Language Fundamentals Summary

Claudia Niederée, Joachim W. Schmidt, Michael Skusa
Software Systems Institute

Object-oriented Analysis and Design 1999/2000

c.niederee@tu-harburg.de
http://www.sts.tu-harburg.de

Keywords

- carry fixed semantics
- must not be used as identifiers elsewhere in the program

abstract	**default**	**if**	**private**	**throw**
boolean	**do**	**implements**	**protected**	**throws**
break	**double**	**import**	**public**	**transient**
byte	**else**	**instanceof**	**return**	**try**
case	**extends**	**int**	**short**	**void**
catch	**final**	**interface**	**static**	**volatile**
	char	**finally**	**long**	**super**
	while			
class	**float**	**native**	**switch**	
const*	**for**	**new**	**synchronized**	
continue	**goto***	**package**	**this**	

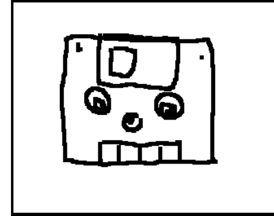
Comments

- /* Simple comments are enclosed like this */
- // Are the start of a one-line-comment
- /** Documentation comments can also be included */
- All lines included start with **
- Documentation comments are evaluated by the documentation tool of the JDK.
- Results in HTML pages.
- Special tags within the comments for information often included lead to special formatting in HTML:
  - @param
  - @return
  - @exception
  - @author
  - @see
  - @version

Documentation Comments

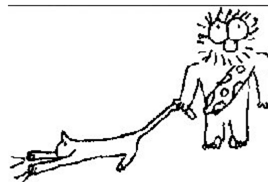
```
/**
 * Compares two objects for <B>equality</B>.
 * Returns a boolean that indicates whether this
  Object
 * is equivalent to the specified Object.
 * @param   obj   the object to compare with
 * @return  true if these Objects are equal;
 *                false otherwise.
 */
public boolean equals(Object obj) { ...
```

Data



- In Java, almost everything is an object.
- Objects are described by reference types like the type String, classes or arrays
- Some simple data types, however, are frequently used as programming basis.
- These so-called primitive types are built-in types, implying a behaviour different from reference types

Primitives



- Built-in types: not object handles, but variables on the stack like C.

  boolean, char (Unicode), byte, short, int, long, float, double
- Consequences for bindings and comparisons.
- Size of each data type is machine independent.

Binding and Comparing Primitives

- Binding a variable to a primitive value means copying this value into the variable's storage cell.

```
int a = 100;
int b = 75;
```

a 100 b 75

- Assigning a new value to an existing variable means replacing the original value by a copy of the new value.

```
a = b;
```

a 75 b 75

- A comparison is done by comparing the bits of the copied values.

```
a == b;
```


Whole Numbers

- Primitive types for whole numbers are byte (8 bits), short (16 bits), int (32 bits), long (64 bits)
- Literals can be expressed in decimal, hexadecimal or octal digits.
- Octals start with '0', hexadecimals with '0X' or '0x'.
- A whole number is of type int unless the suffix 'L' or 'l' for long is added.

int (decimal)	**int** (octal)	**int** (hexadecimal)	**long** (decimal)	**long** (octal)	**long** (hexadecimal)
0 2	0372	0xDada	1l	0777L	0XC0B0L

Operations on Whole Numbers

- Comparison: ==, !=, <, >, <=, >=
- Unary: +, -
- Multiplication, etc: *, /, %
- Addition, etc.: +, -
- Increment and decrement: ++, --
  (Side effects!)
- Shifts: <<, >>, >>>
- Complement: ~
- Bitwise operations: &, |, ^

Scope of Whole Number Values

Type	Range
byte	-128 to 128
short	-32768 to 32767
int	-2147483648 to 2147483647
long	-9223372036854775808 to 9223372036854775808

Exceptions on Whole Numbers

- Neither overflow nor underflow is tested for whole numbers.
- 0 as second operand of / or % throws an ArithmeticException.

```
int i = 1000000;
System.out.println(i*i);
long l = i;
System.out.println(l*l);
System.out.println(20296/(l-i));
```

-727379968

1000000000000

Exception

Floating-Point Numbers

- Primitive types for floating-point numbers are float (32 bits) and double (64 bits)
- A floating-point literal has the following components:
  - a whole-numbered part, at least one digit,
  - a decimal point,
  - a decimal part, at least one digit,
  - an exponent, starting with the letter 'E' or 'e'
  - a type identification, i.e. the letter 'F' or 'f' for float or 'D' or 'd' for double.
- At least one of these!

float: 2.f .3F 0f 7.38473e+18f

double: 2. .3d 1e2 0.0

Order on Floating-Point Numbers

- Floating-point numbers include special values:
  - POSITIVE_INFINITY, NEGATIVE_INFINITY
  - 0.0, -0.0
  - NaN (Not-a-Number)
- Infinity-values instead of overflow
- NaN is the result of invalid operations.

```
0.0 == -0.0                  true
0.0 > -0.0                   false
Math.max(0.0, -0.0)          0.0
Double.NaN == 5.6            false
Double.NaN < 3.1415          false
Double.NaN != 9347.234       true
```


Operations on Floating-Point Numbers

- Comparison: ==, !=, <, >, <=, >=
- Unary: +, -
- Multiplication, etc: *, /, %
- Addition, etc.: +, -
- Increment and decrement: ++, --
  (Side effects!)

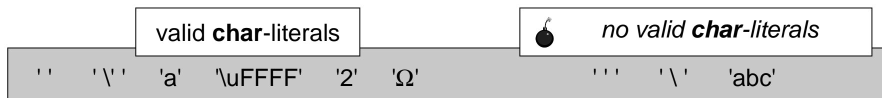
Unicode Characters

- The primitive types for characters is char (16 bits Unicode)
- A literal is formed by a single character or escape character included in single quotes.
- Escape characters are used for characters that cannot represented in any other way:

\ b	**/* backspace */**	**\ r**	**/* carriage retrun */**
\ t	**/* horizontal tab */**	**\ "**	**/* double quote " */**
\ n	**/* line feed */**	**\ '**	**/* simple quote' */**
\ f	**/* form feed */**	**\ **	**/* backslash\ */**

valid **char**-literals

*no valid **char**-literals*



Operations on Characters

- Comparison: ==, !=, <, >, <=, >=
- Unary: +, -
- Multiplication, etc: *, /, %
- Addition, etc.: +, -
- Increment and decrement: ++, -- (Side effects!)
- Shifts: <<, >>, >>>
- Complement: ~
- Bitwise operations: &, |, ^

Careful! **char**s are often converted to **int**s!

Booleans

- The primitive type used to express that something can have exactly two different states is called boolean.
- The two literals are true and false.
- Operations on booleans:
  - Comparison: ==, !=
  - Negation: !
  - binary logic: &, |, ^
  - short-circuit-evaluation logic: &&, ||

Short-Circuit-Evaluation

- The evaluation of a short-circuit-expression is a partial evaluation from left to right.
- As soon as it can be reassured, that the result has a certain value, the evaluation is stopped.
- Quite useful in some situations.

If x == 0, the result is *false*

x!=0 && 1000/x > 100

If x == 0, division by zero !!

x!=0 & 1000/x > 100

Ternary if-else

```
int a,b;
...
b = a >= 0 ? a : -a;
```

- Does the same as

```
int a,b;
...
if (a >= 0) b = a; else b = -a;
```

- A ternary if-else always results in a value
- Can also be used for side-effects
- Shouldn't be used too often for the sake of readable code

Controlling Program Flow

- Variable declaration and assignment
- Using operations on primitive data types
- Iteration
- Conditionals

Statements

- There are two general kinds of statement
  - simple statements
  - compound statements
- Simple statements end with a semicolon.
- Compound statements encompass several statements and are enclosed in brackets.
- Compound statements are also called blocks.
- Blocks are important for scoping.

Variables

- Variables have to be declared before they are used.

```
int a;
```

- They can be assigned changing values according to their declared type.

```
a = 3    a= 5 + 2;     a = "Oh no!";
```

- Declaration and assignment can be done within a single statement.

```
int b = 8;
```

- Some operations involve implicit assignments

```
b = ++a;
```

a 6 b 6

- Variables that are used but not initialized cause compile-time errors.

Scoping

```
{ /* <-- Beginning of scope 1 */
  int x = 12;
  /* Only x available */
  { /* <-- Beginning of scope 2 */
     int q = 96;
     /* Cannot redefine x here! */
     /* Both x & q available */
  } /* <-- End of scope 2 */
  /* Only x available */
  /* q out of scope */
} /* <-- End of scope 1 */
```


if-else

```
int a;
...
if (a >= 0)
  System.out.println("Already positive");
else {
 a = -a;
 System.out.println("Turned positive");
}
```

- The statements can either be simple (terminated by semicolon) or compound in braces.
- else is optional

switch

```
char c;
...
switch(c) {
  case 'a':
  case 'o':
  case 'u':
              System.out.println("dark vowel");
              break;
  case 'e':
  case 'i':
              System.out.println("bright vowel");
              break;
  default:
              System.out.println("no vowel");
}
```


What does switch do?

- Tests all case-conditions and executes the statement behind the colon, if the condition is true.
- If a condition is true and the attached statement is followed by a **break**, switch is finished.
- If no match occurs, the **default**-statement is executed.
- switch only works on integral values like **int** or **char**.

for

```
int [] a = ...;
int sum = 0;

for (int i = 0; i < 10; i++) {
  sum = sum + a[i];
}
```

- performs intialization before first iteration, conditional testing at the beginning of each iteration and some form of stepping at the end of each iteration
- all three parts are optional

while

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  ++i;
}
```

- Tests a condition and executes the loop-statements as long as the condition is true.

do-while

```
int i = 0;
do {
  System.out.println(i);
  ++i;}
while (i < 10)
```

- Executes the loop-statements until the condition is false.
- Mind the difference with while: The statements are at least executed once here, even if the condition is never true.