Methoden und Wrapperklassen

CoMa-Übung IV

TU Berlin

07.11.2012

Themen der Übung

1. Organisatorisches
2. Methoden
3. Wrapper-Klassen

Organisatorisches:

- Im Pool nur auf die Abgabeliste setzen, wenn ihr wirklich fertig seid!
- Christians Abgaben in seinem Tutorium
- Bei Gruppenwechsel:
- Umtragen mit Tutor im Unix-Pool
- eMail mit Daten der veränderten Personen an Madeleine

Methoden

Grundlagen

- Methoden sind Javas Konzept von *Algorithmen*.
- Methoden haben festgelegte Eingaben und Ausgaben, sowie eine Folge von Befehlen, die aus der Eingabe die Ausgabe erzeugen.
- Methoden gehören in Java immer zu einer Klasse oder einem Objekt.
- Diese Klasse bzw. dieses Objekt wird dann **Eigentümer** der Methode genannt.

`Math.min(int a, int b)`

- Algorithmus, der das Minimum zweier Werte zurückgibt.
- Eingabe: zwei `int`-Werte.
- Ausgabe: der Minimum der beiden `int`-Werte.
- Gehört zu der Klasse `Math`.

Vorteile von Methoden

Vorteil 1: Übersicht

- Die Benutzung von Methoden teilt Programme in viele kleine Algorithmen auf.
- Gibt dem Programm eine Struktur.
- Kleinere Algorithmen sind leichter verständlich.

Vorteil 2: Wiederverwendbarkeit

- Methoden können vom ganzen Programm verwendet werden
- Jeder Algorithmus muss nur *einmal* geschrieben werden
- Spart Arbeit
- Macht nachträgliche Änderungen leichter
- Vermeidet unnötige Code-Wiederholungen.

Deklaration von Methoden

Bestandteile einer Methode

Methoden bestehen aus zwei Teilen, dem **Methodenkopf** und dem **Methodenrumpf**. Der Methodenkopf besteht aus

- *Modifizieren*,
- einem *Rückgabetyp*,
- dem *Methodennamen*,
- und einer *Parameterliste*.

Der Methodenrumpf besteht aus den Anweisungen der Methode.

Signatur einer Methode

Unter der Signatur einer Methode versteht man ihren Methodennamen und die Typen ihrer Parameterliste. Modifikatoren, der Rückgabetyp und der Methodenrumpf gehören **nicht** dazu.

Bestandteile einer Methode – Beispiel 1

Erinnerung: Die Signatur einer Methode besteht aus ihrem Namen und der Liste der Typen ihrer Parameter.

Beispiel

```
1 public static long sum(long a, long b) {
2     return a + b;
3 }
```

- Methodenkopf:
  - *Modifizierer*: `public` und `static`
  - *Rückgabetyp*: `long`
  - *Methodenname*: `sum`
  - *Parameterliste*: `long a, long b`
- Methodenrumpf:
  - Besteht nur aus der Anweisung `return a + b;`
- Die Signatur der Methode ist `sum(long,long)` – der Methodenname ist `sum` und die Methode hat zwei Parameter, beide vom Typ `long`.

Rückgaben in Methoden

`return`

- Der Befehl `return;` beendet die Ausführung einer Methode und gibt nichts zurück.
- Der Befehl `return a;` beendet die Ausführung einer Methode und gibt `a` an den Aufrufer zurück.

`return` und der Rückgabetyp

- Der Befehl `return;` darf nur in Methoden benutzt werden, deren Rückgabetyp `void` ist.
- Der Befehl `return a;` darf nur in Methoden benutzt werden, deren Rückgabetyp **nicht** `void` ist; außerdem muss der Typ von `a` zu dem Rückgabetyp der Methode passen.
- Hat eine Methode einen Rückgabetyp, der nicht `void` ist, **muss** die Methode mit einem `return`-Befehl enden, der einen passenden Typ zurückgibt.

Klassenmethoden

Methoden in Java können zwei Arten von Eigentümern haben: *Klassen* und *Objekte*. Demetsprechend lassen sie sich in zwei Kategorien aufteilen: **Klassenmethoden** und **Objektmethoden**.

Klassenmethoden

- Werden durch den Modifizierer `static` in der Methoden-Deklaration gekennzeichnet (ohne `static` Objektmethode).
- Beispiel: `public static void main(String[] args)`
- Arbeiten nur mit Klassen und brauchen keine Objekte.
- Werden üblicherweise in der Form `Klassenname.Methodenname(Parameterliste)` aufgerufen.
- Objektmethoden hingegen **müssen** an einem Objekt aufgerufen werden.
- Wir arbeiten zunächst nur mit Klassenmethoden.

Bestandteile einer Methode – Beispiel 2

Erinnerung: Die Signatur einer Methode besteht aus ihrem Namen und der Liste der Typen ihrer Parameter.

Beispiel

```
Character firstChar(String str) {
    return str.charAt(0);
}
```

- Methodenkopf:
  - *Modifizierer*: keiner
  - *Rückgabetyp*: `Character`
  - *Methodenname*: `firstChar`
  - *Parameterliste*: `String str`
- Methodenrumpf:
  - Besteht nur aus der Anweisung `return str.charAt(0);`
- Die Signatur der Methode ist `firstChar(String)` – der Methodenname ist `firstChar` und die Methode hat einen Parameter vom Typ `String`.

Namen und Typen

Namenskonvention für Methoden

- Methodennamen werden mit einem Kleinbuchstaben begonnen.
- Methodennamen fangen üblicherweise mit einem Verb an.
- Methoden dürfen nicht wie Schlüsselwörter heißen – `class`, `int`, `double`, ... sind also tabu.
- Methode heißt wie Schlüsselwort → Fehler beim Kompilieren.

Parameter- und Rückgabetypen

- Typen für Parameter können sein:
  - jeder primitive Datentyp (`boolean`, `byte`, `char`, ...),
  - jeder Referenztyp (`Character`, `String`, ...),
- Der Rückgabetyp kann sein:
  - jeder für Parameter erlaubte Typ,
  - der spezielle Rückgabetyp `void`, falls nichts zurückgeben wird.

Parameter und Signatur

Parameter

- Wie Variablen in der Form `Datentyp parametername` deklariert.
- Jeder Parameter muss einen Typ haben.
- → Kurzform `double a,b` ist nicht erlaubt!
- Mehrere Parameter werden durch `,` getrennt.
- Methode mit leerer Parameterliste → die Liste besteht nur aus `()`.
- Parameter werden wie Variablen benutzt in der Methode.

Signatur

- Der Kompiler sucht Methoden anhand ihrer Signatur in den Methoden des Eigentümers.
- → Jede Methode eines Eigentümers muss eine eindeutige Signatur haben!
- → Kompiler-Fehler sonst.

Bestandteile einer Methode – Beispiel 3

Erinnerung: Die Signatur einer Methode besteht aus ihrem Namen und der Liste der Typen ihrer Parameter.

Beispiel

```
public void hello() {
    System.out.println("Hello");
}
```

- Methodenkopf:
  - *Modifizierer*: `public`
  - *Rückgabetyp*: `void`
  - *Methodenname*: `hello`
  - *Parameterliste*: keine Parameter
- Methodenrumpf:
  - Besteht nur aus der Anweisung `System.out.println("Hello");`
- Die Signatur der Methode ist `hello()` – der Methodenname ist `hello` und die Methode hat keine Parameter.

Deklaration von Methoden – Zusammenfassung

Deklaration von Methoden

Eine Methodendeklaration hat in Java folgenden Aufbau:

- *Modifizieren*,
- gefolgt von *genau einem Rückgabetyp*,
- dem *Methodennamen*,
- einer (möglicherweise leeren) Liste von *Parametern*,
  - die in runde Klammern ( ) eingeschlossen ist,
  - deren Parameter durch , getrennt sind,
- und dem *Methodenrumpf*,
  - der in geschweifte Klammern { } eingeschlossen ist.

Diese Bestandteile müssen in genau dieser Reihenfolge erscheinen.

Signatur einer Methode

Die Signatur einer Methode besteht aus ihrem Namen und der Liste der Typen ihrer Parameter.

Methodenaufruf

Aufruf einer Methode

- Der Aufruf einer Methode besteht aus 3 Teilen:
  - Dem *Eigentümer* der Methode,
  - dem *Namen* der Methode,
  - und *Argumenten* für die von der Methode erwarten Parametern.
- Die Syntax für den Aufruf dieser Bestandteile ist: `Eigentümer.Methodenname(Parameter1,Parameter2,...)`
- `Eigentümer.` kann weggelassen werden, wenn die aufgerufene Methode in derselben Klasse ist wie die aufrufende.

Argumente

- Argumente sind konkrete Werte für die Parameter einer Methode.
- z.B. kann 2 ein Argument für einen `int`-Parameter einer Methode sein.
- Der Typ eines Arguments muss zum Typ des Parameters passen.

Methodenaufruf

Aufruf einer Methode

- Der Aufruf einer Methode besteht aus 3 Teilen:
  - Dem *Eigentümer* der Methode,
  - dem *Namen* der Methode,
  - und *Argumenten* für die von der Methode erwarten Parametern.
- Die Syntax für den Aufruf dieser Bestandteile ist: `Eigentümer.Methodenname(Parameter1,Parameter2,...)`

Beispiel 1 – `Math.round(double d)`

```
1 double temperature = 5.6;
2 long roundedTemperature = Math.round(temperature);
```

- Eigentümer: die Klasse `Math`
- Methodenname: `round`
- Argumente: die `double`-Variable `temperature`

Methodenaufruf

Aufruf einer Methode

- Der Aufruf einer Methode besteht aus 3 Teilen:
  - Dem *Eigentümer* der Methode,
  - dem *Namen* der Methode,
  - und *Argumenten* für die von der Methode erwarten Parametern.
- Die Syntax für den Aufruf dieser Bestandteile ist: `Eigentümer.Methodenname(Parameter1,Parameter2,...)`

Beispiel 2 – `Math.max(int a, int b)`

```
1 int a = 3;
2 int maximum = Math.max(a,4);
```

- Eigentümer: die Klasse `Math`
- Methodenname: `max`
- Argumente: die `int`-Variable `a` und die `int`-Konstante 4

Methodenaufruf

Aufruf einer Methode

- Der Aufruf einer Methode besteht aus 3 Teilen:
  - Dem *Eigentümer* der Methode,
  - dem *Namen* der Methode,
  - und *Argumenten* für die von der Methode erwarten Parametern.
- Die Syntax für den Aufruf dieser Bestandteile ist:
  `Eigentümer.Methodenname(Parameter1,Parameter2,...)`

Beispiel 3 – `Scanner.nextDouble()`

```java
Scanner scanner = new Scanner(System.in);
double temperature = scanner.nextDouble();
```

- Eigentümer: das Objekt `scanner`
- Methodenname: `nextDouble`
- Argumente: die Methode erwartet keine

Übergabe von Argumenten

Was passiert bei der Übergabe von Argumenten an Methoden?

```java
public static void changeA(int a) {
  a = a + 1;
}

public static void main(String[] args) {
    int a = 1;
    changeA(a);
    System.out.println(a);
}
```

Wird ein Variable als Argument an eine Methode übergeben, wird eine **Kopie** ihres Werts erzeugt und an die Methode übergeben.

- Methoden verändern also keine Variablen in der aufrufenden Methode.
- `changeA` hat also keinen Einfluss auf den Wert von `a` in der `main`-Methode.

Verschachtelte Methodenaufrufe

```java
public static int multiply(int a, int b) {
  return a*b;
}

public static int square(int a)  {
    return multiply(a,a);
}

public static void main(String[] args) {
    System.out.println(square(2));
}
```


Verschachtelte Methodenaufrufe

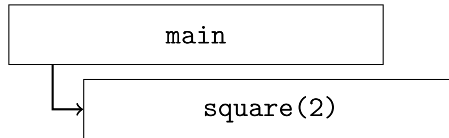
```java
public static int multiply(int a, int b) {
  return a*b;
}

public static int square(int a)  {
    return multiply(a,a);
}

public static void main(String[] args) {
    System.out.println(square(2));
}
```

→ (Zeile 9)

`main`

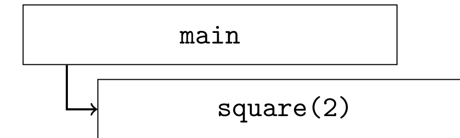
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a) {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



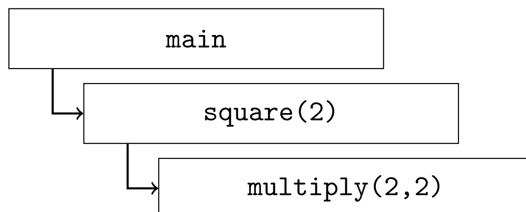
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a) {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



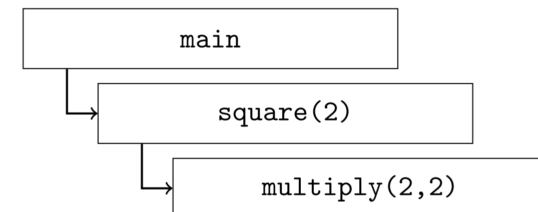
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a) {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



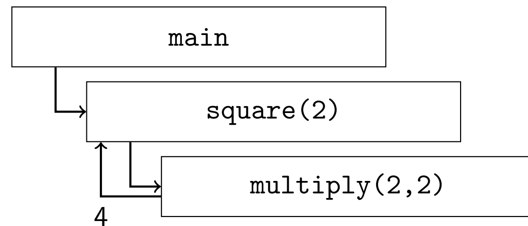
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a) {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



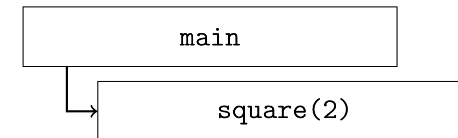
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a)  {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



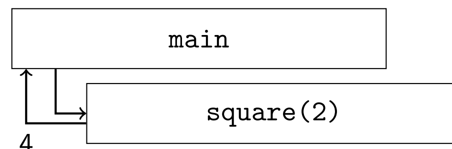
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a)  {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



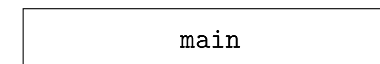
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a)  {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



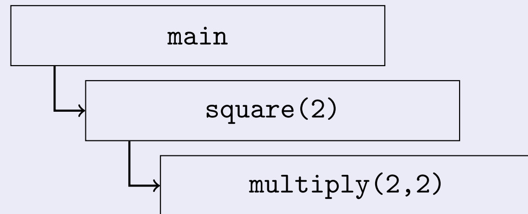
Verschachtelte Methodenaufrufe

```
1  public static int multiply(int a, int b) {
2    return a*b;
3  }
4
5  public static int square(int a)  {
6      return multiply(a,a);
7  }
8
9  public static void main(String[] args) {
10     System.out.println(square(2));
11 }
```



Der Methoden-Stack

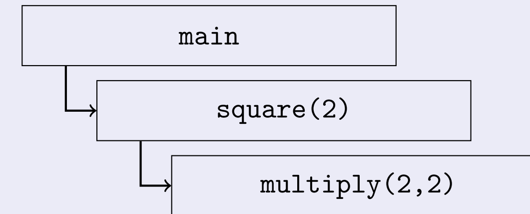
Stack



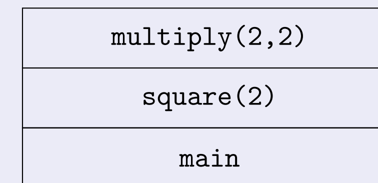
- Java merkt sich verschachtelte Methoden-Aufrufe, indem es diese Aufrufe auf ein *Stack* (Stapel) legt

Der Methoden-Stack

Stack



- Java merkt sich verschachtelte Methoden-Aufrufe, indem es diese Aufrufe auf ein *Stack* (Stapel) legt



Der Stacktrace

Der Stacktrace

- Tritt ein Fehler auf, gibt Java euch den *Stacktrace* aus.
- → ist im Wesentlichen der Inhalt des Methodenstacks, zu dem Zeitpunkt, als der Fehler aufgetreten ist.
- Die zuletzt aufgerufene Methode steht dabei oben.
- Für die Fehlersuche heißt das:
  - Geht den Stacktrace von oben nach unten durch, bis ihr in der ersten Klasse ankommt, die von euch geschrieben ist.
  - Schaut euch die Zeile an, in der Fehler verursacht wird und behebt die Ursache.

Der Stacktrace

Der Stacktrace

- Tritt ein Fehler auf, gibt Java euch den *Stacktrace* aus.
- → ist im Wesentlichen der Inhalt des Methodenstacks, zu dem Zeitpunkt, als der Fehler aufgetreten ist.
- Die zuletzt aufgerufene Methode steht dabei oben.
- Für die Fehlersuche heißt das:
  - Geht den Stacktrace von oben nach unten durch, bis ihr in der ersten Klasse ankommt, die von euch geschrieben ist.
  - Schaut euch die Zeile an, in der Fehler verursacht wird und behebt die Ursache.

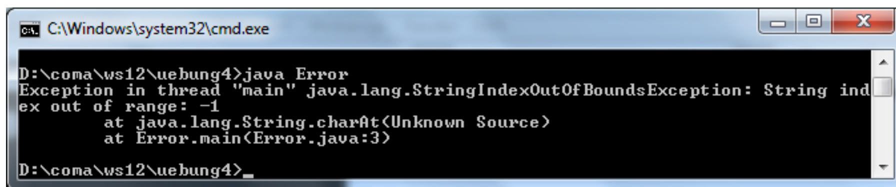
```
1 public class Error {
2     public static void main(String[] args) {
3         System.out.println("Hello".charAt(-1));
4     }
5 }
```

Der Stacktrace

Der Stacktrace

- → ist im Wesentlichen der Inhalt des Methodenstacks, zu dem Zeitpunkt, als der Fehler aufgetreten ist.
- Die zuletzt aufgerufene Methode steht dabei oben.

```
1 public class Error {
2     public static void main(String[] args) {
3         System.out.println("Hello".charAt(-1));
4     }
5 }
```



Einige Fallstricke

Nicht-statische Methoden aus statischen Aufrufen

```
1 public int square(int a) {
2     return multiply(a,a);
3 }
4
5 public static void main(String[] args) {
6     System.out.println(square(2));
7 }
```

- Objektmethode ohne Objekt aus Klassenmethode aufgerufen → Fehler!

Nicht erreichbarer Code

```
1 public static void main(String[] args) {
2     return;
3     int a = 1;
4 }
```

- Zeile 3 ist nie erreichbar → Fehler!

Einige Fallstricke (2)

Fehlendes Return

```
1 public int doSomething(int a) {
2   if (a > 0) {
3       return 0;
4   } else if (a <= 0) {
5       return 0;
6   }
7 }
```

- Fehler wegen nicht sichergestelltem `return`

Gleiche Signatur

```
1 public static long round(double d)
2
3 public void round(double d)
```

- Signatur nicht eindeutig (Modifizierer und Rückgabetyp zählen nicht)

Überladene Methoden

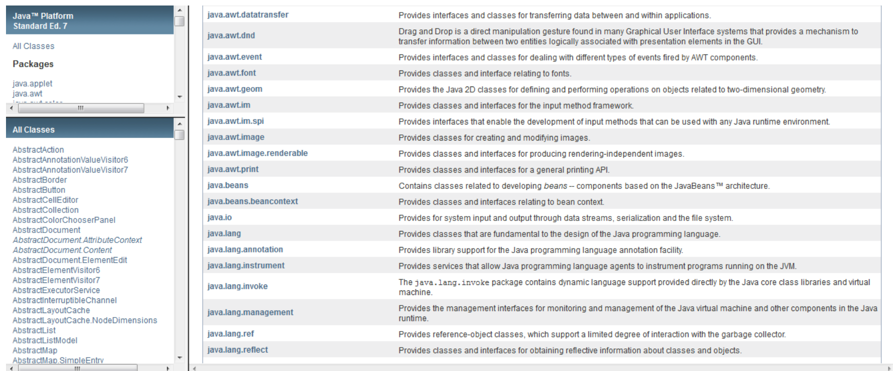
Überladene Methoden

- Methoden mit gleichem Namen werden **überladene Methoden** genannt.
- Für Java okay, solange der Kompiler die Methoden anhand der Parameterliste unterscheiden kann.
- Es gibt z.B. eine Menge `System.out.println()`-Methoden für verschiedene Datentypen
- Kann benutzt werden, um Standardwerte für Parameter festzulegen:

```
1 public static double round(double d, int precision) {
2     ...
3 }
4
5 public static double round(double d) {
6     return round(d, 2);
7 }
```

Die API

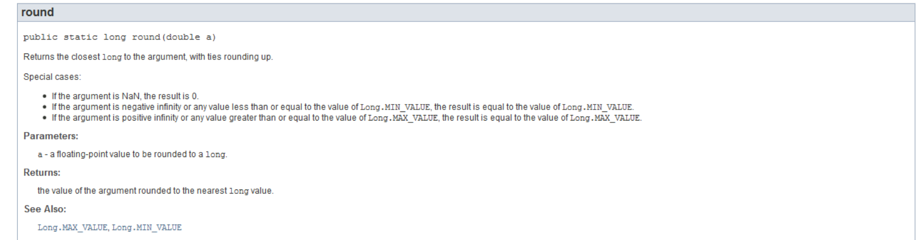
- Dokumentiert alle Klassen des JDKs und ihre Methoden
- `http://docs.oracle.com/javase/7/docs/api/`
- Nicht importierte Klassen finden sich unter `java.lang`



java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing *beans* -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.instrument	Provides services that allow Java programming language agents to instrument programs running on the JVM.
java.lang.invoke	The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.
java.lang.management	Provides the management interfaces for monitoring and management of the Java virtual machine and other components in the Java runtime.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.

Die API

- Beispiel: `Math.round`
- `http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html#round(double)`
- API beschreibt Methoden-Signatur mit Modifizierern
- Eingabe & Ausgabe der Methode und Sonderfälle



round

`public static long round(double a)`

Returns the closest `long` to the argument, with ties rounding up.

Special cases:

- If the argument is NaN, the result is 0.
- If the argument is negative infinity or any value less than or equal to the value of `Long.MIN_VALUE`, the result is equal to the value of `Long.MIN_VALUE`.
- If the argument is positive infinity or any value greater than or equal to the value of `Long.MAX_VALUE`, the result is equal to the value of `Long.MAX_VALUE`.

Parameters:

`a` - a floating-point value to be rounded to a `long`.

Returns:

the value of the argument rounded to the nearest `long` value.

See Also:

`Long.MAX_VALUE`, `Long.MIN_VALUE`

Wrapperklassen

Java bietet für jeden primitiven Datentyp eine zugehörige **Wrapperklasse** an. Diese Wrapperklassen erlauben es

- ihre primitive Datentypen als `String` zu formatieren,
- einen `String` in ihren primitiven Datentypen zu parsen,
- primitive Datentypen als Referenzen zu betrachten (später wichtig).

Primitiver Datentyp	Wrapperklasse
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Wrapperklassen (2)

Erzeugen von Wrapper-Objekten

- Mittels `new` → erzeugt immer ein neues Objekt.

```java
Integer i = new Integer(1);
```

- Mittels `Wrapperklasse.valueOf` → erzeugt ein neues Objekt per `new`

```java
Integer j = Integer.valueOf(1);
```

- Ausnahme: bei `Byte`, `Short`, `Integer`, `Long` und Werten zwischen -128 und 127 wird auf Objekte aus einem Cache zurückgegriffen
- Durch implizite Umwandlung aus einem primitiven Typ → benutzt `Wrapperklasse.valueOf`

```java
Integer j = 10;
```

Umwandlung von primitiven Typen und Wrapperklassen

Boxing und Unboxing

- Java wandelt automatisch zwischen primitiven Typen und Wrapperklassen um (**Autoboxing**).
- Die Umwandlung von primitiven Typen in ein Wrapperklasse nennt man (**Boxing**).

```
1 Integer j = 10;
```

- Die Umwandlung von einer Wrapperklasse in einen primitiven Typen nennt man (**Unboxing**).

```
1 int k = new Integer(10);
```

Hinweis

Der Wert eines Wrapper-Objekts ändert sich sein Leben lang nicht.

Fallstricke von Wrapperklassen

Fallstricke, Teil I

```
1 Integer i = new Integer(1);
2 Integer j = new Integer(1);
3 System.out.println(i >= j); // true
4 System.out.println(i <= j); // true
5 System.out.println(i == j); // false
```

- In Zeile 3 & 4 wird Unboxing benutzt, in Zeile 5 nicht.

Fallstricke, Teil II

```
1 Integer i = new Integer(10);
2 Integer j = Integer.valueOf(10);
3 Integer k = 10;
4 System.out.println(i == j);   // false
5 System.out.println(j == k);   // true
6 System.out.println(i == k);   // false
```

- `i` wird explizit neu erzeugt, `j,k` nicht (da der Wert in -128..127 liegt)