügt werden
- Mit dem Index kann direkt auf die Elemente zugegriffen werden
- Listen werden mithilfe eines Arrays oder als verkettete Liste implementiert

```
List cities = new ArrayList();
cities.add("London");
cities.add("Paris");
cities.add("London");
cities.add("New York");
...
for (int i = 0; i < cities.size(); i++) {
    String city = (String) cities.get(i);
    System.out.println(city);
}
accounts.remove(0);
```

Sets

- Sets enthalten keine Duplikate, und die Elemente haben keine bestimmte Reihenfolge (ausser bei TreeSets)
- Um auf die Elemente zuzugreifen, muss mit einer for-each-Schlaufe über das Set iteriert werden
- Sets werden mithilfe von Maps implementiert

```
Set cities = new HashSet();
cities.add("London");
cities.add("Paris");
cities.add("London");
cities.add("New York");
...
for (Object city : cities) {
    System.out.println(city);
}
cities.remove("London");
```

Maps

- Maps bilden Objekte (Schlüssel) auf andere Objekte (Werte) ab
- HashMaps verwenden den Hashcode der Schlüsselobjekte, um die Wertobjekte effizient zu finden
- Die Einträge von TreeMaps sind nach den Schlüsseln geordnet

```
Map population = new HashMap();
population.put("London", 8_538_689);
population.put("Paris", 2_240_621);
population.put("London", 8_538_694);
population.put("New York", 8_491_079);
...
for (Object city : population.keySet()) {
    System.out.println(city + ": " + population.get(city));
}
population.remove("London");
```

Iteratoren

- Alle Collections implementieren das Interface Iterable und stellen somit einen Iterator zur Verfügung
- Iteratoren sind vor allem nützlich, wenn während einer Iteration Elemente aus der Collection entfernt werden müssen

```
List accounts = new LinkedList();
...
Iterator iter = accounts.iterator();
while (iter.hasNext()) {
    Account a = (Account) iter.next();
    if (a.getBalance() == 0)
        iter.remove();
}
```

Generics

Typisierte Collections

- Collections können mit einem Typparameter versehen werden
- Es können dann nur Objekte vom entsprechenden Typ hinzugefügt werden (Typsicherheit)
- Beim Lesen der Elemente sind keine Typumwandlungen notwendig

```
List<Account> accounts = new ArrayList<Account>();
accounts.add(new Account(...));
accounts.add("London");                    // Compiler-Fehler
...
for (int i = 0; i < accounts.size(); i++) {
    Account a = accounts.get(i);
    System.out.println(a);
}
```

Typisierte Iteratoren und Comparator

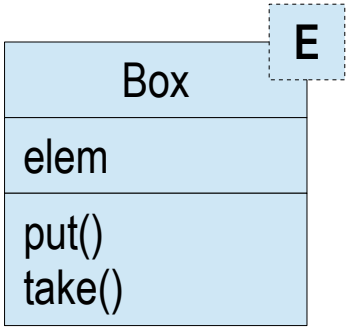
- Iteratoren typisierter Collections haben denselben Typparameter wie die Collection
- Zum Sortieren von Collections können typisierte Comparator verwendet werden

```
Iterator<Account> iter = accounts.iterator();
while (iter.hasNext(); ) {
    Account a = iter.next();
    System.out.println(a);
}

Collections.sort(accounts, new Comparator<Account>() {
    public int compare(Account a1, Account a2) {
        return a1.getBalance() - a2.getBalance();
    }
});
```

Generische Klassen

- Eine Klasse kann typisiert werden, indem sie mit einem formalen Typparameter versehen wird

```
public class Box<E> {
    private E elem;
    public void put(E elem) {
        if (this.elem != null) throw new IllegalStateException();
        this.elem = elem;
    }
    E take() {
        E temp = elem; elem = null; return temp;
    }
}

Box<Account> box = new Box<Account>();
box.put(new Account(...));
Account a = box.take();
```



Generische Methoden

- Generische Klassen können als Parameter- oder Rückgabetyp von Methoden verwendet werden
- Der entsprechende Typparameter muss in der Signatur der Methode aufgeführt werden

```
public class Util {
    public static <E> void reverse(List<E> elems) {
        for (int i = 0; i < elems.size() / 2; i++) {
            int j = elems.size() - i - 1;
            E e = elems.get(i); elems.set(i, list.get(j)); elems.set(j, e);
        }
    }
}

List<Account> accounts = ...
Util.reverse(accounts);
```

Beschränkte Typparameter

- Ein beschränkter Typparameter steht für Typen, die eine gemeinsame Basisklasse haben

```
public class Util {
    public static <E extends Number> double sum(List<E> elems) {
        double sum = 0;
        for (E elem : elems) {
            sum += elem.doubleValue();
        }
        return sum;
    }
}

List<Integer> integers = ...
Util.sum(integers);
```

Generics und Vererbung

- Es gibt keine Typbeziehung zwischen den Instanziierungen einer typisierten Collection, selbst wenn die Typparameter kompatibel sind
- Der Grund liegt darin, dass nach einer Zuweisung die Typsicherheit nicht mehr gewährleistet wäre

```
List<Integer> integers = new ArrayList<>();
integers.add(42);
...
List<Number> numbers = integers;          // Compiler-Fehler
numbers.add(3.14159);
integers.get(1);
```

Wildcard-Typen

- Ein Wildcard-Typ ist eine Collection, deren Elemente von unbekanntem Typ sind
- Jede Instanziierung der entsprechenden Collection ist typkompatibel zum Wildcard-Typ
- Wildcard-Typen dürfen keine Elemente hinzugefügt werden

```
List<Integer> integers = new ArrayList<>();
integers.add(42);
...
List<?> elems = integers;
Object elem = elems.get(0);
elems.add(3.14159);                    // Compiler-Fehler
```

Ein- und Ausgabe

Dateien und Pfade

- Die Klasse File repräsentiert Dateipfade und stellt u.a. folgende Methoden zur Verfügung:
  - exists() gibt an, ob eine Datei mit dem Pfad existiert
  - length() gibt die Grösse der Datei zurück
  - isFile(), isDirectory() geben den Typ der Datei an
  - canRead(), canWrite(), canExecute() geben die Zugriffsrechte an
  - createNewFile() erzeugt eine neue Datei
  - mkdir() erzeugt ein Verzeichnis
  - delete() löscht die Datei

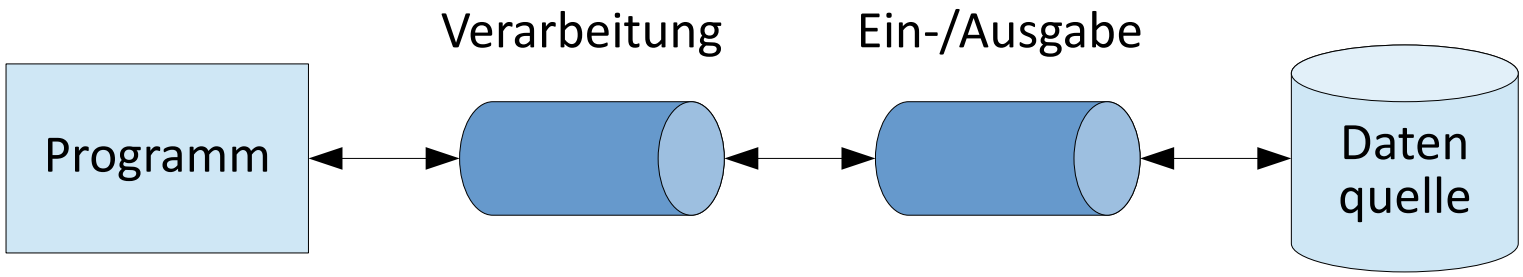
Verzeichnisse

- Die Methode listFiles() der Klasse File erlaubt, die Dateien eines Verzeichnis abzufragen
- Mit einem FileFilter können die Dateien zudem gefiltert werden

```
File dir = new File(dirname);
File[] files = dir.listFiles(new FileFilter() {
    public boolean accept(File file) { return file.isFile(); }
});
for (File file : files) {
    System.out.println(file.getName() + " (" + file.length() + " bytes)");
}
```

Streams

- Die Ein- und Ausgabe erfolgt über sogenannte Streams
- Streams erlauben das iterative Lesen und Schreiben von Daten
- Streams können zu einer Pipeline verkettet werden
- Es gibt Streams zur eigentlichen Ein-/Ausgabe bzw. Streams zur Verarbeitung von Daten



Ein-/Ausgabe-Streams

Je nach Art der zu lesenden bzw. schreibenden Daten wird zwischen Byte- und Character-Streams unterschieden

Datenquelle	Byte-Streams	Character-Streams
File	FileInputStream FileOutputStream	FileReader FileWriter
Array	ByteArrayInputStream ByteArrayOutputStream	CharArrayReader CharArrayWriter
String		StringReader StringWriter
Pipe	PipedInputStream PipedOutputStream	PipedReader PipedWriter

Verarbeitungs-Streams

Verarbeitung	Byte-Streams	Character-Streams
Umwandlung von Bytes in Characters		InputStreamReader OutputStreamWriter
Umwandlung primitiver Typen in Bytes	DataInputStream DataOutputStream	
Umwandlung von Objekten in Bytes	ObjectInputStream ObjectOutputStream	
Pufferung von Daten	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Filterung von Daten	FilterInputStream FilterOutputStream	FilterReader FilterWriter
Formatierung	PrintStream	PrintWriter

Schliessen von Streams

- Streams sollten geschlossen werden, wenn sie nicht mehr gebraucht werden
- Da Streams das Interface AutoCloseable implementieren, kann dazu das try-with-resources-Statement verwendet werden

```
try (FileReader reader = new FileReader(name)) {
    int c = reader.read();
    ...
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

Lesen und Schreiben von Binärdaten

- Ein InputStream erlaubt das Lesen, ein OutputStream das Schreiben von Binärdaten

```
try (FileInputStream in = new FileInputStream(sourcename);
     FileOutputStream out = new FileOutputStream(targetname)) {

    byte[] buffer = new byte[1024];
    while (true) {
        int nbytes = in.read(buffer);
        if (nbytes == -1) break;
        out.write(buffer, 0, nbytes);
    }
}
```

Lesen und Schreiben von Textdaten

- Ein Scanner erlaubt das zeilenweise Lesen, ein PrintWriter das zeilenweise Schreiben von Textdaten
- Beim Lesen und Scheiben von Textdaten muss ein Zeichensatz angegeben werden

```
try (Scanner in = new Scanner(System.in);
     PrintWriter out = new PrintWriter(new OutputStreamWriter(
        new FileOutputStream(filename), "UTF-8"))) {

   while (true) {
      String line = in.nextLine();
      if (line.isEmpty()) break;
      out.println(line);
   }
}
```

Objekt-Serialisierung

- Objekt-Serialisierung ist ein Mechanismus zur Umwandlung von Objekten oder ganzen Objektbäumen in Byte-Arrays und umgekehrt
- Damit die Objekte einer Klasse serialisiert werden können, muss die Klasse das leere Interface Serializable implementieren
- Mit dem Modifier transient können einzelne Felder von der Serialisierung ausgeschlossen werden

```
public class Message implements Serializable {
    private Date date = new Date();
    private String text;

    public Message(String text) { this.text = text; }
    public Date getDate() { return date; }
    public String getText() { return text; }
}
```

Lesen und Schreiben von Objekten

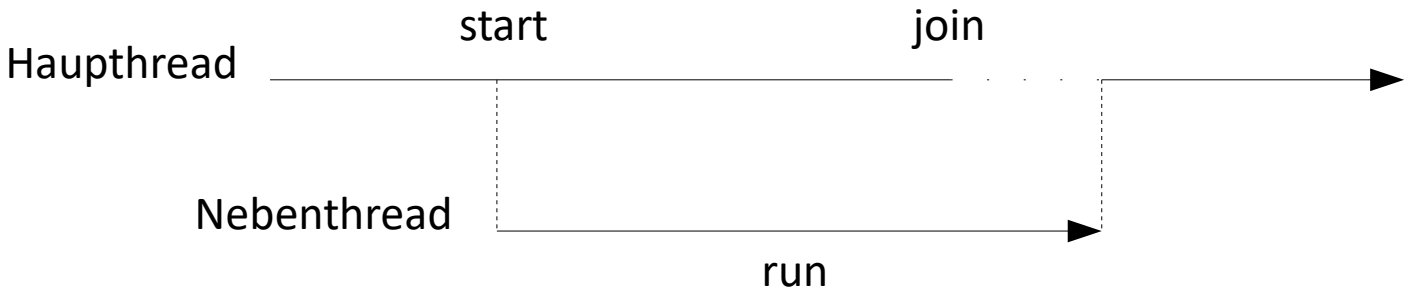
- Die Klassen ObjectInputStream und ObjectOutputStream erlauben das Lesen und Schreiben von Objekten

```
try (Scanner scanner = new Scanner(System.in);
      ObjectOutputStream out = new ObjectOutputStream(
          new FileOutputStream(filename))) {
    Message msg = new Message(scanner.nextLine());
    out.writeObject(msg);
}

try (ObjectInputStream in = new ObjectInputStream(
          new FileInputStream(filename))) {
    Message msg = (Message) in.readObject();
    System.out.println(msg.getDate() + "\n" + msg.getText());
}
```

Threads

Einführung

- Threads sind parallele Ablaufeinheiten innerhalb eines Prozesses und werden u.a. eingesetzt bei
  - Ein-/Ausgabe-Operationen, die blockieren können
  - der Behandlung externer Ereignisse
  - der Ausführung periodischer Aufgaben

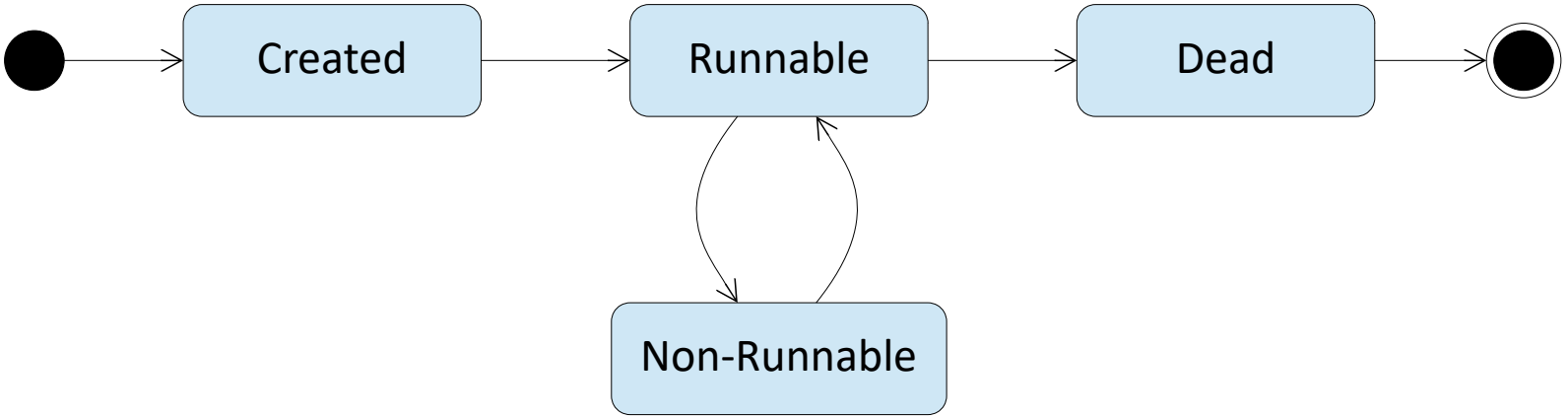


Klasse Thread

- In Java werden Threads durch Objekte der Standard-Klasse Thread repräsentiert
- Ein Thread-Objekt enthält in der run()-Methode den Code, der im zugehörigen Thread ausgeführt werden soll
- Über das Thread-Objekt kann der Thread gesteuert werden:
  - start() startet den Thread
  - join() wartet auf die Terminierung des Threads
  - interrupt() setzt ein Flag, um den Thread vorzeitig zu beenden
  - setPriority() bestimmt die Priorität des Thread
- Statische Methoden beziehen sich auf den ausführenden Thread:
  - currentThread() gibt das Thread-Objekt zurück
  - sleep() unterbricht den Thread für eine bestimmte Zeit
  - interrupted() prüft das Interrupted-Flag

Thread-Zustände

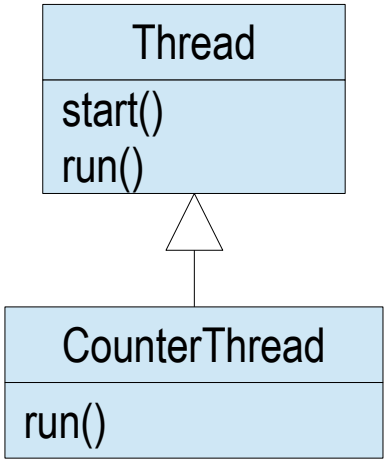
- Ein Thread durchläuft verschiedene Zustände
  - Created: das Thread-Objekt existiert, aber der zugehörige Thread wurde noch nicht gestartet
  - Runnable: der Thread wurde gestartet und ist lauffähig
  - Non-Runnable: der Thread ist nicht lauffähig, weil er z.B. in einer Ein-/Ausgabe-Operation blockiert ist
  - Dead: der Thread hat terminiert



Erzeugen eines Threads (Variante 1)

- Um einen Thread zu erzeugen, wird von der Standardklasse Thread eine Klasse abgeleitet, die run()-Methode überschrieben, die Klasse instanziiert und die start()-Methode aufgerufen
- Das Runtime-System erzeugt dann einen Thread, welcher die run()-Methode des Thread-Objekts ausführt

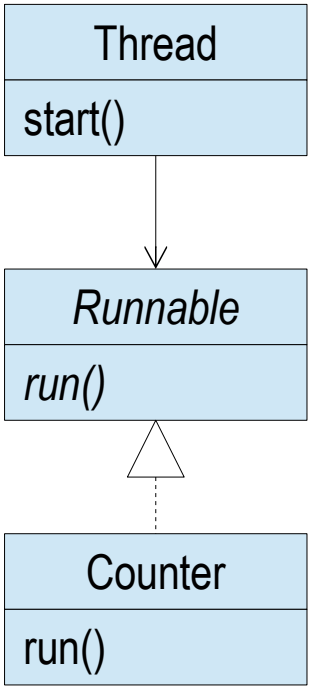
```
public class CounterThread extends Thread {
    public void run() {
        for (int i = 1; i <= 100; i++) System.out.println(i);
    }
}

Thread thread = new CounterThread();
thread.start();
System.out.println("Thread started");
...
```



Erzeugen eines Threads (Variante 2)

- Um ein Thread-Objekt zu erzeugen, kann auch die Klasse Thread instanziiert und dem Konstruktor ein Objekt übergeben werden, welches das Interface Runnable implementiert

```
public class Counter implements Runnable {
    public void run() {
        for (int i = 1; i <= 100; i++) System.out.println(i);
    }
}

Counter counter = new Counter();
Thread thread = new Thread(counter);
thread.start();
System.out.println("Thread started");
...
```



Parameterübergabe

- Da die run()-Methode keine Parameter hat, müssen solche als Felder des Thread-Objekts übergeben werden
- Mittels join() kann auf die Terminierung eines Threads gewartet werden

```
public class SummationThread extends Thread {
    private int max, sum = 0;
    public SummationThread(int max) { this.max = max; }
    public int getSum() { return sum; }
    public void run() {
        for (int n = 1; n <= max; n++) sum += n;
    }
}
SummationThread thread = new SummationThread(100);
thread.start();
...
thread.join();
System.out.println(thread.getSum());
```

Terminieren eines Threads

- Ein Thread terminiert, wenn er die run()-Methode verlässt
- Um einen Thread vorzeitig zu beenden, kann mittels interrupt() ein Flag gesetzt werden, welches der Thread mittels interrupted() periodisch prüfen kann
- Wenn der Thread blockiert ist, wird eine InterruptedException geworfen

```
public class InfiniteCounterThread extends Thread {
    public void run() {
        int i = 1;
        while (!Thread.interrupted()) {
            try { Thread.sleep(1000); } catch(InterruptedException e) { break; }
            System.out.println(i++);
        }
    }
}
Thread thread = new InfiniteCounterThread();
thread.start(); ... thread.interrupt();
```

Synchronisieren von Methoden

- Um Zugriffskonflikte auf die Felder eines Objekts zu vermeiden, können seine Methoden als synchronized markiert werden
- Für dasselbe Objekt kann dann immer nur ein Thread gleichzeitig eine der entsprechenden Methoden ausführen (exklusiver Ausschluss)

```
public class Buffer {
    private Object[] elems = new Object[...];
    private int size = 0;

    public synchronized void put(Object obj) {
        for (int i = size++; i > 0; i--) { elems[i] = elems[i - 1]; }
        elems[0] = obj;
    }
    public synchronized Object get() {
        return elems[--size];
    }
}
```

Warten auf eine Bedingung

- Mittels wait() kann ein Thread innerhalb einer synchronisierten Methode auf eine Bedingung warten, bis diese von einem andern Thread mittels notify() im gleichen Objekt signalisiert wird
- Warten mehrere Threads, so können mittels notifyAll() alle Threads geweckt werden

```
public class SynchronizedBuffer {
    ...
    public synchronized void put(Object obj) {
        ...
        notify();
    }
    public synchronized Object get() throws InterruptedException {
        while (size == 0) wait();
        return elems[--size];
    }
}
```