6. Ersetzbarkeit und co

Quelle: ep2-06_Ersetzbarkeit_Dynamisches-Binden_Methoden-aller-Klassen_toString().pdf
Beinhaltet: Ersetzbarkeit Dynamisches Binden Methoden aller Klassen toString()

Eigenschaften von Untertypbeziehungen

1. **Reflexivität**:
   - **T ist Untertyp von T**: Jede Klasse oder jedes Interface ist ein Untertyp von sich selbst.
2. **Transitivität**:
   - **Wenn R Untertyp von S und S Untertyp von T, dann ist auch R Untertyp von T**: Wenn eine Klasse oder ein Interface ein Untertyp von einem anderen ist, und das andere ein Untertyp von einem weiteren, dann ist der erste auch ein Untertyp des dritten.
3. **Antisymmetrie**:
   - **Wenn S Untertyp von T und T Untertyp von S, dann sind S und T gleich**: Zwei Typen sind nur dann gleich, wenn sie sich gegenseitig als Untertypen haben.
4. **Interface und Untertypbeziehungen**:
   - **Wenn S ein Interface ist und Untertyp von T, dann ist auch T ein Interface**: Wenn ein Interface ein Untertyp eines anderen ist, muss das andere ebenfalls ein Interface sein.
5. **Klassen und Untertypbeziehungen**:
   - **Wenn T eine Klasse ist und S ein Untertyp von T, dann ist auch S eine Klasse**: Ein Untertyp einer Klasse muss ebenfalls eine Klasse sein.
6. **Vererbung von `Object`**:
   - **T ist eine Klasse, dann ist T Untertyp von Object**: Jede Klasse ist ein Untertyp von `Object`, der Wurzelklasse in Java.
7. **Null und Untertypbeziehungen**:
   - **Null ist kein Objekt von T, obwohl vom Compiler so akzeptiert**: `null` wird vom Compiler als gültiger Wert für jede Referenz akzeptiert, ist aber kein tatsächliches Objekt eines bestimmten Typs.

Ersetzbarkeit

Wesentliche Aspekte der Ersetzbarkeit

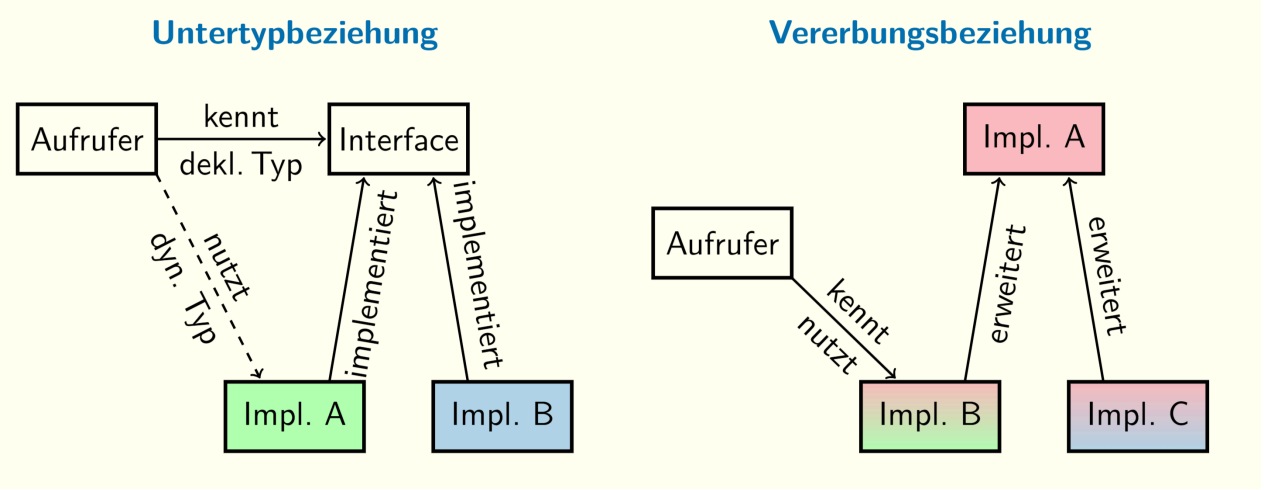
- **Ersetzbarkeit** bedeutet, dass ein Objekt eines Untertyps (`S`) dort verwendet werden kann, wo ein Objekt des Obertyps (`T`) erwartet wird. Dies passiert hauptsächlich bei:
  - **Zuweisung** (z. B. einem Wert vom Typ `T` kann ein Wert vom Typ `S` zugewiesen werden),
  - **Parameterübergabe** (z. B. ein Parameter vom Typ `T` kann durch ein Argument vom Typ `S` ersetzt werden, wenn `S` ein Untertyp von `T` ist).

Methode und Ersetzbarkeit

- Wenn **S Untertyp von T ist** und **T eine Methode beschreibt**, dann beschreibt auch **S eine Methode mit (vereinfacht) gleicher Signatur**. Die Methode kann in Objekten beider Typen (`S` und `T`) aufgerufen werden.
- **Voraussetzung**:
  - Die **Methodenbeschreibungen** (Kommentare) in den Klassen und Interfaces von `S` und `T` müssen übereinstimmen.
  - **Konsistenz der Kommentare**:
    - Kommentar in `S` beschreibt dieselbe Methode wie der Kommentar in `T`.
    - Kommentar in `S` kann konkreter sein, der Kommentar in `T` kann abstrakter sein.
    - **Compiler prüft nicht die Konsistenz von Kommentaren** – diese müssen vom Entwickler sichergestellt werden.

Wozu Ersetzbarkeit?

- **Wiederverwendung durch Bilden konsistenter Versionen**:
  - Ein Teil der Software kann durch eine neue Version ersetzt werden, wenn die neue Version ein Untertyp der alten Version ist.
  - Der Rest der Software bleibt dabei unverändert und benötigt keine Anpassungen.
- **Wiederverwendung innerhalb einer Anwendung**:
  - Ein Programmteil kann mit verschiedenen Datenstrukturen umgehen, wenn diese Datenstrukturen Untertypen eines gemeinsamen Obertyps sind.
  - Dadurch wird die Flexibilität und Wartbarkeit des Codes verbessert, da man verschiedene Implementierungen des gleichen Typs verwenden kann.


Untertypbeziehung
Aufrufer
kennt
dekl. Typ
Interface
nutzt
dyn. Typ
implementiert
implementiert
Impl. A
Impl. B
Vererbungsbeziehung
Impl. A
Aufrufer
kennt
nutzt
erweitert
erweitert
Impl. B
Impl. C

Welche Methode wird ausgeführt?

Deklarierter Typ vs. Dynamischer Typ

1. **Deklarierter Typ**:
   - Der **deklarierte Typ** bestimmt, welche **Methoden** aufgerufen werden können.
   - Es handelt sich um den Typ, der bei der **Deklaration** einer Variablen oder eines Parameters angegeben wird.
   - Beispiel: In der Methode `testAssoc(Assoc assoc)` ist der deklarierte Typ der Parameter `assoc` der Typ `Assoc`.
2. **Dynamischer Typ**:
   - Der **dynamische Typ** (auch zur Laufzeit bestimmter Typ genannt) bestimmt, in welcher **Klasse** eine Methode zur Laufzeit ausgeführt wird.
   - Der dynamische Typ ist der Typ des Objekts, das zur Laufzeit der Methode zugewiesen wird, nicht der deklarierte Typ.
   - Beispiel: Wenn in `testAssoc(new TreeAssoc())` das Objekt `new TreeAssoc()` übergeben wird, ist der dynamische Typ `TreeAssoc`, obwohl der deklarierte Typ `Assoc` ist.

Dynamisches Binden

1. **Was ist dynamisches Binden?**
   - **Dynamisches Binden** bedeutet, dass die Auswahl der Methode, die zur Laufzeit ausgeführt wird, basierend auf dem **dynamischen Typ** des Objekts erfolgt.
   - Das bedeutet, dass bei einem Methodenaufruf nicht sofort festgelegt wird, welche Implementierung ausgeführt wird. Die tatsächliche Implementierung wird zur Laufzeit anhand des **dynamischen Typs** des Objekts bestimmt.
2. **Vorteil von dynamischem Binden:**
   - **Vermeidung von bedingten Anweisungen**:
     - Ohne dynamisches Binden müsste man manuell prüfen, welcher Typ das Objekt hat, und dann die Methode entsprechend aufrufen, wie im Beispiel:

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

     - Dynamisches Binden ermöglicht es, solche **bedingten Anweisungen** zu vermeiden. Stattdessen wird die korrekte Methode zur Laufzeit automatisch aufgerufen.
3. **Beispiel für dynamisches Binden**:

```
class Assoc {
    public void put(String key, String value) {
        System.out.println("Assoc put");
    }
}

class TreeAssoc extends Assoc {
    @Override
    public void put(String key, String value) {
        System.out.println("TreeAssoc put");
    }
}

class SimpleAssoc extends Assoc {
    @Override
    public void put(String key, String value) {
        System.out.println("SimpleAssoc put");
    }
}
```

```java
public static void testAssoc(Assoc assoc) {
    assoc.put("key", "value");  // Dynamisches Binden
}

public static void main(String[] args) {
    testAssoc(new TreeAssoc());  // Gibt "TreeAssoc put" aus
    testAssoc(new SimpleAssoc());  // Gibt "SimpleAssoc put" aus
}
```

- In diesem Beispiel wird beim Aufruf von `assoc.put("key", "value")` die Methode zur Laufzeit basierend auf dem **dynamischen Typ** des Objekts (`TreeAssoc` oder `SimpleAssoc`) ausgeführt.

Statisches Binden

1. **Wann tritt statisches Binden auf?**
   - Statisches Binden passiert, wenn die Methode zur **Kompilierungszeit** (also vor der Ausführung) festgelegt wird. Es gibt mehrere Szenarien, in denen statisches Binden verwendet wird:
2. **Fälle, in denen statisches Binden auftritt**:
   - **Die Methode ist** `static` : Statische Methoden sind zur Kompilierungszeit bekannt, und der Methodenaufruf wird nicht zur Laufzeit aufgelöst.
   - **Die Methode ist** `private` : Private Methoden sind ebenfalls nicht vererbbar und werden zur Kompilierungszeit aufgelöst.
   - **Die Methode ist** `final` : Wenn eine Methode als `final` deklariert ist, kann sie nicht überschrieben werden, und der Methodenaufruf ist zur Kompilierungszeit bekannt.
   - **Der deklarierte Typ hat keine Untertypen**: Wenn der deklarierte Typ keine Untertypen hat (es gibt keine Vererbung), kann der Compiler genau bestimmen, welche Methode aufgerufen wird.
   - **Der Compiler kann den Aufruf eindeutig zuordnen**: In manchen Fällen kann der Compiler ohne Vererbung und ohne Polymorphismus den Methodenaufruf eindeutig festlegen.
3. **Beispiel für statisches Binden**:

```java
class Assoc {
    public static void put(String key, String value) {
        System.out.println("Assoc static put");
    }
}

class TreeAssoc extends Assoc {
    @Override
    public void put(String key, String value) {
        System.out.println("TreeAssoc put");
    }
```

```
}

public static void main(String[] args) {
    Assoc assoc = new TreeAssoc();
    assoc.put("key", "value");  // Hier wird statisches Binden
angewendet, weil put() statisch ist
}
```

- Da die Methode `put()` hier **statisch** ist, wird die Methode zur Kompilierungszeit festgelegt, unabhängig vom dynamischen Typ des Objekts.

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`	Themen in EP2
`boolean equals(Object obj)`	vergleicht `this` mit `obj`	
`int hashCode()`	Hash-Code von `this`	
`String toString()`	String-Darstellung von `this`	
`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` $\rightarrow$ 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

toString()

Besonderheiten von `toString()`

1. **Verwendung von `toString()`**:
   - Wenn `x` eine **Variable eines Referenztyps** ist, dann entspricht:

     ```
     "" + x
     ```

     dem Aufruf:

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

   - `System.out.print(x)` entspricht `System.out.print(x.toString())`.
     - Dies bedeutet, dass `x.toString()` automatisch aufgerufen wird, wenn ein Objekt als Argument an `System.out.print()` übergeben wird.
2. **Debugging**:
   - Der **Wert von** `x` wird im Debugger durch `x.toString()` dargestellt.
3. **Standardimplementierung von `toString()`**:
   - Die **Implementierung in** `Object` liefert einen String im Format:

     ```
     Klassenname@874
     ```

     - Dies gibt den **Klassenname** und den **Hashcode** des Objekts zurück.
4. **Bedingung**:
   - Der Ausdruck:

     ```
     x.toString().equals(x.toString())
     ```

     ergibt `true`, wenn `x` unverändert bleibt.

Verwendbarkeit von `toString()`

1. **Verwendbarkeit**:
   - `x.toString()` ist für jeden **Ausdruck eines Referenztyps** ausführbar, sofern `x != null`.
   - Dies funktioniert nur, weil **jedes Objekt durch eine Klasse erzeugt wird** (mit der Ausnahme von **Arrays**).
2. **Vererbung**:
   - Jeder Ausdruck eines Referenztyps hat eine **Klasse als dynamischen Typ**, und jede Klasse ist ein **Untertyp von** `Object`.

- Die Methode `toString()` ist in `Object` definiert.

3. **Überschreiben von `toString()`**:
   - Die Methode `toString()` ist nur dann wirklich sinnvoll, wenn sie in jeder Klasse überschrieben wird.
   - Der Überschreibungszweck ist, dass `toString()` dann einen sinnvollen **String** liefert, der das Objekt beschreibt.

Informationsgehalt von `toString()`

1. **Informationsgehalt im Ergebnis**:
   - Der Informationsgehalt des Ergebnisses von `toString()` ist frei wählbar, aber es gibt gängige Konventionen:
     - **"an object"**: Versteckt die Informationen vollständig vor dem Benutzer.
     - **"a T"**: Gibt den **dynamischen Typ** des Objekts zurück, nützlich für Debugging.
     - **"T@874"**: Ähnlich der Standardimplementierung von `toString()` in `Object`. Zeigt den **dynamischen Typ** und eine eindeutige Nummer (oft Hashcode) des Objekts.
     - **"[1, 2, 3]"**: Zum Beispiel, der **Inhalt einer Datenstruktur**, was eine inhaltliche Gleichheit darstellt.
2. **Automatische Generierung**:
   - `toString()` kann automatisch generiert werden, um die inhaltliche Gleichheit von Objekten (insbesondere von Collections) widerzuspiegeln.
   - **Automatisch generierte `toString()`-Methoden** kennen jedoch keine **Semantik** und liefern daher oft **falschen Informationsgehalt**.
     - Zum Beispiel wird bei einer Collection ohne semantische Bedeutung der Inhalt als String ausgegeben, aber es könnte in manchen Fällen nicht den gewünschten Informationsgehalt liefern.
3. **Aussehen**:
   - Die Standardimplementation von `toString()` entspricht nur den **minimale Anforderungen**, die die Darstellung eines Objekts in Form eines Strings betrifft.

Zusammenfassung

- `toString()` ist eine Methode, die in jeder Klasse überschrieben werden sollte, um sinnvolle Informationen über ein Objekt bereitzustellen.
- Die **Standardimplementierung von `toString()`** gibt nur den Klassennamen und den Hashcode zurück, was oft nicht aussagekräftig ist.
- Der **Informationsgehalt von `toString()`** ist flexibel und kann für Debugging, Fehlerverfolgung oder die Darstellung des Inhalts von Datenstrukturen angepasst werden.

- Automatisch generierte `toString()`-Methoden liefern in der Regel nur grundlegende Informationen und sind nicht immer semantisch korrekt.