OOP with Java

5. Operators and Expressions

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

website

- So far, we have assigned two things to variables

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable

Expressions

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable
- Actually, we can assign much more complex stuff, namely *expressions*

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable
- Actually, we can assign much more complex stuff, namely *expressions*
- The assignment of a variable actually is something like

```
[variableName] = [expressionOfCompatibleType]
```

Expressions

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable
- Actually, we can assign much more complex stuff, namely *expressions*
- The assignment of a variable actually is something like

```
[variableName] = [expressionOfCompatibleType]
```

- A literal value of a type, say a number, is an expression

Expressions

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable
- Actually, we can assign much more complex stuff, namely *expressions*
- The assignment of a variable actually is something like

```
[variableName] = [expressionOfCompatibleType]
```

- A literal value of a type, say a number, is an expression
- A variable itself can be used in an expression as well

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable
- Actually, we can assign much more complex stuff, namely *expressions*
- The assignment of a variable actually is something like

```
[variableName] = [expressionOfCompatibleType]
```

- A literal value of a type, say a number, is an expression
- A variable itself can be used in an expression as well
- The assignment itself is an expression (of the type of the assigned variable)

Expressions

- So far, we have assigned two things to variables:
  1. a value literal of the right (compatible) type
  2. the value of another, compatible variable
- Actually, we can assign much more complex stuff, namely *expressions*
- The assignment of a variable actually is something like
  `[variableName] = [expressionOfCompatibleType]`
- A literal value of a type, say a number, is an expression
- A variable itself can be used in an expression as well
- The assignment itself is an expression (of the type of the assigned variable)
- Operators can be grouped by parentheses `()`

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows. . .
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows. . .

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows. . .
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows. . .
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows. . .

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows. . .
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows. . .
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows. . .
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!). . .
`A%B`	the result of the rest of the integer division of `A` and `B`

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows. . .
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows. . .
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows. . .
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!). . .
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign
`A<<B`	the result of shifting `A` by `B` bits to the left

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign
`A<<B`	the result of shifting `A` by `B` bits to the left
`A|B`	the result of the bit-wise "or" of `A` and `B`

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows. . .
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows. . .
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows. . .
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!). . .
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign
`A<<B`	the result of shifting `A` by `B` bits to the left
`A|B`	the result of the bit-wise "or" of `A` and `B`
`A&B`	the result of the bit-wise "and" of `A` and `B`

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign
`A<<B`	the result of shifting `A` by `B` bits to the left
`A|B`	the result of the bit-wise "or" of `A` and `B`
`A&B`	the result of the bit-wise "and" of `A` and `B`
`A^B`	the result of the bit-wise "xor" of `A` and `B`

`A` and `B` are expressions of the right numerical type

Integer Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign
`A<<B`	the result of shifting `A` by `B` bits to the left
`A|B`	the result of the bit-wise "or" of `A` and `B`
`A&B`	the result of the bit-wise "and" of `A` and `B`
`A^B`	the result of the bit-wise "xor" of `A` and `B`
`~A`	the result of the bit-wise "not" of `A`

`A` and `B` are expressions of the right numerical type

`A+B`	the result of the addition of `A` and `B`, careful with overflows...
`A-B`	the result of the subtraction of `A` and `B`, careful with overflows...
`A*B`	the result of the multiplication of `A` and `B`, careful with overflows...
`A/B`	the result of the integer division of `A` by `B`, careful result is truncated (no fractions!)...
`A%B`	the result of the rest of the integer division of `A` and `B`
`A>>B`	the result of shifting `A` by `B` bits to the right without touching the sign
`A>>>B`	the result of shifting `A` by `B` bits to the right, shifting everything, also the highest-order bit/sign
`A<<B`	the result of shifting `A` by `B` bits to the left
`A|B`	the result of the bit-wise "or" of `A` and `B`
`A&B`	the result of the bit-wise "and" of `A` and `B`
`A^B`	the result of the bit-wise "xor" of `A` and `B`
`~A`	the result of the bit-wise "not" of `A`

`A` and `B` are expressions of the right numerical type

- integer arithmetic is exact, i.e., $((a-b)-a)+b=0$, but since we have only 8, 16, 32, or 64 bits, the range of numbers we can represent is limited
- thus, if, e.g., $a+b$ is outside of the range of numbers we can represent, it will be "wrapped back in", i.e., we get the wrong result

Listing: A program computing with integer values.

```java
/** Examples for integer arithmetic */
public class IntegerArithmetic {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    int res; // declare int variable res

    res = 5 + 4; // store 5 + 4 in variable "res"
    System.out.println(res); // prints 9
    res = res + 4; // store res + 4 in variable "res"
    System.out.println(res); // prints 13
    res = res + 4; // store res + 4 in variable "res"
    System.out.println(res); // prints 17
    res = 171 / res; // _integer_ divide 171 by "res" (17)
    System.out.println(res); // prints 10
    res = res * 7; // multiply "res" with 7
    System.out.println(res); // prints 70
    res = res % 8; // rest of the integer division of "res" (70) by 8
    System.out.println(res); // prints 6

    res = 3 * 6 + 10 - 4 * 5; // = ((3 * 6) + 10) - (4 *5)
    System.out.println(res); // prints 8

    res = 3 * ((6 + 10) - 4) * 5; // now with different grouping
    System.out.println(res); // prints 180
  }
}
```

Example for Overflow in Integer Arithmetic

Listing: A program encountering an integer overflow.

```java
/** Examples for integer arithmetic overflow */
public class IntegerOverflow {

  /** The main routine
   * @param args
   *           we ignore this parameter for now */
  public static final void main(String[] args) {
    int res; // declare int variable res

    res = 1_000_000; // store 1_000_000 in variable "res"
    System.out.println(res); // prints 1000000
    res = res * 1000; // store res * 1000 in variable "res" = 1_000_000_000
    System.out.println(res); // prints 1000000000

    res = res * 3; // store res * 3 in variable "res", should be 3_000_000_000
    System.out.println(res); // prints -1294967296: res has overflown,
                             // it does _not_ have value 3_000_000_000, but
                             // 3_000_000_000 - Integer.MAX_VALUE + Integer.MIN_VALUE + 1
  }
}
```

Example for Bit Shifting with Integers

Listing: A program performing integer bit shifting.

```java
/** Examples for bit shifting in integer expressions */
public class IntegerBitShifting {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    int res; // declare int variable res

    res = 128; // store 128 = 2^7 in variable "res" (we use "^" here as power operator, not as xor...)
    System.out.println(res); // prints 128
    res = res << 2; // shift res two bits to the left: get 2^(7+2) = 2^9 = 512
    System.out.println(res); // prints 512, as  res<<x is equivalent to  res*2^x

    res = res >> 3; // shift res three bits to the right: get 2^(9-3) = 2^6 = 64
    System.out.println(res); // prints 64, as  res>>x is equivalent to res/(2^x)

    res = 0b11000000_00000000_00000000_00000000; // store 3 << 30 in binary form in res
    System.out.println(res); // prints -1073741824 (highest-order bit in the two's complement determines sign)

    res = 0b11000000_00000000_00000000_00000000 >> 1; // shift -1073741824 right by 1 without touching sign
    System.out.println(res); // prints -1073741824 / 2 = -536870912, >>x is equivalent to signed div by 2^x

    res = 0b11000000_00000000_00000000_00000000 >>> 1; // shift -1073741824 right by 1 and shift sign stuff too
    System.out.println(res); // prints -1610612736, 0b11000000_00000000_00000000_00000000 would be
                             // unsigned int 3221225472 and 3221225472/2 = 1610612736
  }
}
```

Examples for Integer Bit Operators

Listing: A program working with bit operators on integer values.

```java
/** Examples for bit operators in integer expressions */
public class IntegerBitOperators {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    int res; // declare int variable res

    res = 1; // store 1 in variable "res"
    System.out.println(res); // prints 1
    res = res | 1; // binary or with 1, result still 1
    System.out.println(res); // prints 1
    res = res | 8; // binary or with 8, result still 0b1001 = 9
    System.out.println(res); // prints 9

    res = res & 24; // binary or of res and 24, where 24 = 8 | 16
    System.out.println(res); // prints 8

    res = res ^ 9; // binary xor of 8 and 9, where 9 = 8 | 1, leaves 1
    System.out.println(res); // prints 1

    res = ~res; // binary not of 1, set all bits except the first 1
    System.out.println(res); // prints -2
  }
}
```

Floating Point Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, overflows become infinity...

`A` and `B` are expressions of the right numerical type

Floating Point Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, overflows become infinity...
`A-B`	the result of the subtraction of `A` and `B`, overflows become infinity...

`A` and `B` are expressions of the right numerical type

Floating Point Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, overflows become infinity...
`A-B`	the result of the subtraction of `A` and `B`, overflows become infinity...
`A*B`	the result of the multiplication of `A` and `B`, overflows become infinity...

`A` and `B` are expressions of the right numerical type

`A+B`	the result of the addition of `A` and `B`, overflows become infinity...
`A-B`	the result of the subtraction of `A` and `B`, overflows become infinity...
`A*B`	the result of the multiplication of `A` and `B`, overflows become infinity...
`A/B`	the result of the integer division of `A` by `B`, overflows become infinity...

`A` and `B` are expressions of the right numerical type

Floating Point Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, overflows become infinity...
`A-B`	the result of the subtraction of `A` and `B`, overflows become infinity...
`A*B`	the result of the multiplication of `A` and `B`, overflows become infinity...
`A/B`	the result of the integer division of `A` by `B`, overflows become infinity...
`A%B`	the result of the rest of the integer division of `A` and `B`

`A` and `B` are expressions of the right numerical type

Floating Point Expressions: Basic Operators

`A+B`	the result of the addition of `A` and `B`, overflows become infinity...
`A-B`	the result of the subtraction of `A` and `B`, overflows become infinity...
`A*B`	the result of the multiplication of `A` and `B`, overflows become infinity...
`A/B`	the result of the integer division of `A` by `B`, overflows become infinity...
`A%B`	the result of the rest of the integer division of `A` and `B`

`A` and `B` are expressions of the right numerical type

- due to the limited number of bits in mantissa and exponent, floating point arithmetic is not necessarily exact, i.e., $((a-b)-a)+b$ may be different from 0 sometimes

Examples for Floating Point Arithmetic

Listing: A program working with floating point arithmetic.

```
/** Examples for floating point arithmetic */
public class FloatingPointArithmetic {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    double res1, res2; // declare double variable res1 and res2

    res1 = 5d + 4.1d; // store 5 + 4.1 in variable "res1"
    System.out.println(res1); // prints 9.1
    res1 = res1 + 4.1d; // store res1 + 4.1d in variable "res1"
    System.out.println(res1); // prints 13.2
    res1 = res1 + 4.1d; // store res1 + 4.1d in variable "res1"
    System.out.println(res1); // prints 17.299999999999997: double precision is limited (ca. 15 decimals)!

    res2 = 171d / res1; // divide 171 by "res1" (which is almost but not exactly 17.3)
    System.out.println(res2); // prints 9.884393063583817 (which is a good approximation)
    res2 = res1 * res2; // multiply res1 with res2, i.e., with 171/res1, we get 171/(171/res1)
    System.out.println(res2); // prints 171.0, that worked well!
    res2 = (171d / res1) * 17.3d; // (171d/res1)*17.3d ... res1 would ideally be 17.3, but is not
    System.out.println(res2); // prints 171.00000000000003: a bit off due to limited precision

    res1 = (((10d / 8d) * 8d) - 10.1d) + 0.1d; // this should be 0
    System.out.println(res1); // prints 3.608224830031759E-16: limited precision (about 15 decimal places!)

    res1 = ((10.7d - 0.12d) - 10.7d + 0.12d); // this should be 0
    System.out.println(res1); // prints 7.771561172376096E-16: limited precision (about 15 decimal places!)

    res1 = (8.5d % 4.1d); // compute the rest of the integer division of 8.5 and 4.1
    System.out.println(res1); // should be about (8.5-(2*4.1))=0.3, is 0.3000000000000007
  }
}
```

Example: Vertical Ball Throw

- Assume a $x_0 = 1.8m$ tall person throws a ball vertically upwards into the air with $v_0 = 10m/s$ initial velocity.
- Where is the ball after $t = 1.5s$?

$$x(t) = x_0 + v_0 * t - 0.5 * g * t^2, \text{ where } g = 9.80665 \quad (1)$$

How would you compute the position in a program?

Example: Vertical Ball Throw

Listing: A program computing $x(t)$.

```
/**
 * A ball is thrown vertically upwards into the air by a 1.8m tall person
 * with velocity 10m/s. Where is it after t=1.5 seconds?<br/>
 * x(t) = x_0 + v_0 * t - 0.5 * g * t^2
 */
public class VerticalBallThrow {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    double x0 = 1.8d; // initial vertical position
    double v0 = 10d; // initial velocity upwards
    double g = 9.80665d; // free fall acceleration downwards
    double t = 1.5d; // the time
    double xt = x0 + (v0*t) - 0.5d*g*t*t; // x(t) = x_0 + v_0 * t - 0.5 * g * t^2
    System.out.println(xt); // prints 5.767518750000001
  }
}
```

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types
`V*=E`	multiply the value of `V` with value of expression `E`, store the result in `V` and return it; only numerical types

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types
`V*=E`	multiply the value of `V` with value of expression `E`, store the result in `V` and return it; only numerical types
`V|=E`	compute the bit-based "or" of the the value of `V` and the value of expression `E`, store the result in `V` and return it; only `boolean`

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types
`V*=E`	multiply the value of `V` with value of expression `E`, store the result in `V` and return it; only numerical types
`V|=E`	compute the bit-based "or" of the the value of `V` and the value of expression `E`, store the result in `V` and return it; only `boolean`
`VO=E`	Generalization of the above: `O` can be any binary (two-argument) operator: Apply `O` to values of variable `V` and expression `E`, store the result of `O` in variable `V` and return it; `V` and `E` must have compatible types

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types
`V*=E`	multiply the value of `V` with value of expression `E`, store the result in `V` and return it; only numerical types
`V|=E`	compute the bit-based "or" of the the value of `V` and the value of expression `E`, store the result in `V` and return it; only `boolean`
`VO=E`	Generalization of the above: `O` can be any binary (two-argument) operator: Apply `O` to values of variable `V` and expression `E`, store the result of `O` in variable `V` and return it; `V` and `E` must have compatible types
`V++`	add `1` to numerical variable `V`, return the value that `V` had *before* adding

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Assignment Operators: Any Type in, Same Type out

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types
`V*=E`	multiply the value of `V` with value of expression `E`, store the result in `V` and return it; only numerical types
`V|=E`	compute the bit-based "or" of the the value of `V` and the value of expression `E`, store the result in `V` and return it; only `boolean`
`VO=E`	Generalization of the above: `O` can be any binary (two-argument) operator: Apply `O` to values of variable `V` and expression `E`, store the result of `O` in variable `V` and return it; `V` and `E` must have compatible types
`V++`	add `1` to numerical variable `V`, return the value that `V` had *before* adding
`++V`	add `1` to numerical variable `V`, return the value that `V` has *after* adding

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

`V=E`	store the value of expression `E` in variable `V` *and return it*
`V+=E`	add value of expression `E` to variable `V`, store the result in `V` and return it; only numerical types
`V*=E`	multiply the value of `V` with value of expression `E`, store the result in `V` and return it; only numerical types
`V|=E`	compute the bit-based "or" of the the value of `V` and the value of expression `E`, store the result in `V` and return it; only `boolean`
`VO=E`	Generalization of the above: `O` can be any binary (two-argument) operator: Apply `O` to values of variable `V` and expression `E`, store the result of `O` in variable `V` and return it; `V` and `E` must have compatible types
`V++`	add `1` to numerical variable `V`, return the value that `V` had *before* adding
`++V`	add `1` to numerical variable `V`, return the value that `V` has *after* adding
`V--`	subtract `1` from numerical variable `V`, return the value that `V` had *before* the subtraction
`--V`	subtract `1` to numerical variable `V`, return the value that `V` has *after* the subtraction

`V` is a variable, `E` is an expressions of a compatible type, `O` is any of the binary (two-argument) operators discussed for that type

Examples for In-Place Operators

Listing: A program working with in-place operators.

```java
/** Examples for in-place operators */
public class InPlaceOperators {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    int i = 1, j = 5; // declare the two integer variables i and j, where i=1 and j=5

    System.out.println(i); // prints 1
    i += j; // add j and i, store result in i, i.e., i=i+j = (1+5)
    System.out.println(i); // prints 6
    j *= i; // multiply j with i, store result in j (j = (j*i) = 6*5
    System.out.println(j); // prints 30
    j /= ++i; // first, add 1 to i (store result in i) and return it, then divide j by it and store in j
    System.out.println(i); // prints 7
    System.out.println(j); // prints 30/7 = 4
    i *= j++; // first, add 1 to j (store result but return old value), then multiply with i and store in i
    System.out.println(j); // prints 4+1 = 5
    System.out.println(i); // prints 7 * 4 = 28
    i |= (j ^= 3); // first do binary xor of j with 3, store in j, then binary or of result with i and store
    System.out.println(j); // prints (5)^3 = (1 | 4) ^ 3 = 2 | 4 = 6
    System.out.println(i); // prints (28) | 6 = (16 | 8 | 4) | 2 = 30
    i *= ((--j) * (i++)) - j; // this is a tough cooky: i = 30 * [((6-1) * 30)) - 5] <-- never do such stuff
    System.out.println(i); // prints 30*145=4350 .. (the final multiplication takes the original i value)
  }
}
```

Comparison Operators: Any Type in, `boolean` out

`A==B`	`true` if and only if `A` has the exact same value as `B`

`A` and `B` are expressions of compatible types

Comparison Operators: Any Type in, `boolean` out

`A==B`	`true` if and only if `A` has the exact same value as `B`
`A!=B`	`true` if and only if `A` does not have the exact same value as `B`

`A` and `B` are expressions of compatible types

Comparison Operators: Any Type in, `boolean` out

`A==B`	`true` if and only if `A` has the exact same value as `B`
`A!=B`	`true` if and only if `A` does not have the exact same value as `B`
`A>B`	`true` if and only if `A` has a greater value as `B`, numerical types only

`A` and `B` are expressions of compatible types

Comparison Operators: Any Type in, `boolean` out

`A==B`	`true` if and only if `A` has the exact same value as `B`
`A!=B`	`true` if and only if `A` does not have the exact same value as `B`
`A>B`	`true` if and only if `A` has a greater value as `B`, numerical types only
`A>=B`	`true` if and only if `A` has a greater or equal value as `B`, numerical types only

`A` and `B` are expressions of compatible types

Comparison Operators: Any Type in, `boolean` out

`A==B`	`true` if and only if `A` has the exact same value as `B`
`A!=B`	`true` if and only if `A` does not have the exact same value as `B`
`A>B`	`true` if and only if `A` has a greater value as `B`, numerical types only
`A>=B`	`true` if and only if `A` has a greater or equal value as `B`, numerical types only
`A<B`	`true` if and only if `A` has a smaller value as `B`, numerical types only

`A` and `B` are expressions of compatible types

Comparison Operators: Any Type in, `boolean` out

`A==B`	`true` if and only if `A` has the exact same value as `B`
`A!=B`	`true` if and only if `A` does not have the exact same value as `B`
`A>B`	`true` if and only if `A` has a greater value as `B`, numerical types only
`A>=B`	`true` if and only if `A` has a greater or equal value as `B`, numerical types only
`A<B`	`true` if and only if `A` has a smaller value as `B`, numerical types only
`A<=B`	`true` if and only if `A` has a smaller or equal value as `B`, numerical types only

`A` and `B` are expressions of compatible types

Examples for Comparison Operators

Listing: A program working with comparison operators.

```
/** Examples for comparison operators */
public class ComparisonOperators {

  /** The main routine
   * @param args
   *           we ignore this parameter for now */
  public static final void main(String[] args) {
    double a = 5d, b = 6d; // allocate and initialize two double variables
    boolean c = (a == b); // allocate boolean c is true if a==b, false otherwise
    System.out.println(c); // false
    boolean d = (a < b); // allocate boolean d, set to true iff a < b
    System.out.println(d); // true
    boolean e = (c == d); // allocate boolean e, set to true if boolean c == boolean d
    System.out.println(e); // false
    e = (c = d); // careful here: (c = d) is not a comparison but in-place assignment...
    System.out.println(e); // true
    e = (((71d - 0.1d) - 71d) + 0.1d) == 0d; // should be 0, but remember limited precision...
    System.out.println(e); // false: never use == or != with floating point, use >=, <=, <, > only
    e = (5.4d != 4.5d); // is 5.4 different from 4.5?
    System.out.println(e); // true, ok, if that would not work, we would have a serious problem :-)
  }
}
```

Boolean Expressions: Basic Operators

`A&&B`	the result of the Boolean "and" of `A` and `B`

`A` and `B` are `boolean` expressions

Boolean Expressions: Basic Operators

`A&&B`	the result of the Boolean "and" of `A` and `B`
`A||B`	the result of the Boolean "or" of `A` and `B`

`A` and `B` are `boolean` expressions

Boolean Expressions: Basic Operators

`A&&B`	the result of the Boolean "and" of `A` and `B`
`A||B`	the result of the Boolean "or" of `A` and `B`
`A^B`	the result of the Boolean "xor" of `A` and `B`

`A` and `B` are `boolean` expressions

`A&&B`	the result of the Boolean "and" of `A` and `B`
`A||B`	the result of the Boolean "or" of `A` and `B`
`A^B`	the result of the Boolean "xor" of `A` and `B`
`!A`	the result of the Boolean "not"

`A` and `B` are `boolean` expressions

Examples for Boolean Operators

Listing: A program working with Boolean operators.

```
/** Examples for boolean operators */
public class BooleanOperators {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    boolean res; // declare boolean variable res

    res = false || true; // store false "or" true in variable "res"
    System.out.println(res); // prints true
    res = res && false; // store res "and" false  in variable "res"
    System.out.println(res); // prints false
    res = !res; // store "not" res in variable "res"
    System.out.println(res); // prints true
    res = res ^ res; // store res "xor" res in res
    System.out.println(res); // prints false
  }
}
```

Ternary Expression: `boolean` in, any type out

`A?B:C`	the result of `B` if and only if the `boolean` expression `A` evaluates to `true`, the result of `B` otherwise

`A` is a `boolean` expression, `B` and `C` are expressions of compatible types

Examples for the Ternary Operator

Listing: A program working with the ternary operator.

```java
/** Examples for boolean operators */
public class TernaryOperator {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    int a = 5, b = 11; // declare and initialize int variables a=5 and b=11
    double c = (a > b) ? -1d : 1d; // if a>b, set c=-1d; otherwise set c=1d;
    System.out.println(c); // prints 1.0
    c = (a >= (b/2)) ? (2d * c) : (2d / c); // if a>=b/2, set c to 2c else to 2/c
    System.out.println(c); // prints 2.0, since b/2 is 5 due to integer division
    boolean d = (c>a) ? true : (a > b); // if c>a, then set d to true, else set d to (a>b)
    System.out.println(d); // false: since c<a, we check whether a>b, which is false
  }
}
```

`A+B`	return a `String` which starts `A` and continues with the `String` representation of `B` (convert `B` to `String` if necessary)

String Concatenation

`A+B`	return a `String` which starts `A` and continues with the `String` representation of `B` (convert `B` to `String` if necessary)
`A+=B`	append the `String` representation of `B` (convert `B` to `String` if necessary) to the `String` `A` and store the result in `A`

Examples String Concatenation

Listing: A program working with the String concatenation.

```
/** Examples for String concatenation */
public class StringConcatenation {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    String text; // declare String variable text

    text = "Hello␣World!"; //store Hello World! in text //$NON-NLS-1$
    System.out.println(text); // prints "Hello World!"
    text = text + "␣It's␣me!"; //add It's Me! to text //$NON-NLS-1$
    System.out.println(text); // prints "Hello World! It's me!"
    text = "The␣result␣of␣5+6␣is␣" + (5+6); // concatenate (5+6) as string to string //$NON-NLS-1$
    System.out.println(text); // prints "The result of 5+6 is 11"
    text = "The␣result␣of␣5+6␣is␣" + 5 + 6; // concatenate 5 to string then 6 string //$NON-NLS-1$
    System.out.println(text); // prints "The result of 5+6 is 56" <- careful with concatenation, use ()
  }
}
```

Remark: These //NON-NLS-1 things can safely be ignored, they are just there to tell Eclipse that a `String` literal is not internationalized/stored in a resource but to be used as it. Ignore them.

Summary

- We have learned how to compute with the basic types in Java
- We have looked into what to do with integers, floating point numbers, booleans, and strings
- We have learned binary mathematical operators, in-place assignments/updates, string concatenation, bit operators, etc.
- We have also looked a bit more into the limits of the types: All data types occupy a finite, fixed amount of memory and therefore can only represent a finite, fixed amount of values
- We had one productive example already, computing the position of a vertically-upwards thrown ball

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog