Programmieren in Java

Kapitel 2

2. Grundlegende Eigenschaften von Java

2.1. Programmiersprache und Ausführungsplattform

2.2. Klassen und Programmstruktur

2.3. Programm-Erzeugung und –Start

2.4. Packages

2.5. Standard-Bibliothek

2.6. Datentypen

2.7. Strings

2.8. Arrays und Array-Listen

2.9. Die Klasse `Object`

2.10. Aufzählungstypen

2.11. Generische Programmierung

Java als Programmiersprache und Ausführungsplattform (1)

- **Java ist mehr als eine Programmiersprache**

  ◇ In einem ihrer früheren Artikel charakterisiert die Fa. SUN Java wie folgt :

  > *Java : A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language*

  ◇ Einige dieser Eigenschaften werden dadurch realisiert, dass Java auch eine **Plattform** zur **Ausführung** von Java-Programmen zur Verfügung stellt.
  → Java ist **sowohl** eine **Programmiersprache** als **auch** eine **Ausführungsplattform**.

  ◇ Mit Java untrennbar verbunden ist auch eine **Standardbibliothek**, die in ihrem Kernbereich (***Java Core API***) wichtige **Sprachkonzepte implementiert** und damit verwendbar macht. Zumindest dieser Kernbereich wird de facto als Bestandteil der Sprache betrachtet.

- **Elementare Charakteristika der Programmiersprache Java**

  ◇ **Einfachheit** (*simple*)
  Java ist unter Berücksichtigung des **KISS** (*Keep it Small and Simple*) **-Prinzips** in massgebenden Umfang aus der Sprache C++ entwickelt worden. → Syntax und Semantik von Java weisen eine **große Ähnlichkeit mit C++** auf. Wesentliche Bestandteile von Java sind damit C++-Programmierern **vertraut**, so dass Java für sie **leicht erlernbar** ist. Durch das Weglassen zahlreicher redundanter und fehleranfälliger Sprachelemente/-eigenschaften ist Java zudem **schlanker**, **überschaubarer** und **leichter anwendbarer** als C++. Gleichzeitig wird dadurch die **Zuverlässigkeit** der Sprache und der in ihr formulierten Programme **erhöht**.
  Zusätzlich sind **sinnvolle Konzepte** anderer objektorientierter Sprachen, die der Klarheit und Effizienz dienen, in Java integriert worden.

  ◇ **Objektorientiertheit** (*object-oriented*)
  Java ist eine **rein objekttorientierte** Sprache. Alle Funktionalität ist an Objekte bzw Klassen gebunden.
  Auch **Arrays** und **Strings** sind als **Klassen** implementiert.
  Alle Klassen sind – direkt oder indirekt – von der elementaren Basisklasse `Object` abgeleitet, die eine allgemeine Grundfunktionalität zur Verfügung stellt.

  ◇ **Unterstützung verteilter Anwendungen** (*distributed*)
  Java und seine Standardbibliothek stellen durch entsprechende Klassen eine **effiziente High-Level-Unterstützung** zur **Netzwerkkommunikation** zur Verfügung.
  Das **RMI** (*Remote Method Invocation*) **API** ermöglicht den Aufruf von Methoden entfernter Objekte in genau der gleichen Art und Weise wie von lokalen Objekten.
  Ergänzt werden diese Möglichkeiten durch die Fähigkeit des **dynamischen Ladens von Klassen** (und damit auch die Ausführung ihres Codes), nicht nur vom lokalen Rechner sondern auch über ein Netzwerk von entfernten Rechnern.

  ◇ **Robustheit** (*robust*)
  Java ist u.a. mit dem Ziel entworfen worden, sehr **zuverlässige** und **robuste** Software zu entwickeln. Viele – aber natürlich nicht alle – Arten von Programmierfehlern werden durch Java prinzipiell verhindert :

  - ▻ Das **Fehlen eines Preprozessors** und des **Überladens von Operatoren** verhindert, dass tatsächlich anderer Code als im Quellprogramm formuliert, ausgeführt wird.
  - ▻ Java ist eine **streng typsichere Sprache** ("*strongly typed language*"). Durch den Compiler und zur Laufzeit werden ausgedehnte Überprüfungen der richtigen Verwendung des Typ-Systems durchgeführt (Funktionsparameter, Casts)
  - ▻ Das **Fehlen von Pointern und Pointerarithmetik** verhindert viele typische C/C++-Speicherzugriffsfehler
  - ▻ **Array- und Stringgrenzen** werden **beim Zugriff überprüft**. Dies verhindert entsprechende Speicher-Überlauf- und Überschreibungsfehler
  - ▻ Die in Java implementierte **automatische Freigabe** von **dynamisch allokierten** aber nicht mehr benötigten **Speicher** (***Automatic Garbage Collection***) erleichtert die Programmierung und verhindert zahlreiche mit der Speicherallokation und Speicherdeallokation zusammenhängende Fehler.
  - ▻ Das Java-Laufzeitsystem und die Klassen der Standardbibliothek machen **ausgiebigen Gebrauch** vom ***Exception Handling***. Dies erleichtert das Erkennen und Behandeln diverser Laufzeitfehler.

Java als Programmiersprache und Ausführungsplattform (2)

• Elementare Charakteristika der Programmiersprache Java, Forts.

◇ **Sicherheit** (*secure*)
Java stellt **mehrere Sicherheitsschichten** zur Verfügung, die es ermöglichen, Code aus einem a-priori unsicheren Netzwerk (wie z.B. das Internet) herunterzuladen und mit großer – wenn auch nicht 100%-iger Sicherheit – auszuführen (z.B. Applets, dynamisches Laden von Klassen).
Im wesentlichen handelt es sich hierbei um :

- ▻ Java-Programme können weder **direkt zum Speicher zugreifen** (keine Pointer), noch **Array- oder Stringgrenzen verletzen**. Dies stellt einen wesentlichen Schutz gegen "bösartigen" Code dar.
- ▻ Alle Java-Klassen werden **beim Laden** auf **Richtigkeit und Zuverlässigkeit überprüft** ("***byte-code verification***", z.B. Überprüfung auf Stack-Über- oder Unterlauf, illegale Byte-Codes usw)
- ▻ Der Klassenlader sucht immer zuerst nach dem lokalen Vorhandensein einer Klasse. Dies stellt einen gewissen Schutz gegen das "Unterschieben" einer manipulierten Klasse ("*class spoofing*") dar
- ▻ Eine weitere Sicherheitsschicht stellt das "**Sandkasten-Modell**" ("*sandbox model*") dar, das unsicherem Code sicherheitsrelevante Zugriffe verweigert.
  Z.B. ist Applets jeglicher Zugriff zum lokalen Dateisystem verboten.
  Weiterhin benötigen alle Bibliotheksklassen, die sicherheitsrelevante Operationen ausführen (z.B. Dateizugriff oder Netzwerkzugriff), ein `SecurityManager`-Objekt, das sie vor einem entsprechenden Zugriff um Erlaubnis fragen. Die von diesem freizugebenden Zugriffe werden durch ein ***Security Policy File*** festgelegt.

◇ **Unterstützung von Multithreading** (multithreaded)
Java enthält Sprachmittel und Bibliotheksklassen zur einfachen Realisierung von Threads.
Dadurch ist es in Java sehr leicht, Multithreaded-Anwendungen zu realisieren.

• Elementare Charakteristika der Ausführungsplattform Java

◇ **Interpretierte Programmausführung** (interpreted)
Der Java-Compiler erzeugt keinen Maschinencode sondern sogenannten **Byte-Code**, der zur Ausführung von einem **Interpreter**, der **Java Virtuellen Maschine** (***Java Virtual Machine***, **JVM**), abgearbeitet wird. Die JVM bildet die von Java zur Verfügung gestellte **Ausführungsplattform**. Zur Abarbeitung eines Java-Programms muß diese Ausführungsplattform, also die JVM, gestartet werden. Dieser wird der – in einer Datei enthaltene – Byte-Code der Start-Klasse als Parameter übergeben. Die Byte-Codes der weiteren Klassen des Programm werden jeweils bei Bedarf von der JVM geladen. Der Byte-Code kann als "Machinencode" der JVM aufgefasst werden.

◇ **Architektur-Neutralität und Portabilität** (architecture neutral and portable)
Ein Java-Programm ist auf jedem System lauffähig, das eine Java Virtuelle Maschine zur Verfügung stellt.
Der vom Compiler erzeugte **Byte-Code** ist architektur-neutral. Er enthält **keine implementierungsabhängigen Elemente**. Auch die Sprache **Java selbst** ist vollkommen **implementierungsunabhängig**. U.a. legt Java die **Größe** der **einfachen Datentypen** exakt fest. Damit ist – sogar ein übersetztes – Java-Programm **100% portabel**.
Suns Motto für Java lautet deswegen :

Write Once, Run Anywhere

◇ **Dynamisches Verhalten** (dynamic)
Jede Java-Klasse kann zu jeder Zeit in eine laufende Virtuelle Maschine geladen werden. Eine derart **dynamisch geladene Klasse** kann dann **dynamisch instantiiert** werden. Jede geladene Klasse wird durch ein Objekt der Klasse `Class` repräsentiert. Dies ermöglicht die Ermittlung von Informationen über eine Klasse zu Laufzeit (***Reflection API***)

◇ **Hohe Leistungsfähigkeit** (high-performance)
Interpretierter Code ist zwar prinzipiell immer langsamer als direkt ausgeführter Maschinencode, Sun unternimmt aber große Anstrengungen, die JVM so effizient und schnell wie möglich zu realisieren. Heutige JVMs enthalten "***just in time***" **Compiler** (**JIT**), die Byte-Code vor der ersten Ausführung in Maschinen-Code übersetzen. Ergänzt wird dieses Konzept durch adaptive Entscheidungsalgorithmen, die zur Laufzeit den Code bezüglich Leistungsengpässen analysieren und wenig verwendeten Code nicht übersetzen, laufzeit-kritischen dagegen bestmöglich optimieren ("**Hot Spot**" **JVM**).
Darüberhinaus kann besonders zeitkritischer Programm-Code in einer in Maschinen-Code direkt übersetzbaren Sprache formuliert, in nativen Maschinen-Code übersetzt und mittels des ***Java Native Interfaces*** (**JNI**) von der JVM aufgerufen werden.
All diese Maßnahmen ermöglichen es, dass Java-Programme bezüglich der Ausführungsgeschwindigkeit zunehmend mit compilierten C/C++-Programmen konkurieren können.

Klassen in Java - Einführung

• Grundlegendes

◇ **Klassen** sind die **elementaren Bestandteile** jedes Java-Programms.
Sämtlicher Code eines Java-Programms befindet sich innerhalb von Klassen.
Ein Programm kann aus beliebig vielen Klassen bestehen.

◇ Eine Klasse definiert die **Eigenschaften** (den Aufbau) und das **Verhalten** (die Funktionalität) der von ihr instanzierbaren Objekte. Sie ist aus **Klassenkomponenten** aufgebaut.
Klassenkomponenten können sein :
- **Datenkomponenten** (Membervariable) → Variablendefinitionen
  Die Gesamtheit der Datenkomponenten beschreibt den **Aufbau** und – mit ihren jeweiligen konkreten Werten in einem Objekt – den **Zustand** der Objekte (→ Objektvariable).
- **Funktionskomponenten** (Memberfunktionen) → Funktionsdefinitionen
  Diese legen das **Verhalten,** die Fähigkeiten der Objekte fest (→ Objektfunktionen).

◇ Eine Klasse kann auch Komponenten (sowohl Daten- als auch Funktionskomponenten) enthalten, die nicht objekt-spezifisch sind, sondern den Zustand und das Verhalten der Klasse selbst beschreiben.
Sie sind durch den Modifizierer `static` gekennzeichnet.
→ **statische Datenkomponenten** (Klassenvariable) bzw **statische Funktionskomponenten** (Klassenfunktionen)

◇ Zusätzlich kann eine Klassendefinition
- **Konstruktoren** enthalten

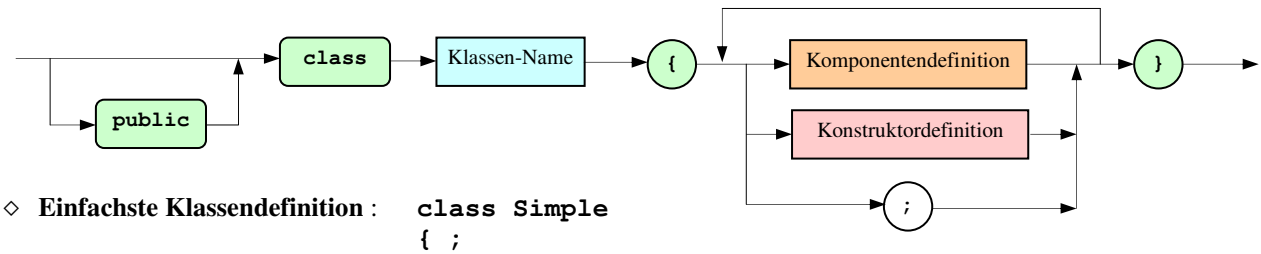
Konstruktoren sind spezielle klassenspezifische Funktionen, die bei der **Objekterzeugung** aufgerufen werden.
Sie tragen immer den Klassennamen und besitzen keinen Rückgabetyp (auch nicht `void`)
Sie werden in Java nicht zu den Klassenkomponenten gerechnet.
Ihre primäre Aufgabe besteht in der Initialisierung der Datenkomponenten des erzeugten Objekts.
Falls eine Klassendefinition keinen Konstruktor enthält, wird vom Compiler implizit ein Default-Konstruktor bereitgestellt.

◇ Eine Klassendefinition kann noch **weitere Bestandteile** enthalten (s. später, Kapitel 4)

◇ Für den Zugriff zu den Klassenkomponenten als auch zu einer Klasse selbst können **Zugriffsberechtigungen** festgelegt werden. Dies erfolgt durch die Angabe von **Zugriffs-Modifizierern**. Wird kein Zugriffs-Modifizierer angegeben, so besteht Zugriff nur von innerhalb des Packages, in dem die Klasse definiert ist.
Soll ein **Zugriff von überall her** möglich sein (**öffentlicher Zugriff**), so muß die Klasse bzw die entsprechende Komponente durch den vorangestellten Zugriffs-Modifizierer **`public`** gekennzeichnet werden.
Für Klassenkomponenten existieren noch die Zugriffs-Modifizierer **`private`** (Zugriff nur von innerhalb der Klasse) und **`protected`** (Zugriff beschränkt auf die Klasse selbst, abgeleitete Klassen und das Package) (→ **Kapselung** !)

• Vereinfachte Syntax der Klassendefinition

(vollständiges Syntax-Diagramm s. Kapitel 4)



◇ **Einfachste Klassendefinition** :

```
class Simple
{ ;
}
```

◇ **Ein etwas sinnvolleres Beispiel :**

```
class Uhr
{ private long actTime;          // Datenkomponente
  void setTime(long time)
  { actTime = time; }
  long tick()                    // Funktionskomponenten
  { ++actTime; }
  void displayClock()
  { /* ... */ };
}
```

Struktur von Java-Programmen (1)

- **Modulkonzept**

  ◇ Ein Java-Programm besteht **nur** aus **Klassen**.

  ◇ Jede **Klasse** muß in **einer einzigen Quelldatei** vollständig **definiert und implementiert** werden.
  Eine Aufteilung in Definitionsdatei (Headerdatei) und Implementierungsdatei wie in C++ existiert in Java nicht.
  Java-Quelldateien erhalten die Extension **`.java`**

  ◇ **Jede Klasse** ist in einer **eigenen Quelldatei** zu definieren.
  Ausnahme : eingebettete Klassen (*nested classes*).
  Der **Quelldatei-Hauptname** muß wie der (Haupt-)**Klassenname** lauten.
  Beispiel : Klasse `Welcome` → Quelldateiname : `Welcome.java`

  ◇ Eine Quelldatei ist die Übersetzungseinheit (Übersetzungs-Modul).
  Für **jede enthaltene Klasse** wird vom Compiler eine eigene **Byte-Code-Datei** erzeugt.
  Die **Byte-Code-Datei** für die (Haupt-)Klasse bekommt den **Hauptnamen** der Quellcode-Datei und die Extension **`.class`**
  Beispiel : Aus `Welcome.java` erzeugt der Compiler `Welcome.class`
  Die Hauptnamen der übrigen Byte-Code-Dateien sind aus den Namen der Haupt-Klasse und der eingebetteten Klassen zusammengesetzt.

- **Programmstruktur**

  ◇ Ein **Linken** der getrennt übersetzten Module (Klassen-Dateien) zu einer – ausführbaren – Programm-Datei **findet** in Java **nicht statt**.

  ◇ Ein Java-Programm ist damit **nicht** in **einer einzigen** Datei zusammengefasst.
  Vielmehr besteht es – entsprechend den im Programm eingesetzten Klassen – aus **einer oder mehreren Dateien**.

  ◇ Eine dieser Dateien enthält die **Start-Klasse** des Programms.

  ◇ Ausgehend von der Start-Klasse werden die **übrigen Klassen** des Programms **referiert**.

- **Programmarten**

  ◇ Es werden **zwei Java-Programmarten** unterschieden :

  ▻ **Applikations-Programme**

  ▻ **Applets**

  ◇ **Applikationsprogramme**
  "Normale" Java-Programme. Sie werden durch **direkten Start der Virtuellen Maschine** (JVM) ausgeführt.
  Hierfür ist dieser beim Start die **Start-Klasse** des Programms als **Parameter** zu übergeben

  ◇ **Applets**
  Diese Programme werden innerhalb eines **Java-fähigen Web-Browsers** (oder im Kontext eines anderen "*applet viewers*") ausgeführt. Ein derartiger Browser (bzw "*applet viewer*") enthält eine – gegebenenfalls über ein Plug-In eingebundene – JVM. Der Applet-Code befindet sich üblicherweise auf einem Server und wird durch ein **`<Applet>`**-Tag im HTML-Code einer Web-Seite referiert. Stößt der Browser beim Interpretieren des HTML-Codes auf ein derartiges `<Applet>`-Tag, startet er seine JVM. Diese lädt von der angegebenen URL den Byte-Code der Start-Klasse des Applets und führt ihn aus.
  Applets sind damit – i.a. kleine – Programme, die in einfacher Art und Weise über das Internet verteilt werden können. Da Applets prinzipiell unzuverlässigen Code enthalten können, unterliegt ihre Ausführung strengen Sicherheitsrestriktionen.
  Der Start und die Ausführung eines Applets unterscheidet sich aber erheblich von Start und Ausführung eines Applikations-Programms.

Struktur von Java-Programmen (2)

- **Start-Klasse eines Java-Applikations-Programms**

  ◇ Die der JVM beim Start zu übergebene Klasse muß – neben möglicherweise beliebig vielen anderen Methoden – eine **statische `main()`-Methode** besitzen.
  Diese `main()`-Methode muß öffentlich zugänglich (**`public`**) und vom Typ **`void`** sein.
  Als formalen Parameter muß sie ein **`String`-Array** besitzen.
  Sie stellt den **Startpunkt der Programmausführung** dar.

  ◇ **Beispiel für ein minimales Java-Programm** :

```
// Welcome.java

public class Welcome
{
  public static void main(String[] args)
  {
    System.out.println("Welcome to the world of Java");
  }
}
```

  ◇ I.a. ist ein Java-Programm **nicht klassenorientiert sondern objekttorientiert**. D.h. im Programm werden **Objekte** erzeugt, die miteinander **kommunizieren**.
  Die **Erzeugung des "ersten" Objekts** erfolgt dann in der `main()`-Methode der Start-Klasse :
  - ▻ Entweder durch **Instantiierung** einer **anderen Klasse** (Start-Klasse dient nur zum Programmstart)
  - ▻ oder durch **Instantiierung** der **eigenen Klasse** (Start-Klasse enthält auch spezifische Programmfunktionalität)

  ◇ Ausgehend vom ersten Objekt werden dann die weiteren Objekte erzeugt.

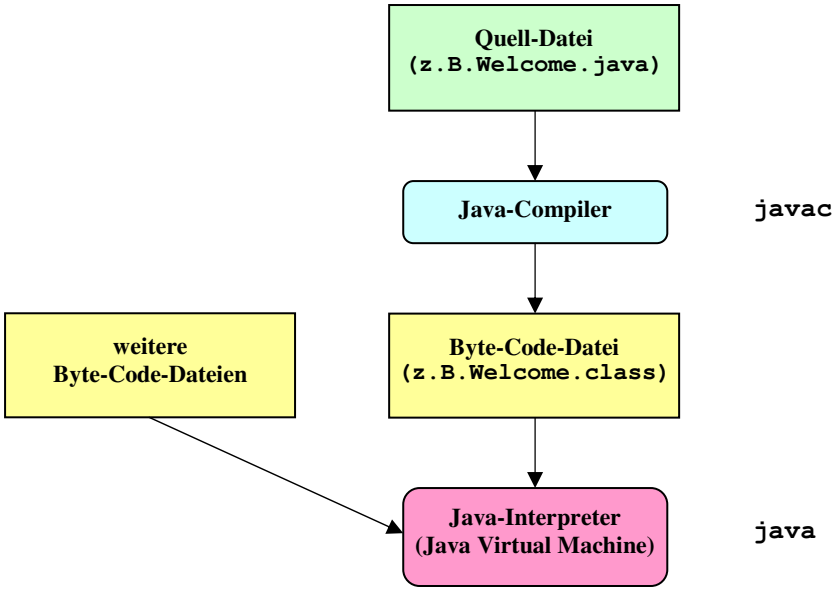
  ◇ **Beispiel für ein minimales Java-Programm mit Objekterzeugung** :

```
// Willkommen.java

public class Willkommen
{
  Willkommen()                                  // Konstruktor
  {
    // nur zur Demonstration
  }

  void begruessung()                            // Memberfunktion
  {
    System.out.println("Willkommen bei Java");
  }


  public static void main(String[] args)
  {
    Willkommen gruss = new Willkommen();        // Erzeugung eines Objekts
    grus.begruessung();
  }
}
```

  ◇ **Anmerkung** :
  Häufig wird auch bei Klassen, die in der späteren Verwendung nicht als Start-Klassen dienen sollen, eine statische `main()`-Methode vorgesehen. Diese Methode enthält dann i.a. Code, der das – weitgehend isolierte – Testen der Klasse ermöglicht.

Erzeugung und Start von Java-Programmen

- **Vom Quellprogramm zur Programmausführung**



- **Aufruf des Java-Compilers (Kommandozeilen-Tool `javac` des JDK)**

  ◇ Der Compiler sucht die zu übersetzende Quelldatei ausgehend vom **aktuellen Arbeitsdirectory**.
  Befindet sich die Datei in einem anderen Directory, ist ein **entsprechender Zugriffspfad** anzugeben.

  ◇ Der Name der zu übersetzenden Quelldatei ist **einschließlich** der **Extension** `.java` anzugeben.

  ◇ **Beispiel** : Quelldatei `Welcome.java` im Directory `E:\Java\Vorl`

  ```
  E:\Java\Vorl>javac Welcome.java
  ```

  ◇ Der Compiler erzeugt – bei Fehlerfreiheit – eine Byte-Code-Datei (hier : **`Welcome.class`**) im Verzeichnis der Quelldatei.

  ◇ Fehler werden zusammen mit der Zeilennummer und dem Quelldateinamen gemeldet.

  ◇ Werden in einer Quelldatei weitere Klassen referiert und findet der Compiler keine entsprechenden `.class`-Dateien, versucht er diese durch zusätzliches Übersetzen der entsprechenden `.java`-Dateien zu erzeugen.
  (Ausnahme : Klassen der Standardbibliothek)

- **Aufruf des Java-Interpreters (Kommandozeilen-Tool `java` des JDK)**

  ◇ Dem Java-Interpreter ist der **Klassenname** (nicht der Dateiname, d.h. keine Extension !) der Start-Klasse als Kommandozeilenparameter zu übergeben.

  ◇ Der Java-Interpreter sucht nach der **Byte-Code-Datei** der Start-Klasse im **aktuellen Directory** und – falls dort nicht vorhanden – in den Directories, die in der **Environment-Variablen `CLASSPATH`** enthalten sind.

  ◇ **Beispiel** : Byte-Code-Datei `Welcome.class` (mit Klasse `Welcome`) im Verzeichnis `E:\Java\Vorl`
  `CLASSPATH`-Variable ist nicht gesetzt.

  ```
  E:\Java\Vorl>java Welcome
  ```

  ◇ Werden von der Start-Klasse **weitere Klassen** benötigt, sucht der Interpreter die entsprechenden Byte-Code-Dateien im **aktuellen Directory** bzw in den Directories der **`CLASSPATH`-Variablen** (Ausnahme : Klassen der Standardbibliothek)

  ◇ Wird eine benötigte Byte-Code-Datei nicht gefunden, wird die **Exception `NoClassDefFoundError`** erzeugt.

Java-Archiv-(JAR-) Dateien (1)

• Grundsätzliches

◇ Das JAR-Dateiformat ermöglicht die **Zusammenfassung von mehreren Dateien** in einer einzigen Datei (→ **JAR-Archiv**). Es ist plattform-unabhängig und basiert auf dem ZIP-Dateiformat.

◇ Es dient im wesentlichen zur Zusammenfassung

- ▻ von mehreren `class`-Dateien in einem einzigen **Bibliotheks-Archiv**
- ▻ von den zu einer **Java-Applikation** (oder einem Applet) gehörenden Dateien (`class`-Dateien und sonstige Dateien, wie benötigte Daten-Dateien usw) zu einer einzigen Datei.

◇ Eine JAR-Datei enthält neben den zusammengefassten Dateien im allgemeinen auch **Meta-Informationen** über bestimmte Eigenschaften und die Verwendung des Archivs.
Diese befinden sich in Dateien, die in dem bei der Archiverzeugung – automatisch – hinzugefügten Directory **`META-INF`** angelegt werden. Die wichtigste dieser Meta-Informations-Dateien ist die Manifest-Datei **`MANIFEST.MF`**.
U.a. enthält diese den `CLASSPATH` für zu verwendende Bibliotheken (außer Standard-Bibliothek)
(Eintrag `Class-Path:`) sowie die Startklasse bei Applikations-Archiven (Eintrag `Main-Class:`)
Anmerkung : Die Erzeugung des Directories `META-INF` und der Datei `MANIFEST.MF` kann unterdrückt werden.

• Erzeugung und Manipulation von JAR-Dateien

◇ Hierfür ist im JDK das Programm **`jar`** (*Java Archive Tool*) enthalten

◇ Die Arbeitsweise dieses Programms wird durch Kommando-**Optionen** gesteuert.

◇ **Überblick** über einige **wichtige** mit dem Programm `jar` ausführbare **Operationen** :

Erzeugung einer `jar`-Datei	`jar` **`cf`** `jar-file input-file(s)`
Erzeugung einer `jar`-Datei für eine Java-Anwendung (gleichzeitige Festlegung der Startklasse)	`jar` **`cfe`** `jar-file startklasse input-file(s)` (Angabe des vollqualifizierten Klassennamens für die Startklasse)
Update einer existierenden `jar`-Datei (Hinzufügen weiterer Dateien, Überschreiben vorhandener Dateien)	`jar` **`uf`** `jar-file input-file(s)`
Ausgabe des Inhalts einer `jar`-Datei	`jar` **`tf`** `jar-file`
Extraktion des Inhalts einer `jar`-Datei	`jar` **`xf`** `jar-file`

Anmerkungen : Für die in ein JAR-Archiv aufzunehmenden Dateien (`input-file(s)`) gilt :
- Mehrere Datei-Angaben sind durch Blanks zu trennen
- Gegebenenfalls ist der jeweilige Zugriffspfad anzugeben.
  Insbesondere trifft dies für `class`-Dateien in einer Package-Struktur zu.
- Die Angabe des Wildcard-Zeichens `*` ist zulässig
- Die Angabe von Directories ist zulässig. Es werden die Directories sowie alle Dateien, die sich in dem Directory und den darunterliegenden Directories befinden, eingebunden

◇ **Beispiel** : JAR-Datei **`PkDemo.jar`** soll zusammenfassen (Dateipfade beziehen sich auf das aktuelle Directory) :
- Klasse `vrl.pk1.PkDem1` (Datei-Pfad : `vrl\pk1\PkDem1.class`)
- Klasse `vrl.pk2.PkDemUse` (Datei-Pfad : `vrl\pk2\PkDemUse.class`)
  Klasse `vrl.pk2.PkDemUse` ist Startklasse

→ **`jar cfe PkDem.jar vrl.pk2.PackDemUse vrl\pk1\PkDem1.class vrl\pk2\PkDemUse.class`**

Java-Archiv-(JAR-) Dateien (2)

- **Start einer Java-Applikation, die sich in einer JAR-Datei befindet**

  ◇ durch Aufruf des Java-Interpreters mit der **`jar-Option`**.
  Statt der Startklasse ist dem Interpreter die JAR.Datei, die die Startklasse enthält, zu übergeben.
  Programmparameter können – wie bei der direkten Angabe einer Startklasse – zusätzlich übergeben werden.

  ◇ **Beispiel** : Start der in **`PkDem.jar`** enthaltenen Applikation.
  Der Applikation ist die Datei `daten.txt` als Programmparameter zu übergeben

  → **`java -jar PkDem.jar daten.txt`**

- **Aufnahme von `CLASSPATH`-Info in eine JAR-Datei**

  ◇ Es gibt Situationen bei denen eine in einer JAR-Datei zusammengefasste Applikation (oder Applet) Klassen benötigt, die weder in der JAR-Datei enthalten noch über die aktuelle `CLASSPATH`-Variable zugänglich sind (Klassen in einer speziellen Bibliothek).

  ◇ Die für den Zugriff zu diesen Klassen notwendige – temporäre – Ergänzung des Klassenpfades kann durch Aufnahme entsprechender Informationen in die **Manifest-Datei** des JAR-Archivs erreicht werden.

  ◇ Dies wird folgendermaßen ermöglicht :

  ▻ Die zusätzlich benötigten Bibliotheksklassen müssen in einer oder mehreren weiteren JAR-Dateien zusammengefasst sein.

  ▻ Diese JAR-Dateien dürfen nicht in der sie benutzenden JAR-Datei enthalten sein.

  ▻ Erstellung einer **Text-Datei**, die den in die Manifest-Datei aufzunehmenden `Class-Path` –Header enthält :

  `Class-Path: jar-file(s)` (Angabe der Zugriffspfade, mehrere durch Leerzeichen getrennt)

  Diese Textdatei muss mit einem **Zeilenende**-Zeichen **abgeschlossen** sein.

  ▻ Bei der Erzeugung des JAR-Archivs ist zusätzlich die Option **`m`** sowie die Manifest-Text-Datei anzugeben :

  `jar cfe`**`m`** `jar-file startklasse` **`manifest-text-file`** `input-file(s)`

  ◇ **Beispiel** : Im Directory `AppTest` sind die beiden Directories `MyLib` und `MyApp` eingetragen.
  Das Directory `MyLib` enthält die JAR-Datei `MyUtil.jar`. Diese fasst einige Utility-Klassen, die von mehreren Applikationen benötigt werden, zusammen.
  Eine dieser Applikationen wird durch die Klassen `App1` und `AppHelp1` gebildet.
  Ihre Klassendateien befinden sich im Directory `MyApp` in einer Directory-Struktur, die ihre Package-Struktur nachbildet : `hm\ee\jvpr1\App1.class` bzw `hm\ee\jvpr1\AppHelp1.class`.
  Startklasse sei die Klasse `App1`.
  Für die Applikation soll im Verzeichnis `MyApp` die JAR-Datei **`App1.jar`** erzeugt werden :

  Erzeugung `der` Text-Datei **`Manifest.txt`** mit dem Inhalt (im Directory `MyApp`) :

  ```
  Class-Path: ..\MyLib\MyUtil.jar
  ```

  Erzeugung der JAR-Datei **`App1.jar`**:

  → `jar cfe`**`m`** `App1.jar hm.ee.jvpr1.App1` **`Manifest.txt`** `hm\ee\jvpr1\*.class`

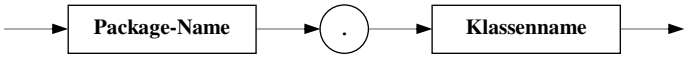
Packages in Java (1)

• Packages als Namensräume

◇ Klassen lassen sich in Java in **Paketen** (***Packages***) zusammenfassen.
Packages sind die Java-Entsprechung der Namensräume (*name spaces*) von C++.

◇ Packages haben einen **Namen** und können **hierarchisch strukturiert** (Package → Unter-Package) sein.
Package-Namen können aus **mehreren Bestandteilen** bestehen, die durch einen Punkt `(.)` voneinander **getrennt** sind.

◇ Damit lässt sich eine Klasse über einen **vollständigen (voll-qualifizierten) Klassennamen** ansprechen.
Dieser besteht aus dem **eigentlichen Klassennamen**, dem der durch einen Punkt `(.)` abgetrennte **Package-Name vorangestellt** ist.



Beispiel : Klasse **`Date`** im Package **`java.util`**
voll-qualifizierter Klassenname : **`java.util.Date`**

◇ Mittels Packages lassen sich **Namenskonflikte**, insbesondere zwischen Klassen unterschiedlicher Hersteller (Bibliotheken), weitgehend **vermeiden**.

◇ **Empfohlene Konvention zur Vergabe von Package-Namen** :
Beginn des Package-Namens mit dem invertierten Internet-Domain-Namen des Herstellers.

Beispiel : Fa. SESA, Internet-Domain-Name : `sesa.com`
Package-Name : **`com.sesa.wmf`**

Es ist auch üblich, aus Gründen der Übersichtlichkeit den **Top-Level-Domain-Namen wegzulassen**.

◇ Klassen, die explizit keinem Package zugeordnet sind, befinden sich im **namenlosen Package** (*unnamed package*).
Für die Verwendung in Bibliotheken sind derartige Klassen nicht geeignet.

• Packages als Directory-Struktur

◇ Ein strukturierter Package-Name wird in eine entsprechende **Directory-Struktur** abgebildet.

◇ **Beispiel** : Klassen-Dateien des Packages **`com.sesa.wmf`** befinden sich im Verzeichnis **`com\sesa\wmf`**

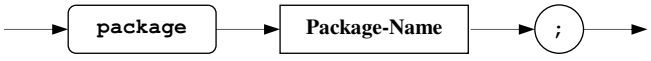
◇ Das **Start-Verzeichnis** einer derartigen Directory-Struktur muss sich im **aktuellen Verzeichnis** oder in einem über die **`CLASSPATH`-Variable** festgelegten Verzeichnis befinden.

◇ Package-Namen ermöglichen damit eine **strukturierte** und dadurch **übersichtliche Ablage** von Klassen-Dateien.

◇ Klassen-Dateien des **namenlosen Package** müssen direkt im aktuellen Directory bzw in einem durch die `CLASSPATH`-Variable festgelegten Verzeichnis liegen.

• Package-Zuordnung einer Klasse

◇ Hierzu dient die **`package`-Vereinbarung**



Beispiel : **`package fhm.ee.vorl;`**

◇ Die `package`-Vereinbarung muss am **Beginn einer Quelldatei** (erste Vereinbarung) stehen.
Sie legt fest, dass die nachfolgend definierte Klasse zu dem angegebenen Package gehört.

Packages in Java (2)

- **Verwendung von Klassen des eigenen Packages**
  - ◇ Eine Klasse kann eine andere Klasse ihres eigenen Packages allein mit dem **einfachen Klassennamen** verwenden.
  - ◇ Jede Klasse eines Packages hat Zugriff zu den anderen Klassen desselben Packages, die nicht `private` sind, **auch** wenn sie **nicht** explizit **`public`** deklariert sind.

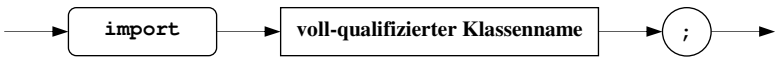
- **Verwendung von Klassen aus anderen Packages**
  - ◇ Eine Klasse kann zu Klassen aus einem anderen Package nur zugreifen, wenn diese **explizit `public`** deklariert sind.
  - ◇ Für die Verwendung bestehen zwei unterschiedliche Möglichkeiten :
    - ▻ Verwendung des voll-qualifizierten Namens.
    - ▻ Importieren der Klassen und Verwendung der einfachen Klassennamen
  - ◇ **Verwendung des voll-qualifizierten Namens** :

    Beispiel : Verwendung der Klasse `Date` aus dem Package `java.util`

```
// Datum1.java

public class Datum1
{
  public static void main(String[] args)
  { java.util.Date now = new java.util.Date();
    System.out.println(now);
  }
}
```

    Nachteil : Klassennamen können sehr lang und damit unhandlich werden.

- **Importieren von Klassen aus anderen Packages**
  - ◇ Hierzu dient die **`import`-Deklaration**. Diese existiert in zwei Formen :
    - ▻ **Importieren einer einzelnen Klasse**



    - ▻ **Importieren aller Klassen eines Packages** (*type import on demand*), aber nicht von dessen Unter-Packages

  - ◇ **Beispiel** : Verwendung der Klasse `Date` aus dem Package `java.util`

```
// Datum2.java

import java.util.Date;              // oder : import java.util.*;

public class Datum2
{
  public static void main(String[] args)
  { Date now = new Date();
    System.out.println(now);
  }
}
```

  - ◇ **Anmerkung** : Importieren bedeutet **kein Einbinden von Code** anderer Klassen.

Packages in Java (3)

- **Die Zugriffsberechtigung "*package*"**

  ◇ Java kennt neben den Zugriffsberechtigungen *private, protected* und *public* die weitere Zugriffsberechtigung ***package***.

  ◇ Diese Zugriffsberechtigung haben alle Klassenkomponenten einer Klasse, die **nicht explizit** als `public`, `protected` oder `private` gekennzeichnet sind.
  → **Default-Zugriffsberechtigung**

  ◇ Diese Zugriffsberechtigung erlaubt den **Zugriff** durch **alle** Klassen **desselben Packages**.

  ◇ Die Zugriffsberechtigung durch ein Package erstreckt sich nicht auf dessen eventuelle Unter-Packages.
  Klassen aus **Unter-Packages** haben **keinen Zugriff** zu *package*-Komponenten von Klassen ihres Ober-Packages.

  ◇ **Anmerkung** zur Zugriffsberechtigung **`protected`** :
  Sie **umfasst** in Java auch die Zugriffsberechtigung ***package***.
  → Zu Klassenkomponenten mit der Zugriffsberechtigung *protected* können neben den abgeleiteten Klassen auch alle Klassen desselben Packages zugreifen.

  ◇ **Beispiel** :

```
package vorl.pack1;

public class PackAccDemo
{
  int ipa;            // Zugriffsberechtigung package
  protected int ipo;
  public int ipu;

  // ...
}
```

```
package vorl.pack1;

class PackAccDemoUse1
{
  public static void main(String[] args)
  { PackAccDemo demo = new PackAccDemo();
    System.out.println("PackAckDemo.ipa (package)   : " + demo.ipa);
    System.out.println("PackAckDemo.ipu (public)    : " + demo.ipu);
    System.out.println("PackAckDemo.ipo (protected) : " + demo.ipo);
  }
}
```

```
package vorl.pack1.sub;
import vorl.pack1.PackAccDemo;

class PackAccDemoUse2
{
  public static void main(String[] args)
  { PackAccDemo demo = new PackAccDemo();
    //System.out.println("PackAckDemo.ipa (package)   : " + demo.ipa);
    System.out.println("PackAckDemo.ipu (public)    : " + demo.ipu);
    //System.out.println("PackAckDemo.ipo (protected) : " + demo.ipo);
 }
}
```

Die Java Standard-Bibliothek (1)

• Allgemeines

◇ Die Java-Plattform stellt auch eine **Laufzeit-Bibliothek** (API) zur Verfügung.

◇ Diese sehr umfangreiche **Java Standard-Bibliothek** enthält zahlreiche nützliche Klassen für **diverse Anwendungsbereiche**.
Eine Reihe dieser Klassen **implementieren wesentliche Spracheigenschaften** und sind damit sehr eng mit der Sprache selbst verknüpft.

◇ Die Klassen der Standardbibibliothek sind auf verschiedene **APIs** aufgeteilt und in **Paketen** (*Packages*) zusammengefasst. Dabei ist die Zuordnung der einzelnen Klassen zu den verschiedenen Paketen nicht immer ganz stimmig, insbesondere stimmen die API-Grenzen nicht immer mit den Paketgrenzen überein.

◇ Zu vielen der **Haupt-Paketen** existieren **Unter-Pakete**.

◇ Die Pakete der im JDK enthaltenen Java Standard-Bibliothek lassen sich in drei Gruppen einteilen :

- ▻ **Standard-Pakete**. Sie gehören zu jeder Java-Plattform
- ▻ **Standard-Erweiterungs-Pakete**. Sie stellen erweiterte und ergänzende Funktionalitäten zur Verfügung und müssen nicht unbedingt auf jeder Java-Plattform zur Verfügung stehen
- ▻ **Ergänzende Pakete** von Dritt-Herstellern. Sie ergänzen und erweitern ebenfalls die Funktionalität der Bibliothek.

◇ Die Java Standard-Bibliothek ist **ständig erweitert** worden.
(JDK 1.0 : 8 Pakete, JDK 1.4 über 130 Pakete, JDK 5.0 über 170 Pakete, JDK 6.0 203 Pakete)

◇ Sun stellt in der das JDK begleitenden Dokumentation eine ausführliche Beschreibung der Klassen zur Verfügung (***Java Platform API Specification***)

• Überblick über die Standard-Pakete

◇ Die Namen der Standard-Pakete beginnen mit **`java.`**

◇ Die folgende Tabelle gibt nur einen Überblick über die **Haupt-Pakete** dieser Gruppe

`java.applet`	Applets
`java.awt`	*Abstract Windowing Toolkit* : Erzeugung von GUIs, mehrere Unter-Pakete
`java.beans`	Java Beans (Komponenten-Architektur von Java), ein Unter-Paket
`java.io`	Ein-/Ausgabe mittels Streams, Dateien und Serialisierung
`java.lang`	Elementare Sprachunterstützung, fünf Unter-Pakete
`java.math`	Unterstützung von Ganzzahl- und Gleitpunkt-Arithmetik
`java.net`	Unterstützung von Netzwerkanwendungen
`java.nio`	*New I/O Package*, verbesserter I/O-Unterstützung, ab JDK 1.4, vier Unter-Pakete
`java.rmi`	*Remote Method Invocation*, Entfernte Objekt-Kommunikation, vier Unter-Pakete
`java.security`	Sicherheits-Framework, vier Unter-Pakete
`java.sql`	Datenbankzugriff
`java.text`	Erweiterte Textdarstellung und –bearbeitung, Internationalisierung, ein Unter-Paket
`java.util`	Utility-Klassen, *Collection Framework*, spezielle Datenstrukturen, neun Unter-Pakete

Die Java Standard-Bibliothek (2)

• Überblick über die Standard-Erweiterungs-Pakete

◇ Die Namen der Standard-Erweiterungs-Pakete beginnt mit **`javax.`**

◇ Die folgende Tabelle gibt nur einen Überblick über die **Haupt-Pakete** dieser Gruppe (hier : Standard Edition)

`javax.accessibility`	Unterstützung spezieller I/O-Geräte (z.B. für Braille-Zeichen)
`javax.activation`	
`javax.activity`	spezielle Exception-Klassen im Zusammenhang mit der CORBA-Serialisierung
`javax.annotation`	Unterstützung von Annotations, ein Unter-Paket
`javax.crypto`	Unterstützung kryptographischer Operationen, zwei Unter-Pakete
`javax.imageio`	I/O von Bilddateien, mehrere Unter-Pakete
`javax.jws`	Unterstützung von Web Services, ein Unter-Paket
`javax.lang.model`	Unterstützung der Modellierung der Sprache Java, drei Unter-Pakete
`javax.management`	Java Management Erweiterungen, mehrere Unter-Pakete
`javax.naming`	Zugriff zu Namensdiensten, vier Unter-Pakete
`javax.net`	Ergänzung zur Netzwerkunterstützung, ein Unter-Paket
`javax.print`	*Print Service API*, Zugriff zu Print Services, drei Unter-Pakete
`javax.rmi`	Ergänzung zum *RMI-API*, zwei Unter-Pakete
`javax.script`	Scripting API (Definition von Java Scripting Engines)
`javax.security`	Ergänzung zum Sicherheits-Framework, besteht nur aus Unter-Paketen
`javax.sound`	*Sound API*, besteht nur aus Unter-Paketen
`javax.sql`	Ergänzung zum Datenbankzugriffs-API (serverseitig), drei Unter-Pakete
`javax.swing`	*Swing Toolkit*, Erweiterungen zur GUI-Erzeugung, zahlreiche Unter-Pakete
`javax.tools`	Interfaces und Klassen für Tools zum Starten aus Programmen heraus
`javax.transaction`	Unterstützung von Transaktionen, ein Unter-Paket
`javax.xml`	Zugriff zu XML-Dateien, zahlreiche Unter-Pakete

• Ergänzende Pakete von Drittherstellern

◇ Hierzu gehören zahlreiche Pakete unterhalb der **`org.omg.`**-Hierarchie.
Sie stellen im wesentlichen **CORBA-Unterstützung** zur Verfügung

◇ Weitere Pakete befinden sich unter den Hierachien **`org.w3c.`** und **`org.xml.`**
Sie bieten Unterstützung für den Zugriff zu **XML-Dateien**.

◇ Das Paket **`org.ietf.jgss`** stellt ein Framework zur Nutzung von Sicherheitsdiensten (Authentifizierung, Daten-Sicherheit, Daten-Integrität usw) unterschiedlichster Mechanismen (wie z.B. Kerberos) zur Verfügung

• Verwendung der Java Standard-Bibliothek

◇ Zur Verwendung von Komponenten der Standard-Bibliothek in eigenen Programmen werden entsprechende **`import`-Vereinbarungen** benötigt, wenn nicht voll-qualifizierte Namen verwendet werden sollen.

Ausnahme : Das Paket **`java.lang`** ist so **eng** mit **der Sprache verknüpft**, dass es **automatisch** durch den Compiler **eingebunden** wird. Eine explizite `import`-Anweisung ist daher für Komponenten aus diesem Paket nicht erforderlich.

◇ Der Java-Compiler und der Java-Interpreter finden den **Zugriffspfad** zu den **Bibliotheks-Dateien automatisch**. Dieser muss daher **nicht** in die **`CLASSPATH`**-Variable aufgenommen werden (ab JDK 1.2).

Datentypen in Java (1)

• Allgemeines

◇ Java ist eine **streng typisierte** Sprache.
Das bedeutet, dass jede Variable und jeder Ausdruck einen Typ hat, der **zur Compilezeit bekannt** sein muß.

◇ In Java muß jede Variable immer einen **definierten Wert** haben.
Membervariable werden automatisch mit einem **Default-Wert initialisiert**, sofern ihnen nicht explizit ein Wert zugewiesen wird.
Bei **lokalen Variablen** verhindert der Compiler, dass sie ohne explizite Initialisierung bzw Wertzuweisung verwendet werden (***Definite Assignment***).

◇ In Java gibt es **zwei Kategorien** von Datentypen :

- ▻ **einfache** (primitive) **Datentypen** (*primitive types*)
- ▻ **Referenz-Typen** (*reference types*)

• Einfache Datentypen

◇ **Variable** eines einfachen Datentyps enthalten einen **Wert** ihres jeweiligen Typs.

◇ Java kennt **acht einfache Datentypen**

◇ Für jeden Datentyp ist der **Wertebereich** und damit die **Größe** des belegten Speicherplatzes **eindeutig** – unabhängig von der jeweiligen Plattform – **festgelegt**.

◇ Zu jedem primitiven Datentyp ist in der Standard-Bibliothek eine **Wrapper-Klasse** definiert (Package **`java.lang`**). Dies ermöglicht die Anwendung objektorientierter Prinzipien und Möglichkeiten auch auf die einfachen Datentypen.

◇ **Überblick** :

Typname	Größe	Art des Typs	Wertebereich	Default-Wert	Wrapper-Klasse
`byte`	1 Byte	vorzeichenbeh. ganze Zahl	`-128 ... +127`	`0`	**`Byte`**
`short`	2 Bytes	vorzeichenbeh. ganze Zahl	-2^{15} ... $+2^{15}-1$	`0`	**`Short`**
`int`	4 Bytes	vorzeichenbeh. ganze Zahl	-2^{31} ... $+2^{31}-1$	`0`	**`Integer`**
`long`	8 Bytes	vorzeichenbeh. ganze Zahl	-2^{63} ... $+2^{63}-1$	`0`	**`Long`**
`char`	2 Bytes	Unicode-Zeichen	alle Unicode-Zeichen	`'\u0000'`	**`Character`**
`float`	4 Bytes	Gleitpunktzahl (reelle Zahl)	$+/-3.4028...*10^{38}$	`0.0`	**`Float`**
`double`	8 Bytes	Gleitpunktzahl (reelle Zahl)	$+/-1.7976...*10^{308}$	`0.0`	**`Double`**
`boolean`	1 Bit *)	logischer Wert	`false, true`	`false`	**`Boolean`**

*) Die belegte Speichergröße ergibt sich durch die kleinste adressierbare Speichereinheit (meist 1 Byte)

◇ Der Datentyp **`float`** entspricht dem **32-Bit-IEEE-Format** (*single precision*, ANSI/IEEE Standard 754-1985)
Der Datentyp **`double`** entspricht dem **64-Bit-IEEE-Format** (*double precision*, ANSI/IEEE Standard 754-1985)

◇ Der Datentyp **`char`** zählt (zusammen mit **`byte`**, **`short`**, **`int`** und **`long`**) zu den **ganzzahligen Datentypen**.

◇ Die ganzzahligen Datentypen bilden zusammen mit den **Gleitpunkttypen** (**`float`** und **`double`**) die **numerischen Typen**.

◇ Der **logische Datentyp** **`boolean`** muß überall dort verwendet werden, wo ein **logischer Operand erforderlich** ist (logische Operatoren, Steuerausdrücke in den Steueranweisungen, erster Ausdruck im bedingten Auswerte-Operator). Ein **Ersatz durch ganzzahlige Typen** (Werte `0` und `!=0`) ist **nicht zulässig**.

Datentypen in Java (2)

- **Darstellung der Werte der einfachen Datentypen (Konstante, *literals*)**

  ◇ Die Wertedarstellung entspricht im wesentlichen der von C/C++

  ◇ Datentypen **`byte`**, **`short`**, **`int`** und **`long`** (ganzzahlige Datentypen ohne `char`)
  - ➢ Darstellung als **Dezimalzahl** : Ziffernfolge aus `0` bis `9`, Beginn nicht mit `0`
  - ➢ Darstellung als **Oktalzahl** : Beginn mit `0` (nur Ziffern `0` bis `7`)
  - ➢ Darstellung als **Sedezimalzahl** : Beginn mit `0x` oder `0X` (Ziffern `0` bis `9`, `a` bis `f`, `A` bis `F`)
  - ➢ Ziffernfolgen sind **grundsätzlich** (ohne Suffix) vom Typ **`int`**.
    Allerdings muss der durch die Ziffernfolge dargestellte Wert innerhalb des für `int` zulässigen Bereichs liegen.
  - ➢ Ziffernfolgen mit dem **Suffix `l`** oder **`L`** sind vom Typ **`long`**
  - ➢ Jede Ziffernfolge, die einen ganzzahligen Wert darstellt, kann mit dem **Vorzeichen `-`** oder **`+`** versehen werden.

  ◇ Datentypen **`float`** und **`double`**
  - ➢ Exponential- oder Dezimalbruchdarstellung
  - ➢ **ohne Suffix** oder **mit Suffix `d`** oder **`D`** : Datentyp **`double`**
  - ➢ **mit Suffix `f`** oder **`F`** : Datentyp **`float`**
  - ➢ Zusätzlich kann eine Gleitpunktzahl mit dem **Vorzeichen `-`** oder **`+`** versehen werden.

  ◇ Für alle **numerischen Datentypen** sind in den entsprechenden **Wrapper-Klassen** (Package `java.lang`) die folgenden **symbolischen Konstanten** (als `static public`) definiert :

`MAX_VALUE`	größter darstellbarer positiver Wert
`MIN_VALUE`	kleinster darstellbarer positiver Wert

  Für die Datentypen **`float`** und **`double`** sind in den **Klassen `Float`** und **`Double`** zusätzlich definiert :

`NaN`	Not-a-Number (Repräsentation eines ungültigen Werts, z.B. `0/0`)
`NEGATIVE_INFINITY`	negativ unendlich
`POSITIVE_INFINITY`	positiv unendlich

  Verwendung dieser Konstanten immer nur zusammen mit dem jeweiligen Klassennamen (**voll-qualifizierter Name**) :
  z.B. : **`double dv = Double.MIN_VALUE;`**

  ◇ Datentyp **`char`**
  - ➢ Darstellung eines **Einzelzeichens** in einfachen Hochkommata (*single quotes*), z.B.: **`'A'`**, **`'Π'`**, **`':'`**
  - ➢ Darstellung durch eine **C-kompatible Escape-Sequenz** in einfachen Hochkommata:

`'\b'`	Backspace (`BS`)
`'\t'`	Horizontaler Tabulator (*Horizontal Tab*, `HT`)
`'\n'`	Zeilenwechsel (*Newline*, *Linefeed*, `LF`)
`'\r'`	*Carriage Return* (`CR`)
`'\f'`	Seitenwechsel (*Form Feed*, `FF`)
`'\''`	Ersatzdarstellung für einfaches Hochkomma (*single quote*)
`'\"'`	Ersatzdarstellung für doppeltes Hochkomma (*double quote*)
`'\\'`	Ersatzdarstellung für Fluchtsysmbol **`\`**
`'\ooo'`	Oktal-Escape-Sequenz, 1 bis 3 Oktalziffern, maximaler Wert : `0377`

  Anmerkung : Die C-Escape-Sequenzen **`'\v'`**, **`'\a'`**, **`'\?'`** und **`'\xhhh'`** sind in Java **nicht** implementiert.

  - ➢ Darstellung durch eine **Unicode-Escape-Sequenz** in einfachen Hochkommata : `'\uhhhh'` (`h` Sedezimalziffer),
    z.B. : **`'\u0041'`** (== `'A'`), **`'\u00dc'`** (== `'Ü'`), **`'\uffff'`**

Datentypen in Java (3)

• Typkonvertierungen bei einfachen Datentypen

◇ **Typkonvertierungen** (Casts) zwischen allen **ganzzahligen Typen** (einschließlich `char`) und den **Gleitpunkt-Typen** sind grundsätzlich **zulässig**.

◇ **Implizite Typkonvertierungen** bei **typgemischten zweistelligen arithmetischen Operationen** vor Ausführung der Operation (*numeric promotion*) :

- ▻ **Nur ganzzahlige** Operanden (Integer-Operation), **kein Operand** vom Typ **`long`** :
  Alle **Nicht-`int`-Operanden** werden in den Typ **`int`** umgewandelt.
  Die Operation wird als **`int`-Operation** ausgeführt und liefert ein **`int`-Ergebnis**
- ▻ **Nur ganzzahlige** Operanden (Integer-Operation), **ein Operand** vom Typ **`long`** :
  Der andere **Operand** wird in den Typ **`long`** umgewandelt.
  Die Operation wird als **`long`-Operation** ausgeführt und liefert ein **`long`-Ergebnis**
- ▻ **Ein Operand** ist vom Typ **`double`**.
  Der andere **Operand** wird in den Typ **`double`** umgewandelt.
  Die Operation wird als **`double`-Operation** ausgeführt und liefert ein **`double`-Ergebnis**
- ▻ **Ein Operand** ist vom Typ **`float`** und der **andere nicht** vom Typ **`double`**.
  Der andere **Operand** wird in den Typ **`float`** umgewandelt.
  Die Operation wird als **`float`-Operation** ausgeführt und liefert ein **`float`-Ergebnis**

◇ **Implizite Typkonvertierungen** zwischen **numerischen Typen** bei **Wertzuweisungen** (und Initialisierungen) :
Nur zulässig, wenn sichergestellt ist, dass durch die Typkonvertierung **keine Information verloren** gehen kann :

- ▻ Ganzzahl-Werte an Ganzzahl-Variable eines größeren Typs.
- ▻ Ganzzahl-Werte an Gleitpunkt-Variable
- ▻ `int`-Konstante an `byte`-, `short`- oder `char`-Variable, wenn die `int`-Konstante sich auch als Wert des kleineren Datentyps darstellen lässt.

◇ Alle **anderen Umwandlungen** zwischen **numerischen Typen**, wenn also Information verloren gehen könnte, sind nur **explizit** möglich (**Cast-Operator** !).

◇ **Zwischen** dem Typ **`boolean`** und den **numerischen Datentypen** sind **keinerlei Typkonvertierungen** (auch nicht explizit) zulässig.

◇ Werte eines **numerischen Datentyps** lassen sich mit einem **`String`-Operanden** (Bibliotheks-Klasse `String`) mittels des **Konkatenations-Operators (+)** vernüpfen. In einem derartigen Fall wird der **numerische Wert** in seine **dezimale Text-Darstellung umgewandelt** und mit dem `String`-Operanden zu einem neuen String zusammengefasst.

Beispiel :
```
double dv = 74.25;
System.out.println("Wert von dv : " + dv);
```
→ **Ausgabe** : `Wert von dv : 74.25`

◇ Analog lassen sich Werte des Typs `boolean` mit einem **`String`-Operanden** mittels des **Konkatenations-Operators (+)** verknüpfen. In einem derartigen Fall wird der **logische Wert** in seine **Textdarstellung** (entweder `"true"` oder `"false"`) **umgewandelt** und mit dem `String`-Operanden zu einem neuen String zusammengefasst.

Datentypen in Java (4)

- **Referenz-Typen**

  ◇ Sie referieren **Objekte,** d.h. Instanzen einer Klasse.
  Objekte können in Java **nicht direkt als Werte** verwendet werden (z.B. als Funktions-Parameter), sondern **nur über** eine **Referenz**.

  ◇ **Variable** eines **Referenz-Typs** (Referenz-Variable) werden i.a. als "**Objekt-Variable**" betrachtet.
  Tatsächlich sind es aber **Pointer-Variable**, die auf ein Objekt verweisen. Sie belegen nur den Speicherplatz zur Aufnahme einer Objekt-Adresse. Ihr **Wert** ist also eine **Speicheradresse**.
  Bei ihrer Verwendung werden sie **automatisch dereferenziert**. (Anwendung des Punkt- `(.)` Operators zum Komponentenzugriff).

  ◇ Eine Referenz-Variable kann immer nur auf Objekte eines bestimmten Typs bzw bestimmter Typen (Polymorphie !) zeigen.
  Formal werden **drei Arten von Referenz-Variablen** unterschieden :

  - ▻ **Klassen-Typen** (sie verweisen auf Objekte der entsprechenden Klasse oder auf Objekte davon abgeleiteter Klassen)
  - ▻ **Interface-Typen** (sie verweisen auf Objekte, die das jeweilige Interface implementieren)
  - ▻ **Array-Typen** (sie verweisen auf Arrays, Arrays sind in Java ebenfalls – spezielle – Objekte)

  ◇ **Objekte** werden grundsätzlich nur **namenlos dynamisch** auf dem **Heap alloziert**.
  Hierzu dient ein **`new-Ausdruck`** (**`new-Operator`** mit nachgestellten **Konstruktor**-Aufruf).
  Konstruktoren sind spezielle klassenspezifische Funktionen, die zur Erzeugung eines Objekts benötigt werden.
  Sie tragen i.a. den Namen der Klasse des zu erzeugenden Objekts (Ausnahme s. Arrays).
  Konstruktoren können Parameter besitzen.
  Der `new-`Ausdruck liefert die **Adresse des erzeugten Objekts** zurück, die dann einer "Objekt-Variablen" als Wert zugewiesen werden kann.

  Beispiel :
  ```
  String sv = new String("Hallo !");
  ```

  ◇ Die für Referenz-Variable definierten **Vergleichs-Operatoren** **`==`** und **`!=`** beziehen sich auf die **Referenzen** (Adressen) und **nicht** auf die **referierten Objekte**.
  Gleiche Referenzen bedeuten natürlich auch **Gleichheit der referierten Objekte**.
  Ungleiche Referenzen bedeuten aber **nicht**, dass auch die **referierten Objekte ungleich** sein müssen.

  ◇ Als **spezielle Referenz-Konstante** ist der Wert **`null`** definiert.
  Eine Referenz-Variable mit diesem Wert **zeigt auf nichts** (also auf kein Objekt).
  Einer **lokalen** Referenz-Variablen, die auf nichts zeigen soll, muss dieser Wert **explizit zugewiesen** werden.

  Beispiel :
  ```
  String[] sa = null;
  // ...
  sa = new String[10];
  ```

  Für Referenz-Variable, die **Member-Variable** sind, ist `null` der ohne explizite Initialisierung zugewiesene **Default-Wert**.

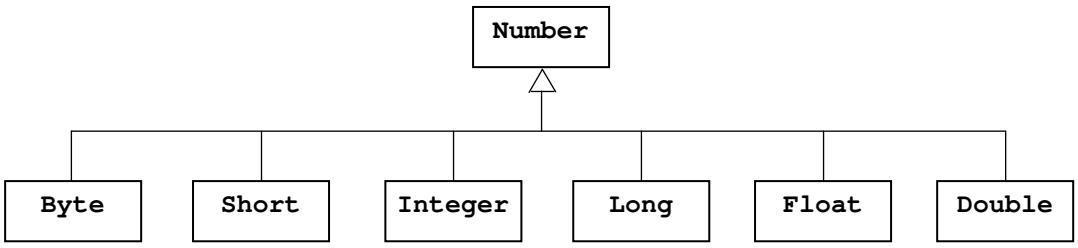
  ◇ Da Java über eine im Hintergrund laufende **automatische *Garbage Collection*** verfügt, ist es weder notwendig noch i.a. sinnvoll und tatsächlich auch nicht möglich, dynamisch allozierten Speicher explizit freizugeben.
  → In Java existiert daher **kein Speicherfreigabe-Operator** (wie z.B. `delete` in C++).

  ◇ Mit dem auf Referenz-Variable anwendbaren Operator **`instanceof`** lässt sich **überprüfen**, ob das **referierte Objekt** von einem **bestimmten Typ** ist.

Datentypen in Java (5)

- **Anmerkungen zu den Wrapper-Klassen für die einfachen Datentypen**

  ◇ Die Wrapper-Klassen für die einfachen Datentypen `byte`, `short`, `int`, `long`, `float` und `double` sind von der **abstrakten Basisklasse** **`Number`** abgeleitet :



  ◇ Die Klasse **`Number`** deklariert die folgenden **Methoden** :

  ▻
```
public byte byteValue()
```
  Rückgabe des gespeicherten (gekapselten) Wertes typgewandelt als `byte`-Wert

  ▻
```
public short shortValue()
```
  Rückgabe des gespeicherten (gekapselten) Wertes typgewandelt als `short`-Wert

  ▻
```
public abstract int intValue()
```
  Rückgabe des gespeicherten (gekapselten) Wertes typgewandelt als `int`-Wert

  ▻
```
public abstract long longValue()
```
  Rückgabe des gespeicherten (gekapselten) Wertes typgewandelt als `long`-Wert

  ▻
```
public abstract float floatValue()
```
  Rückgabe des gespeicherten (gekapselten) Wertes typgewandelt als `float`-Wert

  ▻
```
public abstract double doubleValue()
```
  Rückgabe des gespeicherten (gekapselten) Wertes typgewandelt als `double`-Wert

  Diese Methoden müssen von allen **abgeleiteten Klassen** jeweils **implementiert** werden ("**Getter**"-Methoden).

  ◇ Analog definiert die Wrapper-Klasse **`Character`** für den Datentyp `char` u.a. die folgenden **Methode** :

  ▻
```
public char charValue()
```
  Rückgabe des gespeicherten (gekapselten) `char`-Wertes

  ◇ Analog definiert die Wrapper-Klasse **`Boolean`** für den Datentyp `boolean` u.a. die folgenden **Methode** :

  ▻
```
public boolean booleanValue()
```
  Rückgabe des gespeicherten (gekapselten) `boolean`-Wertes

Umwandlungen zwischen einfachen Java-Datentypen und ihren Wrapper-Klassen

• Verwendung von Werten einfacher Datentypen wie Objekte

◇ Werte einfacher Datentypen und Objekte werden unterschiedlich dargestellt und referiert.
Dadurch lassen sich Werte einfacher Datentypen **nicht direkt** in Situationen, in denen Objekte benötigt werden, verwenden (z.B. als Argument in vielen Methoden, aktueller Typ bei generischen Datentypen)

◇ Um eine derartiger Verwendung trotzdem zu ermöglichen, existieren **Wrapper-Klassen** für alle einfachen Datentypen. Ein Objekt einer Wrapper-Klasse **kapselt** einen Wert des korrespondierenden einfachen Datentyps. Dieser Wert wird bei der Objekterzeugung festgelegt (Konstruktor-Parameter) und kann später nicht mehr geändert werden.

◇ Zwischen Werten einfacher Datentypen ("primitive Werte") und den entsprechenden Objekten der jeweiligen Wrapper-Klasse sind also **Umwandlungen** erforderlich.

• Explizite Umwandlung

◇ **Wert → Objekt** : Mit dem **Konstruktor** der korrespondierenden Wrapper-Klasse.

Beispiel 1 :
```
int iVal1 = 12;
Integer iObj;
iObj = new Integer(iVal1);        // Erzeugung eines neuen Integer-Objekts
```

Beispiel 2 :
```
Double dObj = new Double(0.75); // Erzeugung eines neuen Double-Objekts
```

◇ **Objekt → Wert** : Mit einer **"Getter"-Methode** der Wrapper-Klasse, z.B. **`int intValue()`** der Klasse **`Integer`**

Beispiel 1 :
```
int iVal2 = iObj.intValue();          // De-Wrapping
```

Beispiel 2 :
```
double dVal = dObj.doubleValue();    // De-Wrapping
```

◇ Bis einschliesslich dem JDK 1.4 mussten derartige Umwandlungen immer explizit codiert werden.

• Implizite Umwandlung (*Autoboxing/-unboxing*)

◇ Mit dem JDK 5.0 eingeführt.

◇ Obige Umwandlungen müssen nicht explizit codiert werden, sie werden **automatisch** vom Compiler erzeugt.
→ **Werte einfacher Datentypen** und **Wrapper-Objekte** können völlig **kompatibel zueinander verwendet** werden

◇ **Wert → Objekt** (*Autoboxing*) :

Beispiel 1 :
```
int iVal1 = 12;
Integer iObj;
iObj = iVal1;          // implizite Erzeugung eines neuen Integer-Objekts
```

Beispiel 2 :
```
Double dObj = 0.75; // implizite Erzeugung eines neuen Double-Objekts
```

◇ **Objekt → Wert** (*Autounboxing*) :

Beispiel 1 :
```
int iVal2 = iObj;    // implizites De-Wrapping
```

Beispiel 2 :
```
double dVal = dObj; // implizites De-Wrapping
```

◇ ***Autoboxing/-unboxing*** findet **überall** statt, wo Werte einfacher Datentypen bzw Wrapper-Objekte benötigt werden. Damit können z.B. **mit Wrapper-Objekten** auch "**direkt**" **arithmetische Operationen** vorgenommen werden.

Beispiele :
```
Integer iObj1 = 10;
Integer iObj2 = iObj1 + 5;
++iObj1;
```

◇ ***Autoboxing*** findet allerdings nur dann statt, wenn **keine andere implizit** mögliche **Typkonvertierung** anwendbar ist. Derartige Situationen können z.B. bei **überladenen Funktionen** auftreten.

Beispiel :
```
void func(double dv);
void func(Integer io);
. . .
```
`func(5)` führt zum Aufruf der ersten Methode (**`func(5.0)`**) (**`int → double`**)

Strings in Java - Allgemeines

• Darstellung von Strings

◇ Strings in Java bestehen aus **Unicode-Zeichen**.
Wie in C++ besteht eine **String-Konstante** (*string literal*) aus einer **Folge von 0 oder mehr Zeichen**, die in **doppelte Hochkommata** (*double qoutes*) eingeschlossen sein müssen.
Die Zeichen können auch durch Escape-Sequenzen dargestellt werden.

Beispiele :
```
"\n\"Hallo\""
"\u00ea\u0256"
```

◇ Strings sind in Java **Objekte.**
String-Konstante werden durch Objekte der Bibliotheks-Klasse **`String`** dargestellt.
Der Inhalt derartiger Objekte ist **unveränderlich**.
Zur Darstellung von **veränderlichen Strings** dienen die Bibliotheks-Klassen **`StringBuffer`** (thread-sicher) und **`StringBuilder`** (ab JDK 5.0, nicht thread-sicher, aber schneller, da keine Synchronisation erforderlich) **.**

◇ Eine **String-Variable** ist – wie alle Objekt-Variablen in Java – eine Referenz-Variable. Sie enthält eine **Referenz** auf ein **`String-`** (bzw **`StringBuffer-`** bzw **`StringBuilder`**-) **Objekt**.

Eine **Zuweisung eines neuen Strings** an eine **`String-Variable`** ("Variable vom Typ `String`") bedeutet nicht, dass das referierte `String`-Objekt geändert wird, sondern dass die `String`-Variable ein **anderes `String-Objekt` referiert**.

Beispiel :
```
String str = "Hausboot";
// ...
str = "Segelyacht";   // str zeigt auf ein anderes Objekt !
```

• Direkte Sprachunterstützung für Strings

◇ In der Sprache Java selbst sind einige **Mechanismen zur Erzeugung und Verwendung von Strings** implementiert, die im Zusammenhang mit der Klasse `String` zur Anwendung kommen

◇ Beim Auftritt einer **String-Konstanten** im Quelltext erzeugt der **Compiler** ein **`String-Objekt`**, das mit der String-Konstanten initialisiert wird.

Beispiele :
```
System.out.println("Schoene Gruesse !");
```
Der Compiler erzeugt ein `String`-Objekt mit dem Inhalt `"Schoene Gruesse !"` und übergibt eine Referenz (Adresse) auf dieses Objekt der Methode `println(...)`

```
String s1 = "Das alte Europa";
```
Der Compiler erzeugt ein `String`-Objekt mit dem Inhalt `" Das alte Europa"` und weist der Variablen `s1` eine Referenz (Adresse) auf dieses Objekt zu

◇ Für `String`-Objekte ist der **Operator `+`** überladen als **Konkatenations-Operator**.
Er erzeugt ein **neues `String-Objekt`**, das aus den beiden Operanden zusammengesetzt ist.

Beispiel :
```
s1 = s1 + " hatte Recht !";
```

◇ Der **`+` -Operator** wirkt auch dann als **Konkatenations-Operator**, wenn **nur ein Operand vom Typ `String`** ist.
In einem derartigen Fall wird der **andere Operand implizit** in ein **`String-Objekt` umgewandelt** und mit dem `String`-Operanden zu einem **neuen `String-Objekt` konkateniert**.
Die **implizite Konvertierung** in ein `String`-Objekt (*string conversion*) findet **für jeden einfachen Typ** und für **jeden Referenz-Typ** (Klasse !) statt. Ist die in einer Objekt-Variablen gespeicherte **Referenz** gleich **`null`,** wird der String **`"null"`** erzeugt.

Beispiel :
```
System.out.println("Heute ist : " + new java.util.Date());
```
Ein neu erzeugtes namenloses `Date`-Objekt (Package `java.util`) wird in ein `String`-Objekt (das das Datum als Text enthält) umgewandelt und mit dem `String`-Objekt `"Heute ist : "` zu einem neuen `String`-Objekt zusammengefasst. Eine Referenz auf dieses Objekt wird der Methode `println(...)` als Parameter übergeben.

◇ Eine **String-Konkatenation** kann **auch** mit dem **`+=` -Operator** realisiert werden. Wenn hier der rechte Operand kein `String`-Objekt ist, wird er implizit in ein solches konvertiert.

Die Klasse `String` in Java (1)

- **Allgemeines zur Klasse `String`**
  - ◇ Standard-Klasse zur Darstellung von **nicht veränderlichen Strings**.
  - ◇ Bestandteil des **Package `java.lang`**
  - ◇ Von der Klasse `String` können **keine anderen Klassen abgeleitet** werden (Die Klasse ist **`final`**)
  - ◇ Die Klasse `String` **implementiert** das Interface **`CharSequence`**
  - ◇ Die Klasse `String` enthält zahlreiche **Methoden** zur **Verwendung von Strings**, u.a.
    - zur expliziten Erzeugung von `String`-Objekten (Konstruktoren),
    - zur Ermittlung der Stringlänge
    - zum lesenden Zugriff zu String-Elementen,
    - zum String-Vergleich
    - zum Suchen in Strings
    - zur Extraktion von Teilstrings,
    - zum Kopieren von Strings mit Umwandlung aller Buchstaben in Großbuchstaben bzw Kleinbuchstaben.

- **Konstruktoren der Klasse `String` (Auswahl)**

`public` **`String()`**	Erzeugt ein `String`-Objekt, das den **Leerstring** enthält
`public` **`String(String str)`**	Erzeugt ein `String`-Objekt, das eine Kopie des Inhalts von **`str`** enthält (**Copy-Konstruktor**)
`public` **`String(char[] acs)`**	Erzeugt ein `String`-Objekt, das die **Zeichenfolge** aus **`acs`** enthält. Die Zeichen werden in das neu erzeugte Objekt kopiert.
`public` **`String(byte[] abs)`**	Erzeugt ein `String`-Objekt, das mit der **Zeichenfolge**, die durch **Decodierung** der `byte`-Werte von **`abs`** entsteht, initialisiert ist. Die Decodierung erfolgt für den Default-Zeichensatz der Implementierung
`public` **`String(StringBuffer buf)`**	Erzeugt ein `String`-Objekt, das mit der in **`buf`** enthaltenen **Zeichenfolge** initialisiert ist. Die Zeichen werden in das neu erzeugte Objekt kopiert.
`public` **`String(StringBuilder bld)`**	Erzeugt ein `String`-Objekt, das mit der in **`bld`** enthaltenen **Zeichenfolge** initialisiert ist. Die Zeichen werden in das neu erzeugte Objekt kopiert.

◇ **Hinweis** : Die **explizite Erzeugung** eines `String`-Objekts bei **Initialisierung mit einer String-Konstanten** ist **ineffizient** .

```
String str = new String("Wer rastet, der rostet");
```

Dies führt zunächst zur **impliziten Erzeugung** eines `String`-Objektes das mit der String-Konstanten `"Wer rastet, der rostet"` initialisiert ist.
Anschliessend wird zur **expliziten Erzeugung** eines **weiteren** `String`-Objekts der **Copy-Konstruktor** aufgerufen, dem eine Referenz auf das zuerst erzeugte Objekt übergeben wird.
Eine Referenz auf das explizit erzeugte zweite Objekt wird dann der `String`-Variablen `str` zugewiesen.

Vorzuziehen ist daher die **direkte "Initialisierung**" einer `String`-Variablen mit einer String-Konstanten. In diesem Fall wird **nur ein** `String`-Objekt – implizit – erzeugt

```
String str = "Wer rastet, der rostet";
```

Entsprechendes gilt für die **Wertzuweisung** an `String`-Variable.

Die Klasse `String` in Java (2)

- **Memberfunktionen Klasse `String` (Auswahl)**

◇
```
public int length()
```

▻ Gibt die **Länge** eines `String`-Objekts (= die Anzahl der enthaltenen Unicode-Zeichen) als Funktionswert zurück

◇
```
public char charAt(int index)
```

▻ Liefert das **Zeichen** mit dem **Index** **`index`** als Funktionswert zurück.
▻ Der **zulässige Indexbereich** reicht von **`0`** (erstes Zeichen) bis **`length()-1`** (letztes Zeichen).
▻ Erzeugung einer **`IndexOutOfBoundsException,`** wenn ein **unzulässiger Index** übergeben wird.

◇
```
public boolean contains(CharSequence s)          // ab dem JDK 5.0 vorhanden
```

▻ Überprüfung, ob die **Zeichenfolge** **`s`** (z.B. Inhalt eines `String`-Objekts) im aktuellen Objket **enthalten** ist
▻ Falls ja, wird `true` als Funktionswert zurückgegeben, andernfalls `false`

◇
```
public int indexOf(int ch)
```

▻ Rückgabe des **Index** des **ersten Auftritts** des – als `int`-Wert übergebenen – **Zeichens** **`ch`** als Funktionswert.
▻ Falls das Zeichen `ch` **nicht enthalten** ist, wird der Wert **`-1`** zurückgegeben.
▻ **Beispiel** : **`"hamburger".indexOf('r')`** liefert **`5`**

◇
```
public int indexOf(int ch, int from)
```

▻ Funktionalität wie **`int indexOf(int ch),`** allerdings beginnt die Suche am Index **`from`**
▻ **Beispiel** : **`"hamburger".indexOf('r', 6)`** liefert **`8`**

◇
```
public int indexOf(String str)
```

▻ Rückgabe des **Index** des **ersten Auftritts** des **Teilstrings** **`str.`**
▻ Der zurückgegebene Funktionswert ist der Index des ersten Zeichens des gefundenen Teilstrings.
▻ Falls der Teilstring `str` **nicht enthalten** ist, wird der Wert **`-1`** zurückgegeben.

◇
```
public int indexOf(String str, int from)
```

▻ Funktionalität wie **`int indexOf(String str),`** allerdings beginnt die Suche am Index **`from`**

◇
```
public int lastIndexOf(int ch)
public int lastIndexOf(int ch, int from)
public int lastIndexOf(String str)
public int lastIndexOf(String str, int from)
```

▻ Ähnlich wie die Funktionen `indexOf(...).`
▻ Nur Ermittlung des **Index** des **letzten Auftritts** des Zeichens **`ch`** bzw des Teilsstrings **`str.`**
▻ **Beispiel** : **`"hamburger".lastIndexOf("rindfleisch")`** liefert **`-1`**

Die Klasse `String` in Java (3)

- **Memberfunktionen Klasse `String` (Auswahl), Forts.**

◇
```
public boolean equals(Object obj)
```

- ▻ Überprüfung, ob das übergebene Objekt ein **`String`-Objekt** ist und **gleiche Länge** und **gleichen Inhalt** wie das aktuelle Objekt hat.
- ▻ Falls ja, wird `true` als Funktionswert zurückgegeben, andernfalls `false`

◇
```
public int compareTo(String str)
```

- ▻ Durchführung eines **lexikographischen Vergleichs** zwischen dem aktuellen `String`-Objekt und dem `String`-Objekt `str`
- ▻ Der Vergleich wird mit den **Codewerten** der einzelnen Zeichen bzw der Stringlänge ausgeführt
- ▻ **Funktionswert** : - `0`, wenn beide `String`-Objekte gleich sind
  - `<0`, wenn das aktuelle Objekt "kleiner" als `str` ist
  - `>0`, wenn das aktuelle Objekt "größer" als `str` ist
- ▻ **Beispiel** : **`"haben".compareTo("hat")`** liefert **`-18`**

◇
```
public boolean startsWith(String prefix)
```

- ▻ Überprüfung, ob das **aktuelle Objekt** mit dem **String `prefix` beginnt**.
- ▻ Falls ja, wird `true` als Funktionswert zurückgegeben, andernfalls `false`

◇
```
public String substring(int beg, int end)
```

- ▻ Rückgabe des **Teilstrings**, der am Index **`beg`** beginnt und am Index **`end-1`** endet, als neues `String`-Objekt (genauer : als Referenz auf ein neu erzeugtes `String`-Objekt mit dem entsprechenden Inhalt des Teilstrings).
- ▻ Erzeugung einer **`IndexOutOfBoundsException`**, falls wenigstens einer der übergebenen **Indizees unzulässig** ist oder **`beg>end`** ist.
- ▻ **Beispiel** : **`"hamburger".substring(4,8)`** liefert **`"urge"`**

◇
```
public String toUpperCase()
```

- ▻ Rückgabe eines neuen `String`-Objekts, in dem alle Buchstaben in **Großbuchstaben** konvertiert sind

◇
```
public String replace(char alt, char neu)
```

- ▻ Rückgabe eines **neuen** `String`-Objekts, das aus dem aktuellen Objekt durch **Ersatz** aller Auftritte des **Zeichens `alt`** durch das **Zeichen `neu`** und der Übernahme alle übrigen Zeichen entsteht.
- ▻ Ist das Zeichen **`alt` überhaupt nicht** im aktuellen Objekt **enthalten**, wird kein neues Objekt erzeugt, sondern das **aktuelle Objekt** zurückgegeben.

◇
```
public String trim()
```

- ▻ Rückgabe einer neu erzeugten **Kopie des aktuellen Objekts**, bei der alle **führenden** und **endenden Blanks** sowie **ASCII-Steuerzeichen** (Zeichencode **`<= '0u0020'`**) entfernt sind
- ▻ Falls sowohl das **erste** als auch **das letzte Zeichen** des aktuellen `String`-Objekts einen Zeichencode **`> '0u0020'`** besitzen, wird keine Kopie erzeugt, sondern das **aktuelle Objekt** zurückgegeben

Die Klasse `String` in Java (4)

- **Explizite Umwandlung in eine String-Darstellung**

  ◇ In der Klasse `String` existiert eine mehrfach überladene **statische Memberfunktion** zur Umwandlung von beliebigen Werten der einfachen Datentypen sowie von Objekten beliebiger Klassen in eine String-Darstellung :

  Beispiel für die Umwandlung von `double`-Werten :

```
public static String valueOf(double d)
```

  ◇ Für **jede Klasse** ist in Java die folgende **Memberfunktion** definiert, mit der ein Objekt der jeweiligen Klasse in ein `String`-Objekt (**textuelle Repräsentation** des Objekts !) umgewandelt wird :

```
public String toString()
```

- **Demonstrationsbeispiel zur Klasse `String`**

```
// StringDemo.java

public class StringDemo
{
  public static void main(String[] args)
  { System.out.println("Heute ist : " + new java.util.Date());
    String str = "Aller Anfang";
    str = str + " ist";
    str += " schwer";
    System.out.println(str);
    int len = str.length();
    System.out.println("Laenge des Strings : " + len);
    int idx = 6;
    System.out.println(idx+1 + ". Zeichen : " + str.charAt(idx));
    System.out.println("Index von \"ist\" : " + str.indexOf("ist"));
    char ch = 'r';
    System.out.println("letzter Index von " + ch + " : " + str.lastIndexOf(ch));
    int ie = idx+6;
    System.out.print("Teilstring (" + idx + ',' + ie + ") : ");
    System.out.println(str.substring(idx, ie));
    System.out.println(str.toUpperCase());
    if (str.equals("Aller Anfang ist schwer"))
      System.out.println("Strings sind gleich !");
    else
      System.out.println("Strings sind ungleich !");
    int diff = "haben".compareTo("hat");
    System.out.println("Vergleich von \"haben\" und \"hat\" liefert : "+diff);
  }
}
```

Ausgabe des Programms

```
Heute ist : Thu Sep 04 10:38:41 CEST 2003
Aller Anfang ist schwer
Laenge des Strings : 23
7. Zeichen : A
Index von "ist" : 13
letzter Index von r : 22
Teilstring (6,12) : Anfang
ALLER ANFANG IST SCHWER
Strings sind gleich !
Vergleich von "haben" und "hat" liefert : -18
```

Die Klasse `StringBuffer` in Java (1)

• Allgemeines zur Klasse `StringBuffer`

◇ Thread-sichere Klasse zur Darstellung **veränderlicher Strings**.
Die in Objekten dieser Klasse abgelegte Zeichenfolge kann sowohl bezüglich des **Inhalts** als auch hinsichtlich der **Länge modifiziert** werden.

◇ Bestandteil des **Package `java.lang`**

◇ Die Größe des in einem `StringBuffer`-Objekt vorhandenen Zeichen-Puffers wird als **Kapazität** bezeichnet.
Die Länge des enthaltenen Strings kann kleiner als die Kapazität sein.
Übersteigt bei einer String-Verlängerung die neue Länge die Kapazität, wird automatisch ein neuer Puffer ausreichender Kapazität allokiert. Die **Kapaziät** kann auch **explizit vergrößert** (nicht jedoch verkleinert) werden.

◇ Viele der **Memberfunktionen** der Klasse **`String`** existieren **auch** für die Klasse **`StringBuffer.`**
Zusätzlich sind zahlreiche Memberfunktionen zur **String-Manipulation** implementiert.

◇ Die Klassen **`String`** und **`StringBuffer`** sind **nicht voneinander abgeleitet**.
Beide **implementieren** aber das Interface **`CharSequence.`**
Die Klasse `StringBuffer` **implementiert** darüberhinaus das Interface **`Appendable.`**

◇ Die Klasse **`StringBuffer`** ist nicht ableitbar (Die Klasse ist **`final`**).

• Konstruktoren der Klasse `StringBuffer`

`public` **`StringBuffer()`**	Erzeugt ein **leeres `StringBuffer`-Objekt** mit **Kapazität** von **`16`**
`public` **`StringBuffer(int cap)`**	Erzeugt ein **leeres `StringBuffer`-Objekt**, mit **Kapazität** = **`cap`**
`public` **`StringBuffer(String str)`**	Erzeugt ein **`StringBuffer`-Objekt**, das mit dem Inhalt von `str` initialisiert ist, mit **Kapazität** = `16 + str.length()`
`public` **`StringBuffer`**`(`**`CharSequence seq`**`)` (ab JDK 5.0)	Erzeugt ein **`StringBuffer`-Objekt**, das mit den in **`seq`** enthaltenen Zeichen initialisiert ist, mit **Kapazität** = `16 +` Anzahl Zeichen in `seq`

• Memberfunktionen zur Ermittlung und Änderung der Länge und Kapazität

◇
```
public int length()
```

▻ Gibt die **Länge** des im `StringBuffer`-Objekt enthaltenen Strings als Funktionswert zurück

◇
```
public int capacity()
```

▻ Gibt die Kapazität des `StringBuffer`-Objekts als Funktionswert zurück

◇
```
public void setLength(int neulen)
```

▻ Setzen der Länge des im `StringBuffer`-Objekt enthaltenen Strings
▻ Falls `neulen <` akt. Stringlänge ist, wird der String entsprechend verkürzt (Kapazität bleibt gleich)
▻ Falls `neulen >` akt. Stringlänge ist, wird der String mit `'\u0000'`-Zeichen verlängert
▻ Erzeugung einer **`IndexOutOfBoundsException`**, wenn `neulen<0` ist

◇
```
public void ensureCapacity(int neucap)
```

▻ Sicherstellung, dass die Kapazität `>= neucap` ist (Kapazitätsverkleinerung nicht möglich)

Die Klasse `StringBuffer` in Java (2)

- **Anmerkungen zu den Memberfunktionen zur String-Modifikation**

◇
```
public void setCharAt(int idx, char ch)
```

- ▻ Das **Zeichen** an der Position **`idx`** wird gleich dem Zeichen **`ch`** gesetzt.
- ▻ Erzeugung einer **`IndexOutOfBoundsException`**, wenn ein **unzulässiger Index** (`idx<0` oder `idx>=length()`) übergeben wird.

◇
```
public StringBuffer replace(int beg, int end, String str)
```

- ▻ Der durch **`beg`** (erster Index) und **`end`** (letzter Index +1) festgelegte **Teilstring** wird durch **`str`** **ersetzt**.
- ▻ Erzeugung einer **`StringIndexOutOfBoundsException`**, wenn ein **unzulässiger Index** (`beg<0` oder `beg>length()` oder `beg>end`) übergeben wird.
- ▻ Funktionswert ist eine Referenz auf das aktuelle `StringBuffer`-Objekt

◇
```
public StringBuffer deleteCharAt(int idx)
```

- ▻ **Entfernen** des **Zeichens** an der Position **`idx`** (Verkürzung der Stringlänge um `1`)
- ▻ Erzeugung einer **`StringIndexOutOfBoundsException`**, wenn eine **unzulässige Position** (`idx<0` oder `idx>=length()`) übergeben wird.
- ▻ Funktionswert ist eine Referenz auf das aktuelle `StringBuffer`-Objekt

◇
```
public StringBuffer delete(int beg, int end)
```

- ▻ **Entfernen** des durch **`beg`** (erster Index) und **`end`** (letzter Index +1) festgelegten **Teilstrings**
- ▻ Erzeugung einer **`StringIndexOutOfBoundsException`**, wenn ein **unzulässiger Index** (`beg<0` oder `beg>length()` oder `beg>end`) übergeben wird.
- ▻ Funktionswert ist eine Referenz auf das aktuelle `StringBuffer`-Objekt

◇
```
public StringBuffer reverse()
```

- ▻ Umkehrung der Zeichen-Reihenfolge des im `StringBuffer`-Objekt enthaltenen Strings
- ▻ Funktionswert ist eine Referenz auf das aktuelle `StringBuffer`-Objekt

◇ Die meist verwendetsten Memberfunktionen zur String-Modifikation sind :

```
public StringBuffer append(...)
public StringBuffer insert(int pos, ...)
```

- ▻ Beide Funktionen sind für **unterschiedliche Parametertypen** vielfach **überladen**.
- ▻ Sie **wandeln** den jeweiligen **Parameter** in seine **String-Darstellung** um und
  - hängen dann diesen String an das aktuelle Objekt an (`append()`) bzw
  - fügen ihn an der angegebenen Stelle `pos` ein (`insert()`).
- ▻ Als Funktionswert geben diese Funktionen eine Referenz auf das aktuelle Objekt zurück
- ▻ Bei Übergabe einer unzulässigen Position (`pos<0` oder `pos>=length()`) erzeugen die `insert()`-Funktionen eine **`IndexOutOfBoundsException`**
- ▻ **Beispiel** :
```
StringBuffer strb = new StringBuffer("Ergebnis : ");
int i = 0x7c;
strb.append(i);      // Inhalt von strb : "Ergebnis : 124"
```

Die Klasse `StringBuffer` in Java (3)

- **Erzeugung eines `String`-Objekts aus einem `StringBuffer`-Objekt**

  ◇ Mittels der **`StringBuffer`-Memberfunktion** :

  ```
  public String toString()
  ```

  ▻ Sie erzeugt ein neues `String`-Objekt, dass mit der aktuell im aktuellen `StringBuffer`-Objekt enthaltenen Zeichenfolge initialisiert ist und gibt eine Referenz auf dieses als Funktionswert zurück.

  ▻ Dieses `String`-Objekt ist die String-Repräsentation des `StringBuffer`-Objekts

  ▻ **Beispiel** :
  ```
  StringBuffer strb = new StringBuffer();
  // ... Manipulation von strb
  String str1 = strb.toString();
  ```

  ◇ Mittels des **`String`-Konstruktors** :

  ```
  public String(StringBuffer buf)
  ```

  ▻ **Beispiel** :
  ```
  // ...
  String str2 = new String(strb);
  ```

  ◇ Mittels der **statischen `String`-Memberfunktion** :

  ```
  public static String valueOf(Object obj)
  ```

  ▻ Dieser Funktion ist das `StringBuffer`-Objekt als Parameter zu übergeben.

  ▻ Sie erzeugt ein neues `String`-Objekt, dass mit der aktuell im übergebenen `StringBuffer`-Objekt enthaltenen Zeichenfolge initialisiert ist und gibt eine Referenz auf dieses als Funktionswert zurück.

  ▻ **Beispiel** :
  ```
  // ...
  String str3 = String.valueOf(strb);
  ```

- **Verwendung der Klasse `StringBuffer` durch den Compiler**

  ◇ Der Compiler realisiert eine **mehrfache String-Konkatenation** unter Verwendung eines temporären `StringBuffer`-Objekts.

  Beispiel :
  ```
  String s1 = "Morgenstund";
  String s2;
  s2 = '<' + s1 + '>';
  ```

  Der Ausdruck **`s2 = '<' + s1 + '>'`** wird vom Compiler umgesetzt in :

  ```
  s2 = new StringBuffer().append('<').append(s1).append('>').toString()
  ```

  Hier wird nur ein temporär benötigtes `StringBuffer`-Objekt erzeugt. Da sein Inhalt veränderlich ist, können die zu konkatenierenden Strings bzw Stringrepräsentationen angehängt werden. Eine explizite Erzeugung von `String`-Objekten für die Nicht-`String`-Operanden ist nicht erforderlich.

  ◇ Ohne das temporäre `StringBuffer`-Objekt müsste für jede Stringrepräsentation eines Nicht-`String`-Operanden ein neues `String`-Objekt erzeugt werden. Ausserdem würde nach jedem Zwischenschritt (jeder ausgeführten Konkatenation) zusätzlich ein weiteres `String`-Objekt erzeugt werden.
  Diese zusätzlichen `String`-Objekte werden nach Abschluß der Gesamtoperation nicht mehr benötigt.
  → unnötige Speicherallokationen (und danach wieder durchgeführte Speicherfreigaben).

  Im obigen Beispiel würden erzeugt werden :
  `"<"`, `"<Morgenstund"`, `">"` und **`"<Morgenstund>"`.**
  Nur das letzte `String`-Objekt wird letzten Endes benötigt.

Die Klasse `StringBuilder` in Java

- **Allgemeines zur Klasse `StringBuilder`**

  ◇ Ab dem JDK 5.0 vorhandene alternative Klasse zur Darstellung **veränderlicher Strings**.
  Auch die in Objekten dieser Klasse abgelegte Zeichenfolge kann sowohl bezüglich des **Inhalts** als auch hinsichtlich der **Länge modifiziert** werden.

  ◇ Bestandteil des **Package `java.lang`**

  ◇ Die Klasse `StringBuilder` stellt prinzipiell die **gleiche Funktionalität** bei im wesentlichen **gleichem Methoden-Interface** wie die Klasse **`StringBuffer`** zur Verfügung.

  ◇ Der wesentliche Unterschied zwischen den beiden Klassen besteht darin, dass **`StringBuffer` thread-sicher** ist, während dies für **`StringBuilder` nicht** zutrifft.
  Andererseits arbeiten die Methoden der Klasse **`StringBuilder` schneller**, da in ihnen auf die für eine Thread-Sicherheit notwendige Synchronisation verzichtet werden kann.
  Wenn **sichergestellt** ist, dass der veränderliche String **nur in einem Thread verwendet** werden wird, sollte der Klasse **`StringBuilder`** der **Vorzug** gegeben werden.

  ◇ Die Klasse **`StringBuilder`** steht in **keinerlei Ableitungsbeziehungen** zu den Klassen **`StringBuffer`** und **`String`**.
  Allerdings **implementiert** sie ebenfalls die **Interfaces `CharSequence`** und **`Appendable.`**

  ◇ Die Klasse `StringBuilder` ist ebenfalls **`final`** und damit **nicht ableitbar**.

- **Konstruktoren der Klasse `StringBuilder`** (analog zur Klasse `StringBuffer`)

`public` **`StringBuilder()`**	Erzeugt ein **leeres `StringBuilder`-Objekt** mit **Kapazität** von **`16`**
`public` **`StringBuilder(int cap)`**	Erzeugt ein **leeres `StringBuilder`-Objekt**, mit **Kapazität** = **`cap`**
`public` **`StringBuilder(String str)`**	Erzeugt ein **`StringBuilder`-Objekt**, das mit dem Inhalt von `str` initialisiert ist, mit **Kapazität** = `16 + str.length()`
`public` **`StringBuilder`**`(`**`CharSequence seq`**`)`	Erzeugt ein **`StringBuilder`-Objekt**, das mit den in **`seq`** enthaltenen Zeichen initialisiert ist, mit **Kapazität** = `16 +` Anzahl Zeichen in `seq`

- **Memberfunktionen der Klasse `StringBuilder`**

  Die Memberfunktionen entsprechen den Memberfunktionen der Klasse `StringBuffer`

- **Erzeugung eines `String`-Objekts aus einem `StringBuilder`-Objekt**

  Die Möglichkeiten entsprechen denen der Klasse `StringBuffer`

  ◇ Mittels der **`StringBuilder`-Memberfunktion** :

```
public String toString()
```

  ◇ Mittels des **`String`-Konstruktors** :

```
public String(StringBuilder bld)
```

  ◇ Mittels der **statischen `String`-Memberfunktion** :

```
public static String valueOf(Object obj)
```

Arrays in Java (1)

• Allgemeine Eigenschaften

◇ Arrays sind eine **Zusammenfassung** von **Variablen**, die in einer **Reihenfolge** geordnet sind und als **Elemente** bezeichnet werden.´
Alle Array-Elemente müssen vom **gleichen Typ** sein.

◇ Als **Element-Typ** ist **jeder beliebige Typ** (einfacher Datentyp oder Referenz-Typ) zulässig.

◇ Ein **Array-Element** wird über einen **Index**, der vom Typ `int` sein muß, ausgewählt.
Der Index des **ersten Elements** hat den Wert `0.`

◇ **Arrays** sind in Java **Objekte.**
Array-Typen sind – namenlose – **Klassen**.
Von einer Array-Klasse können **keine** weiteren **Klassen abgeleitet** werden.

◇ **Array-Variable** sind **Referenz-Variable**. Sie **verweisen** auf **Array-Objekte**.
→ Die Definition einer Array-Variablen erzeugt noch kein Array-Objekt, d.h. sie alloziert keinen Speicherplatz für die Array-Elemente.

◇ Eine **Array-Variable** legt **nur den Element-Typ** , **nich**t aber die **Länge eines Arrays** (d.h. die Anzahl seiner Elemente) fest. Sie kann daher **auf Arrays beliebiger Länge** zeigen.

◇ Die **Länge eines Arrays** wird **dynamisch** bei seiner **Erzeugung** (z.B. mittels eines `new`−Ausdrucks) festgelegt. Anschliessend ist sie **nicht** mehr **veränderbar**.

◇ Ein Array kann auch die **Länge** `0` besitzen. Ein derartiges Array ist ein echtes Array-Objekt.
Sinnvoll können derartige Arrays beispielsweise als **Rückgabewert von Funktionen** (genauer : Referenz darauf) auftreten.

◇ **Jedes Array** besitzt die **öffentliche Datenkomponente**

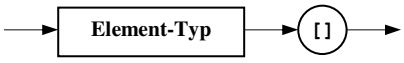
```
public final int length;
```

Sie enthält die **Länge des Arrays**.

◇ Ein **Array-Zugriff** wird zur **Laufzeit** auf **Zulässigkeit überprüft**.
Der Versuch des Zugriffs zu einer **nicht existierenden Array-Komponente** (ungültiger Index) führt zum Werfen der Exception **`ArrayIndexOutOfBoundsException.`**

• Vereinbarung von Array-Variablen

◇ **Angabe eines Array-Typs** :



◇ Diese Typangabe ist bei der Vereinbarung von Array-Variablen zu verwenden.

Beispiele :
```
int[] ia;          // ia kann auf Array-Objekte zeigen, deren Elemente int−Werte sind
String[] names;    // names kann auf Array-Objekte zeigen, deren Elemente Referenzen
                   // auf String−Objekte sind
```

◇ **Element-Typ** eines Arrays kann wiederum ein **Array** sein → **mehrdimensionale Arrays.**
Die Vereinbarung mehrdimensionaler Arrays erfolgt analog zu eindimensionalen Arrays.

Beispiele :
```
double[][] kmat;   // kmat kann auf Array-Objekte zeigen, deren Elemente Referenzen auf
                   // Array-Objekte sind, die double−Werte enthalten
String[][] seite;  // seite kann auf Array-Objekte zeigen, deren Elemente Referenzen
                   // auf Array-Objekte sind, die Referenzen auf String−Objekte enthalten.
```

Arrays in Java (2)

• Definition von Array-Objekten

◇ Definition von Array-Objekten mittels eines **`new`-Ausdrucks**.
Dabei ist die **Länge des Arrays** explizit anzugeben.
Die einzelnen Elemente werden mit ihrem **Defaultwert** initialisiert.

Beispiele :
```
ia = new int[10];          // Speicherallokation für ein Array-Objekt, das 10 int-
                           // Elemente hat. Alle Elemente sind mit 0 initialisiert.

names = new String[3];     // Speicherallokation für ein Array-Objekt, das 3 Elemente hat,
                           // die Referenzen auf String-Objekte sind.
                           // Alle Elemente sind mit null initialisiert.
```

◇ Die **Definition eines Array-Objekts** mittels eines `new`-Ausdrucks kann natürlich auch **zusammen** mit der **Vereinbarung** einer Array-Variablen erfolgen.

Beispiel :
```
float[] pol = new float[6];  // pol zeigt auf ein alloziertes float-Array mit
                             // 6 Elementen (die alle mit 0.0 initialisiert sind)
```

◇ Bei **mehrdimensionalen Arrays** kann im `new`-Ausdruck die **Länge aller Dimensionen** angegeben werden.
Es **reicht** aber aus, nur die **Länge** der **ersten** (am weitesten links stehenden) **Dimension** festzulegen. Die Längen der übrigen Dimensionen müssen dann später festgelegt werden.

Beispiel :
```
kmat = new double[5][4];   // Speicherallokation für ein Array-Objekt, das 5 Elemente hat,
                           // die auf gleichzeitig allozierte double-Arrays, die alle
                           // 4 Elemente haben, zeigen.
                           // Die Elemente der double-Arrays sind mit 0.0 initialisiert
```

Das obige Beispiel ist eine **abkürzende Schreibweise** für die **folgende ausführlichere Formulierung** :
```
kmat = new double[5][];    // Speicherallokation für ein Array-Objekt, das 5 Elemente hat,
                           // die auf double-Arrays zeigen können, aber alle mit null
                           // initialisiert sind.
for (int i=0; i<kmat.length; i++)
  kmat[i] = new double[4]; // Speicherallokation für 5 double-Arrays der Länge 4.
                           // Alle Elemente dieser Arrays sind mit 0.0 initialisiert.
```

Als Vorteil der Java-Implementierung von mehrdimensionalen Arrays ergibt sich, dass die **Element-Arrays** (Arrays in **zweiter** (und gegebenenfalls höherer) **Dimension** eine **unterschiedliche Länge** besitzen können.

◇ Definition von Array-Objekten mittels eines **Array-Initialisierers** (*array initializer*).

Syntax :

Ein Array-Initialisierer kann **bei der Vereinbarung** einer **Array-Variablen** angegeben werden.
Er bewirkt die **Allokation von Speicherplatz** für ein **Array-Objekt** und **initialisiert** die **einzelnen Elemente** mit den angegebenen Initialisierungs-Ausdrücken.
Die **Länge des Arrays** wird **nicht angegeben** .Sie ergibt sich aus der **Anzahl der Initialisierungswerte**

Beispiel :
```
String[] farben = { "rot", "gruen", "blau", "gelb", "weiss" };
// Vereinbarung der Array-Variablen farben. Diese zeigt auf ein gleichzeitig alloziertes Array-
// Objekt, dessen 5 Elemente mit den Referenzen auf die angegeben Farb-Strings, für die ebenfalls
// Speicher alloziert wird, initialisiert werden.
```

Bei **mehrdimensionalen Arrays** können **Array-Initialisierer geschachtelt** werden (Initialisierungs-Ausdrücke sind wiederum Array-Initialisierer)

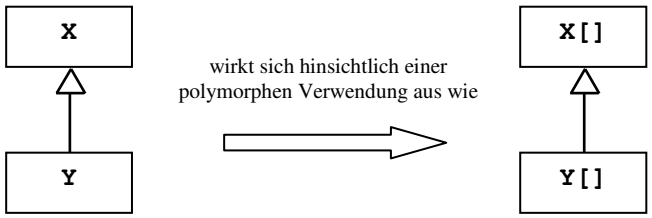
Array-Initialisierer können auch **zusammen** mit einem **`new`-Ausdruck** verwendet werden. Sie sind dem `new`-Ausdruck **nachzustellen**, eine **Array-Länge** darf **nicht angegeben** werden. (sinnvoll zur Erzeugung **anonymer Arrays**)

Arrays in Java (3)

• Arrays und Polymorphie

◇ Obwohl **Array-Typen nicht ableitbar** sind, kann auf sie – sofern es sich um Arrays von Referenz-Typen handelt – **Polymorphie** angewendet werden.

◇ Gegeben seien zwei Klassen **`X`** und **`Y`.**
Die Klasse **`Y`** sei **von** der Klasse **`X`** **abgeleitet**.
Die durch **diese Vererbungsbeziehung** definierte **Polymorphie** wird **auch auf Arrays** von Objekten dieser Klassen **übertragen**.
Bezüglich der **Polymorphie** verhalten sich die Klassen **`Y[]`** und **`X[]`** also so, als ob sie voneinander abgeleitet wären.



Einer **Array-Variablen** vom Typ **`X[]`** kann dadurch auch die Referenz auf ein **`Y[]`−Objekt zugewiesen** werden.

Beispiel :
```
Y[] yArr = new Y[3];
X[] xArr = yArr;        // zulässig, da Y an X zuweisbar ist.
```

◇ **Achtung** : Den Elementen eines Referenz-Typ-Arrays dürfen **nur Referenzen auf solche Objekte** zugewiesen werden, die **auch Variablen seines tatsächlichen Element-Typs** zugewiesen werden können.
Im obigen Beispiel dürfen den Elementen von `xArr` daher nur Referenzen auf `Y`−Objekte zugewiesen werden.

Beispiel :
```
xArr[0] = new Y();      // zulässig, da einer Y−Variablen eine Referenz auf ein Y−Objekt
                        // zugewiesen werden kann
xArr[1] = new X();      // unzulässig, da einer Y−Variablen keine Referenz auf ein X−Objekt
                        // zugewiesen werden darf
```

◇ Die **Zulässigkeit** der Zuweisung an Elemente eines Referenz-Typ-Arrays wird zur **Laufzeit überprüft**.
Beim Versuch einer unzulässigen Zuweisung wird eine Exception vom Typ **`ArrayStoreException`** geworfen.

• Unterstützende Klassen der Standard-Bibliothek

◇ In der Java-Standardbibliothek sind **zwei Klassen** definiert, die **statische Memberfunktionen** zur **Unterstützung** der **Verwendung von Arrays** zur Verfügung stellen
Beide Klassen können **nicht instanziiert** werden.

◇ Klasse **`java.lang.reflect.Array`**
Sie stellt statische Methoden zur dynamischen **Erzeugung** von und zum **Zugriff** zu Arrays zur Verfügung.

◇ Klasse **`java.util.Arrays`**
Sie stellt statische Methoden zur **Bearbeitung von Arrays** zur Verfügung.
Im wesentlichen handelt es sich um Methoden
- zum Suchen in Arrays
- zum Sortieren von Array-Komponenten
- zum Vergleich von Arrays
- zur Zuweisung an Array-Bereiche

◇ Die Klasse **`java.lang.System`** definiert zum **Kopieren von Arraybereichen** die statische Methode

```
public static void arraycopy(Object src, int sPos, Object dest, int dPos, int len)
```

Quell-Objekt (`src`) und Ziel-Objekt (`dest`) müssen Arrays sein

Demonstrationsprogramm zu Arrays in Java

- **Quellcode des Programms (Klasse `ArrayDemo`)**

```
public class ArrayDemo
{
  public static void main(String[] args)
  {
    int[] ia;
    ia = new int[6];
    System.out.println("int-Array : " + ia);
    for (int i=0; i<ia.length; i++)
      System.out.println(i + " : " + ia[i]);
    String[] names = /* new String[]*/ { "Walter", "Franz", "Ilka" };
    System.out.println("String-Array :" + names);
    for (int i=0; i<names.length; i++)
      System.out.println(i + " : " + names[i]);
    double[][] kmat = new double[4][3];
    System.out.println("2-dimensionales double-Array : " + kmat);
    for (int i=0; i<kmat.length; i++)
      System.out.println(i + " : " + kmat[i]);
    System.out.println("erstes Zeilen-Array :");
    for (int i=0; i<kmat[0].length; i++)
      System.out.println(i + " : " + kmat[0][i]);
    System.out.println("weiteres 2-dimensionales double-Array :");
    double[][] gmat = new double[4][];
    for (int i=0; i<gmat.length; i++)
      gmat[i] = new double[i+1];
    for (int i=0; i<gmat.length; i++)
    { System.out.print(i + " : ");
      for (int j=0; j<gmat[i].length; j++)
        System.out.print(gmat[i][j]+ "  ");
      System.out.println();
    }
  }
}
```

- **Ausgabe des Programms**

```
int-Array : [I@108786b
0 : 0
1 : 0
2 : 0
3 : 0
4 : 0
5 : 0
String-Array :[Ljava.lang.String;@119c082
0 : Walter
1 : Franz
2 : Ilka
2-dimensionales double-Array : [[D@1add2dd
0 : [D@eee36c
1 : [D@194df86
2 : [D@defa1a
3 : [D@f5da06
erstes Zeilen-Array :
0 : 0.0
1 : 0.0
2 : 0.0
weiteres 2-dimensionales double-Array :
0 : 0.0
1 : 0.0  0.0
2 : 0.0  0.0  0.0
3 : 0.0  0.0  0.0  0.0
```

Array-Listen in Java

• Eigenschaften von Array-Listen

◇ Array-Listen sind Datenstrukturen, die die Eigenschaften von **Listen** und **Arrays** kombinieren.

◇ Sowohl in Listen als auch in Arrays sind die **Elemente** alle vom **gleichen Typ**.
In beiden Strukturarten sind die Elemente in einer **linearen Reihenfolge** angeordnet und damit damit über eine **Positionsangabe** referierbar.
Der Zugriff zu einem Array-Element erfolgt dabei mittels Indizierung, die **Zugriffszeit** ist unabhängig von der Position.
Der Zugriff zu einem Listen-Element erfolgt über eine Zugriffsfunktion, der die Position (Index) zu übergeben ist.
Die Zugriffszeit hängt i.a. von der Position ab.

◇ Die **Größe** eines Arrays (d.h. die Anzahl der enthaltenen Elemente) ist i.a. feststehend und nicht veränderbar, während die Größe einer Liste variabel ist und jederzeit – durch das Einfügen oder Entfernen von Elementen – geändert werden kann.

◇ **Array-Listen** implementieren im Prinzip in der Größe **veränderliche Arrays** :
- Bezüglich der Zugriffszeit zu einem Element verhalten sie sich wie Arrays (unabhängig von der Position)
- Bezüglich der Elementeanzahl verhalten sie sich wie Listen, d.h. es können Elemente entfernt und hinzugefügt werden
- Die Elemente werden über einen Index (`int`-Wert) referiert, das erste Element hat den Index 0.
  Der Zugriff erfolgt über eine Zugriffsfunktion.

• Realisierung von Array-Listen in Java

◇ Array-Listen werden in Java durch Objekte der Klasse **`ArrayList`** realisiert.
Für diese Klasse gelten – im Gegensatz zu Arrays (namenlose Klassen !) – keine syntaktischen Besonderheiten.
Array-Listen-Objekte werden mittels eines `new`-Ausdrucks erzeugt – mit gleicher Syntax wie für andere benamte Klassen.

◇ Array-Listen-Objekte können sich – wie Array-Objekte – im Element-Typ unterscheiden.
→ Es gibt unterschiedliche **Array-Listen-Typen**.
→ Die Klasse `ArrayList` ist eine **generische Klasse** (genaueres zu generischen Klassen s. später).
Generische Klassen besitzen eine (oder mehrere) **Typ-Parameter**.
Bei der Verwendung einer derartigen Klasse sind i.a. ein aktuelle Typ-Parameter anzugeben. Dadurch wird jeweils ein **parameterisierter Typ** definiert.
Im vorliegenden Fall ist der Typ-Parameter der Typ der Array-Listen-Elemente.
→ Die verschiedenen Array-Listen-Typen sind also parameterisierte Typen, die durch die Klasse `ArrayList` definiert werden.

◇ Die exakte Klassenbezeichnung lautet **`ArrayList<E>.`**
`E` ist der formale Typ-Parameter.
Bei der Verwendung der Klasse (z.B. zur Objekterzeugung) ist `E` durch den jeweiligen Element-Typ als aktueller Typ-Parameter zu ersetzen, dieser muss ein Referenz-Typ sein (Unterschied zu Arrays !)

Beispiel :

```
ArrayList<String> salist = new ArrayList<String>();
```

`ArrayList<String>` ist die Typ-Bezeichnung für "ArrayListe mit `String`-Elementen"
→ Erzeugung eines Array-Listen-Objekts mit `String`-Elementen

◇ **Anmerkung zur Implementierung** :
Ein Array-Listen-Objekt **kapselt** ein **Array seines Element-Typs**. Dieses Array besitzt eine Größe, die größer gleich der Anzahl der Array-Listen-Elemente ist. Sie wird als **Kapazität** der Array-Liste bezeichnet.
Solange die Anzahl der Array-Listen-Elemente kleiner als die Kapazität ist, können problemlos neue Elemente hinzugefügt werden. Reicht die Kapazität zur Aufnahme eines neuen Elements nicht mehr aus, muss ein neues entsprechend größeres internes Array alloziert werden und der Inhalt des alten Arrays in das neue Array kopiert werden.
→ Zeitaufwand (→ sollte nicht zu häufig auftreten).

Die Klasse `ArrayList<E>` in Java (1)

- **Allgemeines**
  - ◇ Die generische Klasse **`ArrayList<E>`** befindet sich im Package `java.util`. Sie ist Bestandteil des *Collection Frameworks*.
  - ◇ Sie dient zur Realisierung von **Array-Listen**
  - ◇ Sie ist von der Klasse `AbstractList<E>` abgeleitet und implementiert die Interfaces `List<E>, Cloneable, Serializable. RandomAccess, Iterable<E>` und `Collection<E>`.

- **Konstruktoren der Klasse `ArrayList<E>`** (Auswahl)

`public ArrayList<E>`	Erzeugt ein leeres `ArrayList`-Objekt für Elemente des Typs `E`. Das Objekt hat eine Anfangs-Kapazität von 10.
`public ArrayList<E>(int initCap)`	Erzeugt ein leeres `ArrayList`-Objekt für Elemente des Typs `E`. Das Objekt hat eine Anfangs-Kapazität von `initCap`

- **Memberfunktionen der Klasse `ArrayList<E>`** (Auswahl)
  - ◇
    ```
    public boolean add(E elem)
    ```
    - ▻ Einfügen des Elements `elem` vom Typ `E` an das Ende der Array-Liste
    - ▻ Funktionswert : `true`
  - ◇
    ```
    public void add(int index, E elem)
    ```
    - ▻ Einfügen des Elements `elem` vom Typ `E` an der Position `index`.
    - ▻ Ein an der Position `index` vorhandenes Element und alle danach kommenden Elemente werden um eine Position weitergeschoben
    - ▻ Liegt `index` ausserhalb der Liste, wird eine `IndexOutOfBoundsException` geworfen
  - ◇
    ```
    public boolean remove(Object elem)
    ```
    - ▻ Entfernen des ersten Auftritts des Elements `elem`, dieses muß tatsächlich vom Typ `E` sein (Die Klasse `Object` ist Basisklasse für alle anderen Klassen in Java, jedes beliebige Element eines Refernztyps – d.h. jedes beliebige Objekt – kann daher als Instanz der Klasse `Object` betrachtet werden).
    - ▻ Alle nach dem entfernten Element enthaltenen Elemente werden um eine Position nach vorn geschoben.
    - ▻ Funktionswert : `true`, wenn das Element `elem` enthalten war,
      `false`, wenn das Element `elem` nicht enthalten war
  - ◇
    ```
    public E remove(int index)
    ```
    - ▻ Entfernen des Elements an der Position `index`.
    - ▻ Alle nach dem entfernten Element enthaltenen Elemente werden um eine Position nach vorn geschoben.
    - ▻ Funktionswert : das entfernte Element
    - ▻ Liegt `index` ausserhalb der Liste, wird eine `IndexOutOfBoundsException` geworfen
  - ◇
    ```
    public void clear()
    ```
    - ▻ Entfernen aller Elemente aus der Liste, die Liste ist anschliessend leer.

Die Klasse `ArrayList<E>` in Java (2)

- **Memberfunktionen der Klasse `ArrayList<E>`** (Auswahl), Forts.

◇
```
public E get(int index)
```

- ▻ Auslesen des Elements an der Position `index`. (Die Liste bleibt unverändert)
- ▻ Funktionswert : das Element an der Position `index`.
- ▻ Liegt `index` ausserhalb der Liste, wird eine `IndexOutOfBoundsException` geworfen

◇
```
public E set(int index, E elem)
```

- ▻ Ersetzen des an der Position `index` befindlichen Elements durch das Element `elem`.
- ▻ Funktionswert : das Element, das sich zuvor an der Position `index` befunden hat..
- ▻ Liegt `index` ausserhalb der Liste, wird eine `IndexOutOfBoundsException` geworfen

◇
```
public void ensureCapacity(int minCap)
```

- ▻ Sicherstellung, dass die Kapazität mindestens `minCap` beträgt (falls erforderlich wird die Kapazität erhöht)

◇
```
public int size()
```

- ▻ Ermittlung der Array-Listen-Größe (== Anzahl der enthaltenen Elemente)
- ▻ Funktionswert : Anzahl der enthaltenen Elemente

◇
```
public E[] toArray(E[] arr)
```

- ▻ Übertragung des Listeninhalts – unter Beibehaltung der Element-Reihenfolge – in ein Array.
- ▻ Falls das als Parameter übergebene Array `arr` groß genug ist, wird der Listeninhalt in dieses Array übertragen, falls es zu klein ist, wird ein neues Array alloziert und als Ziel-Array verwendet.
- ▻ Falls das als Parameter übergebene Array `arr` größer als die Array-Liste ist, werden die nicht belegten Array-Elemente gleich `null` gesetzt.
- ▻ Funktionswert : Array, in das die Listenelemente übertragen wurden.

◇
```
public Object[] toArray()
```

- ▻ Übertragung des Listeninhalts – unter Beibehaltung der Element-Reihenfolge – in ein neu alloziertes Array, dessen Elementtyp tatsächlich `E` ist.
- ▻ Funktionswert : Array, in das die Listenelemente übertragen wurden.

- **Vergleich der Verwendung von Arrays und Arry-Listen**

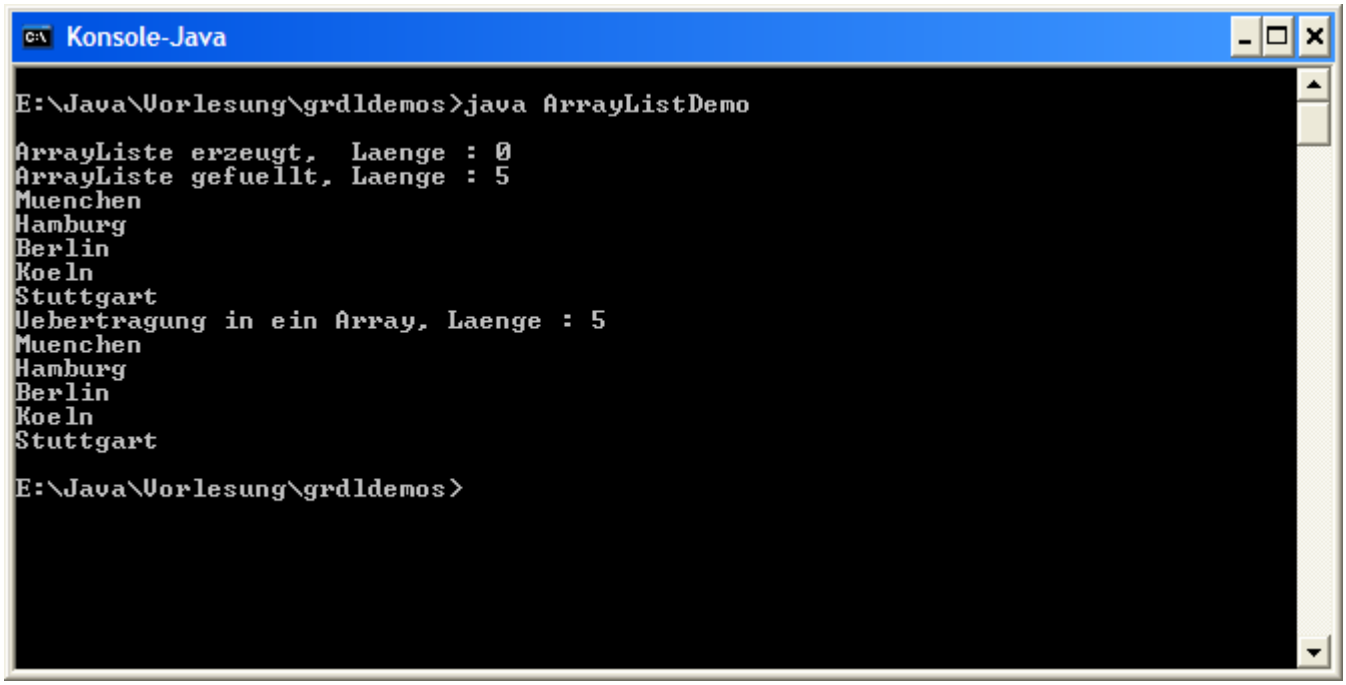
	Array	**Array-Liste**
Variablen-Definition	`String[] a;`	`ArrayList<String> a;`
Objekterzeugung	`a = new String[50];`	`a = new ArrayList<String>();`
Schreiben	`a[i] = "Muenchen";`	`a.set(i, "München");`
Lesen	`String s = a[i];`	`String s = a.get(i);`
Größenermittlung	`int len = a.length;`	`int len = a.size();`

Demonstrationsprogramm zu ArrayListen in Java

- **Quellcode des Programms (Klasse `ArrayListDemo`)**

```
// ArrayListDemo.java

import java.util.*;

public class ArrayListDemo
{
  public static void main(String[] args)
  {
    ArrayList<String> al = new ArrayList<String>();
    System.out.println();
    System.out.println("ArrayListe erzeugt,  Laenge : " + al.size());
    al.add("Muenchen");
    al.add("Hamburg");
    al.add("Berlin");
    al.add("Koeln");
    al.add("Stuttgart");
    System.out.println("ArrayListe gefuellt, Laenge : " + al.size());
    for (int i=0; i<al.size(); ++i)
      System.out.println(al.get(i));
    String[] sa = new String[0];
    sa = al.toArray(sa);
    System.out.println("Uebertragung in ein Array, Laenge : " + sa.length);
    for (int i=0; i<sa.length; ++i)
      System.out.println(sa[i]);
  }
}
```

- **Probelauf des Programms**



Die vereinfachte `for`-Anweisung in Java

• Iteration über alle Elemente eines Arrays (und eines Containers)

◇ Zum sequentiellen Durchlaufen der Elemente eines Arrays wird üblicherweise die – auch aus C/C++ bekannte – (Standard-) **`for`-Anweisung** eingesetzt.
Zum Zugriff zu den Elementen erfordert sie die **explizite Definition** und Verwendung einer "**Laufvariablen**".

◇ **Beispiel** :

```
double[] da;
// ...
for (int i=0; i<da.length; i++)
  System.out.println(da[i]);
```

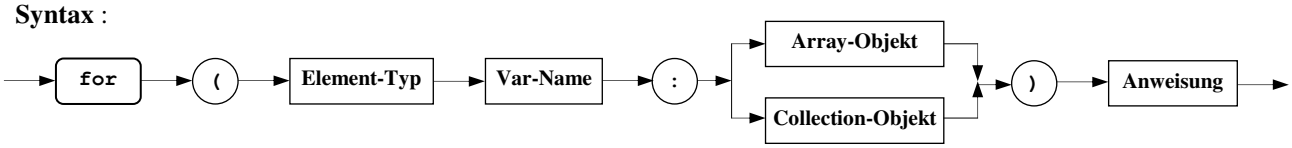
◇ Analoges gilt für das Durchlaufen eines Containers (in Java Collection genannt).
Anstelle der Laufvariablen tritt hier ein **Iterator-**Objekt.

• Vereinfachte `for`-Anweisung (Enhanced `for` Loop, For-Each-Schleife)

◇ Sie wurde mit dem JDK 5.0 zum sequentiellen Durchlaufen **aller Elemente** eines Arrays (bzw einer Collection) eingeführt

◇ Bei ihr wird auf die **explizite Definition** einer **Laufvariablen** (bzw eines Iterator-Objekts) **verzichtet**.

◇ **Syntax** :



◇ **Beispiel** :

```
double[] da;
// ...
for(double el : da)
  System.out.println(el);
```

◇ **Wirkung** : "**Für jedes Element `el` in `da` **"
Der Element-Typ-Variablen `el` wird in jedem Schleifendurchlauf das jeweils nächste Element des Arrays `da` zugewiesen.

◇ Die Gültigkeit der Element-Typ-Variablen `el` ist auf den Schleifenrumpf begrenzt.

◇ Der vom Compiler erzeugte Code benötigt und verwaltet auch hier für den Elementzugriff eine **Laufvariable** (bzw ein Iterator-Objekt). Diese wird vom Compiler **implizit** erzeugt.

• Anwendbarkeit der vereinfachten `for`-Anweisung

◇ Sie ist **kein allgemeiner Ersatz** für die (Standard-) `for`-Anweisung

◇ Sie kann nur zum **sequentiellen** Durchlaufen **aller Elemente** eines Arrays (bzw einer Collection) in **Vorwärtsrichtung** eingesetzt werden (vom ersten zum letzten Element, ohne Überspringen einzelner Elemente)

◇ Sie kann **nicht eingesetzt** werden, wenn

- ▻ ein Element aus dem Array (der Collection) **entfernt** werden soll
- ▻ ein Element innerhalb des Arrays (der Collection) **modifiziert** werden soll
- ▻ innerhalb einer Schleife **mehrere** Arrays (Collections) **parallel** bearbeitet werden sollen

◇ Ihr Einsatz in **geschachtelten Schleifen** ist jedoch **möglich**

Beispiel : Durchlaufen eines zweidimensionalen Arrays

```
double[][] gmat;
// ...
for (double[] row : gmat)
  for (double el : row)
    System.out.println(el);
```

Demonstrationsprogramm zur vereinfachten `for`-Anweisung in Java

- **Quellcode des Programms (Klasse `ForEachDemo`)**

```
// ForEachDemo.java

public class ForEachDemo
{
  public static void main(String[] args)
  {
    double[] da;
    da = new double[6];
    for (int i=0; i<da.length; i++)
      da[i] = i;
    System.out.println("\ndouble-Array (eindimensional) : ");
    for(double el : da)
      System.out.println(el);

    String[] names = { "Walter", "Franz", "Ilka" };
    System.out.println("\nString-Array : ");
    for (String str : names)
      System.out.println(str);

    System.out.println("\ndouble-Array (zweidimensional) :");
    double[][] gmat = new double[4][];
    for (int i=0; i<gmat.length; i++)
      gmat[i] = new double[i+1];
    for (double[] row : gmat)
    { for (double el : row)
        System.out.print(el + "   ");
      System.out.println();
    }
  }
}
```

- **Ausgabe des Programms**

```
double-Array (eindimensional) :
0.0
1.0
2.0
3.0
4.0
5.0

String-Array :
Walter
Franz
Ilka

double-Array (zweidimensional) :
0.0
0.0   0.0
0.0   0.0   0.0
0.0   0.0   0.0   0.0
```

Die Klasse `Object` in Java (1)

• "Mutter" aller Klassen

◇ In Java existiert **nur eine Klassenhierarchie**.
Wurzel dieser Hierarchie ist die Klasse **`Object.`**
→ Alle Klassen haben – direkt oder indirekt – diese Klasse als Basisklasse.
Auch Klassen, die **scheinbar nicht abgeleitet** sind, d.h. die ohne Angabe einer Basisklasse definiert sind, sind tatsächlich **implizit** von **`Object`** **abgeleitet**.

◇ Da eine Variable eines Klassen-Typs auch auf Objekte abgeleiteter Klassen zeigen kann (Laufzeit-Polymorphie !), kann eine **Variable** vom Typ **`Object`** Objekte eines **beliebigen Klassen-Typs** (einschliesslich Array-Typs) **referieren**.

◇ Die Klasse **`Object`** ist im **Package `java.lang`** enthalten.

• Memberfunktionen der Klasse `Object`

◇ Die Klasse `Object` definiert insgesamt **elf Memberfunktionen**, die wegen der Vererbung in allen Klassen zur Verfügung stehen.

◇ **Sechs** dieser Memberfunktionen sind **nicht überschreibbar**.
Die **übrigen** lassen sich in abgeleiteten Klassen **überschreiben**. In Abhängigkeit von der jeweiligen Klasse kann für eine sinnvolle Funktionalität ein Überschreiben notwendig sein.

◇ Die Memberfunktionen von `Object` zerfallen in **zwei Kategorien** :
- Allgemeine **Utility-Funktionen**
- Methoden zur **Thread-Unterstützung**

Hier werden nur kurz die **Utilility-Funktionen** vorgestellt.
Auf die Methoden zur **Thread-Unterstützung** wird **später** im Rahmen der Besprechung von Threads eingegangen.

◇
```
public boolean equals(Object obj)
```

▻ Die Funktion vergleicht das aktuelle Objekt mit dem durch `obj` referierten Objekt.
Sie liefert `true` als Funktionswert, wenn Gleichheit vorliegt, andernfalls **`false.`**

▻ Grundsätzlich ist die Funktion für eine Überprüfung auf **Wert-Gleichheit** der Objekte vorgesehen.

▻ Die **Default-Implementierung** in der Klasse `Object` geht davon aus, dass ein Objekt **nur zu sich selbst gleich** sein kann, d.h. sie setzt Wert-Gleichheit mit **Referenz-Gleichheit** (Identität) gleich.
Sie liefert genau dann `true,` wenn `obj` das aktuelle Objekt referiert (**`obj==this`**).

▻ Für die Implementierung einer echten Wert-Gleichheit, die sich auf den Inhalt (Zustand) der referierten Objekte bezieht, muss die Methode in abgeleiteten Klassen geeignet **überschrieben** werden.
Dies ist für zahlreiche Klassen der Standard-Bibliothek, u.a. auch für die Klasse `String,` erfolgt.

◇
```
public int hashCode()
```

▻ Die Funktion liefert einen **Hash-Code** für das aktuelle Objekt. Jedes Objekt besitzt einen derartigen Hash-Code.
Er ermöglicht die Speicherung von Objekten in Hash-Tabellen (z.B. Klasse `java.util.Hashtable`).

▻ Der Hash-Code eines Objekts darf sich während der Ausführung einer Java-Applikation nicht ändern.
Bei unterschiedlichen Ausführungen derselben Applikation kann er dagegen unterschiedlich sein.

▻ Die **Default-Implementierung** in der Klasse `Object` liefert für **unterschiedliche Objekte unterschiedliche Hash-Codes** (Typischerweise ist dieser gleich der in einen `int`−Wert umgewandelten Speicheradresse des Objekts).

▻ Wenn für die zwei durch **`x`** und **`y`** referierten Objekte **`x.equals(y)`** den Wert **`true`** liefert, muß der für die **beiden Objekte** erzeugte **Hash-Code** auch **gleich** sein (**`x.hashCode()==y.hashCode()`**)

▻ Daher ist es bei Überschreiben der Funktion `equals()` i.a. auch notwendig die Funktion `hashCode()` entsprechend zu überschreiben.

Die Klasse `Object` in Java (2)

- **Memberfunktionen der Klasse `Object`, Forts.**

◇ `protected Object clone() throws CloneNotSupportedException`

- ▻ Die Funktion liefert ein **neues Objekt**, dass ein Clone (eine **Kopie**) des aktuellen Objekts ist.
- ▻ Die tatsächliche Klasse des aktuellen Objekts muß das **Interface `Cloneable`** implementieren.
- ▻ Die **Default-Implementierung** in der Klasse `Object` prüft, ob für das aktuelle Objekt das **Interface `Cloneable`** implementiert ist.
  Ist das **nicht** der Fall, wird die Exception **`CloneNotSupportedException`** geworfen.
  Falls es implementiert ist, wird ein neues Objekt der Klasse erzeugt, dessen Datenkomponenten mit den Werten der entsprechenden Datenkomponenten des aktuellen Objekts initialisiert werden. (**"flache" Kopie**, *shallow copy*)
- ▻ Soll das **neue Objekt** als **"tiefe" Kopie** (*deep copy*) erzeugt werden, muss `clone()` geeignet überschrieben werden.
- ▻ Die Klasse **`Object`** selbst **implementiert** das Interface **`Cloneable` nicht**.
  Das bedeutet, dass der Aufruf von `clone()` für ein Objekt der Klasse `Object` zum Werfen der Exception `CloneNotSupportedException` führt.

◇ `public String toString()`

- ▻ Die Funktion liefert eine **String-Repräsentation** des aktuellen Objekts.
- ▻ Die **Default-Implementierung** in der Klasse `Object` erzeugt ein `String`–Objekt, dessen Inhalt aus dem Klassennamen des aktuellen Objekts, dem Zeichen `'@'` und der sedezimalen Darstellung seines Hash-Codes zusammengesetzt ist.
- ▻ Soll eine andere das Objekt kennzeichnende String-Darstellung erzeugt werden, muss die Methode `toString()` geeignet überschrieben werden
- ▻ Die Methode `toString()` wird immer dann **implizit** aufgerufen, wenn eine Objekt-Referenz in einem String-Konkatenations-Ausdruck auftritt.

◇ `public final Class<?> getClass()`

- ▻ Die Funktion liefert die **Instanz** der Klasse **`Class<T>`,** die die tatsächliche Klasse des aktuellen Objekts beschreibt. Der Typ-Parameter `T` steht für die repräsentierte Klasse (allgemein : für den repräsentiertenTyp)
- ▻ Die Klasse `Class` ist eine wesentliche Komponente der Reflection-Fähigkeit von Java (enthalten im Package `java.lang`). Objekte dieser Klasse repräsentieren andere Klassen (sowie Interfaces und die primitiven Datentypen) und stellen charakteristische Informationen über diese zur Verfügung.
  Sie können nicht explizit erzeugt werden (es gibt keinen öffentlichen Konstruktor), sondern werden durch die JVM beim Laden einer Klasse automatisch generiert.
- ▻ U.a. stellt die Klasse `Class` die Memberfunktion **`public String getName()`** zur Verfügung.
  Diese liefert den vollqualifizierten Namen der repräsentierten Klasse als String zurück.
- ▻ Die Methode **`getClass()`** kann **nicht überschrieben** werden.

◇ `protected void finalize() throws Throwable`

- ▻ Diese Funktion wird vom **Garbage Collector** für ein Objekt **aufgerufen**, für das keine Referenz mehr existiert und das anschliessend vernichtet werden soll.
- ▻ Der **Zweck** dieser Funktion liegt in der **Durchführung von Bereinigungsaufgaben** vor der endgültigen Objektvernichtung (z.B. **Freigabe von Resourcen**, wie z.B. I/O-Verbindungen oder **Schließen von Dateien** usw).
  Die Funktion kann somit als eine Art Ersatz für einen Destruktor aufgefasst werden.
- ▻ Die **Default-Implementierung** in der Klasse `Object` führt keinerlei Aktionen aus.
- ▻ Sollen für Objekte einer bestimmten Klasse spezielle Bereinigungsaktionen ausgeführt werden, muss die Funktion für diese Klasse überschrieben werden.

Die Klasse `Object` in Java (3)

- **Demonstrationsprogramm zu Memberfunktionen der Klasse `Object`**

```
// ObjTest.java

public class ObjTest implements Cloneable
{
  private String name;

  public ObjTest(String str)
  {
    name=str;
  }

  public String toString()
  {
    return super.toString()+" ("+name+")";
  }

  public static void main(String[] args)
  {
    ObjTest o1 = new ObjTest("Test1");
    System.out.println("o1.toString() : " + o1.toString());
    Object o2 = new ObjTest("Test2");
    System.out.println("o2             : " + o2);
    System.out.println("o1.hashCode() : " + o1.hashCode());
    System.out.println("o2.hashCode() : " + o2.hashCode());
    System.out.println("o2.equals(o1) : " + o2.equals(o1));
    o2=o1;
    System.out.println("\nnach o2=o1 :");
    System.out.println("o2             : " + o2);
    System.out.println("o2.equals(o1) : " + o2.equals(o1));
    try
    {
      o2=o1.clone();
    }
    catch (CloneNotSupportedException e)
    {
      System.out.println(e.toString());
    }
    System.out.println("\nnach o2=o1.clone() :");
    System.out.println("o2             : " + o2);
    System.out.println("o2.equals(o1) : " + o2.equals(o1));
  }
}
```

- **Ausgabe des Programms**

```
o1.toString() : ObjTest@119c082 (Test1)
o2            : ObjTest@1add2dd (Test2)
o1.hashCode() : 18464898
o2.hashCode() : 28168925
o2.equals(o1) : false

nach o2=o1 :
o2            : ObjTest@119c082 (Test1)
o2.equals(o1) : true

nach o2=o1.clone() :
o2            : ObjTest@eee36c (Test1)
o2.equals(o1) : false
```

Aufzählungstypen in Java (1)

- **Allgemeines**

  ◇ Aufzählungstypen wurden mit dem **JDK 5.0** in Java eingeführt

  ◇ Sie ermöglichen die **Definition von Wertemengen**, die keinen semantischen Bezug zu Zahlen oder logischen Werten haben.

  ◇ Die einzelnen Werte, aus denen ein Aufzählungstyp besteht (Enum-Werte), besitzen einen bei der Typdefinition festgelegten **Namen**.
  Ausserdem ist jedem Enum-Wert eine durch die Definitions-Reihenfolge festgelegte **Ordnungszahl** zugeordnet.
  Der erste Wert besitzt die Ordnungszahl `0`, der zweite die Ordnungszahl `1` usw.

  ◇ **Werte** in **verschiedenen Aufzählungstypen** können den **gleichen Namen** besitzen.

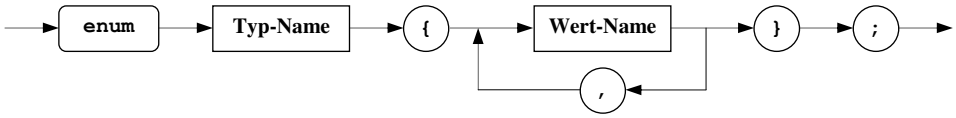
  ◇ Java-Aufzählungstypen sind – anders als C/C++-Aufzählungstypen – **typsicher** :
  Der Compiler stellt sicher, dass einer Aufzählungstyp-Variablen nur gültige Werte ihres Aufzählungstyps zugewiesen werden können.

  ◇ Anders als in C/C++ steht der **Name** eines **Enum-Werts** auch zur **Laufzeit** zur Verfügung und kann z.B. ausgegeben werden.

  ◇ **Enum-Werte** können als **`case`-Label** in **`switch`-Anweisungen** verwendet werden.

- **Definition**

  ◇ In der **einfachsten Form** (elementare Definition) werden Aufzählungstypen wie in C/C++ definiert.



  Beispiele ; `enum Jahreszeit { WINTER, FRUEHLING, SOMMER, HERBST };`
  `enum Farbe { ROT, GRUEN, BLAU, GELB, GRAU };`

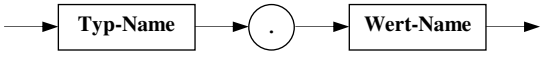
  ◇ Über die Möglichkeiten von C/C++ hinausgehend können auch noch **komplexere Formen** der Typ-Definition verwendet werden.

- **Verwendung**

  ◇ Aufzählungstypen können **wie andere Typen verwendet** werden, z.B. zur **Definition von Variablen**

  Beispiele : `Jahreszeit saison;`
  `Farbe anstrich;`

  ◇ **Aufzählungstyp-Werte** (Enum-Werte) werden mit ihrem **vollqualifizierten Namen** angesprochen :



  ◇ **Aufzählungstyp-Variablen** können **Aufzählungstyp-Werte zugewiesen** werden.

  Beispiele : `saison = Jahreszeit.SOMMER;`
  `anstrich = Farbe.GRUEN;`

  ◇ Anders als in C/C++ sind zwischen **Enum-Werten** und **`int`-Werten weder implizite noch explizite Typ-Konvertierungen** möglich.

Aufzählungstypen in Java (2)

• Implementierung

◇ Aufzählungstypen in Java sind **Klassen**, die besondere Eigenschaften besitzen
Aufzählungstyp-Variable sind somit **Referenz-Variable**.

◇ Alle Aufzählungstypen sind implizit von der abstrakten Bibliotheksklasse **`Enum`** abgeleitet. (Package **`java.lang`**)
Diese besitzt u.a. je eine **Datenkomponente** zur Speicherung des **Namens** des **Aufzählungstyp-Werts** (Typ `String`) sowie der dem Wert zugeordneten **Ordnungszahl** (Typ `int`)

◇ Die einzelnen **Aufzählungstyp-Werte** sind **Objekte** ihrer jeweiligen Klasse (**Enum-Objekte**).
Sie werden implizit beim Laden der Klasse erzeugt und als öffentlich zugreifbare **statische Datenkomponenten** gespeichert. (Verwendung über ihren vollqualifizierten Namen !, s. oben)
Eine **explizite Instantierung** einer Aufzählungstyp-Klasse ist **nicht möglich**.

◇ Wie jede Klasse kann ein Aufzählungstyp auch **Memberfunktionen** und **weitere Datenkomponenten** besitzen.

◇ Neben den für jeden Aufzählungstyp **standardmässig vorhandenen Memberfunktionen** (teilweise geerbt von `Object` und `Enum`) können bei seiner Definition **weitere Methoden und Datenkomponenten** festgelegt werden.

• Standardmässig definierte Memberfunktionen von Aufzählungstypen (Auswahl)

`E` bezeichnet den **jeweiligen Aufzählungstyp**

◇
```
public static E[] values()                    /* automatisch vom Compiler generiert */
```

▻ Die Funktion gibt ein Array zurück, das **alle Aufzählungstyp-Werte** (Enum-Objekte !) in der Reihenfolge ihrer Definition enthält

◇
```
public String toString()                      /* definiert in Enum */
```

▻ Die Funktion liefert den **Namen** des aktuellen **Aufzählungstyp-Wertes** (Enum-Objektes) zurück.

◇
```
public final int ordinal()                    /* definiert in Enum */
```

▻ Die Funktion liefert die **Ordnungszahl**, die mit dem aktuellen **Enum-Objekt** verknüpft ist, zurück

▻ Die Ordnungszahl entspricht der Reihenfolge der Enum-Objekte (Enum-Werte) in der Typ-Definition.
Der erste Enum-Wert hat die Ordnungszahl `0`.

◇
```
public final int compareTo(E obj)             /* definiert in Enum */
```

▻ Die Funktion **vergleicht** das aktuelle Enum-Objekt mit dem als Parameter übergebenen Enum-Objekt bezüglich ihrer **Ordnungszahlen**

▻ Das aktuelle Enum-Objekt und das Parameter-Enum-Objekt müssen vom **gleichen Aufzählungstyp** sein

▻ **Funktionswert** :
`<0`, wenn das aktuelle Objekt `<` dem Parameter-Objekt ist
`==0`, wenn das aktuelle Objekt `==` dem Parameter-Objekt ist
`>0`, wenn das aktuelle Objekt `<` dem Parameter-Objekt ist

◇
```
public final boolean equals(Object obj)       /* definiert in Enum */
```

▻ Die Funktion **vergleicht** das aktuelle Enum-Objekt mit dem als Parameter übergebenen Objekt

▻ **Funktionswert** :
`true`, wenn das Parameter-Objekt gleich dem aktuellen Enum-Objekt ist
`false`, wenn das Parameter-Objekt nicht gleich dem aktuellen Enum-Objekt ist

Aufzählungstypen in Java (3)

- **Aufzählungstypen mit selbstdefinierten Methoden und Datenkomponenten**

  ◇ Aufzählungstypen können mit einer beliebigen Anzahl zusätzlicher Memberfunktionen und Datenkomponenten definiert werden.
  Dies eröffnet ein weites Spektrum an Gestaltungsmöglichkeiten für sehr effizient einsetzbare Aufzählungstypen.

  ◇ Für die entsprechenden Aufzählungstyp-Definitionen gilt eine erweiterte **komplexere Syntax**.

  ◇ Üblicherweise gilt eine Memberfunktion für alle Objekte des entsprechenden Typs. Sie legt ein für alle Objekte gleichartiges Verhalten fest.
  Bei Aufzählungstypen besteht darüberhinaus die Möglichkeit, für jeden Enum-Wert (Enum-Objekt) ein spezifisches Verhalten festzulegen.
  Hierzu kann man die entsprechende Methode in der Aufzählungstyp-Definition als `abstract` deklarieren und sie dann für jeden Enum-Wert konkret definieren. (→ wert-spezifische Methoden, *constant specific methods*)

  ◇ **Beispiel für einen Aufzählungstyp mit selbstdefinierter für alle Enum-Werte geltender Memberfunktion** :

```
enum Wochentag
{
  Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag;

  public boolean istWochenende()
  {
    return this==Samstag || this==Sonntag;
  }
};
```

  ◇ Wenn ein Aufzählungstyp mit **zusätzlichen Datenkomponenten** definiert wird,
  - ▻ muss auch ein **Konstruktor** definiert werden, mit dem die Datenkomponenten initialisiert werden können
  - ▻ und müssen für jeden Enum-Wert entprechende **aktuelle Werte** für die Datenkomponenten bereitgestellt werden.

  Anmerkung : Auch wenn der Konstruktor `public` definiert wird, lassen sich **explizit keine Enum-Objekte anlegen**.

  ◇ **Beispiel für einen Aufzählungstyp mit einer zusätzlichen Datenkomponente**

```
enum Jahreszeit2
{
  WINTER   ("Dezember bis Februar"),
  FRUEHLING("Maerz bis Mai"),
  SOMMER   ("Juni bis August"),
  HERBST   ("September bis Oktober");

  private String bereich;

  public Jahreszeit2 (String monate)
  {
    bereich = monate;
  }

  public String getBereich()
  {
    return bereich;
  }
};
```

Demonstrationsprogramm zu Aufzählungstypen in Java

- **Quellcode des Programms (Klasse `EnumDemo2`)**

```
// EnumDemo2.java

enum Wochentag
{
  Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag;

  public boolean istWochenende()
  { return this==Samstag || this==Sonntag; }
};


enum Jahreszeit2
{
  WINTER   ("Dezember bis Februar"),
  FRUEHLING("Maerz bis Mai"),
  SOMMER   ("Juni bis August"),
  HERBST   ("September bis Oktober");

  private String bereich;

  public Jahreszeit2 (String monate)
  { bereich = monate; }

  public String getBereich()
  { return bereich; }
};


public class EnumDemo2
{
  public static void main(String[] args)
  { Wochentag tag = Wochentag.Samstag;
    System.out.println();
    System.out.println(tag + " gehoert zum Wochenende : " + tag.istWochenende());

    Jahreszeit2 saison = Jahreszeit2.HERBST;
    System.out.println("aktuelle Jahreszeit : " + saison);

    System.out.println("\nDie Jahreszeiten :");
    for (Jahreszeit2 jz : Jahreseit2.values())
      System.out.println(jz + " umfasst " + jz.getBereich());
  }
}
```

- **Ausgabe des Programms**

```
Samstag gehoert zum Wochenende : true

aktuelle Jahreszeit : HERBST

Die Jahreszeiten :
WINTER umfasst Dezember bis Februar
FRUEHLING umfasst Maerz bis Mai
SOMMER umfasst Juni bis August
HERBST umfasst September bis Oktober
```

Generische Programierung in Java (1)

• Allgemeines

◇ Unter **generischer Programmierung** versteht man die Formulierung von **Programm-Code**, der in seiner **wesentlichen Funktionalität unabhängig von** Repräsentationsdetails **konkreter Datentypen** ist.
Der Code wird einmal allgemein formuliert und lässt sich dann für unterschiedliche konkrete Datentypen verwenden.

◇ Ein **typisches Beispiel** für generische Programmierung bilden i.a. **Container-Bibliotheken** (z.B. STL in C++, *Collection Framework* in Java).
Container speichern und verwalten Daten (Objekte) unterschiedlicher Typen. Ihre jeweilige Funktionalität ist dabei i.a. unabhängig von den konkreten Typen der Daten.
Beispiel : Ein Stack für Integer-Werte besitzt die gleiche Funktionalität wie ein Stack für Strings oder ein Stack für `Person`-Objekte.

◇ Generische Programmierung lässt sich **realisieren** mittels

- ➢ **Polymorphie** (zumindest eingeschränkt)
- ➢ **Typ-Parameterisierung**

◇ In **C++** wird generische Programmierung durch den Einsatz von Klassen- und/oder Funktions-**Templates** ermöglicht, die mit Typ-Parametern arbeiten

• Generische Programmierung in Java bis einschliesslich dem JDK 1.4

◇ Für eine – eingeschränkte – generische Programmierung steht lediglich das Konzept der **Polymorphie** zur Verfügung.

◇ Beispielweise verwalten die Container des ***Collection Frameworks*** Objekte der Klasse **`Object`**.
Da in Java jede Klasse direkt oder indirekt von `Object` abgeleitet ist, können die Container für **Daten beliebigen Referenz-Typs** eingesetzt werden.
Das *Collection Framework* ist somit **generisch programmiert**.
Auch in anderen Stellen der Java Standard-Bibliothek wird diese Art der generischen Programmierung verwendet.

◇ Im Code, der derartigen Code – z.B.das *Collection Framework* – **verwendet**, müssen gegebenenfalls **explizite Type-Casts** in den tatsächlich konkret verwendeten Datentyp vorgenommen werden.
Der Anwendungsprogrammierer ist für die Wahl der richtigen Type-Casts verantwortlich. Die Verwendung falscher Typ-Casts kann vom Compiler i.a. nicht erkannt werden, sondern wird erst durch das Auftreten von Laufzeitfehlern entdeckt.

◇ **Beispiel** :

```
// GenProgDemo14.java
// Demonstration der Anwendung generischer Programmierung mittels Polymorphie
// (bis JDK 1.4)

import java.util.*;

public class GenProgDemo14
{
  public static void main(String[] args)
  {
    Stack stk = new Stack();
    stk.push(new Integer(123));
    Integer i = (Integer)stk.pop();          // expliziter Type-Cast erforderlich !
    System.out.println("Wert vom Stack : " + i);
    stk.push("Ein String");                  // fehlerfrei uebersetzt
    i = (Integer)stk.pop();                  // Laufzeitfehler ClassCastException
  }
}
```

Generische Programierung in Java (2)

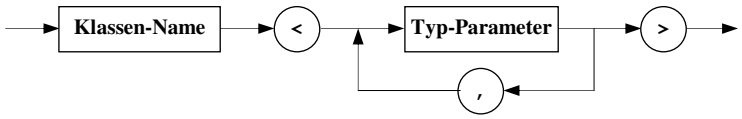
• Generische Programmierung in Java ab dem JDK 1.5

◇ Mit dem JDK 1.5 wurde in Java das Konzept der ***Generics*** als neues Sprachelement eingeführt.
Dieses sieht die Definition **generischer Klassen** (und Interfaces) sowie **generischer Methoden** (einschliesslich generischer Konstruktoren) vor.
Dadurch lässt sich nunmehr auch in Java eine auf **Typ-Parameterisierung** basierende **generische Programmierung** realisieren.

◇ ***Generics*** in Java weisen eine **formale Ähnlichkeit** zu den **Templates** in C++ auf. Sie **unterscheiden** sich aber **grundlegend** von diesen bezüglich ihrer **Implementierung** und ihrer Behandlung durch den Compiler.

◇ Mit der **Definition** einer **generischen Klasse** (bzw eines generischen Interfaces) werden **formale Typ-Parameter** festgelegt. Diese werden in Java auch als **Typ-Variable** (*Type Variable*) bezeichnet.
Die formalen Typ-Parameter werden nach dem Klassennamen (bzw Interface-Namen) angegeben, in spitzen Klammern eingeschlossen und durch Kommata getrennt → **Klassenbezeichnung** einer **generischen Klasse** (bzw eines Interfaces).

Syntax :

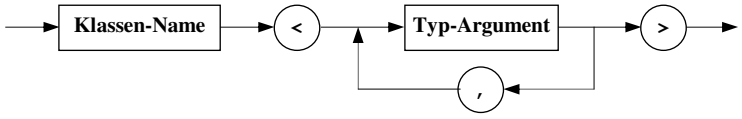


Die Namen der Typ-Parameter sollten per Konvention möglichst nur aus einem Grossbuchstaben bestehen.

Beispiele : `Stack<E>`
`Map<K,V>`

◇ Zur **Verwendung** einer generischen Klasse (bzw eines generischen Interfaces) müssen die formalen Typ-Parameter durch **aktuelle Typ-Parameter** ersetzt werden. Diese werden in der Java-Syntax als **Typ-Argumente** (*Type Arguments*) bezeichnet. (Es muss sich um Referenz-Typen handeln. Einfache Datentypen sind nicht zulässig.)
Hierdurch wird ein aus der generischen Klasse (bzw eines Interfaces) erzeugter konkreter Typ festgelegt
→ **parameteriserter Typ**

Syntax :



Durch **jeden neuen Satz** von **Typ-Argumenten** wird **ein neuer parameterisierter Typ** definiert.
Eine **generische Klasse** (bzw ein generisches Interface) **definiert** also **mehrere** (i.a. beliebig viele) **Typen**.

◇ Die Verwendung von parameterisierten Typen **vereinfacht** die **Programmierung**, explizite Type-Casts sind nicht mehr erforderlich.
Der **Compiler** kann die **fehlerhafte Verwendung** parameteriserter Typen **erkennen**. Es wird **typsicherer Code** erzeugt.

◇ **Beispiel** :

```
// GenProgDemo15.java
// Demonstration der Anwendung generischer Programmierung mittels
// Typ-Parameterisierung (Generics, ab JDK 1.5)

import java.util.*;

public class GenProgDemo15
{
  public static void main(String[] args)
  {
    Stack<Integer> stk = new Stack<Integer>();
    stk.push(new Integer(123));      // oder : stk.push(123);
    Integer i = stk.pop();           // expliziter Type-Cast nicht erforderlich
    System.out.println("Wert vom Stack : " + i);
    stk.push("Ein String");          // Compiler-Fehler
  }
}
```

Generische Programierung in Java (3)

• Parameterisierte Typen und Vererbung

◇ Eine eventuelle **Ableitungsbeziehung** zwischen **Typ-Argumenten** wird **nicht** auf die entsprechenden **parameterisierten Typen übertragen** (Unterschied zu Arrays !) → **Typ-Invarianz**

◇ **Beispiel** : Die Klasse **`Integer`** ist von der Klasse **`Number`** **abgeleitet**. Eine `Number`-Variable kann damit auch auf ein `Integer`-Objekt zeigen :

```
Number num = new Integer(25);
```

Dagegen besteht zwischen den parameterisierten Typen **`Stack<Integer>`** und **`Stack<Number>`** **keinerlei Kompatibilität** :

```
Stack<Number> = new Stack<Integer>()   // unzulässig !!!
```

◇ Eine der Vererbung entsprechende **Kompatibilitätsbeziehung** zwischen **parameterisierten Datentypen** wird durch die Verwendung von **Wildcards** und **Typbegrenzungen** (*Type Bounds*) bei den **Typ-Argumenten** ermöglicht

• Wildcards als Typ-Argumente

◇ Die Verwendung des **Wildcard-**Zeichens **`?`** als Typ-Argument definiert einen parameterisierten Typ, zu dem alle anderen aus derselben generischen Klasse erzeugbaren Typen zuweisungs-kompatibel sind.
Dieser **Wildcard-Typ** wirkt als eine Art "Basisklasse" aller anderen parameterisierten Typen derselben Klasse.

◇ **Beispiel** :

```
Stack<?> stk = new Stack<Integer>(); // zulässig !!!
stk = new Stack<Number>();           // zulässig !!!
stk = new Stack<String>();           // zulässig !!!
```

◇ Wildcard-Typen sind **nur eingeschränkt verwendbar**.
Sie eignen sich nur für **Situationen**, die **keine Festlegung** auf ein **konkretes Typ-Argument** benötigen
Beispielsweise lässt sich eine Utility-Methode zum **Leeren eines beliebigen Stacks** mit einem **Parameter** vom Typ **`Stack<?>`** definieren :

```
// GenProgWildcardDemo.java

import java.util.*;

public class GenProgWildcardDemo
{
  public static void main(String[] args)
  { Stack<Integer> stk = new Stack<Integer>();
    stk.push(3); stk.push(5); stk.push(7);
    System.out.println("Stack enthielt " + clearStack(stk) + " Elemente");
  }

  public static int clearStack(Stack<?> s)  // Rueckgabe Anzahl entfernter Elemente
  { int i = 0;
    while (!s.empty())
    { s.pop();
      ++i;
    }
    return i;
  }
}
```

◇ Aus der Sicht des Compilers sind die Eigenschaften eines Wildcard-Typs bezüglich seiner Typ-Argumente völlig unbekannt. Er kann daher bezüglich der Verwendung der Typ-Argumente keine Überprüfungen vornehmen.
Deshalb sind sowohl **Lese-** als auch **Schreibzugriffe**, die von den Typ-Argumenten abhängen, **nicht zulässig**.

Ausnahmen : - Da jedes Typ-Argument von `Object` abgeleitet sein muss, ist ein **Lesen von `Object`-Objekten immer zulässig** (z.B. können `Integer`-Objekte jederzeit als `Object`-Objekte gelesen werden)
- Der Wert `null` kann jeder Referenz-Variablen zugewiesen werden. Daher kann **`null`** **immer** als ein zu jedem Typ-Argument kompatibler Wert **geschrieben** werden (z.B. kann auf jedem beliebigen Stack der Wert `null` abgelegt werden).

Generische Programierung in Java (4)

• Typbegrenzte Wildcards als Typ-Argumente

◇ Zu einem Wildcard-Typ mit dem Typ-Argument `?` sind alle aus derselben Klasse erzeugbaren parameterisierten Typen kompatibel.
Häufig wird aber eine **eingeschränkte Kompatibiltät** benötigt, die auf die Berücksichtigung der **zwischen den Typ-Argumenten** bestehenden **Ableitungsbeziehungen begrenzt** ist.

◇ Derartige eingeschränkte Kompatibilitäten werden durch **typbegrenzte Wildcards** (*bounded wildcards*) als Typ-Argumente realisiert.
Es gibt zwei Arten der Begrenzung :

- ▻ Wildcards mit **oberer Begrenzung** (*Upper Type Bound*)
- ▻ Wildcards mit **unterer Begrenzung** (*Lower Type Bound*)

◇ **Wildcards mit oberer Begrenzung** (**covarianter** Wildcard-Typ):
Zu einem parameterisierten Typ mit einem derartigen Typ-Argument sind alle Typen kompatibel deren Typ-Argument gleich der oberen Begrenzung ist oder von dieser abgeleitet ist

Syntax :

Beispiel : `Stack<? extends Number> stk;`
Hierzu sind alle `Stack`-Typen kompatibel, deren Typ-Argument der Typ `Number` oder ein davon abgeleiteter Typ ist :

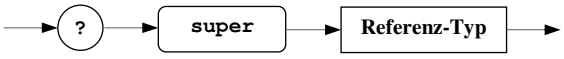
```
stk = new Stack<Integer>();     // zulässig !!!
stk = new Stack<Object>();      // unzulässig !!!
```

Allgemein gilt : **`C<A>` ist kompatibel zu `C<? extends B>`,**
wenn `A` kompatibel ist zu (abgeleitet ist von) **`B`**

Bei Wildcard-Typen mit oberer Begrenzung sind **Schreibzugriffe**, die von den Typ-Argumenten abhängen, grundsätzlich **nicht zulässig**. (Ausnahme : Schreiben des Wertes `null`).
Lesezugriffe dagegen sind **erlaubt**, da ein Objekt einer abgeleiteten Klasse immer als Objekt einer seiner Basisklassen verwendet werden kann.
Z.B. ist ein `Integer`-Objekt immer auch ein `Number`-Objekt, d.h. es kann als solches ausgelesen werden.
Umgekehrt kann ein `Number`-Objekt nicht als `Integer`-Objekt behandelt werden, was beim Schreiben eines `Number`-Objekts in einen – konkret vorhandenen – `Integer`-Stack notwendig wäre.

◇ **Wildcards mit unterer Begrenzung** (**contravarianter** Wildcard-Typ):
Zu einem parameterisierten Typ mit einem derartigen Typ-Argument sind alle Typen kompatibel deren Typ-Argument gleich der unteren Begrenzung ist oder ein Basis-Typ dieser ist

Syntax :



Beispiel : `Stack<? super Number> stk;`
Hierzu sind alle `Stack`-Typen kompatibel, deren Typ-Argument der Typ `Number` oder ein Basis-Typ von `Number` ist :

```
stk = new Stack<Integer>();     // unzulässig !!!
stk = new Stack<Object>();      // zulässig !!!
```

Allgemein gilt : **`C<A>` ist kompatibel zu `C<? super B>`,**
wenn `B` kompatibel ist zu (abgeleitet ist von) **`A`**

Bei Wildcard-Typen mit unterer Begrenzung sind **Lesezugriffe**, die von den Typ-Argumenten abhängen, grundsätzlich **nicht zulässig**. (Ausnahme : Lesen von `Object`-Objekten ist immer zulässig).
Die aus einem – konkret vorhandenen – Basisklassen-Stack auslesbaren Objekte können nicht als Objekte eines abgeleiteten Typs (der die Begrenzung darstellt) betrachtet werden.
Schreibzugriffe dagegen sind **erlaubt**. Z.B. kann ein `Number`-Objekt immer auf einen – konkret vorhandenen – `Object`-Stack abgelegt werden (oder ein `Integer`-Objekt auf einem `Number`-Stack)