OOP with Java

4. Types, Variables, and Assignments

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

1. Introduction to Data Types
2. Variables
3. Limits of Variables
4. Compatible Types
5. Summary

website

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
    - limited in size

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and
  - ultimately be broken down to bits

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and
  - ultimately be broken down to bits
- In Java, we can distinguish different sorts of data types

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and
  - ultimately be broken down to bits
- In Java, we can distinguish different sorts of data types:
  - the type `boolean`, which uses its bits to represent a Boolean decision of `true` or `false`

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and
  - ultimately be broken down to bits
- In Java, we can distinguish different sorts of data types:
  - the type `boolean`, which uses its bits to represent a Boolean decision of `true` or `false`,
  - integer types, which use their bits to represent whole numbers, i.e., subsets of $\mathbb{Z}$

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and
  - ultimately be broken down to bits
- In Java, we can distinguish different sorts of data types:
  - the type `boolean`, which uses its bits to represent a Boolean decision of `true` or `false`,
  - integer types, which use their bits to represent whole numbers, i.e., subsets of $\mathbb{Z}$,
  - floating point types, which use their bits to represent fractional numbers, i.e., subsets of $\mathbb{R}$

Types of Data Types

- A data type describes the nature of data that a variable or expression can contain/represent
- Our computers use memory which is
  - limited in size and
  - ultimately consists of bits (binary values `0` and `1`)
- Therefore, all data that we can represent must be
  - limited in size and
  - ultimately be broken down to bits
- In Java, we can distinguish different sorts of data types:
  - the type `boolean`, which uses its bits to represent a Boolean decision of `true` or `false`,
  - integer types, which use their bits to represent whole numbers, i.e., subsets of $\mathbb{Z}$,
  - floating point types, which use their bits to represent fractional numbers, i.e., subsets of $\mathbb{R}$, and
  - the type `char`, which uses its bits to represent a character

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \ldots 2^7-1$, i.e., $-128 \ldots 127$

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$
`int`	signed 32 bit integer, whole numbers from range $-2^{31} \dots 2^{31}-1$, i.e., $-2147483648 \dots 2147483647$

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$
`int`	signed 32 bit integer, whole numbers from range $-2^{31} \dots 2^{31}-1$, i.e., $-2147483648 \dots 2147483647$
`long`	signed 64 bit integer, whole numbers from range $-2^{63} \dots 2^{63}-1$, i.e., $-9223372036854775808 \dots 9223372036854775807$

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$
`int`	signed 32 bit integer, whole numbers from range $-2^{31} \dots 2^{31}-1$, i.e., $-2147483648 \dots 2147483647$
`long`	signed 64 bit integer, whole numbers from range $-2^{63} \dots 2^{63}-1$, i.e., $-9223372036854775808 \dots 9223372036854775807$
`float`	signed 32 bit floating point number (1 sign bit, 8 bit signed exponent, 23 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-149},(2-2^{-23})*2^{127}] \cup \{0,-\infty,\infty,\emptyset\}$

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$
`int`	signed 32 bit integer, whole numbers from range $-2^{31} \dots 2^{31}-1$, i.e., $-2147483648 \dots 2147483647$
`long`	signed 64 bit integer, whole numbers from range $-2^{63} \dots 2^{63}-1$, i.e., $-9223372036854775808 \dots 9223372036854775807$
`float`	signed 32 bit floating point number (1 sign bit, 8 bit signed exponent, 23 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-149}, (2-2^{-23}) * 2^{127}] \cup \{0, -\infty, \infty, \emptyset\}$
`double`	signed 64 bit floating point number (1 sign bit, 11 bit signed exponent, 52 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-1074}, (2-2^{-52}) * 2^{1023}] \cup \{0, -\infty, \infty, \emptyset\}$

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$
`int`	signed 32 bit integer, whole numbers from range $-2^{31} \dots 2^{31}-1$, i.e., $-2147483648 \dots 2147483647$
`long`	signed 64 bit integer, whole numbers from range $-2^{63} \dots 2^{63}-1$, i.e., $-9223372036854775808 \dots 9223372036854775807$
`float`	signed 32 bit floating point number (1 sign bit, 8 bit signed exponent, 23 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-149}, (2-2^{-23})*2^{127}] \cup \{0, -\infty, \infty, \emptyset\}$
`double`	signed 64 bit floating point number (1 sign bit, 11 bit signed exponent, 52 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-1074}, (2-2^{-52})*2^{1023}] \cup \{0, -\infty, \infty, \emptyset\}$
`char`	a single 16 bit unicode character, can be any character from any of the major languages

Data Types

- in Java, we can process data of several basic primitive data types

`boolean`	either `true` or `false`
`byte`	signed 8 bit integer, whole numbers from range $-2^7 \dots 2^7-1$, i.e., $-128 \dots 127$
`short`	signed 16 bit integer, whole numbers from range $-2^{15} \dots 2^{15}-1$, i.e., $-32768 \dots 32767$
`int`	signed 32 bit integer, whole numbers from range $-2^{31} \dots 2^{31}-1$, i.e., $-2147483648 \dots 2147483647$
`long`	signed 64 bit integer, whole numbers from range $-2^{63} \dots 2^{63}-1$, i.e., $-9223372036854775808 \dots 9223372036854775807$
`float`	signed 32 bit floating point number (1 sign bit, 8 bit signed exponent, 23 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-149}, (2-2^{-23}) * 2^{127}] \cup \{0, -\infty, \infty, \emptyset\}$
`double`	signed 64 bit floating point number (1 sign bit, 11 bit signed exponent, 52 bit unsigned mantissa + hidden bit), subset of real numbers from $\pm[2^{-1074}, (2-2^{-52}) * 2^{1023}] \cup \{0, -\infty, \infty, \emptyset\}$
`char`	a single 16 bit unicode character, can be any character from any of the major languages
`String`	a piece of text, at most 2147483647 characters (but literals are limited to 65536 characters), (not actually a primitive type)

Variables

- A variable is a container which has a specific data type and can store exactly one value of that type

Variables

- A variable is a container which has a specific data type and can store exactly one value of that type
- Variables are declared with statements of the form `[type] [variableName];` which creates a variable with name `variableName` and of type `type`.

Variables

- A variable is a container which has a specific data type and can store exactly one value of that type
- Variables are declared with statements of the form `[type] [variableName];` which creates a variable with name `variableName` and of type `type`.
- We can store a value in a variable by using statements of the form `[variableName] = [expression]`, where `variableName` is the name of the variable and expression must be an expression of the right type

Variables

- A variable is a container which has a specific data type and can store exactly one value of that type
- Variables are declared with statements of the form `[type] [variableName];` which creates a variable with name `variableName` and of type `type`.
- We can store a value in a variable by using statements of the form `[variableName] = [expression]`, where `variableName` is the name of the variable and expression must be an expression of the right type
- When a program is executed, variables exist in the RAM assigned to the process. After the process has terminated, they disappear.

Examples for `boolean` Variables

Listing: A program allocating, initializing, and printing `boolean` variables.

```java
/** Examples for boolean variables */
public class BooleanVariables {

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

    var = true; // set variable var to true
    System.out.println(var); // prints true

    var = false; // set variable var to false
    System.out.println(var);// prints false

    var = false; // set variable var to false
    var = true; // set variable var to true
    System.out.println(var); // prints true

    boolean a = false, b = true, c = false; // allocate and initialize three variables
    System.out.println(a); // print the value of a, which is false
    System.out.println(b); // print the value of b, which is true
    System.out.println(c); // print the value of c, which is false
  }
}
```

Examples for byte Variables

Listing: A program allocating, initializing, and printing byte variables.

```java
/** Examples for byte variables */
public class ByteVariables {

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

    var = -1; // set variable var to -1
    System.out.println(var); // prints -1

    var = -128; // set variable var to -128
    System.out.println(var); // prints -128

    var = 1_2_7; // set variable var to 127
    System.out.println(var); // prints 127

    byte hex = 0x10; // set hex to hexadecimal 10, which is 1*16+0 = 16: starts with "0x"
    System.out.println(hex); // prints 16

    byte bin = 0b0110_1111; // set bin to binary 01101111, which is 1+2+4+8+32+64=111: starts with
        "0b"
    System.out.println(bin); // prints 111
  }
}
```

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10:
  - `1234` means $4 * 10^0 + 3 * 10^1 + 2 * 10^2 + 1 * 10^3 = 4 * 1 + 3 * 19 + 2 * 100 + 1 * 1000$

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10:
  - `1234` means
    $4 * 10^0 + 3 * 10^1 + 2 * 10^2 + 1 * 10^3 = 4 * 1 + 3 * 19 + 2 * 100 + 1 * 1000$
  - `10100` means
    $0 * 10^0 + 0 * 10^1 + 1 * 10^2 + 0 * 10^3 + 1 * 10^4 = 1 * 100 + 1 * 10000$

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java:
  - `0b1234` is invalid, since only digits `0` and `1` can occur

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java:
    - `0b1234` is invalid, since only digits `0` and `1` can occur
    - `0b10100` means, in base 10, $0 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 = 1 * 4 + 1 * 16 = 20$

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java
- Hexadecimal numbers are relative to base 16 (using digits $0 \dots 9, a \dots f$) and written in form `0x...` in Java

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java
- Hexadecimal numbers are relative to base 16 (using digits $0\ldots9, a\ldots f$) and written in form `0x...` in Java:
  - `0x1234` means, in base 10, $4*16^0+3*16^1+2*16^2+1*16^3=4+3*16+2*256+1*4096=4660$

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java
- Hexadecimal numbers are relative to base 16 (using digits $0\dots9, a\dots f$) and written in form `0x...` in Java:
  - `0x1234` means, in base 10, $4*16^0+3*16^1+2*16^2+1*16^3 = 4+3*16+2*256+1*4096 = 4660$
  - `0x10100` means, in base 10, $0*16^0+0*16^1+1*16^2+0*16^3+1*16^4 = 1*256+1*65536 = 65792$

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java
- Hexadecimal numbers are relative to base 16 (using digits $0\dots 9, a\dots f$) and written in form `0x...` in Java:
  - `0x1234` means, in base 10, $4*16^0+3*16^1+2*16^2+1*16^3=4+3*16+2*256+1*4096=4660$
  - `0x10100` means, in base 10, $0*16^0+0*16^1+1*16^2+0*16^3+1*16^4=1*256+1*65536=65792$
  - `0xef` means, in base 10, $15*16^0+14*16^1=15+14*16=239$

Excursion Binary and Hexadecimal Numbers

- Numbers are always represented relative to a given *base*
- The least significant/important digit is always the right-most one whereas the one with the highest value is on the left side
- We usually use numbers relative to base 10
- Binary numbers are relative to base 2 and written in the form of `0b...` in Java
- Hexadecimal numbers are relative to base 16 (using digits $0\ldots9, a\ldots f$) and written in form `0x...` in Java:
  - `0x1234` means, in base 10, $4*16^0+3*16^1+2*16^2+1*16^3 = 4+3*16+2*256+1*4096 = 4660$
  - `0x10100` means, in base 10, $0*16^0+0*16^1+1*16^2+0*16^3+1*16^4 = 1*256+1*65536 = 65792$
  - `0xef` means, in base 10, $15*16^0+14*16^1 = 15+14*16 = 239$
- Hexadecimal numbers are heavily used when operations on bits (e.g., `|`, `^`, see Lesson 5: *Operators Expressions*) are performend

Examples for short Variables

Listing: A program allocating, initializing, and printing short variables.

```
/** Examples for short variables */
public class ShortVariables {

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

    var = -1; // set variable var to -1
    System.out.println(var); // prints -1

    var = -32768; // set variable var to -32768
    System.out.println(var); // prints -32768

    var = 3_2767; // set variable var to 32767
    System.out.println(var); // prints 32767

    short hex = 0x10; // set hex to hexadecimal 10, which is 1*16+0 = 16: starts with "0x"
    System.out.println(hex); // prints 16

    short bin = 0b0110_1111; // set bin to binary 01101111, which is 1+2+4+8+32+64=111: starts with
        "0b"
    System.out.println(bin); // prints 111
  }
}
```

Examples for `int` Variables

Listing: A program allocating, initializing, and printing `int` variables.

```java
/** Examples for integer variables */
public class IntVariables {

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

    var = -1; // set variable var to -1
    System.out.println(var); // prints -1

    var = -2147483648; // set variable var to -2147483648
    System.out.println(var); // prints -2147483648

    var = 2_147_483_647; // set variable var to 2147483647
    System.out.println(var); // prints 2147483647

    int hex = 0x10; // set hex to hexadecimal 10, which is 1*16+0 = 16: starts with "0x"
    System.out.println(hex); // prints 16

    int bin = 0b0110_1111; // set bin to binary 01101111, which is 1+2+4+8+32+64=111: starts with "0b"
    System.out.println(bin); // prints 111
  }
}
```

Examples for `long` Variables

Listing: A program allocating, initializing, and printing `long` variables.

```java
/** Examples for long variables */
public class LongVariables {

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

    var = -1L; // set variable var to -1; notice the "L" which marks "long" literals for the compiler
    System.out.println(var); // prints -1

    var = -9223372036854775808L; // set variable var to -9223372036854775808
    System.out.println(var); // prints -9223372036854775808

    var = 9_223_372_036_854_775_807L; // set variable var to 9223372036854775807
    System.out.println(var); // prints 9223372036854775807

    long hex = 0x10L; // set hex to hexadecimal 10, which is 1*16+0 = 16: starts with "0x"
    System.out.println(hex); // prints 16

    long bin = 0b0110_1111L; // set bin to binary 01101111, which is 1+2+4+8+32+64=111: starts with "0b"
    System.out.println(bin); // prints 111
  }
}
```

Examples for `float` Variables

Listing: A program allocating, initializing, and printing float variables.

```java
/** Examples for float variables */
public class FloatVariables {

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

    var = -1f; // set variable var to -1; notice the "f" marking the "float" literal for the compiler
    System.out.println(var); // prints -1.0

    float fraction = 0.8f; // allocate and set variable faction to 0.8
    System.out.println(fraction); // prints 0.8

    var = 1.4e-45f; // set variable var to 1.4*10^{-45}, the "aaaExxx" means "aaa*10^{xxx}"
    System.out.println(var); // prints 1.4E-45

    var = 3.4028235e38f; // set variable var to 3.4028235*10^{38}
    System.out.println(var); // prints 4028235E38

    float pi = 3.141592653589793238462643383279502884197169399375105820974944592307816406286f; // set
        pi to, well, π
    System.out.println(pi); // prints 3.1415927 <- precision of float is about 7 decimals
  }
}
```

Examples for `double` Variables

Listing: A program allocating, initializing, and printing double variables.

```java
/** Examples for double variables */
public class DoubleVariables {

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

    var = -1d; // set variable var to -1; notice the "d" marking the "double" literal for the compiler
    System.out.println(var); // prints -1.0

    double fraction = 0.8d; // allocate and set variable faction to 0.8
    System.out.println(fraction); // prints 0.8

    var = 4.9e-324d; // set variable var to $4.9*10^{-324}$, the "aaaExxx" means "$aaa*10^{xxx}$"
    System.out.println(var); // prints 4.9E-324

    var = 1.7976931348623157e308d; // set variable var to $1.7976931348623157*10^{308}$
    System.out.println(var); // prints 1.7976931348623157E308d

    double pi = 3.14159265358979323846264338327950288419716939937510582097494459230781640628 6d; // set pi to, well, $\pi$
    System.out.println(pi); // prints 3.141592653589793 <- precision of double is about 15 decimals
  }
}
```

Examples for `char` Variables

Listing: A program allocating, initializing, and printing `char` variables.

```java
/** Examples for character variables */
public class CharVariables {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(final String[] args) {
    char var; // allocate character variable "var"

    var = 'T'; // set variable var to 'T'
    System.out.println(var); // prints 'T' (without the primes "'")

    var = '\u597d'; // set variable var to unicode char 0x597d
    System.out.println(var); // prints the Chinese character for "good"

    var = '\n'; // set variable var to literal \n, which stands for newline
    System.out.println(var); // prints a newline, i.e., an empty line

    char space = '␣'; // set space to a space character
    System.out.println(space); // prints ' ' (without the primes "'")

    var = '\''; // setting var to ', using escaped single quote
    System.out.println(var); // prints '
  }
}
```

Examples for `String` Variables

Listing: A program allocating, initializing, and printing `String` variables.

```java
/** Examples for String variables */
public class StringVariables {

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

    var = "Hello␣World!"; // set variable var to "Hello World!" //$NON-NLS-1$
    System.out.println(var); // prints "Hello World!" (without the quotation marks)

    var = "Hello\nWorld!"; // set variable var to "Hello\nWorld!" //$NON-NLS-1$
    System.out.println(var); // prints Hello, newline, World!

    String niHao="\u4f60\u597d"; // set variable var to Hello in Chinese //$NON-NLS-1$
    System.out.println(niHao); // prints a "Ni Hao" in Chinese for Hello

    var = "\"Hello\""; // set variable var to "Hello" (using escaped double quotes)
        //$NON-NLS-1$
    System.out.println(var); // prints a "Hello" (without the quotation marks)
  }
}
```

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.

The limits of the type byte

Listing: The limits of the type byte.

```java
/** Examples for the limits of byte variables */
public class ByteLimits {

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

    var = Byte.MIN_VALUE; // set variable var to the minimum byte value
    System.out.println(var); // prints -128

    var = Byte.MAX_VALUE; // set variable var to the maximum byte value
    System.out.println(var); // prints 127

    var = Byte.SIZE; // set variable var to the size in bits of 1 byte
    System.out.println(var); // prints 8
  }
}
```

The limits of the type short

Listing: The limits of the type short.

```java
/** Examples for the limits of short variables */
public class ShortLimits {

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

    var = Short.MIN_VALUE; // set variable var to the minimum short value
    System.out.println(var); // prints -32768

    var = Short.MAX_VALUE; // set variable var to the maximum byte value
    System.out.println(var); // prints 32767

    var = Short.SIZE; // set variable var to the size in bits of 1 short
    System.out.println(var); // prints 16
  }
}
```

The limits of the type `int`

Listing: The limits of the type `int`.

```java
/** Examples for the limits of integer variables */
public class IntLimits {

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

    var = Integer.MIN_VALUE; // set variable var to the minimum int value
    System.out.println(var); // prints -2147483648

    var = Integer.MAX_VALUE; // set variable var to the maximum int value
    System.out.println(var); // prints 2147483647

    var = Integer.SIZE; // set variable var to the size in bits of 1 int
    System.out.println(var); // prints 32
  }
}
```

The limits of the type `long`

Listing: The limits of the type `long`.

```java
/** Examples for the limits of long variables */
public class LongLimits {

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

    var = Long.MIN_VALUE; // set variable var to the minimum long value
    System.out.println(var); // prints -9223372036854775808

    var = Long.MAX_VALUE; // set variable var to the maximum long value
    System.out.println(var); // prints 9223372036854775807

    var = Long.SIZE; // set variable var to the size in bits of 1 long
    System.out.println(var); // prints 64
  }
}
```

The limits of the type `float`

Listing: The limits of the type `float`.

```java
/** Examples for the limits of float variables */
public class FloatLimits {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(final String[] args) {
    float var; // allocate long variable "var"

    var = Float.MIN_VALUE; // set variable var to the minimum positive! float value
    System.out.println(var); // prints 1.4E-45

    var = Float.MIN_NORMAL; // set variable var to the minimum normal float value
    System.out.println(var); // prints 1.17549435E-38

    var = Float.MAX_VALUE; // set variable var to the maximum float value
    System.out.println(var); // prints 3.4028235E38

    int size = Float.SIZE; // set variable size to the size in bits of 1 float
    System.out.println(size); // prints 32

    var = Float.NEGATIVE_INFINITY; // set variable var to negative infinity
    System.out.println(var); // prints -Infinity

    var = Float.POSITIVE_INFINITY; // set variable var to positive infinity
    System.out.println(var); // prints Infinity

    var = Float.NaN; // set variable var to "not a number"
    System.out.println(var); // prints NaN
  }
}
```

The limits of the type `double`

Listing: The limits of the type double.

```java
/** Examples for the limits of double variables */
public class DoubleLimits {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(final String[] args) {
    double var; // allocate long variable "var"

    var = Double.MIN_VALUE; // set variable var to the minimum positive! double value
    System.out.println(var); // prints 4.9E-324

    var = Double.MIN_NORMAL; // set variable var to the minimum normal double value
    System.out.println(var); // prints 2.2250738585072014E-308

    var = Double.MAX_VALUE; // set variable var to the maximum double value
    System.out.println(var); // prints 1.7976931348623157E308

    int size = Double.SIZE; // set variable size to the size in bits of 1 double
    System.out.println(size); // prints 64

    var = Double.NEGATIVE_INFINITY; // set variable var to negative infinity
    System.out.println(var); // prints -Infinity

    var = Double.POSITIVE_INFINITY; // set variable var to positive infinity
    System.out.println(var); // prints Infinity

    var = Double.NaN; // set variable var to "not a number"
    System.out.println(var); // prints NaN
  }
}
```

Losslessly Compatible Types

- It is possible to store values of a smaller type into a variable of a larger type of the same sort without any loss

Losslessly Compatible Types

- It is possible to store values of a smaller type into a variable of a larger type of the same sort without any loss, i.e.

  1. a `byte` value in a `short` variable
  2. a `short` value in an `int` variable
  3. an `int` value in a `long` variable
  4. a `float` value in a `double` variable

Losslessly Compatible Types Example

Listing: Example for lossless compatible types.

```java
/** Examples for lesslessly compatible variables */
public class CompatibleTypes1 {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(final String[] args) {
    byte myByte = -128; // allocate byte variable "myByte"
    short myShort = myByte; // store value of myByte into variable myShort
    System.out.println(myShort); // prints -128

    myShort = 32767; // now store the maximum short value in the variable
    int myInt = myShort; // and copy the value over to the new myInt variable
    System.out.println(myInt); // prints 32767

    myInt = 2147483647; // set myInt to the maximum integer value
    long myLong = myInt; // and copy it to the new long variable myLong
    System.out.println(myLong); // prints 2147483647

    float myFloat = 1.4E-45f; // new float variable with the smallest positive float value
    double myDouble = myFloat; // copy its value into a double variable
    System.out.println(myDouble); // prints 1.401298464324817E-45, which is equivalent to 1.4E-45f
  }
}
```

Integer-to-Floating Point: Potential Loss of Information

- It is possible to store values of an integer type into a floating point variable, but maybe with loss of information

Integer-to-Floating Point: Potential Loss of Information

- It is possible to store values of an integer type into a floating point variable, but maybe with loss of information, i.e.
  1. a `byte` value in a `float` variable (no loss of infos)
  2. a `short` value in an `float` variable (no loss of infos)
  3. an `int` value in a `float` variable (possible loss of infos)
  4. a `long` value in a `float` variable (possible loss of infos)
  5. a `byte` value in a `double` variable (no loss of infos)
  6. a `short` value in an `double` variable (no loss of infos)
  7. an `int` value in a `double` variable (no loss of infos)
  8. a `long` value in a `double` variable (possible loss of infos)

Conversation of Integer Types to `float`

Listing: Examples for the conversation of Integer Types to `float`.

```java
/** Examples for int-to-float compatibility */
public class CompatibleTypes2Float {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(final String[] args) {
    byte myByte = -128; // allocate byte variable "myByte"
    float myFloat = myByte; // store value of myByte into variable myFloat
    System.out.println(myFloat); // prints -128.0: no loss of infos, 8 bit fit into float mantissa

    short myShort = 32767; // now store the maximum short value in the variable
    myFloat= myShort; // and copy the value over to the new myFloat variable
    System.out.println(myFloat); // prints 32767.0: no loss of infos, 16 bit fit into float mantissa

    int myInt = 2147483646; // set myInt to the maximum integer value -1
    myFloat= myInt; // and copy it to the new long variable myLong
    System.out.println(myFloat); // prints 2.14748365E9, .e., rounding up to 2147483650

    long myLong = 9223372036854775806L; // set myLong to the maximum long value - 1
    myFloat= myLong; // and copy it to the new long variable myLong
    System.out.println(myFloat); // prints 9.223372E18, i.e., rounding down to 9223372000000000000L
  }
}
```

Conversation of Integer Types to `double`

Listing: Examples for the conversation of Integer Types to double.

```java
/** Examples for int-to-double compatibility */
public class CompatibleTypes2Double {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(final String[] args) {
    byte myByte = -128; // allocate byte variable "myByte"
    double myDouble = myByte; // store value of myByte into variable myDouble
    System.out.println(myDouble); // prints -128.0: no loss of infos, 8 bit fit into double mantissa

    short myShort = 32767; // now store the maximum short value in the variable
    myDouble= myShort; // and copy the value over to the new myDouble variable
    System.out.println(myDouble); // prints 32767.0: no loss of infos, 16 bit fit into double mantissa

    int myInt = 2147483646; // set myInt to the maximum integer value -1
    myDouble= myInt; // and copy it to the new long variable myLong
    System.out.println(myDouble); // prints 2.147483646E9: no loss of infos, 32 bit fit into double mantissa

    long myLong = 9223372036854775806L; // set myLong to the maximum long value - 1
    myDouble= myLong; // and copy it to the new long variable myLong
    System.out.println(myDouble); // prints 9.223372036854776E18, i.e., rounding up to 9223372036854776000
  }
}
```

Other Compatibilities

- When doing I/O (seel Lesson 28: *I/O and Streams*, a `char` is sometimes represented as `int` value, which allows us to express that no more character can be read from a file as `-1` . . . but this is subject to another lesson (Lesson 28: *I/O and Streams*)

Other Compatibilities

- When doing I/O (seel Lesson 28: *I/O and Streams*, a `char` is sometimes represented as `int` value, which allows us to express that no more character can be read from a file as `-1` . . . but this is subject to another lesson (Lesson 28: *I/O and Streams*)
- Type casting allows us to transform floating point numbers back to integers via truncation, likely resulting in loss of information . . . and this is also subject of a later lesson (Lesson 20: *Type Casts*)

Other Compatibilities

- When doing I/O (seel Lesson 28: *I/O and Streams*, a `char` is sometimes represented as `int` value, which allows us to express that no more character can be read from a file as `-1` ... but this is subject to another lesson (Lesson 28: *I/O and Streams*)
- Type casting allows us to transform floating point numbers back to integers via truncation, likely resulting in loss of information ... and this is also subject of a later lesson (Lesson 20: *Type Casts*)
- Types can be automatically transformed to `String` when appearing in a `String` expression ... and this is subject to a later lesson as well (Lesson 5: *Operators Expressions*).

Summary

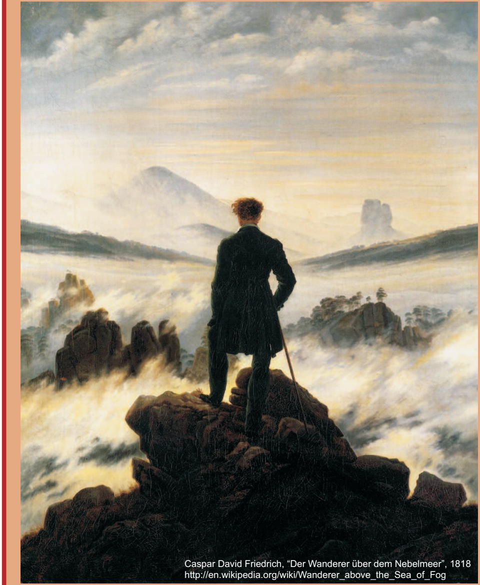
1. We have learned the basic primitive types of Java (plus the non-primitive type `String`).
2. We have learned how to declare variables.
3. We have learned how to store values in variables.
4. We have learned how to print out variables to the console (via `System.out.println(...)`.
5. We have learned that some types are losslessly compatible.
6. We have learned that some conversations are lossy.

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