Java Programmierung

Zeichen, Bits und große Zahlen

Kleine und große Zahlen

Prof. Dr. rer. nat. Andreas Berl

Fakultät für Angewandte Informatik

Technische Hochschule Deggendorf

Lernziele

- Datentypen für Zahlen
- Primitive Zahlentypen
  - Besonderheiten
  - Grenzen
- Wrapperklassen
  - Verwendung
  - Konstanten
  - Methoden
- BigInteger und BigDecimal
  - Erzeugen und Berechnungen
  - MathContext
- Auswahl des richtigen Datentypen

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Datentypen für Zahlen

Es gibt unterschiedliche Datentypen zur Repräsentation von Zahlen

- Primitive Zahlentypen
  - \+ Können zeit- und speichereffizient verarbeitet werden
  - \- Können nicht in Collections oder anderen Datencontainern verwendet werden
  - \- Die Größe bzw. Genauigkeit der primitiven Typen ist beschränkt auf maximal 64 Bit
- Wrappertypen für Zahlen kapseln die primitiven Datentypen in einer objektorientierten Hülle
  - \+ Sind vielseitiger einsetzbar als primitive Datentypen
  - \- Brauchen mehr Speicher und werden weniger effizient verarbeitet als primitive Datentypen
- Die Klassen **`BigInteger`** und **`BigDecimal`**
  - \+ Können Zahlen in beliebiger Größe und Genauigkeit darstellen
  - \- Brauchen entsprechend viel Speicher und Rechenzeit

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Primitive Datentypen

Datentypen für ganze Zahlen

Name	Art	RAM	Wertebereich	Beispiele
`byte`	Mit Vorzeichen	8 Bit	-128 ... +127	`byte b1 = 127;` `byte b2 = -3;`
`short`	Mit Vorzeichen	16 Bit	-32768 ... +32767	`short s1 = 32767;` `short s2 = -17767;`
`int`	Mit Vorzeichen	32 Bit	-2147483648 ... +2147483647	`int i1 = 5000;` `int i2 = -333344545;`
`long`	Mit Vorzeichen	64 Bit	-9223372036854775808 ... +9223372036854775807	`long l2 = -1420;` `long l3 = 13453520L; // Mit "l" oder "L"`

`char`	Ohne Vorzeichen
	Eigentlich: Zeichen
		16 Bit	Alle 16 Bit Unicodezeichen
			0 … 65536
				`char c1 = 97; // Repräsentiert 'a'`
				`char c2 = 'a';`

Datentypen für Gleitkommazahlen

Name	Art	RAM	Wertebereich	Beispiele

`float`	Mit Vorzeichen	32 Bit	$\pm 1{,}4^{-45}$ ...
			$\pm 3{,}4028235^{38}$
				`float f1 = 1.25f; // mit "f" oder "F"`
				`float f2 = -0.000025F;`

`double`	Mit Vorzeichen	64 Bit	$\pm 4{,}9^{-324}$ … $\pm 1{,}79769313486231^{57308}$	`double d2 = -3.25;` `double d3 = 5d; // mit "d" oder "D"`

Primitive Zahlentypen - Besonderheiten

Besondere Schreibweisen bei Literalen

- Präfixe erlauben die Darstellung von Literalen zu bestimmten Basen
  - Binär (Basis 2) mit Präfix „**0b**“ oder „**0B**“
  - Oktal (Basis 8) mit Präfix „**0**“
  - Hexadezimal (Basis 16) mit Präfix „**0x**“ oder „**0X**“
- Unterstriche zur besseren Lesbarkeit

```
int a    = 1_000;
double b = 10_000.0;
int c    = 0b1100_0000_1101_0011;
```

  - **Vorsicht:** Der Unterstrich ist an beliebigen Stellen erlaubt

```
int million = 1_000_0000;
```

Das ist gar keine Million!

Beispiel: Darstellungen der Zahl 16

```
int dezimal     = 16;
int binaer      = 0b10000;
int oktal       = 020;
int hexadezimal = 0x10;
```

Achtung: Eine „Vornull“ verändert Ganzzahlen!

- Schreibweisen ohne Vor- oder Nachkommastelle

```
double d1 = .33;  // 0.33
double d2 = 200.; // 200.0
```

- Exponentenschreibweise mit „**e**“ oder „**E**“

```
double exp1 = 0.3e-2; // 0.003
double exp2 = 0.3e2;  // 30.0
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Primitive Zahlentypen - Besonderheiten

Besonderheiten bei Ganzzahlen

- Rechenoperationen werden in Java immer mit dem Typ **`int`** oder **`long`** durchgeführt
- Die Typen **`char`**, **`byte`** und **`short`** werden dabei automatisch in den Typ **`int`** umgewandelt

```
byte a = 10;
byte b = a * a;
```

Kompiliert nicht wegen automatischer Umwandlung von `a` in ein `int`.

Besonderheiten bei Gleitkommazahlen

- Die Datentypen **`double`** und **`float`** können spezielle Werte annehmen
  - Positive und negative Null: **`0.0`** und **`-0.0`**
    - Die Zahl ist näher an der **`0`** als der Darstellbare positive/negative Wert
    - Achtung: **`0.0 == -0.0`** liefert **`true`** und **`0.0 > -0.0`** liefert **`false`**
  - Positives und negatives Unendlich: **`Infinity`**, **`-Infinity`**
  - Not a number: **`NaN`**

Beispiel: Spezielle Werte von Gleitkommazahlen

```
double d = 1E-322 * -0.0001; // -0.0
double e = 1E300 * 1E20;     // Infinity
double f = 0.0 / 0.0;        // NaN
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Primitive Zahlentypen - Besonderheiten

Kompatibilität von Gleitkommazahlen

- Der Modifikator `strictfp`
  - Sorgt für **plattformübergreifende Kompatibilität** bei Gleitkommaberechnungen
  - Darf in angewendet werden für Methoden, Interfaces und Klassen
  - Ohne `strictfp`
    - Evtl. **unterschiedliche Ergebnisse** auf verschiedenen Plattformen
    - **Bestmögliche Genauigkeit** auf jeder Plattform
  - Mit `strictfp`
    - **Gleiches Ergebnis** auf jeder Plattform
    - Evtl. **ungenauere** Ergebnisse

Beispiel: Klasse mit `strictfp`

```
package geometric;

strictfp class Circle {

  double radius;

  Circle(double radius) {
    this.radius = radius;
  }

  double circumference() {
    return 2 * radius * Math.PI;
  }
  ...
}
```

Kompatible Gleitkomma-berechnungen innerhalb der Klasse.

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Primitive Zahlentypen - Grenzen

Primitive Zahlentypen sind keine Objekte

- Sie können nicht in Collections oder anderen Datencontainern verwendet werden

Primitive Datentypen sind in ihrer Größe und Genauigkeit beschränkt

- Primitive Typen haben eine maximale Größe
  - Überläufe werden **nicht** geprüft!

`i * i` ist zu groß und kann nicht mehr in einem `int` gespeichert werden.

```
int i = 2147483647;
int mult = i * i;
System.out.println(mult); // → 1
```

Bei Überläufen kommt es zu falschen Ergebnissen. Der Fehler wird nicht angezeigt.

- Die Typen `float` und `double` sind ungenau
  - Es kommt sehr leicht zu Rundungsfehlern

Beispiel: Ausgabe von 0.02 mit 20 Nachkommastellen

```
System.out.printf("%.20f%n", 0.01f + 0.01f); // 0,01999999955296516400
System.out.printf("%.20f%n", 0.03d - 0.01d); // 0,01999999999999999700
```

  - Kein Compilerfehler wenn eine Zahl nicht genau dargestellt werden kann

Beispiel: Ungenaue Darstellung einer Zahl

```
System.out.println(2345678.88f); // 2345678.9
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Wrapperklassen

Wrapperklassen kapseln primitive Datentypen in einer objektorientierten Hülle

- Sie können in Datenstrukturen verwendet werden die Objekte benötigen, z.B. in **Collections**
- Sie können für **Generics** genutzt werden, z.B. **`ArrayList<Double>`**

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

Erzeugen von Wrappertypen

- Wrapperklassen sind unveränderlich (Design Pattern: immutable)
  - Ein einmal erzeugter Wrappertyp kann nicht mehr verändert werden
  - Zuweisungen führen zu anderen/neuen Objekten im Arbeitsspeicher
- Wrappertypen können mit Hilfe einer Fabrikmethode oder mit „autoboxing“ erzeugt werden

Beispiel: Erzeugen von Wrappertypen aus primitiven Datentypen

```
Integer b = Integer.valueOf(5); // Fabrikmethode
Integer c = 5;                  // autoboxing - entspricht Integer.valueOf(5);
// Verboten
Integer a = new Integer(5);     // deprecated seit Java 9
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Wrapperklassen - Verwendung

Umwandeln von Wrappertypen

- Umwandeln in primitve Typen
  - Das „autounboxing“ macht aus einem Wrappertyp einen primitiven Typ
- Umwandeln in andere Wrappertypen
  - Wrapperklassen für Zahlen sind von einer gemeinsamen Oberklasse **`Number`** abgeleitet

Beispiel: Autounboxing

```
Integer a = 5;
Integer b = 10;
int c = a;          // autounboxing
int d = a + b + 3; // autounboxing
```

Auch Rechenoperatoren veranlassen „autounboxing“.

Beispiel: Umwandlung mit Methode

```
Integer a = 5;
Byte b = a.byteValue();  // Methode aus Number
Long l = a.longValue();  // Methode aus Number
```

Vergleichen von Wrappertypen

- Vergleiche mit „**<**“, „**>**“ oder „**==**“ gelingen nur mit „autounboxing“

Beispiel: Vergleichen von Wrappertypen

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b);       // false -> kein autounboxing (zwei verschiedene Objekte werden verglichen)
System.out.println(a == (b + 0)); // true  -> autounboxing durch das „+ 0“
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Wrapperklassen
- Konstanten

Wrapperklassen bieten einige hilfreiche Konstanten

- Konstanten aus der Wrapperklasse **`Double`**

```
Double.MIN_VALUE          // Kleinste darstellbare Zahl
Double.MAX_VALUE          // Größte darstellbare Zahl
Double.MAX_EXPONENT       // Größter darstellbarer Exponent
Double.MIN_EXPONENT       // Kleinster darstellbarer Exponent
Double.NaN                // „Not a Number“ z.B. bei Division durch 0
Double.NEGATIVE_INFINITY  // Negativ unendlich
Double.POSITIVE_INFINITY  // Positiv unendlich
Double.MIN_NORMAL         // Kleinste positive darstellbare Zahl
```

- Beispiele

```
System.out.println(Double.MIN_VALUE);     // 4.9E-324
System.out.println(Double.MAX_VALUE);     // 1.7976931348623157E308
System.out.println(Double.MAX_EXPONENT); // 1023 (zur Basis 2)
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Wrapperklassen
- Methoden

Methoden der Wrapperklassen (am Beispiel der Klasse Integer)

- Umwandeln von Wrappertypen in Strings
  - Z.B. `String toBinaryString(int i) // Binäre Repräsentation von i`
- Parsen von Strings in primitive Datentypen
  - Z.B. `static int parseInt(String s, int radix) // Parsen der Zahl s zur Basis radix`
- Methoden um Bitoperationen zu vereinfachen
  - Z.B. `static int reverse(int i) // Dreht die Reihenfolge der Bits um`
- Mathematische Methoden
  - Z.B. `static int max(int i, int j) // Maximum von i und j`

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal

Große und genaue Zahlen

- In vielen Fällen reicht die Größe oder die Genauigkeit von primitiven Datentypen oder Wrappertypen nicht aus
  - Z.B. in kryptografischen Anwendungen
  - Z.B. in der Finanzmathematik
- Das Package **`math`** stellt für diese Fälle zwei Klassen zur Verfügung
  - Die Klasse **`BigInteger`** erlaubt Berechnungen mit beliebig große Ganzzahlen
  - Die Klasse **`BigDecimal`** erlaubt Berechnungen mit beliebig großen bzw. genauen Gleitkommazahlen
  - Beide Klassen passen die Objektgröße an die Größe und Genauigkeit von Berechnungsergebnissen an
- Objekte vom Typ **`BigInteger`** und **`BigDecimal`** sind unveränderlich (Design Pattern: immutable)
  - Ein einmal erzeugter Wert kann nicht mehr verändert werden
  - Zuweisungen führen zu anderen/neuen Objekten im Arbeitsspeicher

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigInteger

Erzeugen von großen Ganzzahlen

- Konstanten

```
static final BigInteger ZERO // 0
static final BigInteger ONE  // 1
static final BigInteger TWO //  2
static final BigInteger TEN // 10
```

- Konstruktoren
  - Mit Hilfe eines **`String`**

```
public BigInteger(String val)              // Positive und negative Zahlen zur Basis 10
public BigInteger(String val, int radix) // Positive und negative Zahlen zur angegebenen Basis
```

  - Es gibt weitere Konstruktoren
    - Erzeugen mit Hilfe von Byte-Arrays
    - Erzeugen von Zufallszahlen und zufälligen Primzahlen
- Fabrikmethoden

```
static BigInteger valueOf(long val)
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigInteger

Berechnungen mit Ganzzahlen

- Berechnungen mit Hilfe von Methodenaufrufen

```
public BigInteger add(BigInteger val)        // Summe
public BigInteger subtract(BigInteger val)   // Differenz
public BigInteger multiply(BigInteger val)   // Produkt
public BigInteger divide(BigInteger val)     // Quotient
public BigInteger remainder(BigInteger val) // Restwert
```

- Mathematische Funktionen

```
public BigInteger pow(int exponent)          // Potenz
public BigInteger abs()                      // Betrag
public BigInteger negate()                   // Multiplikation mit -1
public int signum()                          // Vorzeichen der Zahl
public BigInteger gcd(BigInteger val)        // Größter gemeinsamer Teiler
public BigInteger min(BigInteger val)        // Minimum
public BigInteger max(BigInteger val)        // Maximum
```

- Weitere Funktionen, z.B. logische und bitweise Operationen, erstellen von Primzahlen, usw.

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigInteger

Beispiel: Rechnen mit `int`

```
int i = 10;
int j = 100;
int k = 1000;
System.out.println(i * j + k);
// Ausgabe: 2000

int bigNumber = 2000000000;
System.out.println(bigNumber * bigNumber);
// Ausgabe: -1651507200 (falsch wegen Überlauf)

System.out.println(fakultaetInt(40));
// Ausgabe: 0 (falsch wegen Überlauf)
```

Beispiel: Rechnen mit BigInteger

```
BigInteger i = BigInteger.TEN;
BigInteger j = new BigInteger("100");
BigInteger k = BigInteger.valueOf(1000);
System.out.println(i.multiply(j).add(k));
// Ausgabe: 2000

BigInteger bigNumber = new BigInteger("2000000000");
System.out.println(bigNumber.multiply(bigNumber));
// Ausgabe: 4000000000000000000

System.out.println(fakultaetBigInteger(40));
// Ausgabe: 815915283247897734345611269596115894272000000000
```

Beispiel: Fakultätsfunktion mit `int`

```
public static int fakultaetInt(int n) {
    int value = 1;
    for (int i = 2; i <= n; i++) {
        value = value * i;
    }
    return value;
}
```

Beispiel: Fakultätsfunktion mit BigInteger

```
public static BigInteger fakultaetBigInteger(int n) {
  BigInteger value = BigInteger.ONE;
  for (int i = 2; i <= n; i++) {
    value = value.multiply(BigInteger.valueOf(i));
  }
    return value;
}
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigDecimal

Erzeugen von großen Gleitkommazahlen

- Konstanten

```
static final BigDecimal ZERO //  0
static final BigDecimal ONE  //  1
static final BigDecimal TEN  // 10
```

- Konstruktoren

```
public BigDecimal(String val) // Positive und negative Zahlen.
public BigDecimal(double d)   // Vorsicht, hier kann ein Verlust von Genauigkeit auftreten!
```

- Fabrikmethoden

```
static BigDecimal valueOf(double d)             // Wie beim Konstruktor
static BigDecimal valueOf(long val)             // Erzeugung aus long.
static BigDecimal valueOf(long val, int scale) // Erzeugung aus long. Kommaverschiebung nach links.
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigDecimal

Berechungen mit Gleitkommazahlen

- Berechnungen mit Hilfe von Methodenaufrufen (wie bei **BigInteger**)
  - **add**, **substract**, **multiply**, **devide** **remainder**, **pow**, **abs**, **negate**, **plus**, **signum**, **min**, **max**
- Setzen von Nachkommastellen (**int scale**) und Art der Rundung (RoundingMode **mode**)

```
public BigDecimal setScale(int scale, RoundingMode mode)
```

  - Konstanten der Klasse **java.math.RoundingMode**

```
RoundingMode.UP      / RoundingMode.DOWN      // Runden nach 0.
RoundingMode.CEILING / RoundingMode.FLOOR     // Runden nach positiv/negativ unendlich.
RoundingMode.HALF-UP / RoundingMode.HALF-DOWN // Runden zum nächsten Nachbarn.
RoundingMode.HALF-EVEN  // Runden zum nächsten geraden Nachbarn.
RoundingMode.UNNECESARY // Kein Runden. Exaktes Ergebnis notwendig. Standardeinstellung.
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigDecimal

Berechungen mit Gleitkommazahlen

- Besonderheiten der Division
  - Ohne Rundungsangaben liefern nicht-exakte Ergebnisse (wie z.B. **`1/3`**) eine **`ArithmeticException`**
  - Division für exakte Ergebnisse

```
public BigDecimal divide(BigDecimal val)
```

  - Division für nicht exakte Ergebnisse

```
public BigDecimal divide(BigDecimal val, int scale, RoundingMode mode)
public BigDecimal divide(BigDecimal val, RoundingMode mode) // this.scale wird verwendet
```

Beispiel: Dividieren mit BigDecimal

```
System.out.println(BigDecimal.ONE.divide(BigDecimal.valueOf(3), RoundingMode.HALF_UP));    // 0
System.out.println(BigDecimal.ONE.divide(BigDecimal.valueOf(3), 3, RoundingMode.HALF_UP)); // 0.333
System.out.println(BigDecimal.ONE.divide(BigDecimal.valueOf(3), 4, RoundingMode.UP));      // 0.3334
System.out.println(BigDecimal.ONE.divide(BigDecimal.valueOf(3), 5, RoundingMode.DOWN));    // 0.33333
System.out.println(BigDecimal.ONE.divide(BigDecimal.valueOf(3))); // java.lang.ArithmeticException
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal
- BigDecimal

java.math.MathContext

- Für Zahlen vom Typ **`BigDecimal`** kann ein **`MathContext`** festgelegt werden

  `public MathContext(int precision, RoundingMode mode)`

  `int precision`: Anzahl der signifikanten Stellen (auch Vorkommastellen)

  `RoundingMode mode`: Rundungsmethode

- Anwendungsmöglichkeiten
  - Konstruktoren, z.B.

    ```
    public BigDecimal(double d, MathContext mathContext)
    ```

  - Berechnungen, z.B.

    ```
    public BigDecimal divide(BigInteger val, MathContext mathContext)
    ```

Beispiel: MathContext

```
MathContext mathContext = new MathContext(5, RoundingMode.HALF_UP);
BigDecimal bigDecimal = new BigDecimal(123.456789, mathContext);
System.out.println(bigDecimal); // 123.46 (5 signifikante Stellen)
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

BigInteger und BigDecimal - Vergleichen und Umwandeln

Vergleichen und Umwandeln

- Vergleichen von **BigInteger** und **BigDecimal**
  - Vergleiche mit „<“, „>“ oder „==“ sind nicht möglich
  - Statt dessen muss mit Hilfe von Methoden verglichen werden

```
public int compareTo(BigInteger val)// Vergleich nach Größe
public boolean equals(Object o)      // Vergleich nach Inhalt
```

- Umwandlung in einen **String**

```
public String toString()
```

- Umwandlung in primitive Datentypen
  - **BigInteger** und **BigDecimal** erben Umwandlungsmethoden von der Klasse **Number**

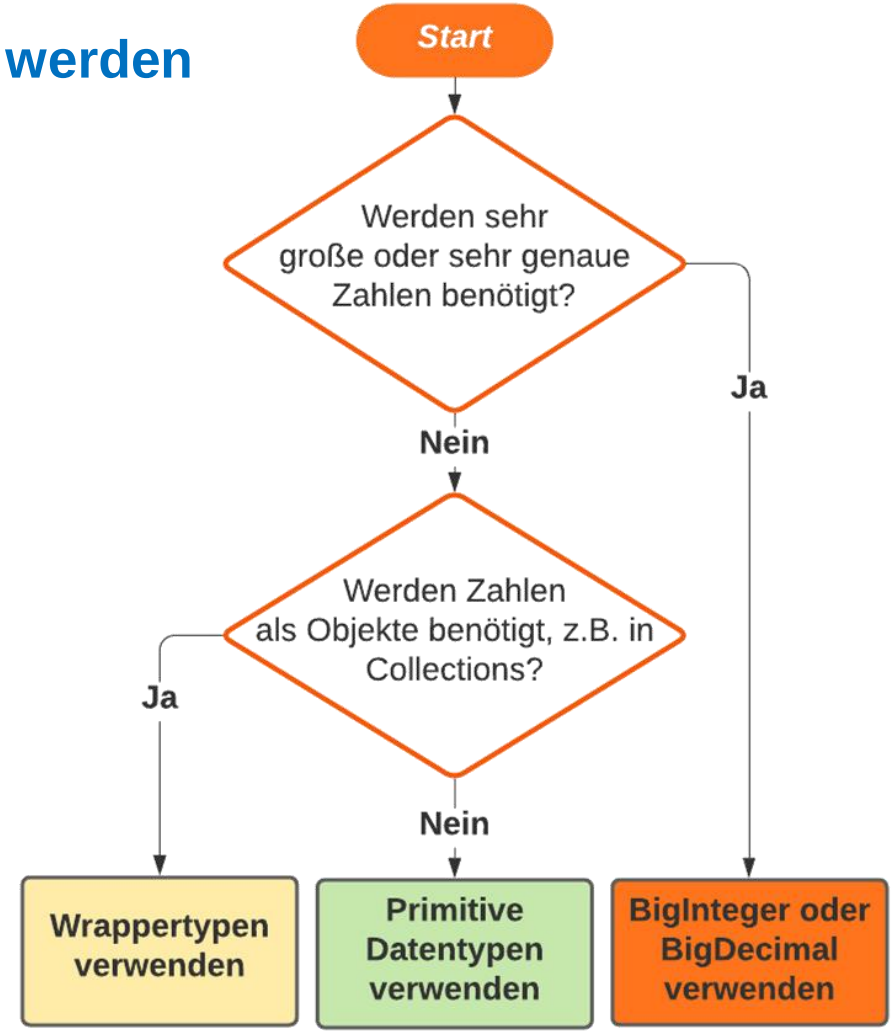
```
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()
```

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Auswahl des richtigen Datentypen

Primitive Datentypen sollten so oft wie möglich verwendet werden

- **Vorteile der primitiven Datentypen**
  - Zeit- und speichereffiziente Verarbeitung
- **Nachteile der nicht primitiven Typen**
  - **Wrappertypen**
    - Durch die objektorientierte Hülle wird mehr Speicher und Rechenzeit benötigt als bei primitive Datentypen
    - Boxing / unboxing muss beachtet werden
  - **BigInteger und BigDecimal**
    - Die beliebige Genauigkeit und Zahlengröße geht auf Kosten der Zeit- und Speichereffizienz
    - Rechenoperatoren (+, -, *. /, …) können nicht direkt verwendet werden



Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Auswahl des richtigen Datentypen - Primitive Datentypen

Ganzzahlige primitive Datentypen

- Der Datentyp **`int`** ist die beste Wahl
  - Ganzzahlige Literale werden in Java als **`int`** interpretiert
  - Der Typ **`int`** bietet die höchste Performanz bei Berechnungen
    - Kleinere Typen (**`byte`** und **`short`**) werden vor Rechenoperationen automatisch in **`int`** umgewandelt

    → Langsamere Berechnungen, schlechte Lesbarkeit und problematische Casts

Primitive Gleitkommatypen

- Der Datentyp **`double`** ist die beste Wahl
  - Gleitkommazahl-Literale werden als **`double`** interpretiert
  - Der Typ **`double`** bietet eine höhere Genauigkeit als **`float`**
- **Aber:** Rechenoperationen mit **`float`** sind meist schneller

Beispiel: Klasse mit kleinen Ganzzahltypen

```
package datentypen;

public class KleineTypen {

  public static void main(String[] args) {
    byte b = 10;
    byte summe = summe(b, (byte) 5);
  }

  static byte summe(byte b1, byte b2) {
    return (byte) (b1 + b2);
  }
}
```

Zwei Casts sind notwendig.

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de

Zusammenfassung

- Datentypen für Zahlen
- Primitive Zahlentypen
  - Besonderheiten
  - Grenzen
- Wrapperklassen
  - Verwendung
  - Konstanten
  - Methoden
- BigInteger und BigDecimal
  - Erzeugen und Berechnungen
  - MathContext
- Auswahl des richtigen Datentypen

Prof. Dr. rer. nat. Andreas Berl – www.andreas-berl.de