

- Software Richtlinien
- Typisierung
- Vertragsbasierte Programmierung
- Fehlertolerante Programmierung
- Portabilität
- Dokumentation

**Typisierte Sprachen (Sprachen, die ein Typsystem besitzen)** kategorisieren Daten und fassen gleichartige Objekte zu einem Datentyp zusammen.

**Heute:** Die modernen Programmiersprachen unterstützen ausgefeilte Typsysteme.

**Früher** (Lisp, Fortran, Cobol,..) oftmals nur rudimentäre Typsysteme.

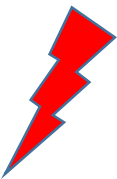
## Typsystem:

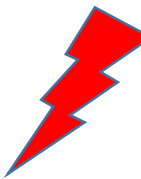
Der Teil eines Compilers oder einer Laufzeitumgebung, der ein Programm auf die korrekte Verwendung der Datentypen überprüft.

## Nutzen:

1. Frühzeitiges Erkennen von Inkonsistenzen
2. Verständlichkeit des Programms

**Bsp (zu 1.):** Syntaktische Bildung von unsinnigen Ausdrücken wird bereits zur Compile Zeit verhindert:

&42 

"41"++ 

# Nutzen der Verwendung von Datentypen

- Frühzeitige Erkennung von Inkonsistenzen
- Verständlichkeit der Codes/Lesbarkeit
- ➔ Letztlich Vermeidung von Laufzeitfehlern.

# Bestandteile eines Typsystems

- Typen (entweder in der Sprache verankert oder mittels Typdefinitionen erzeugt)
- Möglichkeit, Variablen, Funktionsparameter etc. mit einem bestimmten Typ zu deklarieren
- Regeln, nach denen die Werte von Ausdrücken einem bestimmten Typ zugeordnet werden
- Regeln zur Prüfung der Zuweisungskompatibilität von Typen.
- Optional weitere Sprachbestandteile (typbezogene Operatoren, Reflection etc.)

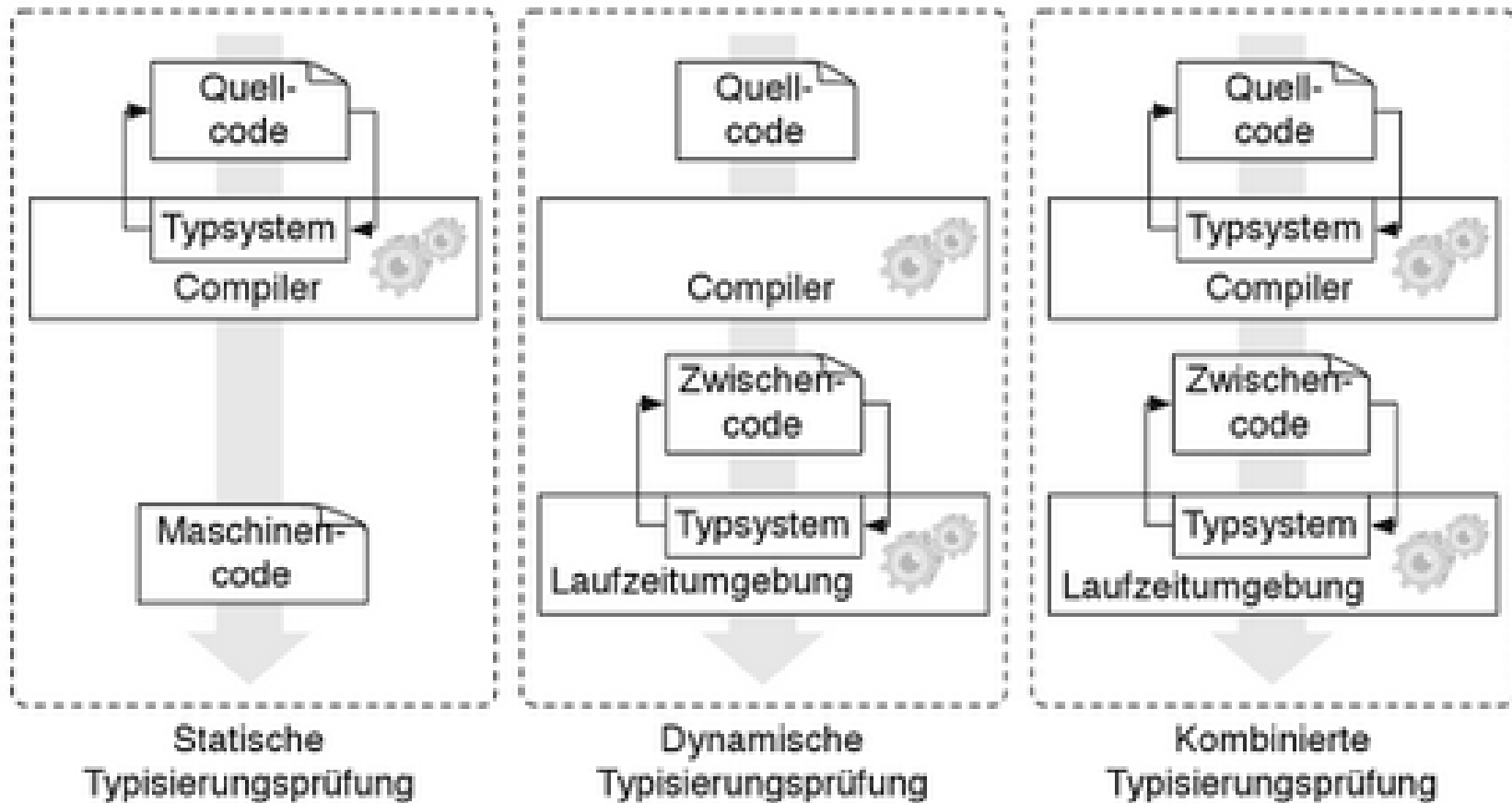
# Aufgaben eines Typsystems

- Erkennen von Typverletzungen
- Typumwandlungen

# Typprüfung

- Statische Typprüfung (static type checking): Prüfung zur Compilezeit.
- Dynamische Typprüfung (dynamic type checking): Prüfung während der Programmausführung.

# Typprüfung





Dynamische Typprüfung (die zwei rechten Bilder der Vorfolie) erhöht die Sicherheit des Programms und kostet Performance.

- ➔ Dynamische Typprüfung wird hauptsächlich bei Programmiersprachen der höheren Abstraktionsebene.
- ➔ Verlass auf statische Typprüfung bei hardwarenaher Programmierung.

# Einteilung der Typisierungsprüfung nach Prinzipien:

- **Statische Typisierung (static typing):**  
Variablen werden zusammen mit ihrem Datentyp im Quelltext deklariert.  
Bsp: C, C#, Java
- **Dynamische Typisierung (dynamic typing):**  
Variablen sind nicht an einen bestimmten Datentyp gebunden.  
Bsp: Smalltalk, PHP, Python

# PHP: dynamic typing

```
<?php
    $mystring = "12";
    $myinteger = 20;
    print $mystring + $myinteger;
?>
```

PHP konvertiert den String „12“ in den Integer 12 → Ergebnis ist 32.

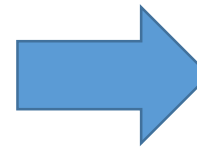
Aber:

\$mystring = “Hans“;

Konvertierung in Integer 0 → Ergebnis 20 ohne Fehlermeldung.

# PHP static typing

```
<?php
    $bool = true;
    print "Bool is set to $bool\n";
    $bool = false;
    print "Bool is set to $bool\n";
?>
```



```
Bool is set to 1
Bool is set to
```

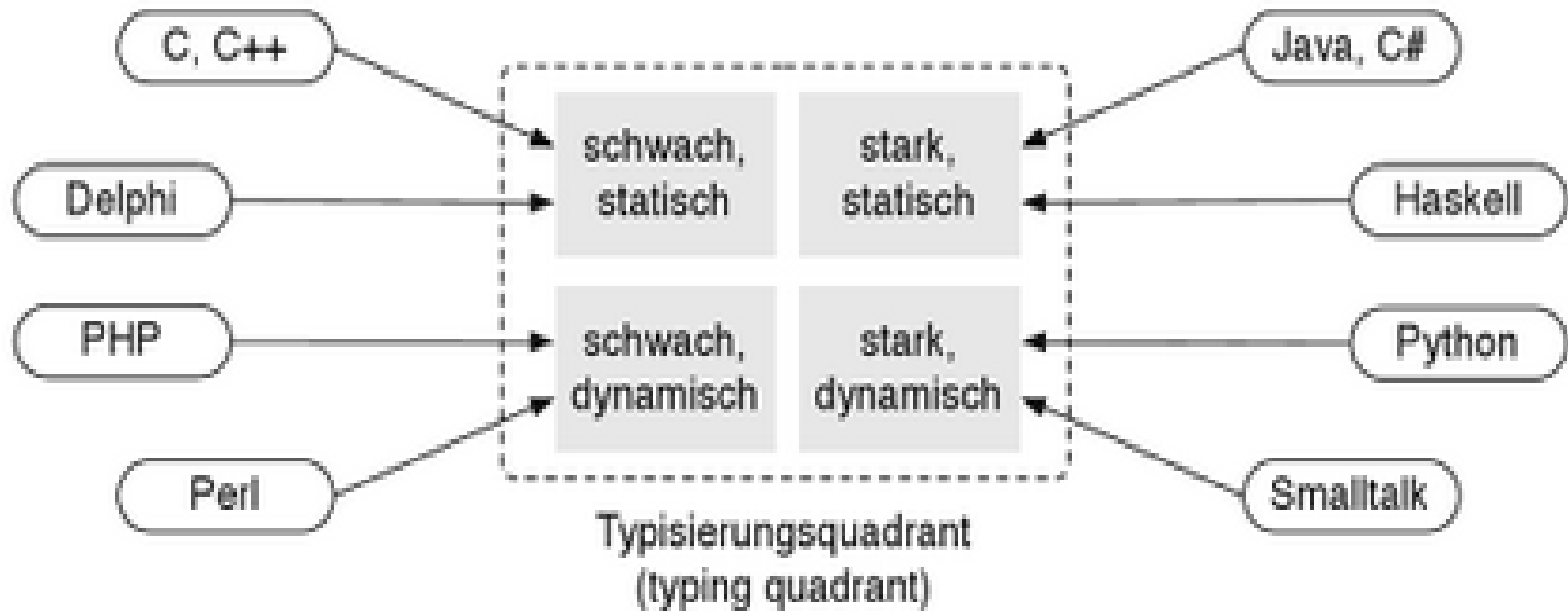
Lösung: expliziter Cast:

```
<?php
    $bool = true;
    print "Bool is set to $bool\n";
    $bool = false;
    print "Bool is set to ";
    print (int)$bool;
?>
```

- **Schwache Typisierung:**  
Datentyp eines Objekts darf beliebig uminterpretiert werden. (mithilfe von Typecasts)
- **Starke Typisierung:**  
Sicherstellung, dass der Zugriff auf alle Objekte und Daten stets typkonform erfolgt.

Diese Begriffe sind nicht absolut zu verstehen.

# Programmiersprachen und Typsysteme



Klassifikation verschiedener Programmiersprachen  
anhand ihrer Typsysteme

# Beispiel in Java

```
public static Integer addMax(Object a, int b, int c){  
    int max= Math.max(b,c);  
    int sum = (int) a + max;  
  
    return sum;  
}
```

ClassCastException,  
falls a nicht gecastet  
werden kann.

Implizite Konvertierung  
zu Integer.

NullPointerException,  
falls a nicht definiert  
ist.

# Grenzen, Fallstricke: Bsp Java

```
public class FooBar {  
  
    public static void main(String[] args) {  
        // create list  
        List list = new ArrayList();  
        list.add(new String("foo"));  
        list.add(new String("bar"));  
        list.add(new Integer(42));  
  
        //print list  
        Iterator iterator = list.iterator();  
        while(iterator.hasNext()){  
            String item = (String)iterator.next();  
            System.out.println(item);  
        }  
    }  
}
```

foo  
bar

Exception in thread "main" [java.lang.ClassCastException](#): java.lang.Integer cannot be cast to java.lang.String  
at demo.FooBar.main([FooBar.java:19](#))



# Lösung: Generics

```
public class FooBar_2 {  
  
    public static void main(String[] args) {  
        // create list  
        List<String> list = new ArrayList();  
        list.add(new String("foo"));  
        list.add(new String("bar"));  
        list.add(new Integer(42));  
  
        //print list  
        Iterator iterator = list.iterator();  
        while(iterator.hasNext()){  
            String item = (String)iterator.next();  
            System.out.println(item);  
        }  
    }  
}
```

Compile Fehler !

## C-Bsp

```
#include <stdio.h>

int speed_limit()
{
    /*speed limit in mph*/
    int limit = 65;
    return limit;
}

int main() {
    /*speed limit in km/h*/
    int limit;
    limit = speed_limit();

    printf("Speed limit = %d km/h \n", limit);

    getchar();

    return 1;
}
```

# C-Bsp- Verbesserung

```
#include <stdio.h>
```

```
typedef int kmh;
```

```
typedef int mph;
```

```
mph speed_limit()
```

```
{
```

```
    /*speed limit in mph*/
```

```
    int limit = 65;
```

```
    return limit;
```

```
}
```

```
int main() {
```

```
    /*speed limit in km/h*/
```

```
    kmh limit;
```

```
    limit = speed_limit();
```

```
    printf("Speed limit = %d km/h \n", limit);
```

```
    return 1;
```

```
}
```

Besser,  
aber noch nicht gut

# C-Bsp, Version 3

```
#include <stdio.h>

struct kmh{
    int value;
};

struct mph{
    int value;
};

typedef struct kmh kmeter_pro_stunde;
typedef struct mph miles_per_hour;

miles_per_hour speed_limit()
{
    /*speed limit in mph*/
    miles_per_hour limit;
    limit.value = 65;
    return limit;
}

int main(){
    /*speed limit in km/h*/
    kmeter_pro_stunde limit;
    limit = speed_limit();

    printf("Speed limit = %d km/h \n", limit);

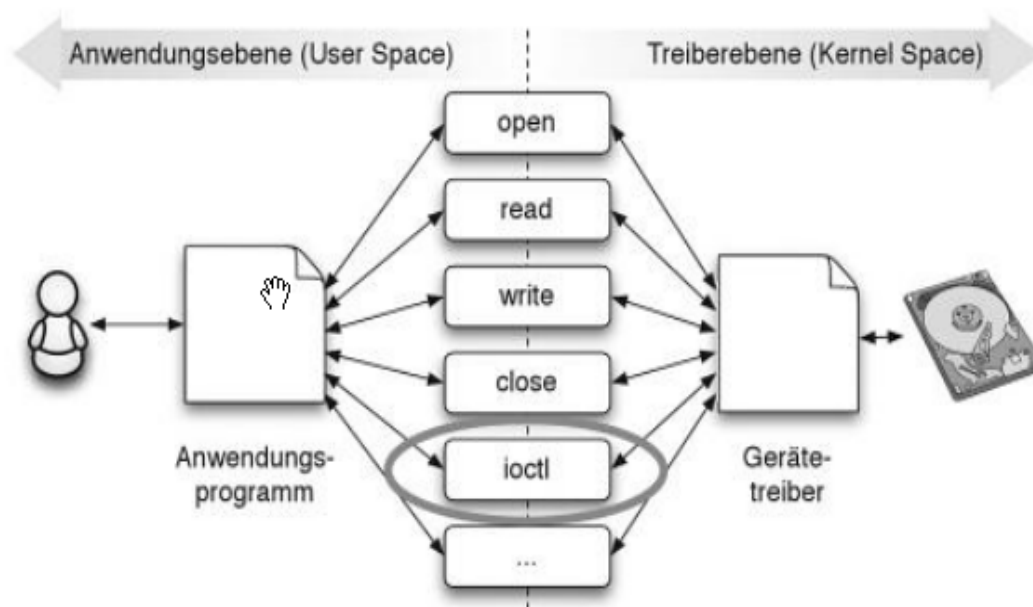
    return 1;
}
```

Fehler beim Kompilieren!

# Generische Schnittstellen

In manchen Anwendungsfällen ist eine strenge Typsicherheit nicht gewünscht.

Bsp: Kommunikation mit Gerätetreibern über eine einheitliche Schnittstelle (für verschiedenste Geräte)



# Generische Schnittstellen

Funktion `int ioctl(int file, int request, ... /*argument list*/);`

Variable Parameterliste ohne Definition der Typen!

Erhöhung der Typsicherheit durch: Verwendung von Container Typen.

Bsp: VARIANT (Bsp in C siehe nächste Seite) als Container für eine Vielzahl von fest definierten Datentypen.

# Variant als selbstbeschreibender Datentyp

variant.c

```
typedef struct {  
    VARTYPE vt;  
    WORD    wReserved1;  
    WORD    wReserved2;  
    WORD    wReserved3;  
    union {  
        LONG    lVal;  
        BYTE    bVal;  
        SHORT   iVal;  
        FLOAT   fltVal;  
        DOUBLE   dblVal;  
        LONG*    plVal;  
        BYTE *   pbVal;  
        SHORT *  piVal;  
        LONG *   plVal;  
        FLOAT *  pfltVal;  
        DOUBLE * pdblVal;  
        ...  
    } value;  
} VARIANT;
```

vt hält die  
Typinformation