Java Quick Reference

Comments

// Everything to the end of the line is ignored. Use for most comments.
/* Everything (possibly many lines) is ignored until a */. Uncommon. Use for commenting out code.
/** Used for automatic HTML documentation generation. */

Identifier Names

- Start identifiers with an alphabetic character (a-z or A-Z), and continue with alphabetic, numeric (0-9), or '_' (underscore) characters. Do not use $.
- Second words in a name should start with an uppercase letter.
- Do not use special Java keywords.
- Class and interface names should start with an uppercase letter (Graphics, String, Car, Motorbike...).
- Variable and method names should start with a lowercase letter (repaint(), x, ...).
- Constants should be all uppercase with underscores between words (BoxLayout.X_AXIS, Math.PI, ...).

Variables - Local, Instance, Class

Variables may be *local*, *instance*, or *static* (class) variables.
Parameters are local variables that are assigned values when the method is called.

	local	**instance**	**static**
Declared where?	In a method.	In class, but not in a method.	In class, using *static* keyword.
Initial value	Assign a value before using. Compiler error if you don't.	Number: zero Object: null Boolean: false *Or initialized in constructor.*	Number: zero Object: null Boolean: false *Or initialized in static initializer.*
Visibility	Only in the same method. No visibility may be declared.	`private`: Only methods in this class. *(Default)*: All methods in same package. `public`: Anyone can see it. `protected`: This class and all subclasses can see it.	Same as instance variables.
Created when?	When the method is entered.	When an instance of the class (object) is created with `new`.	When the program is loaded.
Where in memory?	Call stack.	Heap.	"Permanent" memory.
Destroyed when?	When the method returns.	When there are no more references to the object.	When the program terminates.

Primitive Types

boolean (true/false)
Arithmetic types: byte, short, char, int, long, float, double

Expressions

Parentheses () have three uses:

1. Grouping to control order of evaluation, or for clarity.
   Eg, (a + b) * (c - d)
2. After a method name to enclose parameters.
   Eg, x = sum(a, b);
3. Around a type name to form a *cast*. Eg, i = (int)x;

Operator Precedence

1. Higher precedence are done before lower precedence. 2. Left to right among equal precedence except: unary, assignment, conditional operators.	Remember only 1. unary operators 2. * / % 3. + - 4. *comparisons* 5. && || 6. = *assignments* Use ( ) for all others

Arithmetic Operators

The result of arithmetic operators is double if either operand is double, else float if either operand is float, else long if either operand is long, else int.

`i++`	Add 1 to i
`i--`	Subtract 1 to i
`n + m`	Addition. Eg 7+5 is 12, 3 + 0.14 is 3.14
`n - m`	Subtraction
`n * m`	Multiplication. Eg 3 * 6 is 18
`n / m`	Division. Eg 3.0 / 2 is 1.5 , 3 / 2 is 1
`n % m`	Remainder (Mod) after division of n by m. Eg 7 % 3 is 1

Comparing Primitive Values

The result of all comparisons is boolean (true or false).
==, !=, <, <=, >, >=

Logical Operators

The operands must be boolean. The result is boolean.

`b && c`	"And". *true* if both operands are *true*, otherwise *false*. Short circuit evaluation. *Eg* (*false* && anything) is *false*.
`b || c`	"Or". *true* if either operand is *true*, otherwise *false*. Short circuit evaluation. *Eg* (*true* || anything) is *true*.
`!b`	"Not". *true* if b is *false*, *false* if b is *true*.

Assignment Operators

`=`	Left-hand-side must be an identifier/variable.
`+= -= *= ...`	All binary operators (except && and ||) can be combined with assignment. *Eg* `a += 1` is the same as `a = a + 1`

Casts

Use casts when "narrowing" the range of a value. From narrowest to widest the primitive types are: byte, short, char, int, long, float, double. Objects can be assigned without casting up the inheritance hierarchy. Casting is required to move down the inheritance hierarchy (downcasting).

`(t)x`	Casts x to type t

Object Operators

`co.f`	Member. The `f` field or method of object or class `co`.
`x instanceof co`	*true* if the object `x` is an instance of class `co`.
`s + t`	String concatenation if one or both operands are Strings.
`x == y`	*true* if `x` and `y` refer to the same object, otherwise *false* (even if the values of the objects are the same!).
`x != y`	As above for inequality.

Note: Compare object instances with `.equals()` or `.compareTo()`

`x = y`	Assignment copies the *reference*, not the *object*.

Flow of Control

if Statement

```
//----- if statement with a true clause
   if (expression) {
       statements // do these if expression is true
   }

//----- if statement with true and false clause
   if (expression) {
       statements // do these if expression is true
   } else {
       statements // do these if expression is false
   }

//----- if statements with many parallel tests
   if (expression1) {
       statements // do these if expression1 is true
   } else if (expression2) {
       statements // do these if expression2 is true
   } else if (expression3) {
       statements // do these if expression3 is true
   . . .
   } else {
       statements // do these no expression was true
   }
```

switch Statement

switch chooses one case depending on an integer value.

```
switch (expr) {
   case c1:
         statements // do these if expr == c1
         break;
   case c2:
         statements // do these if expr == c2
         break;
   case c3:
   case c4:
   case c5:  //  Cases can simply fall through.
        // do these if expr ==  any of c3, c4 or c5
         statements
         break;
   . . .
   default:
         statements // do these if expr != any above
}
```

while Loop

```
while (expression) {
   // do these continuously if expression == true
   statements
}
```

for Loop

```
for (initialStmt; testExpr; incrementStmt) {
   // do these continuously if testExpr == true
   statements
}
```

while and for can be *almost* equivalent:

```
int i = 0;
while ( i < 5 ) {
   System.out.print("Hi!");
   i++;
}
```

```
for (i = 0; i < 5; i++) {
   System.out.print("Hi!");
}
```

Other loop controls

All loop statements can be labeled, so that break and continue can be used from any nesting depth.

```
   break;        //exit innermost loop or switch
   break label; //exit from loop label
   continue;     //start next loop iteration
   continue label; //start next loop label
```

Put label followed by colon at front of loop, like this:

```
outer: for (. . .) {
         . . .
         continue outer;
}
```

Exceptions

Simple try...catch for exceptions

```
   try {
       . . . // statements that might cause exceptions
   } catch (exception-type x) {
       . . . // statements to handle exception
   }
```

throw

```
   throw exception-object;
```

Multiple catch clauses and finally clause

Executes first catch clause that specifies the exception class or a super class. The finally clause is always executed (regardless of whether there was an exception or not) so resources can be cleaned up (for example, closing a file):

```
try {
   . . . // statements that might cause exceptions
} catch (exception-type x) {
   . . . // statements to handle exception
} catch (exception-type x) {
   . . . // statements to handle exception
} finally (exception-type x) {
   // statements that will always be executed
   // exception or not.
   . . .
}
```

Strings

String Concatenation

The + operator joins two strings together. If either operand is String, the other is converted to String and concatenated with it. This is a common way to convert numbers to Strings.
If a non-String object is concatenated with a String, its *toString()* method is called. It's useful for debugging to write your own *toString()* method in your classes.

`"abc" + "def"`	`"abcdef"`
`"abc" + 4`	`"abc4"`
`"1" + 2`	`"12"`
`"xyz" + (2+2 == 4)`	`"xyztrue"`
`1 + "2.5"`	`"12.5"`

i = s.length()	length of the string s.

String Comparison (use these instead of == and !=)

i = s.compareTo(t) compares to s.	returns <0 if s<t, 0 if s==t, >0 if s>t
i = s.compareToIgnoreCase(t)	same as above, but upper and lower case are same
b = s.equals(t)	true if the two strings have equal values
b = s.equalsIgnoreCase(t)	same as above ignoring case
b = s.startsWith(t)	true if s starts with t
b = s.endsWith(t)	true if s ends with t

Searching (all "indexOf" methods return -1 if not found)

i = s.indexOf(t)	index of the first occurrence of String t in s.
i = s.indexOf(t, i)	index of String t at or after position i in s.
i = s.lastIndexOf(t)	index of last occurrence of t in s.
i = s.lastIndexOf(t, i)	index of last occurrence of t on or before i.

Strings - Getting parts

c = s.charAt(i)	char at position i in s.
s1= s.substring(i)	substring from index i to the end of s.
s1= s.substring(i, j)	substring from index i to BEFORE index j.

Strings - Creating a new string from the original

s1= s.toLowerCase()	new String with all chars lowercase
s1= s.toUpperCase()	new String with all chars uppercase
s1= s.trim()	with whitespace deleted from front and back
s1= s.replace(cs2, cs3)	with all cs2 substrings replaced by cs3

StringBuilder

Faster String modification, more memory and CPU efficient.

sb = new StringBuilder()	Creates empty StringBuilder
sb = new StringBuilder(s)	Creates StringBuilder with String s.
sb = sb.append(x)	Appends x (any type) to end of sb.
sb = sb.insert(offset, x)	Inserts x (any type) at position offset.
sb = sb.setCharAt(index, c)	Replaces char at index with c
sb = sb.deleteCharAt(i)	Deletes char at index i.
sb = sb.delete(beg, end)	Deletes chars at index beg to end.
sb = sb.reverse()	Reverses the contents.
sb = sb.replace(beg, end, s)	Replace chars beg to end with s.

indexOf, lastIndexOf, charAt, equals, substring just like String!

Arrays

To use and manipulate many data elements, either primitives or objects. All elements must be of the same type. Arrays don't expand!

Examples:

```
int[] scores;  // Declares scores as array of integers.
scores = new int[12]; // Allocate memory, 12 values.
int[] scores = new int[12]; // Combined in one line.
```

Initialize an array

If no initial values are specified for array elements, array elements are initialized to zero for numbers, null for object references, and false for booleans.
Create and initialize in one line:

```
String[] names = {"Mickey", "Minnie", "Donald"};
```

Or in several lines:

```
String[] names = new String[3];
names[0] = "Mickey";
names[1] = "Minnie";
names[2] = "Donald";
```

Accessing elements of the array

```
scores[5] = 86; // Assign value 86 to the 5th value.
scores[i]++;  // Increment score of element number i.
```

Iterating over an array

Size of an array can be found using *length*, eg, scores.length.

```
// Using standard for loop.
int[] scores = new int[12];
```
...Set values in scores array.
```
int total = 0;
for (int i = 0; i <
scores.length; i++) {
    total += scores[i];
}
```

```
//Using enhanced for loop.
int[] scores = new int[12];
```
. . . Set values in the scores array.
```
int total = 0;
for (int scr : scores) {
    total += scr;
}
```

Two-dimensional arrays

Almost always processed with nested for loops. Example:

```
static final int ROWS = 2;
static final int COLS = 4;
. . .
int[][] a2 = new int[ROWS][COLS];
. . .
//... Print array in rectangular form
for (int i =0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        System.out.print(" " + a2[i][j]);
    }
    System.out.println("");
}
```

Scanner

The main use of java.util.Scanner is to read values from System.in or a file.

sc = new Scanner(System.in);	Scanner which reads from System.in.
sc = new Scanner(s);	Scanner which reads from String s.

Most common "next" input methods.

s = sc.next()	Returns next "token", more or less a "word".
s = sc.nextLine()	Returns an entire input line as a String.
x = sc.nextXYZ()	Returns value of type XYZ: Int, Double, Boolean, Byte, Float, Short.
b = sc.hasNext()	True if another token is available to be read.
b = sc.hasNextLine()	True if another line is available to be read.
b = sc.hasNextXYZ()	True if another XYZ is available to be read.

Text File Input / Output

Example:

```
public static void copyFile(File fromFile, File
                           toFile) throws IOException {
   Scanner freader = new Scanner(fromFile);
   BufferedWriter writer = new BufferedWriter(
                                 new FileWriter(toFile));

   //... Loop as long as there are input lines.
   String line = null;
   while (freader.hasNextLine()) {
      line = freader.nextLine();
      writer.write(line);
      writer.newLine();   // Write end of line.
   }

   //... Close reader and writer.
   freader.close();  // Close to unlock.
   writer.close(); // Close to unlock & flush to disk.
}
```