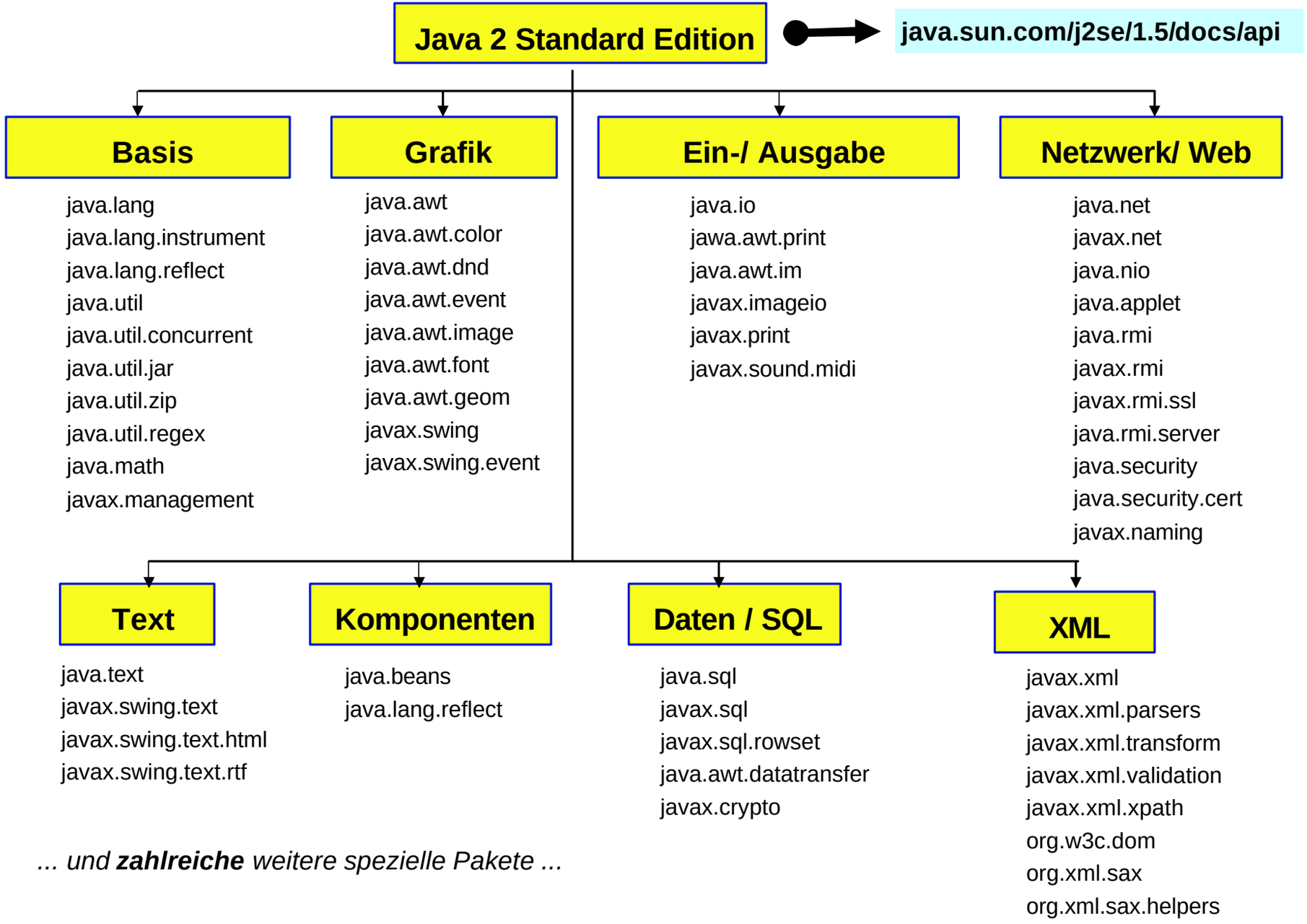
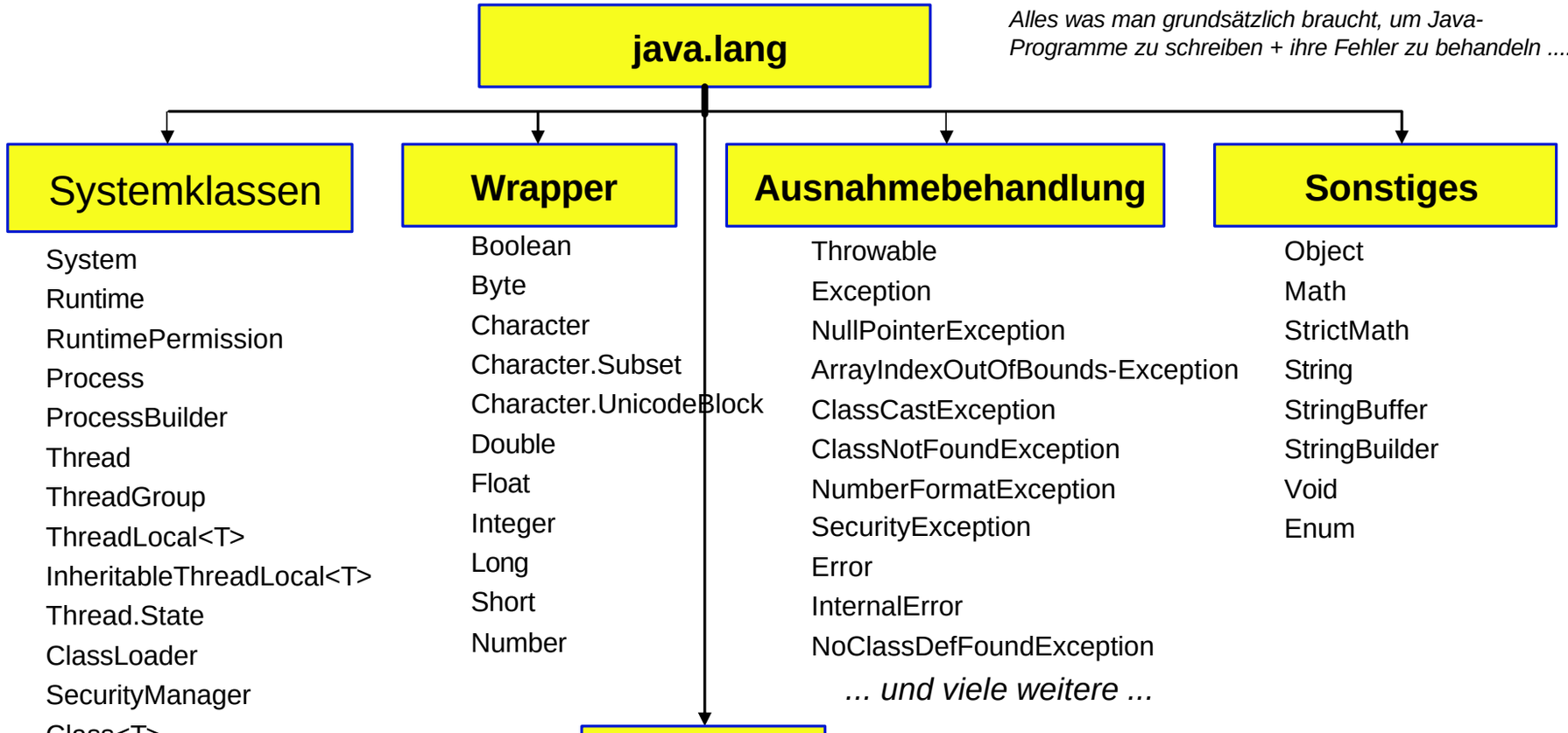
E6 J2SE 5.0 - Fundamentale Pakete



*... und **zahlreiche** weitere spezielle Pakete ...*

Klassen + Interfaces des Pakets `java.lang` der J2SE 5.0



Wichtigste Informationsquelle →
API-Dokumentation von Sun:
java.sun.com/j2se/1.5/docs/api

Wichtiges Package :

Wegen elementarer Bedeutung durch Compiler automatisch importiert

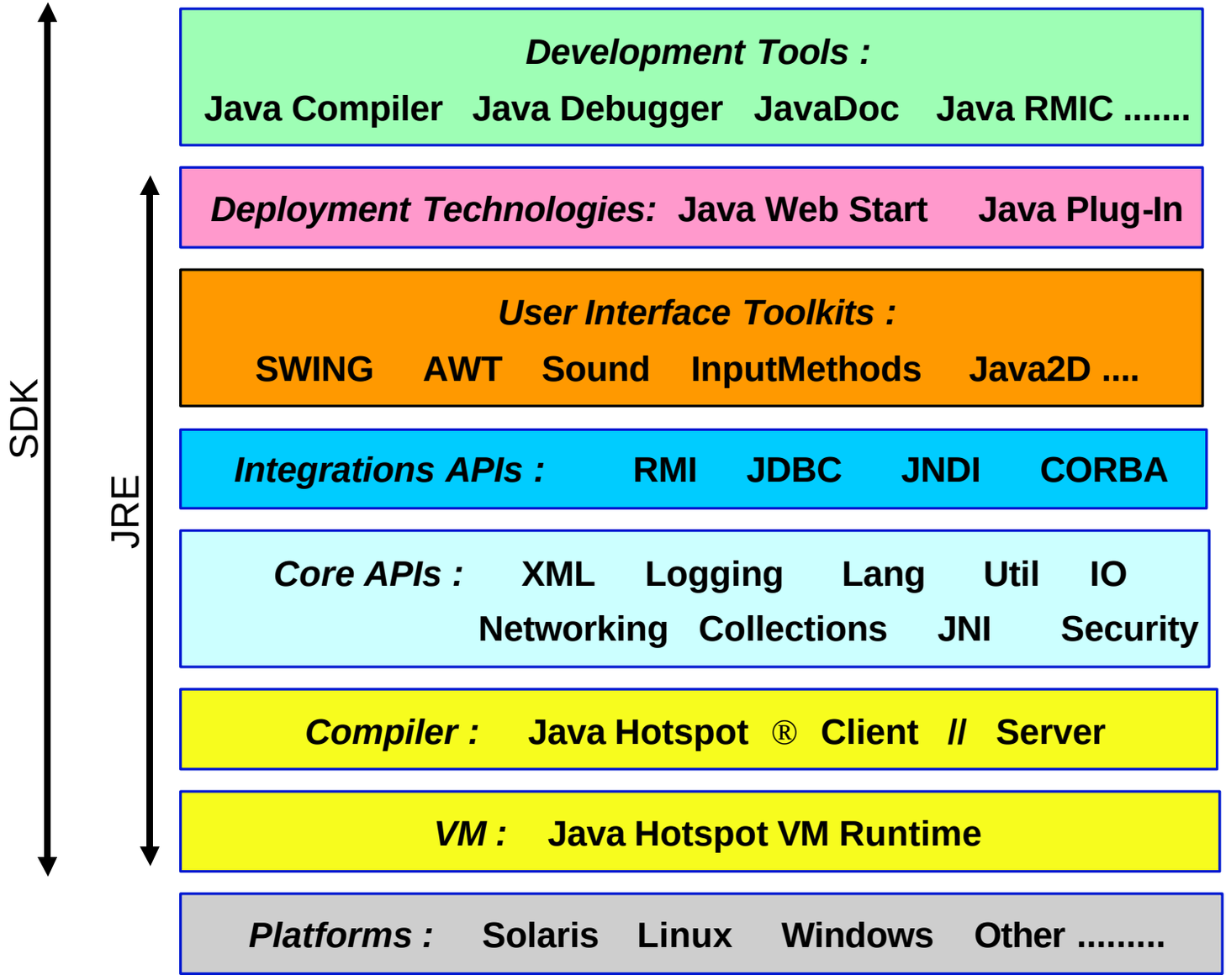
⇒ Kein explizites:

`import java.lang.*`;

zur Verwendung erforderlich!

J2SE: Komponenten, Teilsysteme und APIs

Quelle: Sun©

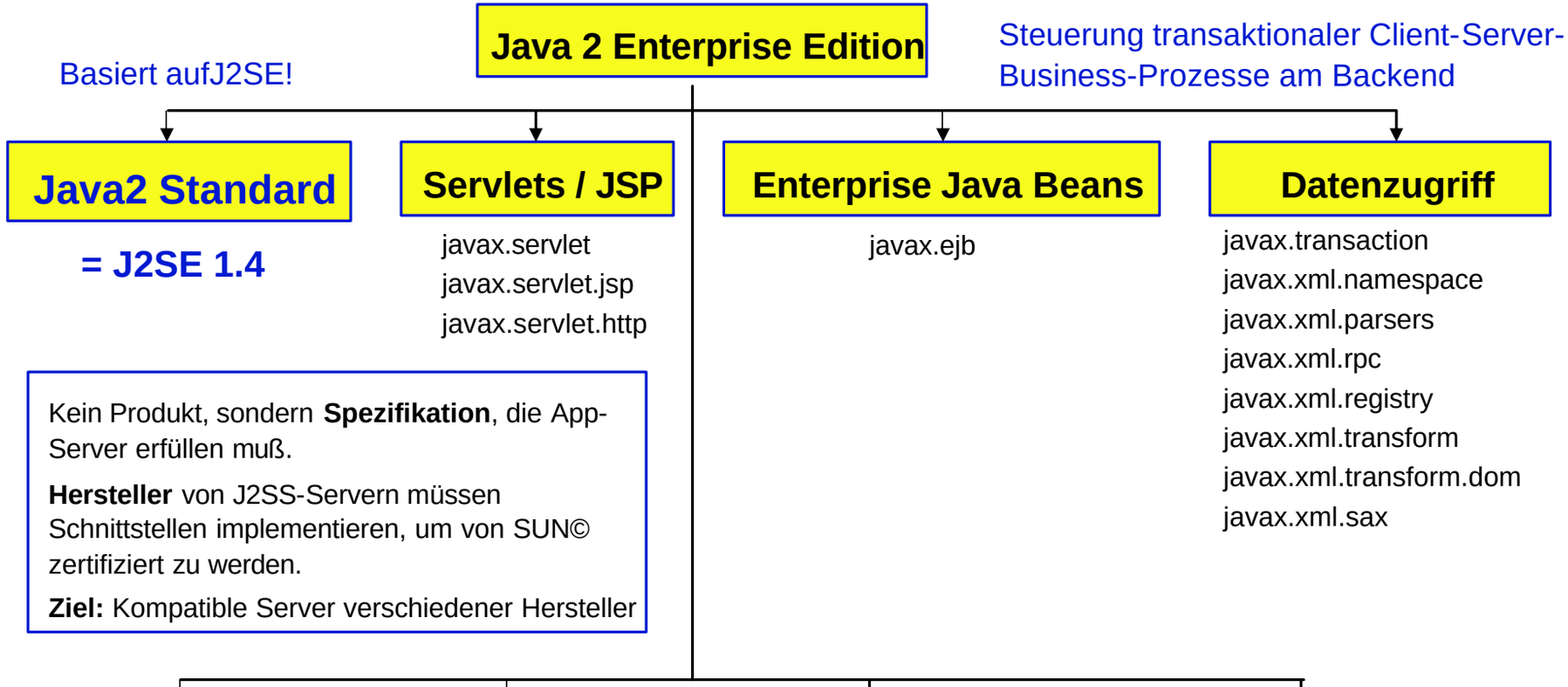


Java2 Enterprise Edition - "Fundamentale" Pakete

Einsatz J2EEauf **Applikationsservern:**

Steuerung transaktionaler Client-Server-Business-Prozesse am Backend

Basiert aufJ2SE!



Kein Produkt, sondern **Spezifikation**, die App-Server erfüllen muß.

Hersteller von J2SS-Servern müssen Schnittstellen implementieren, um von SUN© zertifiziert zu werden.

Ziel: Kompatible Server verschiedener Hersteller

... und weitere spezielle Pakete ...

Sun © liefert Referenzimplementation:

→ java.sun.com/j2ee/1.4/download.html

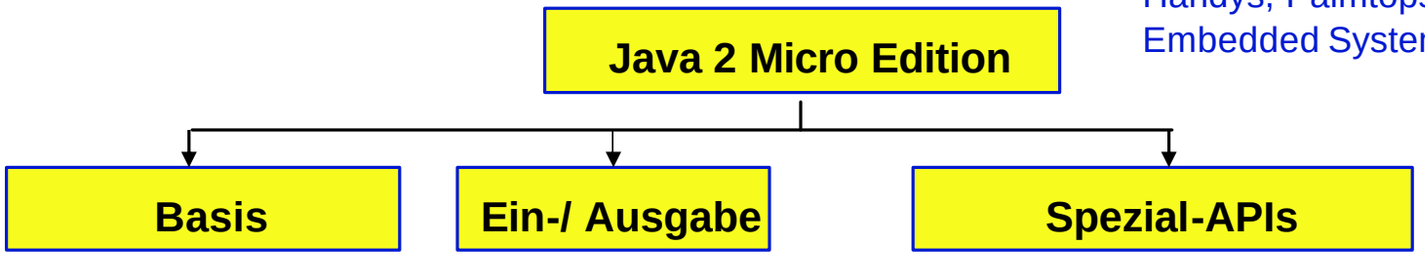
OpenSource J2EE-Server **JBoss** © :

→ www.jboss.com/downloads/index

Java2 Micro Edition - "Fundamentale" Pakete

Einsatz der **J2ME** auf Geräten mit **geringer Hardwareausstattung**:

Handys, Palmtops, Smartphones, Embedded Systems, ...



java.lang
java.util

java.io

javax.microedition.io
javax.microedition.lcdui
javax.microedition.rms

Erweiterungen im Bereich GUI, Netz, ...

Basiert auf Miniversion der virtuellen Maschine = **KVM**

Noch manches im Fluß .....

J2ME Wireless Toolkit 2.5 =

Umgebung KToolbar, Emulatoren, J2ME, Dokumentation

Quelle: java.sun.com /javame

Nützliche Klassen aus J2SE-Paketen

Mathematische Funktionen und Konstanten

Klasse: java.lang.Math

Klasse enthält nur statische Konstanten und Methoden

Konstanten: Math.E Math.PI

Methoden: **(**teilweise in mehreren überladenen Fassungen für verschiedene primitive Datentypen)

Abs() acos() asin() atan() atan2() cbrt() ceil() cos() cosh() exp() expm1() floor()
hypot() log() log10() log1p() max() min() pow() random() rint() round() signum() sin()
sinh() sqrt() tan() tanh() toDegrees() toRadians() ...

Definition der Methoden siehe: java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html

```
//import java.lang.Math;  nicht erforderlich, da Standardpaket java.lang
public class Mathe  {
   public static void main( String[] args) {
            IO.writeln( "Pi = " + Math.PI + "    e = " + Math.E );
            double d = Math.max( Math.PI, Math.E );
            IO.writeln( "Cosinus(Pi) = " + Math.cos( Math.PI) );
            IO.writeln( "Kubikwurzel von 27 ist: " + Math.cbrt(27) );
   }
}
```

Zufallszahlen-Erzeugung

Klasse: java.util.Random

Erzeugung von Zufallszahlen

Konstruktoren Random()

Random (long seed)

Ganzzahlige Zufallszahl im gesamten Wertebereich von int oder long:

int nextInt() long nextLong()

Zufallszahl zwischen 0 (incl) und n (excl):

int nextInt(**n**)

Fließkomma-Zufallszahl im gesamten Wertebereich von float oder double:

float nextFloat()

double nextDouble()

Boolesche Zufallswerte true / false:

boolean nextBoolean()

Übergebenes Array mit **Zufallsbytefolge** gefüllt:

void nextBytes(byte[] bytes)

Gaußverteilte Fließkomma-Zufallszahlen, zentriert um 0.0 :

nextGaussian()

Ändern des Startwerts des Generators

void setSeed(long seed) *und weitere .....*

```
import java.util.Random ;

public class Lotto  {
   public static void main( String[] args) {
         int zahl ;
         Random generator = new Random( ) ;

          zahl = generator.nextInt( 50 ) ;
          //  liefert Zahlen im Bereich 0 bis 49 !
   }
}
```

Initialisieren des Generators:

Random( ) ;

→ intern mittels Systemzeit

→ liefert bei jedem Programmlauf stets **andere** Folge von Zahlen!

Random(long seed) ;

→ explizit mittels Wert seed

→ liefert stets **gleiche** Folge von Zahlen!

Stoppuhr / Timer

Klasse: java.lang.System

Methode:

System.currentTimeMillis( ) ;

Liefert Anzahl der Millisekunden seit 1.1.1970 als long-Wert

Verstrichene Zeit erhaltbar durch ***Differenzbildung*** der Werte zweier Aufrufe

Anwendung:

- Testen von Algorithmen
- Evaluieren in welchen Programmteilen die meiste Zeit verbraucht wird

In J2SE 5.0 zusätzlich die Methode:

System.nanoTime()

Liefert Zeitwert des genauesten Systemzeitgebers in Nanosekunden.

Allerdings wird Nanosekundengenauigkeit sicherlich nicht erreicht.

```
// import java.lang.* ;
//      Braucht nicht explizit importiert werden:

public class Timer  {
   public static void main( String[] args) {
          long start ;
          long end ;

          start = System.currentTimeMillis( )  ;

          int test = 0 ;
          for ( int i = 1;  i<100;  i++ )  {
                test = test + i ;
           }

          end = System.currentTimeMillis( ) ;

          long dauer = end - start ;

          IO.writeln( "Dauer [ms] =  " + dauer ) ;
   }
}
```

Darstellung von Dezimalzahlen

Klasse: java.text.DecimalFormat

Formatierung von **Dezimalzahlen** vor deren Ausgabe

Konstruktor :

DecimalFormat(String **pattern**)

Erzeugt ein Objekt mit dem durch **pattern** vorgegebenen Format. String **pattern** kann sich zusammensetzen aus Platzhaltern für :

Ziffer, führende Nullen werden nicht angezeigt

0 Ziffer, führende Nullen werden angezeigt

. Dezimalpunkt (landesspezifisch)

, Tausendertrennung (landesspezifisch)

% Darstellung als Prozentzahl

E Trennt Platzhalter für Mantisse und Exponent

Alle anderen Zeichen werden direkt in den formatierten String übernommen !

Methoden: (... *und zahlreiche weitere ...*)

String **format**(double **x**) String **format**(long **x**)

Gibt Inhalt von **x** als gemäß Pattern-Vorgabe **formatierten** String zurück.

```
import java.text.DecimalFormat ;
public class Format {
  public static void main( String[] args) {
      DecimalFormat f1 = new DecimalFormat( "###,###.##" ) ;
      DecimalFormat f2 = new DecimalFormat( "Wert: 000,000.00000 Euro" ) ;
      DecimalFormat f3 = new DecimalFormat( "Prozente = ###.## %" ) ;
      DecimalFormat f4 = new DecimalFormat( "#.#E000" ) ;
      String s = f1.format( 24522.4567 ) ;     IO.writeln( s );
      s = f2.format( 98.765 ) ;                IO.writeln( s ) ;
      s = f1.format( 98.765 ) ;                IO.writeln( s ) ;
      s = f3.format( 55.123456 ) ;             IO.writeln( s ) ;
      s = f4.format( 0.123456789 ) ;           IO.writeln( s ) ;
      ausgabe( f4, 4568.56 ) ;
  }
  // Methode zur formatierten Ausgabe:
  public static void ausgabe( DecimalFormat f, double d )  {
      IO.writeln(  f.format( d )  ) ;
  }
}
```

Ausgabe:

```
24.522,46
Wert: 000.098,76500 Euro
98,76
Prozente = 5512,35 %
1,2E-001
4,6E003
```

Zerlegen von Strings

Klasse java.util.StringTokenizer

Zerlegung von Zeichenketten in definierte Einheiten = ***Token***

Definition der **Tokens :** Was soll als **Trennzeichen zwischen** zwei Tokens betrachtet werden?

Konstruktoren :

StringTokenizer(String str)

⇒ **Leerzeichen** als Trennzeichen

StringTokenizer(String str, String delim)

⇒ **delim** als Trennzeichen

Zugriff auf Tokens: (u.a.)

Anzahl von Tokens, die noch im String vorhanden sind = Anzahl möglicher Aufrufe der Methode nextToken()

int countTokens()

Abfrage, ob noch Tokens vorhanden sind :

boolean hasMoreTokens()

boolean hasMoreElements()

Abgreifen des nächsten Tokens :

String nextToken() Object nextElement()

```
import java.util.StringTokenizer ;
public class Tokens {

   public static void main( String[] args) {

      String ein = "HalloÖihrÖLeÖser" ;
      String trenn = "Ö" ;

      StringTokenizer st =
         new StringTokenizer(ein,trenn);

      int n = st.countTokens( ) ;
      IO.writeln("Tokenanzahl = " + n) ;

      while( st.hasMoreTokens( ) ) {
            IO.writeln( st.nextToken( ) );
        }
    }
}
```

Systemzugriff

Klasse: java.lang.System

Kapselt **Funktionen der JVM**. Zugriff auf Standard-Ein- und -Ausgabe, auf dyn.Bibliotheken, Umgebungsvariablen, ...

Von System können keine Objecte erzeugt werden - nur **statische** Klassenmethoden

void gc( )

Bittet JVM, Garbadge Collector zu starten

void exit(int status)

Beendigung der laufenden JVM (und damit des Programms) mit dem Exit-Code *status*

Properties getProperties( )
String getProperty(String key)

Abfrage der System-Properties, falls Zugriff darauf erlaubt ist.

Key ist der Schlüssel der Properties - wird auch ausgegeben, wenn man über Property-Objekt alle zugänglichen Properties ausgibt.

```
// import java.lang.*
import java.util.Properties ;
public class SysInfo {
   public static void main( String[] args) {
      // alle auf einmal ausgeben:
      Properties p = System.getProperties( ) ;
      p.list( System.out ) ;

       // Spezielle Property ansprechen:
       String key = "java.version" ;
       String prop = System.getProperty( key ) ;
        IO.writeln( prop ) ;
    }
}
```

Bsp:

java.version	java.vendor	java.home
java.vm.version	java.vm.vendor	java.vm.name
java.class.version	java.class.path	java.library.path
java.io.tmpdir	java.compiler	java.ext.dirs
os.name	os.arch	os.version
file.separator	path.separator	line.separator
user.name	user.home	user.dir

Systemzugriff Klasse: java.lang.System

Abfrage von Umgebungsvariablen

In der J2SE 5.0 existieren in **java.lang.System** zwei Methoden zur Abfrage der Umgebungsvariablen des Betriebssystems.

Methode: **`public static String getenv( String name )`**

liefert den Wert der Umgebungsvariablen `name` (ungleich `null`, sonst `NullPointerException`) als `String` zurück. Falls die Umgebungsvariable `name` auf dem System nicht definiert ist, wird `null` zurückgeliefert. Auf Unix muss `name` case-sensitiv sein, auf Windows typischerweise nicht.

Methode: **`public static Map<String, String> getenv()`**

liefert eine nicht-veränderliche `String-Map` mit den aktuellen Umgebungsvariablen des Betriebssystems. Falls das System keine Umgebungsvariablen unterstützt, wird eine leere Map zurückgeliefert. Bei Abfrage eines Umgebungswertes, der nicht vom Typ `String` ist, wird eine `ClassCastException` geworfen.

Falls ein ***Security Manger*** existiert, wird in beiden Fällen aus Sicherheitsgründen dessen Methode `checkPermission()` gerufen und je nach Konfiguration eventuell eine `SecurityException` geworfen.

Systemzugriff Klasse: java.lang.Runtime

Start von **Fremdprozessen** :

Runtime getRuntime( )

Liefert Runtime-Objekt für laufende JVM

Process exec(String cmd)

Vollständige Pfadangabe des aufzurufenden Programms/ System-Kommandos incl Parametern.
Voraussetzung: Ausreichende Rechte !!

long totalMemory( )

Liefert Größe des Systemspeichers in Bytes

long freeMemory( )

Liefert Anzahl der nichtbelegten Bytes

```
public class Prozesse  {   // Alles Nötige in java.lang
   public static void main( String[] args)   throws Exception {
      Runtime rt = Runtime.getRuntime( ) ;
      IO.writeln("Systemspeicher  = " + rt.totalMemory( )  ) ;
      IO.writeln("Freier Speicher = " + rt.freeMemory( )  ) ;

      Process p = rt.exec(  "notepad" ) ;
      IO.promptAndReadString( "Drücke Taste!" ) ;
      p.destroy( ) ;
}
```

Klasse **Process** repräsentiert gestarteten Prozess :

void destroy( )

Beenden des gestarteten Prozesses

Mit J2SE 5.0 wird Aufgabe von **Runtime** durch die neue Klasse **ProcessBuilder** übernommen, die zusätzliche Information über den gestarteten Prozess liefert .....

Systemzugriff — Klasse: java.lang.ProcessBuilder

Start von **Fremdprozessen mittels ProcessBuilder:**

ProcessBuilder(List<String> command)

ProcessBuilder(String ... command)

Erzeugen ProcessBuilder-Objekt mit Angabe Programmnamen und variabler Zahl von Aufrufargumenten

ProcessBuilder command(List<String> command)

ProcessBuilder command(String ... command)

Setzen des Programmnamens und variabler Zahl von Aufrufargumenten

List< String > command()

Liefert gesetzten Programmnamen und Argumente als String-List

ProcessBuilder directory(File directory)

Setzen des Arbeitsverzeichnisses

File directory( )

Liefert Arbeitsverzeichnis

Auch bei ProcessBuilder repräsentiert die Klasse **Process** den gestarteten Prozess :

void destroy( )

Beenden des gestarteten Prozesses

Map< String, String > environment()

Liefert betriebssystemspezifische Umgebungwerte des ProcessBuilders

Process start()

Start des Prozesses

... und weitere Methoden, zur Festlegung der Ausgabe von Fehlermeldungen

...

Klasse: java.lang.ProcessBuilder


```
import java.io.File;
public class Prozesse {
  public static void main( String[] args) throws Exception {
    ProcessBuilder pB = new ProcessBuilder( "notepad", "Demo1.txt" ) ;
    pB.directory( new File( "C:/WINNT/" ) ) ;
    IO.writeln( pB.command() ) ;
    IO.writeln( pB.directory() ) ;
    IO.writeln( "Umgebungsinfo: \n" + pB.environment().toString() ) ;
    Process p1 = pB.start() ;
    pB.command(  "notepad", "Demo2.txt" ) ;
    // Für ProcessBuilder-Objekte darf start() mehrfach aufgerufen werden:
    Process p2 = pB.start() ;
    IO.promptAndReadString( "Drücke Taste!" ) ;
     p1.destroy() ;
     p2.destroy() ;
  }
}
```

Aufruf des notepad-Editors unter Windows mit zwei abgelegten Textfiles ....

Ausgabe: (nur teilweise .....)

[notepad, Demo1.txt]

C:\WINNT

Umgebungsinfo:

{PROCESSOR_ARCHITECTURE=x86,
LOGONSERVER=

// und noch vieles mehr .....

Klassendeskriptoren **Klasse: java.lang.Class**

Jede Klasse + Interface läßt sich durch Objekt der Klasse **Class** beschreiben = **Klassendeskriptor**: Gewinnung von Metainformationen über Klassen zur Laufzeit.

Class hat **keine Konstruktoren**. Objekte vom Typ **Class** erzeugt, wenn Klassen geladen werden.

Methoden: *(... und viele andere zum Ermitteln der Methoden, Konstruktoren, Pakete, ...)*

Class getClass( )

Liefert Klassendeskriptor für Klasse auf deren Objekt die Methode aufgerufen wurde:

(Methode getClass() *von* Object *an alle Klassen vererbt)*

String s = "Hallo" ; **Class cs = s.getClass( ) ;**

String getName()

Liefert vollständigen Paketnamen der Klasse bzw des Interfaces

static Class forName(String className)

Liefert Klassendeskriptor für Klasse / Interface mit Namen className. (Vollständige Paketangabe nötig: zB *java.lang.String*)

boolean isInterface( )

Liefert true, wenn Klassendeskriptor ein Interface darstellt

Class getSuperclass( )

Liefert Klassendeskriptor für Oberklasse der Klasse bzw null wenn keine Oberklasse (oder nur Object) vorhanden ist

Class[ ] getInterfaces( )

Liefert Array von Klassendeskriptoren für alle Interfaces die von Klasse implemtiert werden bzw von denen Interface abgeleitet ist

Object newInstance( ) // dynamische Objektkonstruktion

Erzeugt ein neues Objekt der Klasse die durch den zugehörigen Klassendeskriptor repräsentiert wird.

java.lang.**Class**

Ein kleines Tool zum Ermitteln der Klassenhierachie und der Menge der implementierten Interfaces mittels Klassesdeskriptoren

```
class ClassInfo {

   public static void main( String[] args) throws Exception  {

        String s = IO.promptAndReadString("Klassen- oder Interface-Name: ") ;

       Class cs = Class.forName( s ) ;
       IO.writeln( "Info für : " + cs.getName( ) ) ;

       Class csSup = cs.getSuperclass( ) ;
        while( csSup != null ) {
             IO.writeln( "Oberklasse = " + csSup.getName( ) ) ;
             csSup = csSup.getSuperclass( ) ;     //Hochsteigen in der Klassenhierachie
       }

        Class[ ] csArr = cs.getInterfaces( ) ;
         for( int i = 0;  i<csArr.length;  i++ ) {
               IO.writeln( "Interface: " + csArr[i].getName( ) ) ;
         }
   }
}
```

Weitere Anwendungen: Bau von Entwicklungsumgebungen ® Klassencode wird eingegeben, gespeichert, mit Klassendeskriptor erfasst und mit Klassendeskriptor Objekt der Klasse erzeugt sowie deren Methoden gerufen .....

Clonen von Objekten - flache Kopien: Interface java.lang.**Cloneable**

Das **Interface Cloneable** enthält ***keine*** Methoden, sondern dient als reiner Marker!

Eine **Klasse** implementiert **Cloneable** nur deshalb, um dadurch ***anzuzeigen***, dass mittels der geerbten Methode **Object.clone()** ***flache Kopien*** ihrer Objektinstanzen durch direkte Zuweisung der Attributwerte hergestellt werden dürfen.

Wird die Methode **clone()** für Objekte einer Klasse aufgerufen, die **Cloneable** *nicht* implementiert, so wird eine **CloneNotSupportedException** geworfen - eine flache Kopie wird nicht unterstützt.

Klassen, die Cloneable implementieren sollten die *von* **Object** *geerbte Methode* **protected clone()** *durch eine* **public** *Variante überschreiben* - und dadurch eine frei aufrufbare *klassenspezifische Kopierweise* implementieren - oder intern einfach **super.clone()** aufrufen.

Das Interface enthält selbst *nicht* die **clone()**-Methode. Somit ist *nicht* gewährleistet, dass eine Klasse, die **Cloneable** implementiert, tatsächlich über eine *klassenspezifische* **public clone()**-Methode verfügt!

Will man in der überschriebenen Variante von **clone()** mittels **super.clone()** die von **Object** geerbte Version aufrufen, so *muss* **Cloneable** implementiert werden, da **Object.clone()** prüft, ob das aktuelle Objekt vom Typ **Cloneable** ist.

```
public class Mitarbeiter implements Cloneable  {
    public int id;     public double gehalt;
    public Object clone( ) {
        try {
          Mitarbeiter m = new Mitarbeiter;
          m.id = this.id;       m.gehalt = this.gehalt;
          return m;
          // oder: return  super.clone();
        }  catch ( CloneNotSupportedException e ) {
              return null;  // Klonen gescheitert
        }
}
```