STATIC EN HET SINGLETON PATTERN

Wijnand Schepens

Logging (1)


```
void foo(int x)
{
    System.out.println("in foo(), x=" + x);
    // ...
}
```

+ Eenvoudig

- Achteraf verwijderen?
- Selectief verwijderen?
- Naar bestand?

Logging (2)


```
public class Logger
{
    public void log(String s)
    {
        System.out.println(s);
    }
}

void foo(int x)
{
    Logger logger = new Logger();
    logger.log("in foo(), x=" + x);
    // ...
}
```

+ Logger kan apart wijzigen

bv. output naar bestand

- Dit is vervelend!

Logging (3)


```
public class Logger
{
    public void log(String s)
    {
        System.out.println(s);
    }
}

void foo(int x, Logger logger)
{
    logger.log("in foo(), x=" + x);
    // ...
}
```

Beter, maar nog steeds vervelend
(overal doorgeven !?)

Logging (4)


```
public class Logger
{
    public static void log(String s)
    {
        System.out.println(s);
    }
}

void foo(int x)
{
    Logger.log("in foo(), x=" + x);
    // ...
}
```

+ static is handig

- geen polymorfisme mogelijk!

Logging (5a)


```
public interface Logger
{
    public void log(String s);
}

public class StandardLogger implements Logger
{
    public void log(String s)
    {
        System.out.println(s);
    }
}
```

Logging (5b)

7

```
public class LogSystem
{
    private static Logger logger = ...;

    public static Logger getLogger()
    {
        return logger;
    }
    public static void setLogger(Logger logger_)
    {
        logger = logger_;  // geen this!!
    }
}
```


Logging (5c)

8

```
void foo(int x)
{
    LogSystem.getLogger().log("in foo(), x=" + x);
    // ...
}
```


Logging (5d)

9

```
int main(String[] args)
{
    LogSystem.setLogger( new MyLogger() );

    // ...
}
```


Echte logging libraries

10

- java.util.logging
- Log4j
- Slf4j
- ...

Singleton

11

```
public class DBAccess
{
    private static DBAccess instance = new DBAccess();

    private DBAccess() { ... }

    public static DBAccess getInstance()
    {
        return instance;
    }

    public List<Customer> getCustomers() { ... }
    // ...
}
```

Enige instantie!

Niemand anders kan er een maken!

Singleton (gebruik)

12

```
void foo()
{
    DBAccess dba = DBAccess.getInstance();

    List<Customer> customers = dba.getCustomers();
    // ...
}
```

static

Singleton (lazy)

13

```
public class DBAccess
{
    private static DBAccess instance = null;

    private DBAccess() { ... }

    public static DBAccess getInstance()
    {
        if (instance == null)
            instance = new DBAccess();
        return instance;
    }
}
```


Voor- en nadelen

14

- Minder koppelingen (overal bereikbaar)
- Opgelet met thread safety!!
  - Eventueel delen `synchronized` maken
- Is polymorfisme nog mogelijk??
- Zuinig mee omspringen