

Primitive Data Types

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at

<http://www.risc.jku.at>

Primitive Data Types

Each variable/constant has an associated data type.

- Primitive data types are built into the Java Virtual Machine.

`boolean, byte, short, int, long, char, float, double.`

- Other data types are implemented by programmers.

`String.`

A data type consists of a set of values and operations on these values.

Booleans

Data type boolean.

- Two constants:

- true and false.

- Program:

```
boolean b = true;  
System.out.println(b);
```

- Output:

```
true
```

The type of logical values.

Boolean Operations

Logical connectives.

Operator	Description
!	“not” (negation)
&&	“and” (conjunction)
	“or” (disjunction)

- Example:

```
System.out.println(!true);  
System.out.println(true && false);  
System.out.println(true || false);
```

```
false
```

```
false
```

```
true
```

Boolean Operations

- Precedence rules:

- ! binds stronger than &&.
- && binds stronger than ||.
- ((!a) && b) || c can be written as

!a && b || c

- Short-circuited evaluation:

- (false && *any*) is false.
- (true || *any*) is true.
- Second argument is not evaluated, if result already determined by first one.

Short-circuited evaluation will become important later.

Equality and Inequality

Relational operations defined on any data type:

Operator	Description
==	equal to
!=	not equal to

- Program:

```
boolean b = (1 == 0);  
System.out.println(b);
```

- Output:

```
false
```

- Do we need the inequality operator?

We can decide about equality/inequality for any data type.

Integers

Data types `byte`, `short`, `int`, `long`.

- Integral numerical values in a certain range:

Type	Size	Minimum Value	Maximum Value
<code>byte</code>	8 bits	-128	127
<code>short</code>	16 bits	-32768	32767
<code>int</code>	32 bits	-2147483648	2147483647
<code>long</code>	64 bits	-9223372036854775808	9223372036854775807

- Predefined constants:

Type	Minimum Value	Maximum Value
<code>byte</code>	<code>Byte.MIN_VALUE</code>	<code>Byte.MAX_VALUE</code>
<code>short</code>	<code>Short.MIN_VALUE</code>	<code>Short.MAX_VALUE</code>
<code>int</code>	<code>Integer.MIN_VALUE</code>	<code>Integer.MAX_VALUE</code>
<code>long</code>	<code>Long.MIN_VALUE</code>	<code>Long.MAX_VALUE</code>

Arithmetic Operations

- Literals:
 - 0, 9, 47113414.
 - Larger than Integer.MAX_VALUE: 49203281234L.
- Operations:

Operator	Description
+	unary plus and addition
-	unary minus and subtraction
*	multiplication
/	truncated division
%	remainder

- Truncated division: $-17/2 = -8$.
 - Remainder: $a = (a/b) * b + (a\%b)$.
 - What is $-17\%2$?

Arithmetic Operations

- Example:

```
int a = 2; int b = 3;  
a = ((a+b)*7)/a;  
System.out.println(a);
```

17

- Division by zero:

```
int a = 1/0;
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Main.main(Main.java:5)
```

- Usual precedence rules:

- $a*b+a/b$
- $(a*b)+(a/b)$

Overflow

Integer types only represent subset of \mathbb{Z} .

- Sum of two positive `int` values is not necessarily positive:

```
int a = 1500000000;  
int b = 1000000000;  
int c = a+b;  
System.out.println(c);
```

-1794967296

- **Integer overflow** has occurred:

$$\begin{aligned} & (1500000000 + 1000000000) - \text{MAX_VALUE} + (\text{MIN_VALUE} - 1) \\ &= 2500000000 - 2147483647 + (-2147483648 - 1) \\ &= 352516353 - 2147483649 \\ &= -1794967296. \end{aligned}$$

Programmer must ensure that this does not happen.

Widening Conversions

- In arithmetic, byte and short values are converted to int:

```
byte a = 100;  
byte b = 100;  
System.out.println(a+b);
```

200

- Mixing of different integer types possible:

```
byte a = 127;  
short b = 255;  
int c = a*b+1;  
System.out.println(c);
```

32386

byte and short are promoted to int.

Widening Conversions

- In arithmetic with long, int values are converted to long:

```
int a = Integer.MAX_VALUE;  
long b = Integer.MAX_VALUE;  
System.out.println(a+b);
```

4294967294

- In an assignment, the smaller type is promoted to the larger type:

```
byte a = 127;  
short b = a;  
int c = b;  
long d = c;  
System.out.println(d);
```

127

Widening conversions are automatically performed.

Narrowing Conversions

- Narrowing conversions are not automatically performed:

```
int a = 127;  
short b = a;
```

```
Main.java:6: possible loss of precision  
found    : int  
required: short
```

- Problem with widening conversions:

```
short a = 1;  
short b = a+1;
```

```
Main.java:6: possible loss of precision  
found    : int  
required: short
```

Narrowing Conversions

- A **type cast** is required to convert to a narrower datatype.

- *(type) expression*
- value of *expression* is converted to *type*.

```
int a = 127;  
short b = (short)a;
```

- Type cast binds stronger than arithmetic operations:

```
short a = 1;  
short b = (short)(a+1);
```

- `(short)a+1` would be useless.

Narrowing conversions require an explicit type cast.

Relational Operations

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Arithmetic promotion is applied:

```
short i = -1;
int j = 1;
boolean b = (i <= j);
System.out.println(b);
```

true

- Does the following work?

```
boolean b = (-2 <= 1) || (1/0 == 1);
```

Characters

Data type char.

- The Unicode characters.
 - About 2^{16} characters, two bytes of memory.
- Character literals: ' *char* '

– Program:

```
char letter = 'a';  
char newline = '\n';  
char backslash = '\\';  
System.out.print(letter);  
System.out.print(newline);  
System.out.println(backslash);
```

– Output:

```
a  
\
```

Special Characters

- Already known escape sequences:

- `\", \\, \n, \t`

- Further escape sequences:

- Single quote: `\'`

- Hexadecimal Unicode value: `\unnnn`

- Program:

```
char letter = '\u0041';  
System.out.println(letter);
```

- Output:

A

Characters as Integers

char is an integer data type smaller than int.

- Number interpreted as Unicode value.

```
char letter = 65;  
System.out.println(letter);
```

A

- Usage like any other integer type (e.g., short).

```
boolean b = ('A' <= 'B')  
System.out.println(b);
```

true

- Promotion and type cast:

```
int i = 'A'+1;  
System.out.println((char)i);
```

B

Ordering Properties

- Upper-case letters are densely ordered:
 - 'A', 'B', ..., 'Z'.
- Lower-case letters are densely ordered:
 - 'a', 'b', ..., 'z'.
- Decimal digits are densely ordered:
 - '0', '1', ..., '9'.
- Example:
 - How to determine whether a character is a letter?

Look up Unicode table to find ordering properties.

Floating Point Numbers

Data types `float` and `double`.

- Fractional numerical values: -1.5 , 0.0 , 3.1415927 .
 - Only **approximated** (no exact representation).
 - **Rounding errors** may occur in computations: $1.5 + 1.5 \neq 3.0$
- Floating point:
 - No fixed number of digits after decimal point.
 - Values close to zero are approximated with higher accuracy.

Type	Size	Minimum Value	Maximum Value	Significant digits
<code>float</code>	32 bits	ca. $-3.4 * 10^{38}$	ca. $3.4 * 10^{38}$	7
<code>double</code>	64 bits	ca. $-1.7 * 10^{308}$	ca. $3.4 * 10^{308}$	15

`double` represents larger values and with greater accuracy.

Floating Point Literals

Literals: at least one digit after the decimal point.

- By default, literals are of type `double`.

```
double g = 2.718281828459;
```

- Append `F`, to denote literal of type `float`.

```
float f = 3.1415927F;
```

Floating Point Operations

- Printing:

```
float f = 1.5F;  
System.out.println(f);
```

1.5

- Arithmetic:

Operator	Description
+	unary plus and addition
-	unary minus and subtraction
*	multiplication
/	division
%	remainder

$$a = (\text{int})(a/b) * b + (a\%b).$$

Special Floating Point Values

- “Infinity”

- Result of a computation is too big:

```
double f = 2.718281828459E300;  
double g = f*f;  
System.out.println(g);
```

Infinity

- Division by zero:

```
System.out.println(-1.0/0.0)
```

-Infinity

- “Not a Number”

```
System.out.println(0.0/0.0);
```

NaN

Conversions

- float is promoted to double.
 - Mixed arithmetic and assignments.
- Integer values are promoted to float values:

```
float f = 1.5F;  
long l = 1;  
System.out.println(f+l);
```

2.5

- Narrowing conversion requires type cast:

```
double f = 2.718281828459;  
float g = (float)f;  
int h = (int)f;
```

2.718281828459

2.7182817

2

Pitfalls

Floating point arithmetic is tricky.

```
float f = 3.141592F;  
float g = 3.141593F;  
System.out.println(f*f*f*f*f*f*f*f*f*f*f*f*f*f*f*f*g);  
System.out.println(g*f*f*f*f*f*f*f*f*f*f*f*f*f*f*f*f);  
  
9122149.0  
9122147.0
```

- Associativity does not hold $(a * (b * c)) \neq (a * b) * c$.
- Small rounding errors can accumulate to large overall errors.

Use floating points only when you need them and use them with care.

Relational Operations

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Example:

```
int i = -1;
double d = 1.5;
boolean b = (i <= d);
System.out.println(b);
```

true

- Exact comparisons do not make sense:

- Wrong: $f == g$
- Better: $\text{Math.abs}(f-g) < e$

Strings

Sequences of characters represented by Unicode numbers.

```
String letter = "a";  
String empty = "";  
String word = "hello";  
String sentence = "The ball is red.";  
System.out.println(letter);  
System.out.println(empty);  
System.out.println(word);  
System.out.println(sentence);
```

Output:

a

hello

The ball is red.

String Literals

- String literals are delimited by double quotes.
- The double quote has to be escaped in the literal.

```
String doubleQuote = "\"";  
System.out.println(doubleQuote);
```

Output:

"

While character literals are delimited by single quotes, string literals are delimited by double quotes.

Unicode Codes

Unicode values may be used to denote characters in literals:

```
String letter = "\u0041";  
System.out.println(letter);
```

Output:

A

Useful to denote characters that are not available on keyboard.

String Operations

- String concatenation: $s1+s2$

```
String s1 = "abc"; String s2 = "def";  
String s = s1+s2;  
System.out.println(s);
```

Output:

```
abcdef
```

- String length: $s.length()$

```
String s = "abc";  
int n = s.length();  
System.out.println(n);
```

Output:

```
3
```

String Operations

- Character extraction: `s.charAt(i)`

```
String s = "abcd"; char ch = s.charAt(2);  
System.out.println(ch);
```

Output:

c

- Unicode value extraction: `s.codePointAt(i)`

```
String s = "abcd"; int c = s.codePointAt(2);  
System.out.println(c);
```

Output:

99

Index *i* must be less than `s.length()`.

String Operations

- Substring extraction: `s.substring(i,j)` and `s.substring(i)`

```
String s = "abcdefg";  
String t1 = s.substring(2,5);  
String t2 = s.substring(2);  
System.out.println(t1);  
System.out.println(t2);
```

Output:

```
cde  
cdefg
```

Index *i* denotes the position of the first character of the substring;
index *j* denotes the position of the first character that is not in the
substring any more.

String Operations

- String search: `s1.indexOf(s2)` and `s1.lastIndexOf(s2)`

```
String s = "abcabc";  
System.out.println(s.indexOf("bc") + " " + s.lastIndexOf("bc"));
```

Output:

1 4

- Generalization: `s1.indexOf(s2, i)` & `s1.lastIndexOf(s2, i)`

```
String s = "abcabc";  
System.out.println(s.indexOf("bc", 2) + " " + s.lastIndexOf("bc", 3));
```

Output:

4 1

The optional index *i* denotes the position where the search starts.

String Comparison

- Do not compare strings by `==` and `!=`
 - `"abcdef" == "abc" + "def"` may **not** be true.
- String comparison: `s1.equals(s2)`
 - Returns true if `s1` and `s2` have the same length and the same character in every position.

```
String s = "abcdef";  
System.out.println(s.equals("abc" + "def"));
```

Output:

```
true
```

Use `equals` to compare strings!

Lexicographical String Comparison

- `s1.compareTo(s2)` and `s1.compareToIgnoreCase(s2)`
 - Result is zero, if `s1` and `s2` are equal, negative, if `s1` is before `s2`, positive, if `s1` is after `s2`.
 - First difference from the left determines the order between `s1` and `s2`.

```
String s = "AB";  
System.out.println(s.compareTo("B") < 0);  
System.out.println(s.compareTo("AC") < 0);  
System.out.println(s.compareTo("ABC") < 0);
```

Output:

```
true  
true  
true
```

The well-known “phone book ordering”.

Basic Input

No simple input for basic data types in Java.

- Own input methods: `kwm.jar (Input.README)`

Method	Description
<code>Input.readBoolean()</code>	reads boolean value
<code>Input.readByte()</code>	reads byte value
<code>Input.readShort()</code>	reads short value
<code>Input.readInt()</code>	reads int value
<code>Input.readLong()</code>	reads long value
<code>Input.readChar()</code>	reads char value
<code>Input.readFloat()</code>	reads float value
<code>Input.readDouble()</code>	reads double value
<code>Input.readString()</code>	reads String value

- Reads one line from standard input.
 - Converts it to the specified data and returns the result.

Input Example

```
System.out.print("Enter first float: ");  
float f1 = Input.readFloat();  
System.out.print("Enter second float: ");  
float f2 = Input.readFloat();  
System.out.print("The product of " + f1 + " and " + f2 + " is ");  
System.out.println(f1*f2);
```

Enter first float: 2.7

Enter second float: -1.4

The product of 2.7 and -1.4 is -3.78

What happens, if the input has wrong format?

Error Handling

- Control methods:

Method	Description
<code>Input.isOkay()</code>	was last input okay?
<code>Input.hasEnded()</code>	has input stream ended?
<code>Input.getError()</code>	get message of last error
<code>Input.clearError()</code>	clear error flag and message

- Check after input, whether result is okay.

- If yes, use the returned value.
- If not, input has either ended or error has occurred.
- If error, get error message and clear error.

Details will be shown later.