**Ersetzbarkeit**
**Dynamisches Binden**
**Methoden aller Klassen**
**toString()**

# Eigenschaften von Untertypbeziehungen

$T$ ist Untertyp von $T$ (reflexiv)

wenn $R$ Untertyp von $S$ und $S$ Untertyp von $T$, dann auch $R$ Untertyp von $T$ (transitiv)

wenn $S$ Untertyp von $T$ und $T$ Untertyp von $S$, dann $S$ und $T$ gleich (antisymmetrisch)

ist $S$ ein Interface und Untertyp von $T$, dann ist auch $T$ ein Interface

ist $T$ eine Klasse und $S$ Untertyp von $T$, dann ist auch $S$ eine Klasse

ist $T$ eine Klasse, dann ist $T$ Untertyp von `Object`

jeder dynamische Typ ist eine Klasse

`null` ist kein Objekt von $T$, obwohl vom Compiler so akzeptiert

# Das Wesentliche: Ersetzbarkeit

wenn $S$ Untertyp von $T$, ist ein Objekt von $S$ verwendbar wo Objekt von $T$ erwartet

→ hauptsächlich bei Zuweisung und Parameterübergabe

wenn $S$ Untertyp von $T$ und $T$ beschreibt eine Methode, dann beschreibt auch $S$ eine Methode mit (vereinfacht) gleicher Signatur

→ Methode in allen Objekten der deklarierten Typen $S$ und $T$ aufrufbar

Voraussetzung: Methodenbeschreibungen (Kommentare) in $S$ und $T$ passen zusammen

→ Kommentar in $S$ beschreibt dieselbe Methode wie Kommentar in $T$

→ Kommentar in $S$ kann konkreter, Kommentar in $T$ abstrakter sein

→ Compiler kann Konsistenz von Kommentaren nicht prüfen – **unsere Aufgabe**

# Aufgabe: Deklarierter und dynamischer Typ

In Gruppen zu 2 bis 3 Personen:

Angenommen, die Variable `x` ist folgendermaßen deklariert: `A x = new B();`

Wie lautet der deklarierte und dynamische Typ von `x`?
Welche Untertypbeziehungen gelten zwischen `A`, `B` und `Object`?
Für welche Typen `T` wird `(T)x` zur Laufzeit sicher keinen Fehler melden?

Zeit: 2 Minuten

# Wozu Ersetzbarkeit?

Wiederverwendung durch Bilden konsistenter **Versionen**

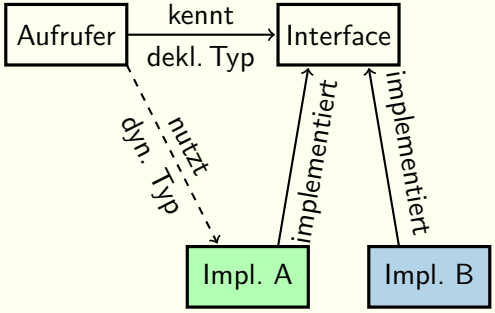
Teil der Software wird durch neue Version ersetzt;
Rest davon nicht betroffen, wenn neue Version Untertyp der alten Version ist

Wiederverwendung **innerhalb einer Anwendung**

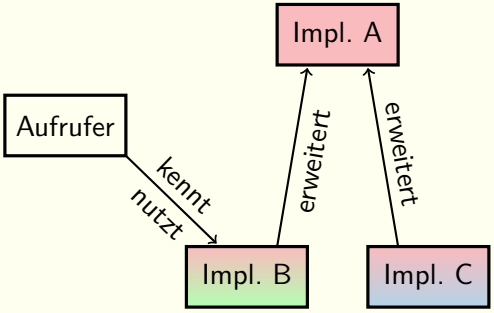
ein Programmteil kann mit unterschiedlichen Datenstrukturen umgehen,
wenn die Datenstrukturen Untertypen eines erwarteten Obertyps sind

# Ersetzbarkeit versus Vererbung

**Untertypbeziehung**




**Vererbungsbeziehung**



# Welche Methode wird ausgeführt?

**deklarierter Typ** bestimmt, welche Methoden **aufrufbar** sind

**dynamischer Typ** bestimmt, in welcher Klasse eine Methode **ausgeführt** wird
(erst zur Laufzeit bekannt, daher Aufruf **dynamisch** an Methode **gebunden**)

```
public static void testAssoc(Assoc assoc) {
    ...
    assoc.put(...);
    ...
}


    ...
    testAssoc(new TreeAssoc());
    ...
```

deklarierter Typ ist `Assoc`

aufrufbar weil `put` in `Assoc`,
ausgeführt in `TreeAssoc`
(zur Laufzeit bestimmt)

dynamischer Typ ist `TreeAssoc`,
war bei vorigem Aufruf `SimpleAssoc`

# Aufgabe: Vorhersehbarkeit der Ausführung

In Gruppen zu 2 bis 3 Personen:

Der Compiler kann nicht vorhersehen, in welcher Klasse eine Methode auszuführen ist.
Folglich werden wir das beim Programmieren auch nicht vorhersehen können.
Ist das ein Problem beim Verstehen eines Programms?

Zeit: 2 Minuten

# Dynamisches Binden statt `if`

```
assoc.put(...);
```

entspricht

```
if (assoc.getClass() == SimpleAssoc.class) {
    execute assoc.put(...) in SimpleAssoc;
} else if (assoc.getClass() == TreeAssoc.class) {
    execute assoc.put(...) in TreeAssoc;
}
```

gezielte Verwendung von dynamischem Binden vermeidet bedingte Anweisungen (siehe `DynBindList`)

# Statisches Binden

kein dynamisches Binden beim Methodenaufruf wenn

- Methode ist `static`
- oder Methode ist `private`
- oder Methode ist (im deklarierten Typ) `final`
- oder der deklarierte Typ kann keine Untertypen haben
- oder der Compiler kann feststellen, wo die Methode auszuführen ist

daher dynamisches Binden meist nur bei einem kleinen Anteil der Methodenaufrufe

# Aufgabe: Dynamisches versus statisches Binden

In Gruppen zu 2 bis 3 Personen:

Betrachten Sie ein Programm (z.B. die Lösung eines Aufgabenblatts).

Welche der Methodenaufrufe in diesem Programm erfolgen sicher über dynamisches Binden, welche möglicherweise, und welche verwenden sicher statisches Binden?

Zeit: 2 Minuten

# Erben und Überschreiben von Methoden

```
class X extends Y { ... }
```

`X` ist Unterklasse von `Y`,
`Y` ist Oberklasse von `X`

`X` **erbt** nicht-statische Methoden und Variablen von `Y`:
was nicht-statisch in `Y` definiert ist, ist automatisch auch in `X` definiert;
wenn `private`, dann nur über geerbte Methoden sichtbar

ererbbare Methoden können **überschrieben** werden:
wird eine in `Y` definierte Methode in `X` neu definiert,
dann existiert in `X` die neu definierte Methode, sie wird nicht von `Y` übernommen

`final` Methoden dürfen nicht überschrieben werden

# java.lang.Object

jede Klasse ist Untertyp von `Object`

→ Methoden von `Object` existieren in jeder Klasse:

| | |
|---|---|
| `Class getClass()` | dynamischer Typ von `this` |
| `boolean equals(Object obj)` | vergleicht `this` mit `obj` |
| `int hashCode()` | Hash-Code von `this` |
| `String toString()` | String-Darstellung von `this` |

Themen in EP2

| | |
|---|---|
| `Object clone()` | erzeugt eine Kopie von `this` |
| `void notify()` | Aufwecken eines auf `this` wartenden Threads |
| `void notifyAll()` | Aufwecken aller auf `this` wartender Threads |
| `void wait(...)` | aktueller Thread muss auf `this` warten |
| `void finalize()` | Destruktor, kaum verwendet |

# Überschreiben der Methoden von `Object`

`getClass` ist `final` → darf nicht überschrieben werden

`toString`, `equals` und `hashCode` werden häufig überschrieben

komplexe Einschränkungen auf `equals` und `hashCode` in `Object`, die durch Subtyping auf Untertypen übertragen werden; `equals` und `hashCode` müssen gemeinsam überschrieben werden

`toString` durch Sonderbehandlung tief in Java integriert

# Aufgabe: Übertragung von Einschränkungen

In Gruppen zu 2 bis 3 Personen:

Warum müssen Bedingungen, die für Methoden in `Object` formuliert sind, auch in allen anderen Klassen gelten?

Zeit: 2 Minuten

# Besonderheiten von `String toString()`

Wenn `x` Variable eines Referenztyps, dann

`"" + x` entspricht `"" + x.toString()`

`System.out.print(x)` entspricht `System.out.print(x.toString())`

Wert von `x` im Debugger durch `x.toString()` dargestellt

Implementierung in `Object` liefert Strings wie `"Klassenname@874"`

Bedingung: `x` unverändert $\rightarrow$ `x.toString().equals(x.toString())`

# Verwendbarkeit von `String toString()`

`x.toString()` ausführbar für *jeden* Ausdruck `x` eines Referenztyps wenn `x != null`

funktioniert nur weil

- jedes Objekt durch eine Klasse erzeugt wurde (Arrays als Ausnahme)
- jeder Ausdruck eines Referenztyps eine Klasse als dynamischen Typ hat
- jede Klasse Untertyp von `Object` ist
- `String toString()` in `Object` definiert ist

ist nur sinnvoll wenn

- `String toString()` in jeder Klasse überschrieben ist,
  sodass das Ergebnis entsprechende Objekte sinnvoll beschreibt

# Aufgabe: Objektvergleich mittels `toString()`

In Gruppen zu 2 bis 3 Personen:

Nehmen wir an, `x.toString().equals(y.toString())` sei für zwei beliebige Objekte `x` und `y` erfüllt. Warum folgt daraus nicht, dass `x` und `y` gleich sind?

Zeit: 2 Minuten

# Informationsgehalt von `toString()`

Informationsgehalt im Ergebnis von `toString()` frei wählbar

Beispiele:

| | |
|---|---|
| `"an object"` | versteckt Informationen gänzlich vor User |
| `"a T"` | für Typ `T`, zeigt dynamischen Typ für Debugging |
| `"T@874"` | wie in `Object`, dynamischen Typ und eindeutige Nr., Identität |
| `"[1, 2, 3]"` | z.B. Inhalt einer Datenstruktur, inhaltliche Gleichheit |

`toString()` automatisch generierbar für angenäherte inhaltliche Gleichheit, aber

- Generator kennt keine Semantik → oft falscher Informationsgehalt
- Aussehen erfüllt nur minimale Wünsche