universität innsbruck

Die Klasse Object

Programmiermethodik

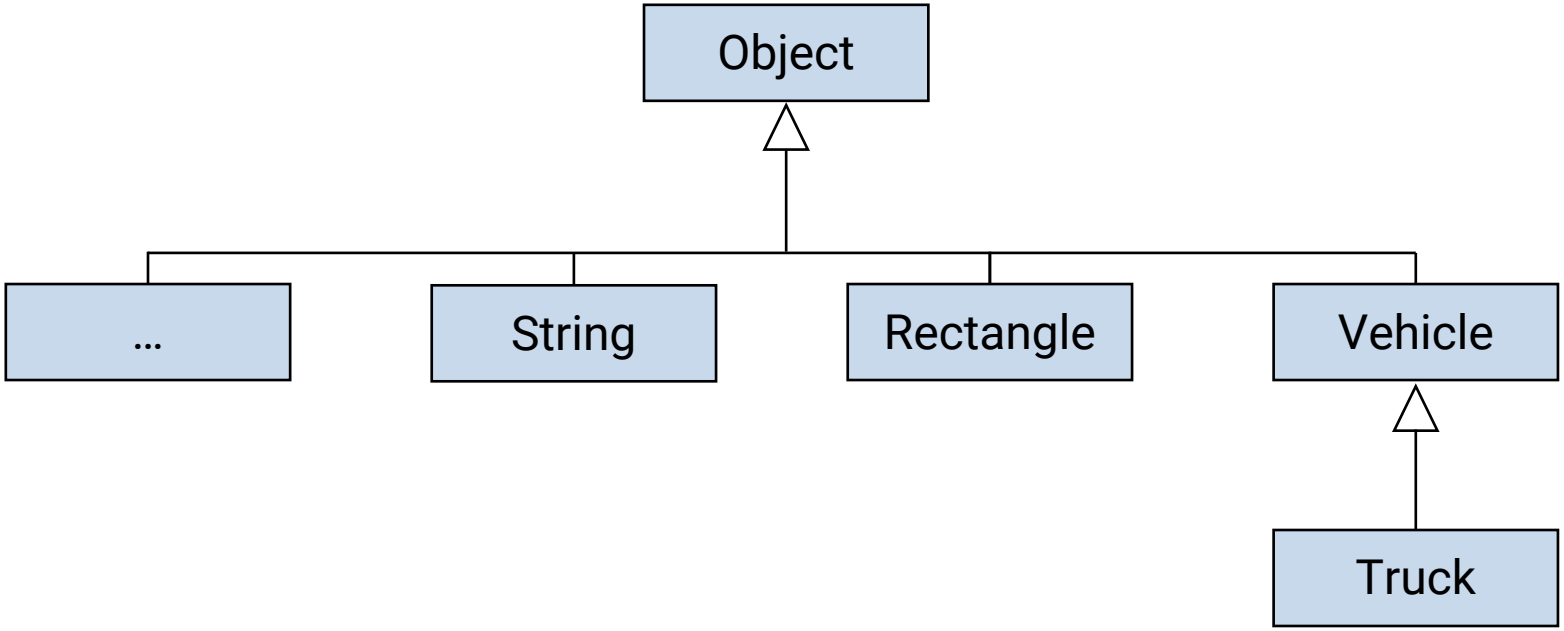
Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Die Wurzelklasse Object

- Jede Klasse, die von keiner Klasse erbt, erbt automatisch von der Wurzelklasse `Object`.
  - Folgende Definitionen sind äquivalent:

```
public class className {…}
public class className extends Object {…}
```

- Damit werden alle Klassen direkt oder indirekt von `Object` abgeleitet.



Vordefinierte Methoden

- `Object` bietet einige Methoden an, die an jede Java-Klasse vererbt werden.
- Methoden können überschrieben werden.
- Auszug:

  `public String toString()`
  - Liefert eine lesbare Repräsentation des Objekts.

  `public boolean equals(Object x)`
  - Prüft ob das Zielobjekt und x gleich sind.

  `public int hashCode()`
  - Liefert eine Kennnummer (Hashcode) des Objekts.

  `protected Object clone()`
  - Liefert eine Kopie des Objekts.

  `protected void finalize()`
  - Wird vom Garbage Collector aufgerufen.
  - Problematisch

- Vollständige Übersicht in der Java 17 API Dokumentation

toString

- Die Implementierung aus der Klasse `Object` gibt den Klassennamen gefolgt von einem @-Zeichen und der hexadezimale Repräsentation des Hash-Codes zurück.
  - Beispiel: `at.ac.uibk.pm.inheritance.rectangle.Rectangle@630`
- Sollte in der Regel in allen Subklassen überschrieben werden, sofern sie nicht in einer Superklasse entsprechend überschrieben wurde.
  - Bei Hilfsklassen, von welchen kein Exemplar erzeugt werden kann, und bei Enums ist ein Überschreiben typischerweise nicht erforderlich.
- Wenn möglich, sollten alle relevanten Informationen in der String-Repräsentation abgebildet werden.
- Eine gute `toString`-Implementierung hilft beim Verwenden der Klasse.
- Die `toString`-Methode wird automatisch aufgerufen, wenn z.B.
  - ein Objekt der Methode `println`, `print` oder `printf` übergeben oder
  - eine Stringkonkatenation durchgeführt wird.

Beispiel toString

```java
public class Rectangle {
    private int width;
    private int length;
    …
    @Override
    public String toString() {
        return "Rectangle{" + "width=" + width + ", length=" + length + '}';
    }
    …
}
```

```java
public class RectangleApplication {
    public static void main(String[] args) {
        Rectangle rectangle1 = new Rectangle(20, 3);
        System.out.println(rectangle1);
    }
}
```

```
Rectangle{width=20, length=3}
```

src/at/ac/uibk/pm/inheritance/rectangle/Rectangle.java

equals (1)

- Implementierung aus der Klasse `Object` vergleicht nur Referenzen.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- Eine korrekte Implementierung erfüllt:
  - Reflexivität

    `x.equals(x) → true`
  - Symmetrie

    `x.equals(y) == y.equals(x)`
  - Transitivität

    `x.equals(y) → true`, `y.equals(z) → true` dann gilt auch `x.equals(z) → true`
  - Konsistenz

    Zwei Objekte müssen bei wiederholten Aufrufen immer das gleiche Ergebnis liefern, so lange sie sich nicht verändert haben.
  - Für alle Objekte x, die nicht gleich null sind, gilt:

    `x.equals(null) → false`

equals (2)

- Es kann zu Problemen beim Verwenden dieser Methode führen, wenn dieser Vertrag nicht eingehalten wird.
- Fundamentales Problem:
  - Eine Klasse, von welcher Exemplare erzeugt werden können, wird erweitert.
  - Die abgeleitete Klasse erweitert die Klasse mit zusätzlichen Objektvariablen.
  - Der `equals`-Vertrag kann nicht eingehalten werden, ohne das Prinzip der Ersetzbarkeit zu verletzen.
  - Durch Komposition kann dieses Problem umgangen werden.
- Empfehlung für eigene `equals`-Methode:
  1. Mittels Referenzvergleich die Referenzen vergleichen. Sofern diese gleich sind, `true` zurückgeben.
  2. Mit dem `instanceof-Operator` überprüfen, ob der Parameter den korrekten Typ hat. Falls nicht, `false` zurückgeben.
  3. Für jedes signifikante Feld der Klasse, überprüfen, ob das Feld mit dem entsprechenden Feld der Pattern-Variable übereinstimmt. Falls alle übereinstimmen, wird `true` zurückgegeben, sonst `false`.

Beispiel equals

```
public class Rectangle {
    private int width;
    private int length;
    …
    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (!(other instanceof Rectangle rectangle)) {
            return false;
        }
        return width == rectangle.width && length == rectangle.length;
    }
    …
}
```

src/at/ac/uibk/pm/inheritance/rectangle/Rectangle.java

hashCode

- Für jedes Objekt sollte eine eindeutige Kennnummer (nicht immer möglich) produziert werden.
- Wird vor allem im Collection-Framework (z.B. HashMap; wird noch besprochen) benutzt.
- Die Methoden `equals` und `hashCode` hängen zusammen und sollten immer gemeinsam überschrieben werden.
- Anforderungen:
  - Wenn zwei Objekte gemäß `equals` gleich sind, müssen sie auch den gleichen Hash produzieren.
  - Die Umkehrung gilt nicht, d.h. zwei Objekte können den gleichen Hash haben, aber verschieden sein.
  - Der Wert von `hashCode` muss immer gleich bleiben, solange sich die für `equals` signifikanten Objektvariablen des Objekts nicht ändern.

Empfehlung für eigene hashCode-Methode

1. `int`-Variable `result` mit dem Hash `c` der ersten signifikanten Objektvariable initialisieren (siehe 2.a.).
2. Für jede weitere signifikante Objektvariable `a`:
   a. Einen `int`-Hash c berechnen:
      i. Falls a einen primitiven Typ hat und der Typ `type` ist, `Type.hashCode(a)` verwenden.
      ii. Falls a eine Objektreferenz und `null` ist, 0 verwenden.
      iii. Falls a eine Objektreferenz und nicht `null` ist, rekursiv `hashCode` aufrufen.
      iv. Falls a ein `Array` ist, für alle signifikanten Elemente den Hash, wie in 2.a. beschrieben, ermitteln. Falls alle Felder signifikant sind, `Arrays.hashCode` verwenden.
   b. Den Hash `c` zu `result` hinzufügen:
      `result = 31 * result + c`
3. `result` zurückgeben

Falls Performance irrelevant ist, kann die statische Methode `Objects.hash` verwendet werden, um den Hash beliebig vieler Variablen zu berechnen.

Beispiel hashCode

```
public class Rectangle {
    private int width;
    private int length;
    …
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = Integer.hashCode(width);
        result = prime * result + Integer.hashCode(length);
        return result;
    }
    …
}
```

```
public class Rectangle {
    …
    @Override
    public int hashCode() {
        return java.util.Objects.hash(width, length);
    }
    …
}
```

clone

- Ein Kopier-Konstruktor eignet sich zum Kopieren eines Objekts ohne Vererbung.
- Kopieren bei einem dynamischen Typ kann nur durch eine dynamisch gebundene Methode erfolgen – `clone`-Methode.
- `clone` hat den Zugriffsschutz `protected`.
  - Damit kann die Methode nicht von außen aufgerufen werden, wenn sie nicht überschrieben wird.
  - Beim Überschreiben muss der Zugriffsschutz gelockert werden (`public`).
- Voraussetzung für Anwendbarkeit: Klasse muss
  - das Interface `Cloneable` implementieren,
  - eine eigene öffentliche Methode `clone()` implementieren,
  - in `clone` eine Kopie des Superklassenobjekts mit `super.clone()` erzeugen,
  - in `clone` alle Datenelemente veränderlicher Klassen einzeln mit clone-Aufrufen auf diese Objekte kopieren.
- `clone` kann nur verwendet werden, wenn jede verwendete Klasse `clone` korrekt implementiert.

finalize

- Ist seit Java 9 als `@Deprecated` markiert und sollte somit nicht verwendet werden
- Ist inhärent problematisch
  - Kann unter anderem zu Performanceeinbußen, Deadlocks und Ressourcen-Lecks führen.
  - Ausführungszeitpunkt ist nicht festgelegt (kann unbegrenzt aufgeschoben werden).
- Wenn eine Klasse Ressourcen die sie hält explizit wieder freigeben muss, sollte eventuell `AutoCloseable` implementiert werden.
- Durch die `Cleaner` und `PhantomReference` Klassen werden effizientere und flexiblere Wege Ressourcen wieder freizugeben bereitgestellt.
- Mehr zum Thema lernen wir am Ende des Semesters bei JVM.

Quellen

- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018