

---

# Das Singleton

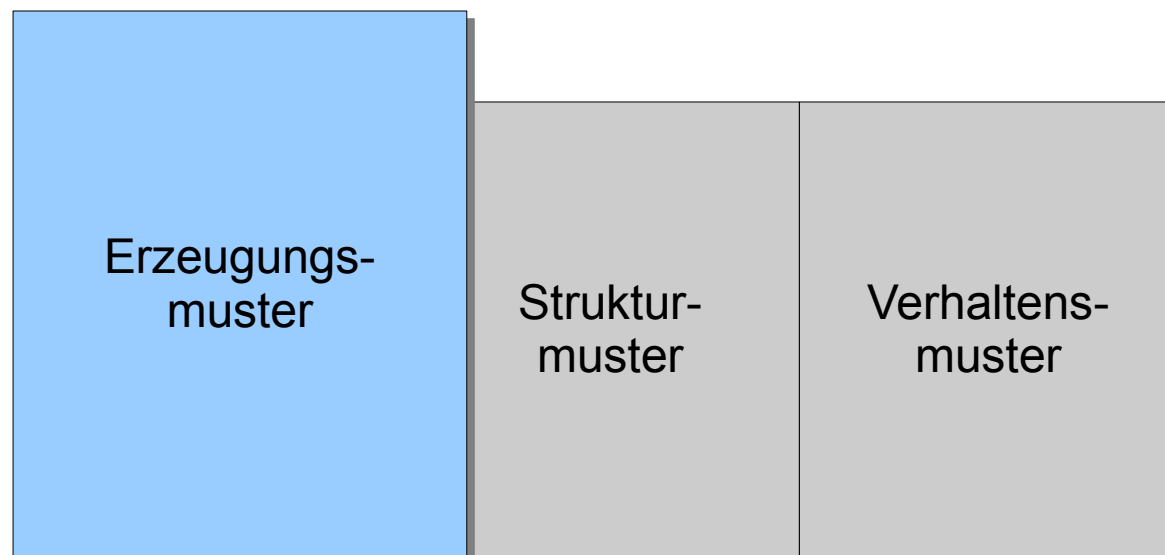
Alleine gegen alle



# Singleton

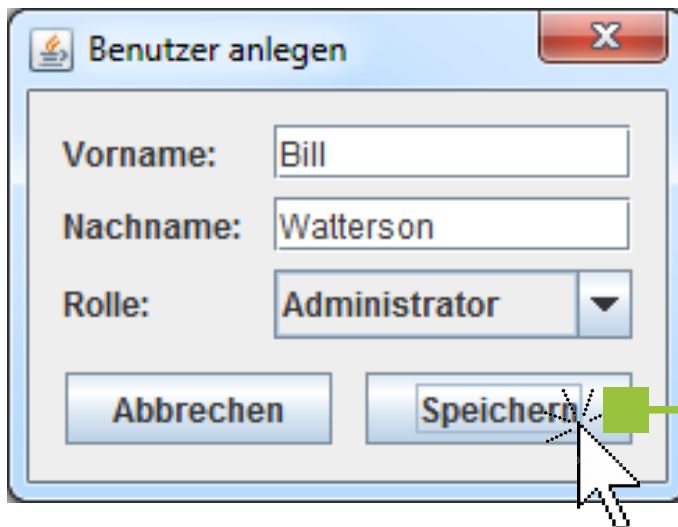
---

- Klassifikation
  - Objektbasiertes Erzeugungsmuster
  - Sehr langlebig
- Alternativname: Einzelstück



# Einleitendes Beispiel

- Benutzerverwaltung mit Zugangskarten
- Dialog zum Anlegen eines neuen Benutzers



A screenshot of a Windows-style dialog box titled 'Benutzer anlegen' (Create User). The dialog has a standard title bar with a close button (X). Inside, there are three input fields: 'Vorname:' (First Name) with the value 'Bill', 'Nachname:' (Last Name) with the value 'Watterson', and 'Rolle:' (Role) with a dropdown menu showing 'Administrator'. At the bottom, there are two buttons: 'Abbrechen' (Cancel) and 'Speichern' (Save). A green square highlights the 'Speichern' button, and a green arrow points from this square to the list of actions on the right.

- Zugangskarte ausstellen
- Benutzer in Datenbank speichern
- Datenmodell (Cache) aktualisieren

# Einleitendes Beispiel

```
public class NewUserDialog extends JDialog {

    // [...]
    private final JButton saveButton;

    public NewUserDialog() {
        super();
        this.saveButton = new JButton("Speichern");
        this.saveButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                saveUser();
            }
        });
    }

    private void saveUser() {
        final User toSave = new User(firstName(), lastName(), role());
        // TODO: Print authorization card
        // TODO: Save in database
        // TODO: Refresh data model
    }
}
```

# Einleitendes Beispiel

```
public class NewUserDialog extends JDialog {

    // [...]
    private final JButton saveButton;

    public NewUserDialog() {
        super();
        this.saveButton = new JButton("Speichern");
        this.saveButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                saveUser();
            }
        });
    }

    private void saveUser() {
        final User toSave = new User(firstName(), lastName(), role());
        CardPrinter.getInstance().printAuthorizationFor(toSave);
        // TODO: Save in database
        // TODO: Refresh data model
    }
}
```

# Einleitendes Beispiel

```
public class NewUserDialog extends JDialog {

    // [...]
    private final JButton saveButton;

    public NewUserDialog() {
        super();
        this.saveButton = new JButton("Speichern");
        this.saveButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                saveUser();
            }
        });
    }

    private void saveUser() {
        final User toSave = new User(firstName(), lastName(), role());
        CardPrinter.getInstance().printAuthorizationFor(toSave);
        Database.getInstance().save(toSave);
        // TODO: Refresh data model
    }
}
```

# Einleitendes Beispiel

```
public class NewUserDialog extends JDialog {  
  
    // [...]  
    private final JButton saveButton;  
  
    public NewUserDialog() {  
        super();  
        this.saveButton = new JButton("Speichern");  
        this.saveButton.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                saveUser();  
            }  
        });  
    }  
  
    private void saveUser() {  
        final User toSave = new User(firstName(), lastName(), role());  
        CardPrinter.getInstance().printAuthorizationFor(toSave);  
        Database.getInstance().save(toSave);  
        Application.getInstance().getDataModel().users().refresh();  
    }  
}
```

Singletons

Train Wreck Line – zuviele Aufrufe in einer Zeile



# Motivation

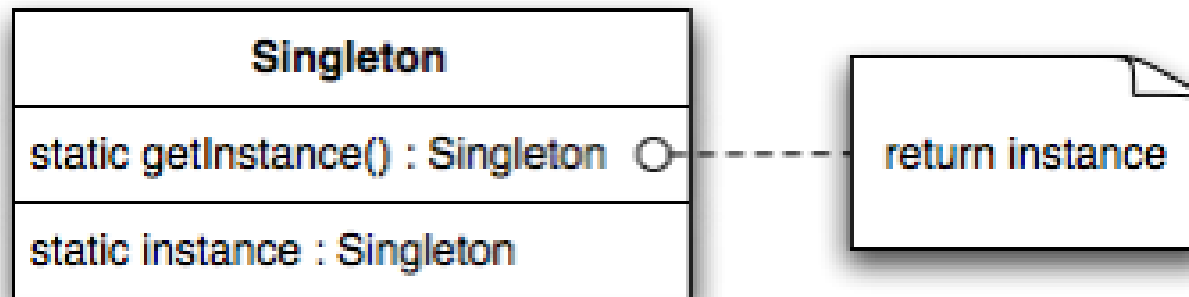
---

- Beschränkt die Instanzen eines Typs
  - Üblicherweise auf „maximal eine Instanz“
- Kapselt die Verwaltung dieser Instanz
- Bietet Zugriffspunkt auf die Instanz
- Mit anderen Worten
  - Globale Variable ohne Zugriffsschutz
  - Einfacher Ausweg bei Architekturproblemen



# UML-Diagramm (Klassen)

---



# Implementierung

---

- Implementierung muss zwei Kriterien erfüllen
  - Nur eine Instanz
  - Globaler Zugriff
- In objektorientierten Sprachen oft nicht einfach
- In Java lange Zeit nicht vollständig umsetzbar

# Implementierung in Java

---

## Einfache Implementierung

```
public class MySingleton {  
    public MySingleton() {  
        super();  
    }  
  
    public MySingleton getInstance() {  
        return this;  
    }  
}
```

# Implementierung in Java

---

## Privater Konstruktor verhindert Instanziierung

```
public final class MySingleton {  
  
    private MySingleton() {  
        super();  
    }  
  
    public MySingleton getInstance() {  
        return this;  
    }  
}
```

# Implementierung in Java

---

Zugriffspunkt muss auf Klassenebene sein

```
public final class MySingleton {  
  
    private MySingleton() {  
        super();  
    }  
  
    public static MySingleton getInstance() {  
        return this;  
    }  
}
```

# Implementierung in Java

---

Instanz muss auf Klassenebene bekannt sein

```
public final class MySingleton {  
  
    private static MySingleton instance;  
  
    private MySingleton() {  
        super();  
    }  
  
    public static MySingleton getInstance() {  
        return instance;  
    }  
}
```

# Implementierung in Java

---

Instanz muss angelegt sein

```
public final class MySingleton {  
  
    private static MySingleton instance = new MySingleton();  
  
    private MySingleton() {  
        super();  
    }  
  
    public static MySingleton getInstance() {  
        return instance;  
    }  
}
```

# Implementierung in Java

## Eager Initialization – Frühe Initialisierung

```
public final class MySingleton {  
  
    private static MySingleton instance = new MySingleton();  
  
    private MySingleton() {  
        super();  
    }  
  
    public static MySingleton getInstance() {  
        return instance;  
    }  
}
```

- Vorteil: Direkte Implementierung
- Vorteil: Laufzeitverhalten bekannt
- Nachteil: Erzeugungsarbeit immer beim Systemstart (längere Boot Time)



# Implementierungen in Java

## Lazy Initialization – naive Implementierung

```
public class MySingleton {  
  
    private static MySingleton instance = null;  
  
    private MySingleton() {  
        super();  
    }  
  
    public static MySingleton getInstance() {  
        if (null == instance) {  
            instance = new MySingleton();  
        }  
        return instance;  
    }  
}
```



Exkurs: `null == instance` ist in vielen Sprachen zu bevorzugen

Programmierstil nennt sich Yoda Notation

# Implementierungen in Java

## Lazy Initialization – Threading beachten!

```
public class MySingleton {  
  
    private static MySingleton instance = null;  
  
    private MySingleton() {  
        super();  
    }  
  
    public static synchronized MySingleton getInstance() {  
        if (null == instance) {  
            instance = new MySingleton();  
        }  
        return instance;  
    }  
}
```

- synchronized-Modifizier an Methoden ist nachteilig
- Lock festgelegt und öffentlich bekannt

# Implementierungen in Java

## Lazy Initialization – Gekapseltes Thread-Lock

```
public class MySingleton {  
  
    private static MySingleton instance = null;  
    private static Object lock = new Object();  
  
    private MySingleton() {  
        super();  
    }  
  
    public static MySingleton getInstance() {  
        synchronized (lock) {  
            if (null == instance) {  
                instance = new MySingleton();  
            }  
            return instance;  
        }  
    }  
}
```

- Viel Aufwand für den „As-Late-As-Possible“-Effekt

# Implementierungen in Java

---

Enum-based – die einzig vollständig korrekte Art, in Java ein Singleton zu implementieren

```
public enum MySingleton {  
    INSTANCE;  
}
```

- Sprachfeature (seit Java 5)
- Garantiert nur ein Exemplar von INSTANCE
- Enum können ansonsten verwendet werden wie normale Klassen
  - Variablen
  - Methoden

# Varianten

---

- Eager Initialization
  - Einfache Implementierung
  - Früher Ressourcenbedarf
- Lazy Initialization
  - Aufwendige Implementierung
  - Spätestmöglicher Ressourcenbedarf
- Enum-based
  - Implementierung basiert auf Sprachfeature
  - Früher Ressourcenbedarf
  - Korrekt, aber ungewohnt

# Vorteile des Singleton

---

- Garantiert systemweit nur eine Instanz
  - Bei korrekter Implementierung
- Einfacher Zugriff auf Instanz
  - Einfach zu verstehen
  - Einfach anzuwenden
  - Unmittelbarer „Erfolg“

# Nachteil: Globaler Zugriff

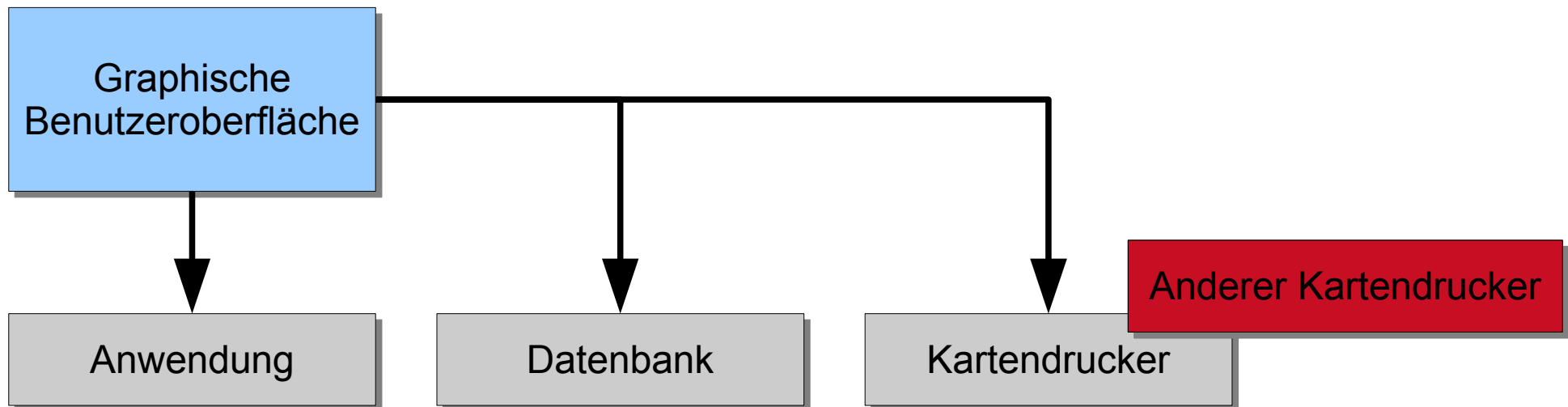
- Globaler Zugriff auf Instanz
- Konkreter Klassenname in jedem Zugriff
  - Keine Möglichkeit für polymorphen Aufruf

```
public class NewUserDialog extends JDialog {  
    // [...]  
    private void saveUser() {  
        final User toSave = new User(firstName(), lastName(), role());  
        CardPrinter.getInstance().printAuthorizationFor(toSave);  
        Database.getInstance().save(toSave);  
        Application.getInstance().getDataModel().users().refresh();  
    }  
}
```

- Systemarchitektur wird korrumpiert
  - Jeder hat Zugriff von überall auf alles

# Nachteil: Starke Kopplung

- Kopplung an den konkreten Typ
- Keine polymorphen Aufrufe zu haben bedeutet, den Singleton-Code nahezu zu „Inlinen“
- Es gibt kaum eine stärkere Kopplung
  - Guter Code ist lose gekoppelt





# Nachteil: Starke Kopplung

- Tests, vor allem Unit Tests sind besonders betroffen
  - Nicht zu testende Abhängigkeiten werden oft durch Mock-Objekte ersetzt
  - Durch die starke Kopplung können die Singletons nicht ersetzt werden
- Singleton-Klassen erlauben keine Unterklassen

```
public class NewUserDialog extends JDialog {  
    // [...]  
    private void saveUser() {  
        final User toSave = new User(firstName(), lastName(), role());  
        CardPrinter.getInstance().printAuthorizationFor(toSave);  
        Database.getInstance().save(toSave);  
        Application.getInstance().getDataModel().users().refresh();  
    }  
}
```

# Erweiterungen

---

- Instanzpool
  - Statt einer einzigen Instanz werden mehrere Instanzen erzeugt und vorgehalten
  - Herausgabe im „Round Robin“-Verfahren
- Multiton
  - Mehrere Instanzen werden erzeugt und vorgehalten
  - Jede Instanz besitzt einen eindeutigen Identifier
  - Herausgabe an Klienten nur über Identifier

| Multiton   |
|--|
| - static instances : Map<String, Multiton>                   |
| + static getInstanceFor(String identifier) : return Multiton |

# Zusammenfassung

- Singleton
- Objektbasiertes Erzeugungsmuster
- Erzeugt und verwaltet die einzige Instanz
- Vereinfacht Architekturentscheidungen
- Hoher Preis, u.a. feste Strukturen
  - Problem für Tests
- Objektorientiertes Äquivalent zur „globalen Variable“

