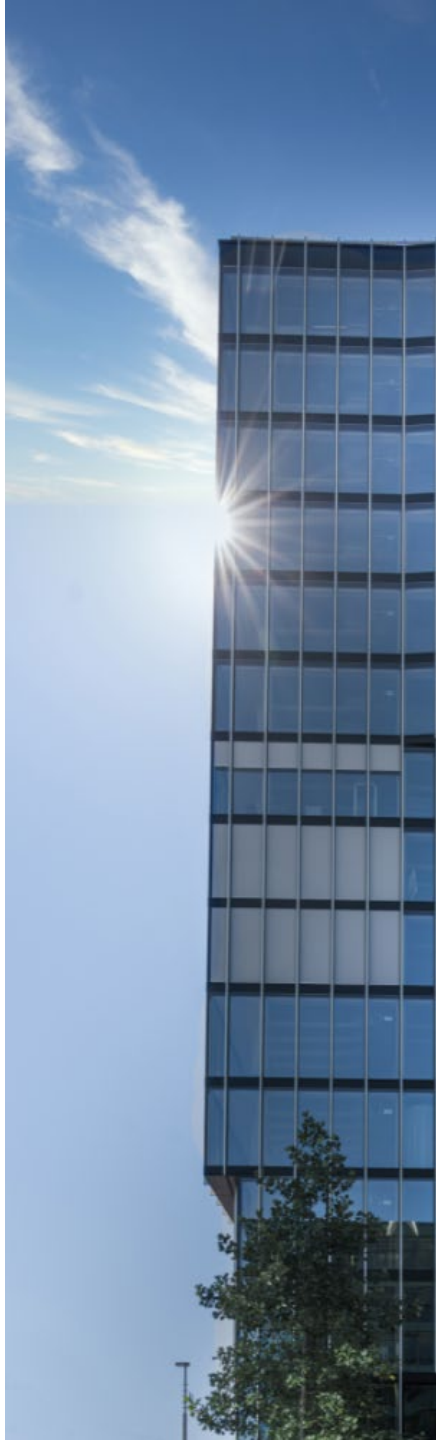
HSLU Hochschule Luzern

Objektorientierte Programmierung

Polymorphie

Roland Gisler

FH Zentralschweiz



Inhalt

- Polymorphie – Was ist das?
- Überladen von Methoden und Konstruktoren
- Überschreiben von Methoden
- Konzept des «Subtyping»
- Statischer und dynamischer Typ
- Parametrisierte Klassen (Generics)
- Zusammenfassung

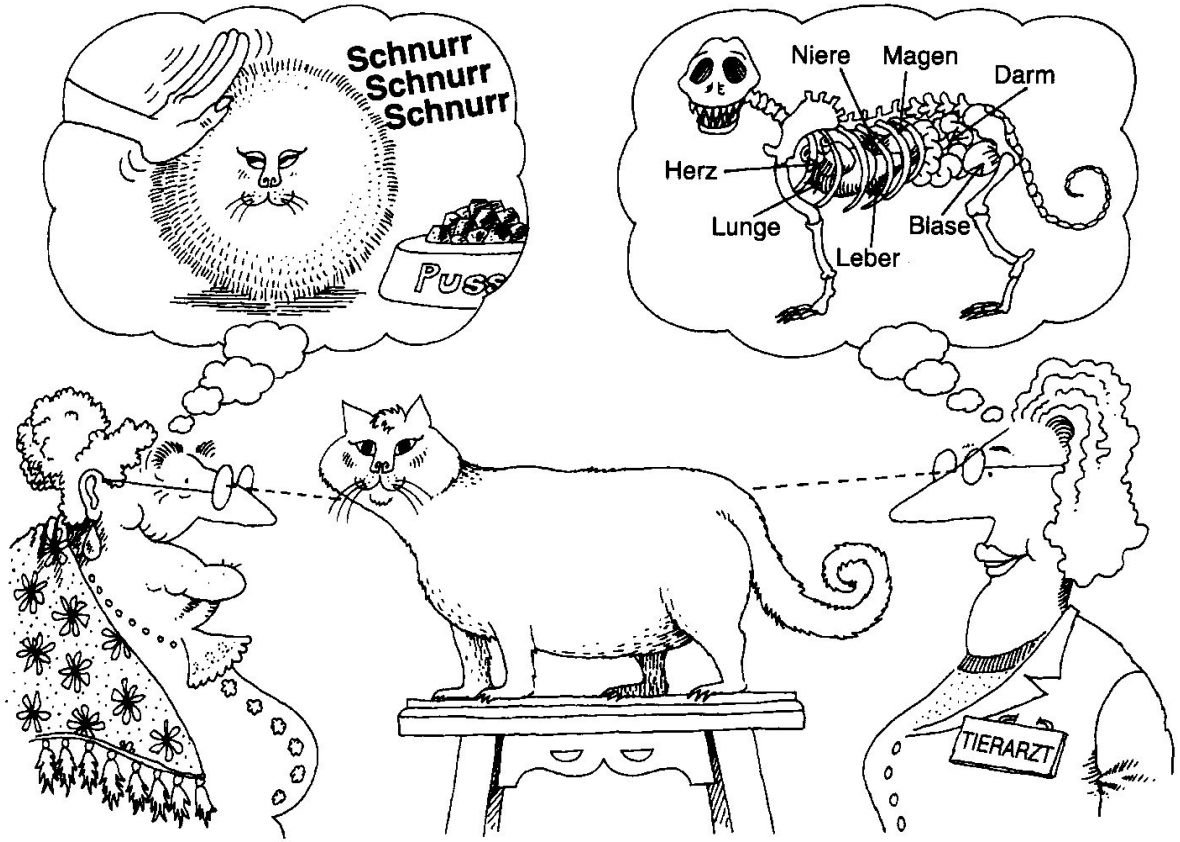
Lernziele

- Sie können den Begriff Polymorphismus anhand von Beispielen erklären.
- Sie kennen die Technik des Überladens und des Überschreibens von Methoden.
- Sie sind mit dem Konzept des Subtyping vertraut.
- Sie können zwischen statischem und dynamischem Typ unterscheiden.
- Sie können generisch implementierte Klassen nutzen.

Polymorphie – Was ist das?

Polymorphie - Vielgestaltigkeit

- Sie erinnern sich: Es liegt auch im Auge des Betrachters…



Die Abstraktion konzentriert sich auf die wesentlichen Charakteristika eines Objekts, relativ zur Perspektive des Betrachters.

Polymorphie / Polymorphismus

- Polymorphie = Vielgestaltigkeit
  - Ist ein zentrales Element der Objektorientierung.
- Die Fähigkeit zur Polymorphie hat zur Folge, dass
  - sich Methoden abhängig vom Parametertyp unterschiedlich verhalten können.
  - Objekte von unterschiedlichen Typen sich bei gleicher Behandlung unterschiedlich verhalten können.
  - die gleiche Klasse für unterschiedliche Typen parametrisiert werden kann.
- Polymorphie führt zu einfacherem Programmcode. Sie ermöglicht die Handhabung von Objekten unterschiedlicher Klassen auf einer allgemeineren Ebene.
  - Programme werden damit flexibler und leichter erweiterbar.

Überladen (overload) von Methoden

Überladen von Methoden (method overloading)

- Sie erinnern sich an die Signatur einer Methode:

```
public float max(float a, float b) { … }
```

➔ Signatur: `max(float, float)`

- Signatur besteht somit aus dem Namen der Methode und ihrer Parameterliste (nur Menge, Typen und Reihenfolge, aber **ohne** Namen).
  - Eine Klasse darf mehrere Methoden mit gleichem Namen haben, solange die Parameterlisten unterschiedlich sind.
- Einsatzmöglichkeiten:
  - «Gleiche» (logische) Funktionalität für unterschiedliche Typen.
  - Vereinfachte Methoden mit weniger Parametern, welche sinnvolle Defaultwerte verwenden.

Überladen von Methoden: Beispiele

- Beispiel **1**: Methode **`max(a, b)`** für zwei verschiedene Parametertypen (**`int`** und **`float`**):

```
public int max(int a, int b) { … }
public float max(float a, float b) { … }
```

- Beispiel **2**: Methode **`increment(n)`** mit optionalem Inkrement: Entweder man setzt das Inkrement explizit, oder aber es wird ein sinnvoller Defaultwert (hier z.B. **1**) verwendet:

```
public int increment(int increment) { … }
public int increment() {
    return this.increment(1);
}
```

Overloading - Empfehlungen

- Nicht alles, was geht ist auch vernünftig! **Schlechtes** Beispiel:

```
foo(int a, int b) { … }
foo(long a, int b) { … }
```

- Empfehlung: Möglichst **klar** unterscheidbare, **eindeutige** Signaturen wählen, das heisst:
  - Möglichst unterschiedliche **Parameteranzahl**.
  - **Eindeutige**, nicht vertauschbare Typenreihenfolge.
  - Nicht zu viele Parameter.
- Bei Überladungen mit unterschiedlicher Parameterzahl erfolgt die Implementation in der Methode mit **maximaler** Parameterzahl.
  - Methoden mit weniger Parametern rufen diese dann mit Defaultwerten auf ➔keine redundanten Implementationen!

Überladen von Konstruktoren

Überladen von Konstruktoren

- Grundsätzlich **identische** Möglichkeiten wie bei Methoden.
- Einzige Spezialität: Aufruf eines überladenen Konstruktors aus einem anderen Konstruktor (derselben Klasse) erfolgt nicht mit dem Namen, sondern mit dem Schlüsselwort **`this(…)`**
- Beispiel:

```
public Person(final int id, final String name) {
    ...
}

public Person(final int id) {
    this(id, "unbekannt");
}
```

Überladen von Konstrukturen - Empfehlungen

- **Identische Empfehlungen** wie bei Overload von Methoden:
- Möglichst klar unterscheidbare, **eindeutige** Parameter wählen:
  - Möglichst unterschiedliche **Parameteranzahl**.
  - **Eindeutige**, nicht vertauschbare Typenreihenfolge.
  - Nicht zu viele Parameter.
- Bei Konstruktoren mit unterschiedlicher Parameterzahl erfolgt die eigentliche Implementation meist im Konstruktor mit der **maximalen** Parameterzahl.
  - Konstruktoren mit weniger Parametern rufen diesen dann mit `this(…)` und sinnvollen (Default-)Werten auf.

  ➔ Keine redundante Implementation, weniger Fehler.

Überschreiben (override) von Methoden

Überschreiben von Methoden

- Überschreiben von Methoden erfolgt **immer** im Kontext einer **Vererbung** oder der **Implementation eines Interfaces**.
- Eine Methode der Oberklasse **kann** (sofern sie nicht als `final` markiert wurde) in einer Unterklasse überschrieben werden.
  - **Identischer Header** mit spezifischer Implementation.
- Die Unterklasse kann über das Schlüsselwort `super.methode(…)` auf die Implementation der Oberklasse zurückgreifen.
  - bedingt natürlich entsprechende Sichtbarkeit.
- Bei der Implementation eines Interfaces, oder wenn die Methoden in der Oberklasse abstrakt sind, wir das Überschreiben (konkrete Implementation) sogar erzwungen.

Beispiel 1: `toString()` von `Object`

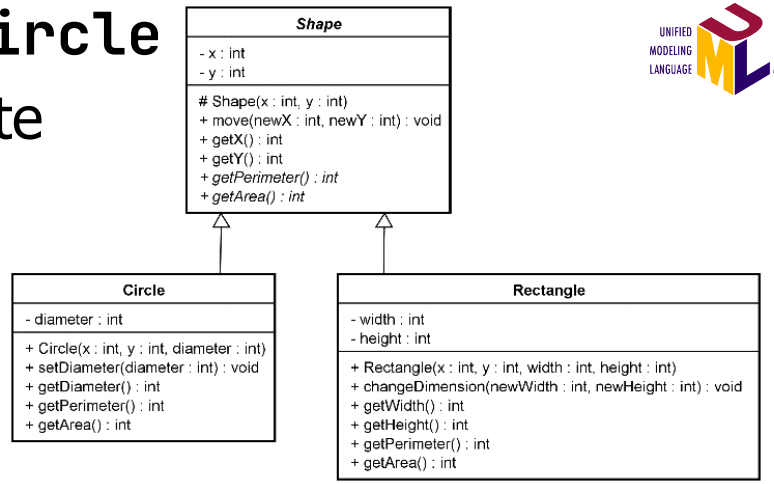
- Jede Klasse in Java erbt letztlich von der Klasse **`Object`**.
- Die Klasse **`Object`** implementiert u.a. eine nicht-finale Methode `String` **`toString()`**, welche dazu dient, eine einfache **`String`**-Repräsentation eines Objektes (z.B. für Logging und Debugging) zu produzieren.
  - Die Implementation von **`Object`** gibt nur Klassenname und Hashcode (als Hexadezimalwert codiert) aus ➔ meist sinnlos.
- **Jede** (!) Klasse sollte diese Methode überschreiben.
  Beispiel für die **`Temperatur`**-Klasse aus der Übung:

```
@Override
public String toString() {
    return "Temperatur[kelvin=" + this.kelvin + "]";
}
```

Beispiel 2: Überschreiben von abstrakten Methoden

- Wird eine abstrakte Klasse durch eine Spezialisierung konkretisiert, werden wir sogar **gezwungen**, die fehlende Implementation der abstrakten Oberklasse in der Unterklasse zu **überschreiben**.
  - Das bezeichnet man in diesem Kontext meist (vereinfacht) als Implementation (der abstrakten Methoden); gleiches gilt auch bei der Implementation von Interfaces.

- Beispiel (➔Details siehe Input O07 Vererbung): Die abstrakte Klasse **`Shape`** wird zu **`Circle`** und **`Rectangle`** spezialisiert: Abstrakte Methoden **`getPerimeter()`** und **`getArea()` müssen** in den Spezialisierungen überschrieben werden.



Beispiel – Überschreiben von Methoden

```
public final class Rectangle extends Shape {
    ...
    @Override
    public int getPerimeter() {
        return (2 * this.width) + (2 * this.height);
    }

    @Override
    public int getArea() {
        return (this.width * this.height);
    }
    ...
}
```

- **`@Override`** ist eine (optionale, aber empfohlene) ➔Annotation, welche sowohl dem Kompiler als auch uns klar macht, dass wir hier **bewusst** eine Methode überschreiben (wollen).

Überschreiben von Konstruktoren?

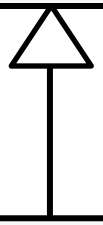
Kein Überschreiben von Konstruktoren

- Konstruktoren können **nicht** überschrieben werden!
  - Weil: Jede Klasse hat einen **eindeutigen** Namen und somit sind ihre Signaturen à priori unterschiedlich.
- Es gibt einen impliziten Default-Konstruktor (ohne Parameter).
- Sobald eine Klasse einen eigenen Konstruktor implementiert, werden alle Konstruktoren der Oberklasse(n) überdeckt.
- Die Konstruktoren der direkten Oberklasse können aber mit **`super(…)`** (analog zu **`this(…)`** beim Overloading) explizit aufgerufen werden.
  - **`super(…)`** muss aber zwingend das erste Statement sein!
  - Fehlt es, wird vom Kompiler implizit ein **`super()`** eingefügt.
  - Kennt die Oberklasse keinen Default-Konstruktor: Fehler!

Beispiel – Aufruf von Konstruktoren der Superklasse

- Konstruktoren der Klasse **`Person`** und der Klasse **`Mitarbeiter`** (welche von **`Person`** erbt):

```
public class Person {
    public Person(final String name) {…}
}
```



```
public class Mitarbeiter extends Person {
    public Mitarbeiter(final String name) {
        super(name);
        this.gehalt = 1000;
    }
}
```

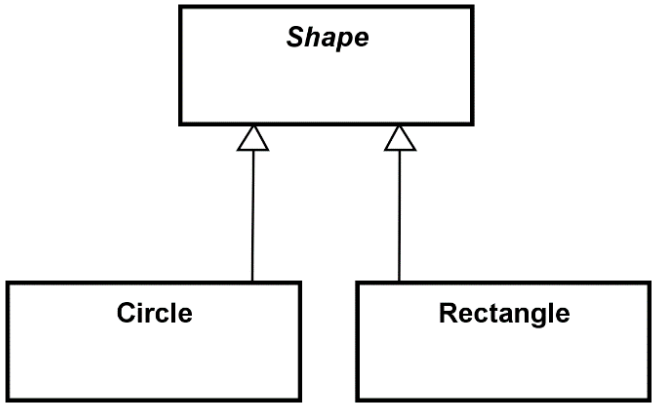
Subtyping und Casting

Konzept des Subtyping (Untertypen)

- Bei Vererbung gilt (sowohl bei Klassen und Interfaces):
  Der Typ einer Unterklasse ist ein Subtyp (Untertyp) des Typs der Oberklasse.
  - Beispiel: Kernobst ist der (Super-)Typ, ein Apfel ist ein Subtyp.
- Bei der Implementation von Interfaces gilt:
  Der Typ der implementierenden Klasse ist ein Subtyp (nimmt die Rolle ein) des Interfaces.
  - «Schaltbar» ist der Typ, die (schaltbare) Lampe ein Subtyp.
- Jede Referenzvariable kann nicht nur Objekte des deklarierten Typs aufnehmen, sondern auch Objekte **aller Subtypen** dieses Typs!
  - Beispiel: In einen Obstkorb kann ich Äpfel und Birnen legen!
  - Diese Polymorphie bezeichnet man auch als ➔ Substitution.

Beispiel für Subtypen mit Klassen

- Beispiel mit **`Shape`**, **`Circle`** und **`Rectrangle`**:
  - **`Circle`** und **`Rectangle`** sind Subtypen von **`Shape`**.
  - Eine Variable vom Typ **`Shape`** kann darum auch Objekte vom Typ **`Rectangle`** und **`Circle`** aufnehmen!



- Folgendes ist also möglich:

```
Shape shape1 = new Circle(…);
Shape shape2 = new Rectangle(…);
```

- **Wichtig**: Über den (Referenz-)Typ **`Shape`** stehen dann **nur** die Eigenschaften von **`Shape`** zur Verfügung!
  - Bleiben aber **trotzdem** ein **`Circle`**- bzw. **`Rectangle`**-Objekte!

Statischer und dynamischer Datentyp

- Mit dem Konzept des Subtyping können (müssen) wir somit zwischen **statischem** und **dynamischem** Typ unterscheiden.
- **Statischer** Typ: Der Typ z.B. einer Variable, welcher zum Programmierzeitpunkt festgelegt wird.
  - Beispiele (lokale Variable, formaler Parameter oder Attribut):
    `Shape form;` ➔ **`Shape`** ist der statische Typ.
    `Circle kreis;` ➔ **`Circle`** ist der statische Typ.
- **Dynamischer** Typ: Der **tatsächliche** Typ des **Objektes** zur Laufzeit, auf welches die Referenz zeigt, kann auch ein Subtyp sein.
  - Beispiel: `Object object = new Circle();`

➔ Während der statische Typ hier «nur» **`Object`** ist, ist der dynamische Typ des Objektes **`Circle`**!

Casting zwischen Typ und Subtyp

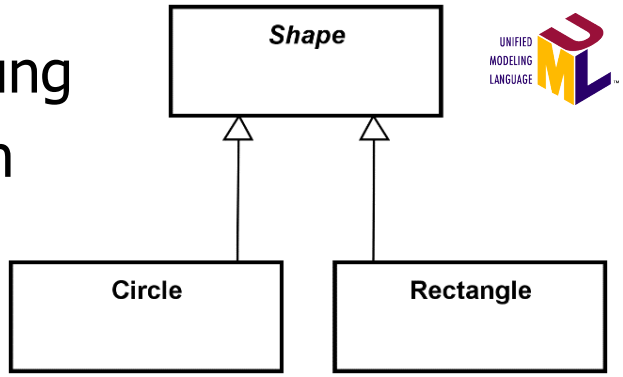
- Innerhalb einer Typhierarchie kann man zwischen Super- und Subtypen casten.
  - Ein Cast verändert aber **nie** den dynamischen Typ des eigentlichen Objektes, sondern ändert **nur** den Typ der Referenz – also nur als was wir es betrachten!

- Da Vererbungshierarchien gerichtete Beziehungen sind, unterscheidet man die Castings in zwei Richtungen:
  - **Upcasting***: Vom Subtyp zum Typ,
    also von der Spezialisierung hoch zur Generalisierung.
  - **Downcasting***: Vom Typ zum Subtyp,
    also von der Generalisierung runter zur Spezialisierung.
- Die Warnung vorweg: Analog zu den Castings zwischen elementaren Datentypen ist auch hier **nicht** alles möglich!

*zur Erinnerung: Vererbungsbäume «hängen» in der Regel.

Upcasting – Cast vom Subtyp zum (Super-)Typ

- **Upcasting**: Die Referenz einer Spezialisierung (z.B. **`Circle`**) wird zu einem generalisierten Typ (z.B. **`Shape`** oder **`Object`**) gecastet.



- Die Referenz eines Subtypes kann **jederzeit** problemlos auf einen Supertyp gecastet werden.
  - Wir können jede Spezialisierung auch immer als ihre Generalisierung betrachten: Ein Apfel ist immer ein Kernobst.
- Upcasting findet **implizit**① statt, kann aber auch explizit② angegeben werden. Folgende Statements sind somit gleichwertig:

```
Shape shape1 = new Rectangle(…); ①
Shape shape2 = (Shape) new Rectangle(…); ②
```

Downcasting – Cast vom (Super-)Typ zum Subtyp

- **Downcasting**: Die Referenz einer Generalisierung wird zu einem spezialisierten Typ gecastet.
- **Achtung**: Das ist nur möglich, wenn der **dynamische** Typ des Objektes kompatibel, also tatsächlich dem gewünschten Cast-Typ (oder einem Subtyp davon) entspricht.
  - Nur wenn ich weiss, dass das «Obst» in meiner Hand tatsächlich ein Apfel ist, kann ich dieses auch als «Apfel» behandeln.
- Downcastings müssen **immer explizit** erfolgen, und es stellt sich teilweise erst zur Laufzeit heraus, ob es zulässig ist (weil abhängig vom dynamischen Datentyp).
  - Bei fehlerhaften/unerlaubten Castings, resultiert ein Laufzeitfehler (runtime error)!

Downcasting – Beispiele

- Als Voraussetzung sei gegeben:

```
Object object = new Rectangle(…);
```

➔ Der statischer Typ ist **`Object`**, der dynamische Typ **`Rectangle`**.

- Die folgenden Downcastings sind somit **erlaubt** und möglich:

```
Shape shape = (Shape) object;
Rectangle rectangle = (Rectangle) object;
```

- **Nicht** möglich / nicht erlaubt sind hingegen:

```
Circle circle = (Circle) object;
Person person = (Person) object;
```

- Downcastings müssen **immer explizit** definiert werden.

Analogie – Vergleich mit Casts bei elementaren Datentypen

- Java macht automatische (implizite) Casts von «kleineren» zu «grösseren» elementaren Datentypen, weil diese (meist) gefahrlos sind. Die Umkehrung geht aber nur explizit, wir müssen Java dabei versichern, dass wir uns der Gefahr eines potenziellen Bereichsüberlaufes und/oder Genauigkeitsverlustes bewusst sind.
- Ähnlich verhält es sich mit Castings zwischen Klassentypen: Während der Cast von der Spezialisierung zur Generalisierung (Upcasting) immer problemlos möglich ist, müssen wir bei der Umkehrung (Downcasting) Java explizit versichern, dass wir wissen, dass ein kompatibler dynamischer Typ vorhanden ist.
- Der Typ eines Objektes lässt sich übrigens auch mit einer Expression abfragen: `(object instanceof KlassenName)`

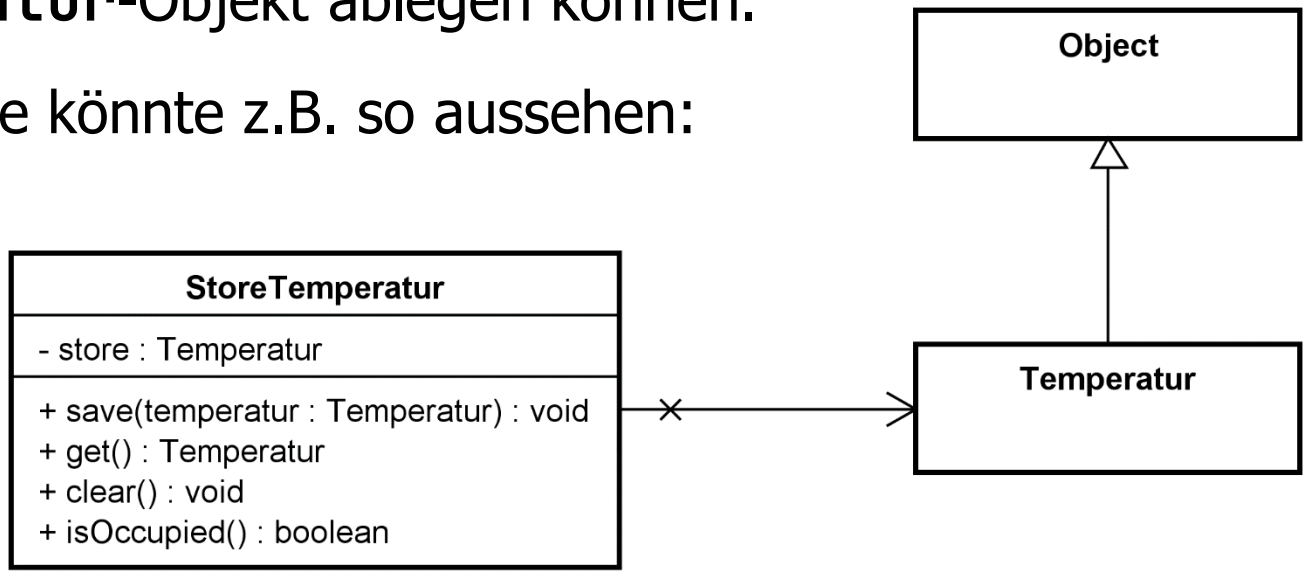
Jedes Objekt in Java ist auch ein `Object`

- Die Klasse **`Object`** ist in der Vererbungshierarchie von Java **die** Basisklasse.
  - Darum kennt auch jede Klasse in Java die Methoden, die auf **`Object`** deklariert sind.
- Durch das Subtyping erklärt sich nun auch, dass einer (Referenz-)Variablen vom Typ **`Object`** tatsächlich ein Objekt **jeder beliebigen** Java-Klasse zugewiesen werden kann.
- Diesen Umstand nutzt man beispielsweise, um flexible Schnittstellen oder Methoden zu implementieren, welche für beliebige (oder zumindest erweiterbare) Mengen von Typen nutzbar sind.
  - Ursprünglich haben viele Datenstrukturen in Java den Typ **`Object`** verwendet, heute arbeitet man aber mit ➔Generics.

Parametrisierte Klassen (Generics)

Ein Beispiel: Speicher für `Temperatur`

- Eine einfache Anforderung: Wir möchten einen einfachen «Speicher» implementieren, in welchen wir ein **einziges `Temperatur`-Objekt** ablegen können.
- Die Klasse könnte z.B. so aussehen:



- Die Klasse ist **typsicher**: Sie kann wirklich nur **`Temperatur`**-Objekte (bzw. auch Subtypen davon, wenn **`Temperatur`** spezialisiert werden darf) aufnehmen.

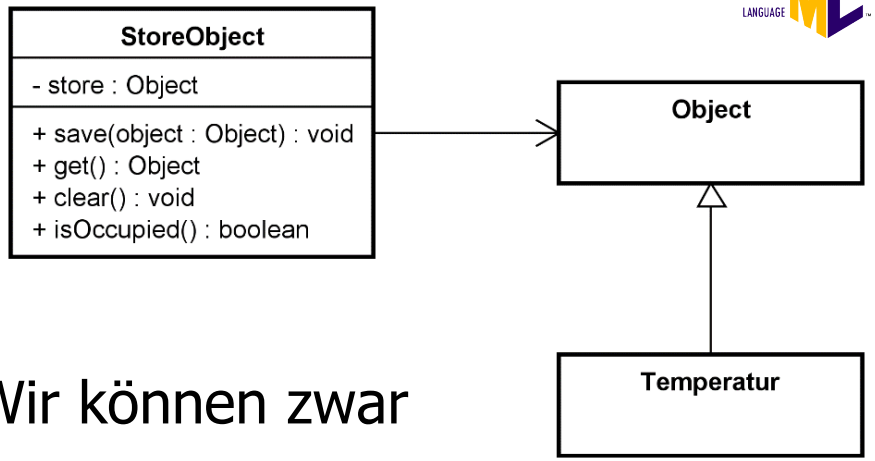
Neue Anforderung: Speicher für `Rectangle` und `Circle`

- Nun möchten wir auch einen Speicher für die beiden Klassen **`Rectangle`** und **`Circle`**!
- **Schlechte** Idee: Die **`StoreTemperatur`**-Klasse zu kopieren und den Datentyp anzupassen wäre ineffizient und würde viele Coderedundanzen erzeugen.
  - Auch wenn wir den Supertyp **`Shape`** verwenden würden.
- **Bessere** Idee: Wir könnten ja **`Object`** als Typ für den **`Store`** verwenden?
  - **`Object`** ist der Supertyp aller Klassen, und kann somit **jedes** beliebige Objekt aufnehmen.

➔ Haben wir damit den **universellen** Speicher erfunden?

Der universelle Speicher: `StoreObject`

- Tatsächlich hätten wir mit dieser Implementation jeden möglichen existierenden und auch zukünftigen Typ abgehandelt!



- Die Sache hat aber einen Haken. Wir können zwar ganz einfach jedes beliebige Objekt darin speichern:

```
storeObject.save(new Temperatur(…));
```

- Aber wenn wir das Objekt mit **`get()`** wieder holen wollen, kriegen wir dieses aber nur über den Supertyp **`Object`**, und müssen es immer wieder explizit auf den tatsächlichen Typ downcasten:

```
Temperatur t = (Temperatur) storeObject.get();
```

➔ Das ist nicht wirklich elegant, war aber bis Java 1.4 normal!

Die Lösung: Parametrisierbare Klassen (Generics)

- Mit Java 1.5 wurden sogenannte **Generics** eingeführt.
- Generics erlauben uns, konkrete Klassen zu schreiben, deren exakt verwendete Typen beim Einsatz **parametrisiert** werden können.
- Bei der Deklaration einer Variablen von einem **generischen Klassentyp** geben wir zusätzlich noch an, für welchen **Datentyp** diese verwendet werden soll:

```
StoreGeneric<Temperatur> tempStore = new StoreGeneric<>();
```

- Hinweis: Da beim Konstruktor zwingend **derselbe** Typ stehen muss, darf man den Typ dort auch weglassen.
  - Die leeren spitzen Klammern "**<>**" bezeichnet man als sogenannten «Diamond»-Operator (seit Java 1.7 verfügbar).

Implementation einer generischen Klasse (optional)

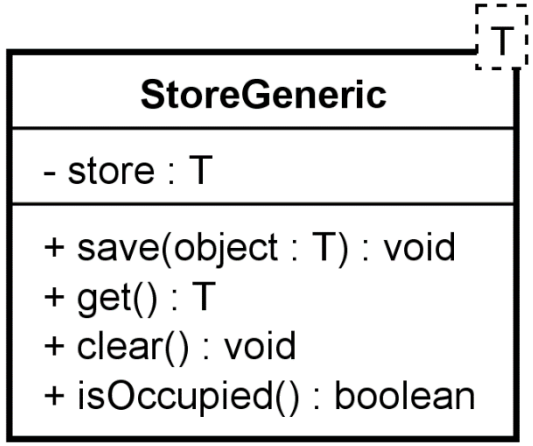
```
public final class StoreGeneric<T> {

    private T store;

    public void save(final T object) {
        this.store = object;
    }

    public T get() {
        return this.store;
    }

    ...
}
```



- Einfacher Tipp für das Verständnis:
  Stellen Sie sich vor, an der Stelle von T würde z.B. **`Object`** stehen!

Hinweise zu generischen Klassen und Interfaces

- Neben generischen Klassen gibt es auch generische Interfaces.
- Ab Java 1.5 wurde ein Grossteil der Klassen von Java (speziell z.B. bei den Datenstrukturen) generisch implementiert.
- Die Verwendung von generischen Klassen ist in der Regel relativ einfach und intuitiv, wenn man das Konzept verstanden hat.
- Die eigene Implementation von generischen Klassen kann hingegen sehr anspruchsvoll und trickreich sein.
- Wir beschränken uns daher vorerst nur auf die Verwendung.

Zusammenfassung

- Polymorphie = Vielgestaltigkeit.
- Überladen (overload) von Methoden und Konstruktoren:
  Gleicher Name, aber unterschiedliche Parameterliste.
- Überschreiben (override) von Methoden:
  Identischer Methodenkopf, (Re-)Implementation in Spezialisierung oder bei der Implementation eines Interface.
- Supertyp und Subtyp (Typhierarchie).
- Statischer und dynamischer Datentyp.
- Generics: Mit Typ parametrisierbare Klassen.

Fragen?