Java: Types, Declarations, Operators, Control Structures, . . .

The boring but necessary basics

Jonas Kvarnström

1 Introduction

This course is intended to help you learn both a *programming paradigm* and a *programming language*. A large part of the course concerns learning high-level and intermediate-level *concepts* and *connections* between those concepts. But to be able to create programs in practice you also need to know quite a few basic low-level details. Here is a reference for these details, that you can use in addition to the lecture slides and the course book you have chosen.

This is not intended to be a self-contained introductory Java text. Instead it is intended for those who take TDDD78, to be used in conjunction with the information from the initial lectures.

Even if you have used Java before, we recommend that everyone read this text from the beginning to the end. This should not take very long, and there are probably some details here that you have forgotten or never learned.

2 Java without OO

This section covers several non-OO aspects of Java: Primitive datatypes, variables, declaration and scoping, operators, expressions, control statements, and so on.

The following is a simple Java program called JavaTest, demonstrating a few statements and control structures. The `main()` method is called when the program starts. It must be `public static void` and must take one argument of type `String[]`.

```
public class JavaTest {
    public static void main(String[] args) {
        System.out.println("Hello, world");
        int i = 1;
        while (i < 1024) {
            System.out.println("i = " + i);
            i = i * 2;
        }
    }
}
```

2.1 Primitive Datatypes

Java provides the following eight primitive datatypes. Arrays are not primitive data types. Strings are not primitive data types and not arrays of characters.

Type	**Bits**	**Values**		
`boolean`		`true or false`		
`byte`	8	`-128`	`to`	`127`
`short`	16	`-32768`	`to`	`32767`
`int`	32	`-2147483648`	`to`	`2147483647`
`long`	64	`-9223372036854775808L`	`to`	`9223372036854775807L`
`char`	16	`0`	`to`	`65535`
`float`	32	`±3.40282347E-45f`	`to`	`±3.40282347E+38f`
`double`	64	`±4.9406564584124654E-324`	`to`	`±1.79769313486231570E+308`

2.1.1 Integer Literals

A *literal* is simply an expression denoting a fixed value. Unless you specify otherwise, *integer* literals are interpreted as values of type `int`. You can use the prefix `0x` to specify a hexadecimal value or the prefix `0` to specify an octal value. Be careful with the `0` prefix: Adding leading zeros to make columns line up is not a good idea (`0020 == 16`).

Value	**Resulting type**
`0`	int
`123456789012345`	not allowed, will not fit in an int
`123456789012345L`	add an 'L' to specify a long value
`(short) 125`	short
`(byte) 125`	byte
`0xCAFEBABE`	hexadecimal
`0777`	octal – *be careful!*
`0777L`	octal long

2.1.2 Floating Point Literals

Unless you specify otherwise, floating point literals are interpreted as values of type `double`. You can use the postfix `d` or `D` to specify explicitly that you want a double-precision value, or you can use the postfix `f` or `F` to specify a single-precision `float` value. You can also use exponential notation.

Value	**Resulting type**
`0.0`	double
`0.0d`	double
`0.0D`	double
`0.0f`	float
`0.0F`	float
`1.7e-12`	double (value is $1.7 * 10^{-12}$)
`1.7e-12f`	float (value is $1.7 * 10^{-12}$)

2.1.3 Character Literals

Character literals are usually written within single quotes. However, Java also exposes the internal numeric representation to the programmer, so you can also write a character literal numerically using its Unicode position. Characters 0 to 127 are identical to plain ASCII.

Value	**Result**
`char c1 = 'x';`	Small letter x
`char c2 = 120;`	Small letter x (ASCII code 120)
`char c3 = '7';`	The digit '7'
`char c4 = 7;`	The ASCII control code BEL (ctrl-G)
`char c5 = '∀';`	For all (Unicode 0x2200)
`char c6 = '⋉';`	Left normal factor semidirect product (Unicode 0x22C9)
`char c7 = c1+2;`	Small letter z (ASCII code 122)

You can also use escape codes to represent unprintable characters:

`'\b'`: Backspace
`'\t'`: Tab
`'\n'`: Newline
`'\f'`: Form feed
`'\r'`: Carriage return
`'\''`: Single quote (quote backslash quote quote)
`'\\'`: A backslash
`'\nnn'`: The character with octal value nnn (`'\012'` == `'\n'`)

You can tell the `javac` compiler which encoding is used for your `.java` source files using the `-encoding` switch. There is also a special Unicode escape sequence that permits you to enter arbitrary Unicode characters even in plain ASCII: `'\u01AB'` is Unicode character number `01AB` (hexadecimal).

```
char snowman = '\u2603';
char tripleIntegral = '\u222d';
```

This processing is done before lexing and parsing, not only within string or char literals!

```
public\u0020clas\u0073\u0020Myclas\u0073 \u007b ...
char newline = '\n';     // OK!
char newline = '\012';   // OK!
char newline = '\u000a'; // Error; \u000a is a real newline!
char newline = '         // Just like having an actual newline
';                       // between the single quotes...
```

String literals are written within double quotes: `"Hello, World"`. You can use the same escape codes as in character literals.

2.2 Identifiers

Any variable, field, class, and so on must have a name. This name must be a proper *identifier*, consisting of a letter or underscore (_) possibly followed by other letters, underscores and digits. Since Java uses Unicode, a "letter" does not have to be an English letter (a-z). A few examples:

```
int foo;
double bar;
void myLongMethodName() { ... }
String HELLO_WORLD;
void mittLångaMetodnamn() { ... }
double π = 3.1415926535;
```

2.2.1 Naming Conventions

There is a set of standard naming conventions for Java code. These conventions are followed by the Java class libraries and by almost all code you will see in articles and other examples. Please follow them during the course, as this will make it easier for other Java programmers to read your code.

- Class names should be nouns. All words are capitalized, and there are no underscores. Examples: `String`, `Border`, `TextArea`, `InheritableThreadLocal`, `URLStreamHandler`.
- Interface names are capitalized like class names, but do not necessarily need to be nouns. Many interface names are adjectives. Example: `List`, `Serializable`, `ActionListener`.
- Ordinary variables, fields, and methods: All words except the first are capitalized, and there are no underscores. Examples: `size`, `width`, `arrayLength`, `addElement()`, `defaultFontSize`.
- Constants (that is, `final static` fields): Only capitals, with underscores between words. Examples: `PI`, `ACCESSIBLE_HYPERTEXT_OFFSET`, `WEEK_OF_YEAR`.
- Packages: A package name consists of a sequence of identifiers, separated by periods. Each identifier should consist of lower case letters. Example: `java.lang`, `java.util`, `se.liu.ida.jonkv.graphics`.

See also `http://java.sun.com/docs/codeconv/index.html`.

2.3 Declarations and Scoping

Variables can be declared at (almost) any point within a method – not only at the beginning of a method. Usually it is a good idea to declare them just before you need to use them. Do not follow the C practice of declaring all variables at the beginning of a function!

Naturally, a variable can be initialized when it is declared. The compiler complains if you try to use an uninitialized variable.

2.3.1 Final Variables; Blank Finals

If you declare a variable `final`, its value cannot be changed. This is very useful for several reasons:

- Catching programming errors – if you know a certain value will not be changed, declaring it `final` ensures that you won't change it by accident due to a typo or similar errors.
- Documenting your intentions – if you declare a variable `final`, anyone reading the code knows that when the variable is used fifty lines below, the variable will still retain its initial value.

Declare your variables as `final` whenever possible!

```
public static void main(String[] args) {
    final double j = complexMethodCall();
    System.out.println("j squared is " + (j*j));
    // ... lots of code ...
    int i = 1;
    while (i < 1024) {
        System.out.println("i = " + i);
        j = j * 2;      // Mistake!
        // Compiler will complain: j is final.
    }
}
```

A *blank final* is declared `final` but not initialized to a specific value. Like any variable, it must be assigned a value before it is used. However, it must not be initialized twice (the compiler verifies this).

Consider the following example. No matter which branch we choose, j will be initialized exactly once before use.

```
public static void main(String[] args) {
    final double j;
    if (someCondition) {
        j = firstComplexMethodCall();
    } else {
        j = secondComplexMethodCall();
    }
    System.out.println("j squared is " + (j*j));
}
```

2.3.2 Scoping

Java uses lexical scoping, with levels defined by brackets {...}. As in C and C++, a variable is visible at its own bracket level and all inner levels.

```
public static void main(String[] args) {
    final int x = 24;
    if (x == 24) {
        final int y = 2 * x;          // x still visible
        System.out.println(x + y);    // x and y visible
    }
    System.out.println(x);            // x still visible
    System.out.println(y);            // COMPILE ERROR!
    final double y = 3.14159 * x;
    System.out.println(y);            // OK, there's a y
}
```

Note that although fields can be hidden by scoping, variables cannot (unlike in C). The following example is not legal Java:

```
public static void main(String[] args) {
    final int x = 24;
    if (someCondition) {
        final int x = 36;             // COMPILE ERROR!
        System.out.println(x);
    }
    System.out.println(x);            // 24
}
```

2.4 Type Conversions (Casting)

You can convert numeric values between the various numeric types.

Java has *automatic promotion* (also called widening or upcasting): If you provide a value of a smaller type where a larger type is expected, no explicit cast is necessary.

```
double d = 12;                // converted to 12.0
void print(int f);
byte b = 100;
print(b);                     // Byte 100 converted to int
```

Java requires *manual demotion* (also called narrowing or downcasting): If you provide a value of a larger type where a smaller type is expected, you must add an explicit cast.

```
int i = 271828.18;            // Not allowed
int i = (int) 271828.18;      // i = 271828 (truncated)
short s = (short) i;          // s = 9684 (lowest 16 bits)
float f = (float) 271828.18   // f = 271828.2 (rounded)
void print(int i);
print((int) 129.8);           // Calls print(129)
```

However, there is a special case for integer constants. These are automatically converted to byte or short, if they are guaranteed to fit in that type. For example, the value 12 below is in fact of type int, but can be assigned to a short or byte variable.

```
int i = 12;
short s = 12;
byte b = 12;
byte b = 200;         // Not allowed
void print(byte b);
print(12);            // No cast necessary
```

2.4.1 Automatic Promotion in Expressions

Java has no built-in operations on byte or short values. Instead, if the operands of a binary operator are of type byte or short, they are promoted to int values before the computation takes place.

```
byte b = 50;
b = b*2;              // Compiler complains
b = (byte) b*2        // OK
```

Java does have operations on `int`, `long`, `float` and `double`, but the two operands need to be of the same type. If they are of different types, the operand of the smaller type is automatically converted to the larger type: `int` $\rightarrow$ `long` $\rightarrow$ `float` $\rightarrow$ `double`.

In the example below, `10` is of type `int` and `0.5` is of type `double`. Both values are converted to `double`, so the expression is equivalent to `10.0*0.5`. The result is `5.0`, which is a `double`. It is not possible to assign the integer variable `x` a double-precision floating point value, so the compiler will complain.

```
int x = 10 * 0.5;              // Error
int x = (int) (10 * 0.5);      // OK
```

Despite these checks, there can of course still be overflow. As in most languges, this is not signalled in any way.

```
int x1 = 131072 * 131072;    // x1 == 0
long x2 = 131072 * 131072;   // x2 == 0
long x3 = 131072L * 131072;  // x3 == 17179869184
```

Note that in the three multiplication expressions above, the types of the operands determine whether an `int`, `long`, `float`, or `double` multiplication is performed. The type of the variable (`x1`, `x2`, `x3`) does not matter, so `long x2 = 131072*131072` results in a 32-bit `int` multiplication. The result is the 32-bit integer 0 (zero), which is then promoted to a 64-bit `long` value 0L. In the last example, however, we multiply the 64-bit `long` 131072L with the 32-bit `int` 131072. This results in a 64-bit multiplication.

2.5 Operators and Expressions

2.5.1 Integer Operators

The following operations are available for integral types (including char):

Binary math	`a+b, a-b, a*b, a/b, a%b` (remainder: 17 % 5 == 2)
Unary math	`+a, -a`
Shift	`a << b` (left), `a >> b` (signed right), `a >>> b` (zero extension)
If-else	`a ? b : c` (if a is true, result is b; otherwise, c)
Relational	`a<b, a<=b, a==b, a>=b, a>b, a!=b` (not equals)

An example: Using the if-else operator to calculate $n!$ (n factorial), where $0! = 1$ and $n! == n*(n-1)!$. For example, $6! = 6*5*4*3*2*1$. Two ways of writing this recursively:

```
public long fac(final int n) {
    final long retval;
    if (n == 0) retval = 1;
    else        retval = n * fac(n-1);
    return retval;
}
public long fac(final int n) {
    final long retval = (n == 0) ? 1 : (n * fac(n-1));
    return retval;
}
```

The last version can be simplified as follows

```
public long fac(final int n) {
    return (n == 0) ? 1 : (n * fac(n-1));
}
```

2.5.2 Floating Point Operators

The following operations are available for floating point types (`float` and `double`):

Binary math	`a+b, a-b, a*b, a/b, a%b` (remainder)
Unary math	`+a, -a`
If-else	`a ? b : c` (if a is true, result is b; otherwise, c)
Relational	`a<b, a<=b, a==b, a>=b, a>b, a!=b` (not equals)

2.5.3 Boolean Operators

The following operations are available for `boolean` expressions:

Binary logical	`a & b, a	b, a ^ b` (and, or, xor; always evaluate a and b)
Unary logical	`!a` (negation)
Short-circuiting	`a && b, a || b` (only evaluate what is needed)
Relational	`a == b, a != b` (equals, not equals)

The short-circuiting logical operators only evaluate the second operand if this is necessary in order to determine the final value of the expression. For example, if `a` is false, `a && b` must be false regardless of the value of `b`, so `b` is not evaluated. If `a` is true, `a || b` must be true regardless of the value of `b`, so `b` is not evaluated. This is better for performance. It may also be useful in order to emulate nested or sequential if statements. The following two snippets are equivalent, in that if the user first answers "no", he is not asked "Are you really sure?":

```
if (askUser("Quit?") && askUser("Are you really sure?")) {
    quit();
}

if (askUser("Quit?")) {
    if (askUser("Are you really sure?")) {
        quit();
    }
}
```

The non-short-circuiting logical operators always evaluate both operands. This is rarely necessary (you usually use the short-circuiting variations), but can be useful in case evaluating an operand has a side effect that is required to take place. For example, suppose that `saveIfNeeded(file)` saves a file if it is needed, and returns true iff it did save a file. Then, the following two snippets are equivalent, in that the program always attempts to save the second file even if the first file was saved.

```
if (saveIfNeeded("foo") | saveIfNeeded("bar")) {
    statusbar.setText("Saved!");
}

boolean saved1 = saveIfNeeded("foo");
boolean saved2 = saveIfNeeded("bar");
if (saved1 || saved2) {
    statusbar.setText("Saved!");
}
```

2.5.4 Assignment; Increment/Decrement

Java has a plain assignment operator, and one assignment version of each binary operator. For example:

```
a = b;
a += b;   // a = a + b;
a *= b;   // a = a * b;
a >>>= 3; // a = a >>> 3;
```

The combined operation/assignment operators have several advantages:

- Less code to write
- The left operand is evaluated only once – compare `arr[f(x)] += 12` with `arr[f(x)] = arr[f(x)] + 12`, where you have to call `f(x)` twice
- Easier to read (although some may disagree): You see immediately that you are incrementing a variable rather than assigning a new completely unrelated value

An assignment is not only a statement – it can also be used as an expression which has a value in itself: The value that is being assigned. In other words, `a = 3` is an expression which has the value 3.

```
int a = 7;
System.out.println(a = 3); // Prints 3
```

There are also specialized operators for incrementing or decrementing the value of a variable or field. The post-increment/decrement variations are attached after the variable, while the pre-increment/decrement variations are prefixed to the variable:

```
a++; // post-increment: a += 1;
b--; // post-decrement: b -= 1;
++a; // pre-increment:  a += 1;
--b; // pre-decrement:  b -= 1;
```

The pre and post variations have exactly the same effect, but when they are considered as expressions, their values are different. The pre-increment operator first increments the variable and then returns the result, while the post-increment first calculates the old value (which will be returned) and then increments the variable:

```
int a = 12, b = 12;
System.out.println(a++);        // Evaluate then increment: 12
System.out.println(++b);        // Increment then evaluate: 13
System.out.println(a);          // 13
System.out.println(b);          // 13
```

2.5.5 Assignment vs. Comparison

Remember the difference between = and ==: A single equals sign means assignment, while a double equals sign means comparison.

```
int i = 100;
i = i * 2;
i = (int) Math.sqrt(4711);
if (i == 100) System.out.println("Yes");
```

As mentioned above, an assignment is not only a statement but also an expression with a value. The value of `j = 100` is 100, so you can write "`i = j = 100;`", or "`System.out.println(j = 100);`". The value of `i == 100` is true or false, so you can write "`boolean equals100 = (i == 100);`", or "`System.out.println(i == 100);`".

2.6 Statements

Every statement ends with a semicolon, except composite statements ending in a block:

```
public class JavaTest {
    public static void main(String[] args) {
        System.out.println("Hello, world");
        int i = 1;
        while (i < 1024) {
            System.out.println("i = " + i);
            i = i * 2;
        } // No semicolon here
        while (i > 12) i /= 2;
    }
}
```

2.6.1 Conditional Statements: `if`, `switch`

There are two conditional statements: `if` and `switch`. The `if` statement works just like in C and C++:

```
if (condition) statement;
if (condition) statement1; else statement2;
if (condition) { st1; st2; } else { st3; st4; st5; st6; }
if (x == 0 || x == 1) {
    System.out.println("At most 1");
} else if (x == 2) {
    System.out.println("Exactly two");
} else if (x == 3) {
    System.out.println("Exactly three");
    System.out.println("Exactly three again");
} else {
    System.out.println("Something else");
}
```

Note that there is no semicolon after the condition. If you do add a semicolon, this is interpreted as an empty statement. In other words, `if (cond); statement;` means "if cond is true, do nothing; then, unconditionally perform statement".

The `switch` statement also works just like in C and C++. Remember to use `break` to end each case. Otherwise, you will fall through to the next case.

```
switch (x + y) {
    case 0:
    case 1:
        System.out.println("At most 1");
        break;
    case 2:
        System.out.println("Exactly two");
        // Fall through to case 3!
    case 3:
        System.out.println("Two or three");
        break;
    default:
        System.out.println("Something else");
}
```

2.6.2 Loops: `for`

A `for` loop takes the following form:

```
for (initialization; condition; update) statement;
```

A simple example:

```
// Print 0, 1, ..., 98, 99
int i;
for (i = 0; i < 100; i++) System.out.println(i);
System.out.println("After loop: " + i); // After loop: 100
```

Variables can be declared in the initialization part. Such variables go out of scope after the loop – they are only valid within the loop. To declare multiple variables of the same type, use a comma as shown below. You cannot declare multiple variables of different types.

```
for (int i = 0; i < 100; i++) System.out.println(i);
for (int i = 0, j = 0;   i < 12;   i++, j *= 2) statement;
```

Note that `for (...)` is immediately followed by the statement. Do not add an extra semicolon. This is interpreted as an empty statement. The following code does nothing twelve times and then performs `statement` exactly once:

```
for (int i = 0; i < 12; i++); statement;
```

2.6.3 Loops: `while` and `do/while`

A `while` loop tests the condition before entering the loop. Therefore, the loop may be executed zero times if the condition is false from the beginning.

```
int i = 0;
while (i < 100) {
    System.out.println(i);
    i++;
}
```

A `do/while` loop tests its condition after executing one iteration. Therefore, the loop is always executed at least once.

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 100);
```

2.6.4 Loops: `break` and `continue`

It is possible to force the program to continue with the next iteration using the `continue` statement. This can be useful in order to avoid deeply nested `if` statements, and works for all three kinds of loops (`for`, `while`, `do/while`). The following two loops are equivalent.

```
while (i++ < 100) {
    if (!test(i)) continue;
    doSomething(i);
    if (!test2(i)) continue;
    doSomethingElse(i);
}
```

```
while (i++ < 100) {
    if (test(i)) {
        doSomething(i);
        if (test2(i)) {
            doSomethingElse(i);
        }
    }
}
```

The `break` statement can be used to exit a loop prematurely (before the loop condition becomes false).

```
int i = 0;
while (i++ < 100) {
    if (!test(i)) break;  // To "After the loop"
    doSomething(i);
    doSomethingElse(i);
}
System.out.println("After the loop");
```

A loop can be *labeled*, and the labels can be used to specify which loop the `break` and `continue` statements refer to.

```
outer:
for (int i = 0; i < 12; i++) {
    inner:
    for (int j = 0; j < 12; j++) {
        while (thisIsTrue) {
            if (condition) continue inner;   // next j
            else if (otherCond) break inner; // exit inner loop
            else if (thirdCond) break outer; // exit outer loop
            ...;
        }
    }
    // destination of "break inner"
}
// destination of "break outer"
```

2.6.5 Exiting a labeled block: `break`

You can also use `break` to exit a labeled block. This works like a structured form of `goto` – there is no ordinary `goto` statement in Java.

```
myblock: {
    doSomething();
    if (condition) break myblock;
    doSomethingElse();
    for (int i = 0; i < 100; i++) {
        doSomethingWith(i);
    }
}
System.out.println("This is the place I jump to");
```

2.6.6 Returning from a method: `return`

A `void` method returns no value. As you might expect, the method returns when there are no more statements to execute:

```
public static void main(String[] args) {
    doSomething();
    doSomethingElse();
    // End of method...
}
```

You can also use the `return` statement to return "prematurely":

```
public static void main(String[] args) {
    doSomething();
    if (condition) return;
    doSomethingElse();
}
```

A non-`void` method must returns a value. The final statement in such a method must always be a `return` statement that specifies the return value.

```
public static int square(double x) {
    return x * x;
    // No code possible here --- we have returned!
}
```

2.6.7 Exception handling: `try/catch/finally`

A `try/catch/finally` statement is used for catching exceptions and ensuring that cleanup code will always be run. This is not covered in detail in this document.

```
try {
    doSomething();
} catch (final NullPointerException e) {
    handleTheException(e);
} catch (final IllegalArgumentException e) {
    handleTheException(e);
} finally {
    closeAllFiles();
}
```

2.6.8 Comments

There are two types of comments: Everything from "`//`" to the end of the line, and everything between "`/*`" and "`*/`".

```
/* This is a comment...
    And so is this... */
public static void main(String[] args) { // Me too
    double /* ugly comment */ x;
    x = 3.1415926535;   // pi
}
```

2.7 Arrays

Arrays will not be covered in detail here. However, a few facts should be mentioned.

Java arrays are objects. The array classes are named `int[]`, `String[]`, `char[]`, and so on. Like any object, an array is allocated with the `new` operator. Indexes are from `0` to `size-1`. The size is available in the `length` data field.

```
int[] myarr = new int[100];    // Allocate array of length 100
myarr[0] = 22;
...;
myarr[99] = 4711;
for (int i = 0; i < myarr.length; i++) {
    System.out.println(i + ": " + myarr[i]);
}
```

2.8 Compilers and Tools; Running a Java Program

Compilers and tools will be not be covered in detail here, but a few facts will be mentioned.

Every class is (usually) in its own .java file (you place the class `JavaTest` in "`JavaTest.java`"). Classes are compiled using `javac`, and you run your programs using the `java` command. You can start any Java class that has a `main` method.

```
>> cat JavaTest.java
public class JavaTest {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
>> javac JavaTest.java
>> java JavaTest
Hello, World!
```

You can also provide command line arguments after the class name on the command line. The arguments end up in the `args[]` array in `main()`.

```
>> cat JavaTest.java
public class JavaTest {
    public static void main(String[] args) {
        System.out.println("Hello, world");
        for (int i = 0; i < args.length; i++) {
            System.out.println("Arg " + i + " is " + args[i]);
        }
    }
}
>> java JavaTest hello "a long argument" 12
Hello, World
Arg 0 is hello
Arg 1 is a long argument
Arg 2 is 12
```

3 Javadoc

Javadoc is a standardized system for documenting packages, classes, methods, and fields. If you have used Java before, you have probably browsed the standard API documentation on the WWW – this documentation is written in Javadoc format and converted to HTML by the javadoc tool.

While Javadoc 1.0 was a rather simple tool that performed only some minimal processing of its input in order to produce a set of HTML documents, the javadoc tool has evolved along with the rest of the Java environment and its functionality has been greatly expanded. The full functionality is covered in detail in the official documentation, which is available on the web. Here, we will provide a more simplistic view, covering mainly the functionality that was available in Javadoc 1.0. The following links will be useful for those who want additional information:

- `http://java.sun.com/j2se/javadoc/` is the official Javadoc home page.
- `http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html` contains documentation for the javadoc tool and shows the technical details involved in writing Javadoc documentation, including a full list of documentation tags.

Also, if you intend to write Javadoc comments for an API that will be used by others, you should *definitely* read the following guidelines. Bad documentation is often worse than no documentation at all!

- `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html` contains a set of style guidelines for writing documentation comments.
- `http://java.sun.com/j2se/javadoc/writingapispecs/index.html` contains a set of requirements for writing proper API documentation in sufficient detail that one does not have to refer to the source code to understand what a method really does (which is all too often the case in documentation written by novices!).

3.1 Writing Javadoc Comments

Package-level documentation is written in an HTML called package.html, which should be placed together with the source code for a specific package.

All other Javadoc documentation is written as a set of Javadoc comments in each .java file, immediately above the class, method, or field that is being documented. A javadoc comment is a special form of block comment which starts with slash-star-star rather than the ordinary slash-star. The comment is written in HTML format with a number of special documentation tags. The following example shows a Javadoc comment for a method declaration.

```
/**
  * Compares the specified object with this map for equality.  Returns
  * <code>true</code> if the given object is also a map and the two Maps
  * represent the same mappings.  More formally, two maps <code>t1</code>
  * and <code>t2</code> represent the same mappings if
  * <code>t1.entrySet().equals(t2.entrySet())</code>.  This ensures that the
  * <code>equals</code> method works properly across different  implementations
  * of the <code>Map</code> interface.
  *
  * @param obj object to be compared for equality with this map.
  * @return <code>true</code> if the specified object is equal to this map.
  */
boolean equals(Object obj);
```

Note the special documentation tags `@param`, used for documenting a method parameter, and `@return`, used for documenting a return value.

The following are the most commonly used tags:

- `@param arg description` – documents the given method parameter and what requirements the parameter must satisfy (is it allowed to be null, and if so, what does that mean?). Use one `@param` tag for each parameter.
- `@return description` – documents the return value of a method.
- `@exception ExceptionClassName reason` – specifies that a certain exception may be thrown by the method, and documents why the exception would be thrown. All checked exceptions should be documented, as well as any non-checked exception that the caller might reasonably want to catch. Use one `@exception` tag for each exception that may be thrown. For example:

  ```
  @throws UnsupportedOperationException This map is immutable
  and cannot be cleared.
  ```
- `@throws ExceptionClassName reason` – an alias for `@exception`.
- `@since version` – the class, method, or field has been present since the given version.
- `@deprecated explanation` – indicates that this class, method, or field should no longer be used. The first sentence should tell the user *why* the method was deprecated and what to use instead. The compiler automatically warns you if you attempt to use a deprecated class or member.

  ```
  @deprecated As of JDK 1.1, replaced by
              {@link #setBounds(int,int,int,int)}
  ```
- `@version version` – indicates the version of a class.
- `@author name` – indicates the author of a class or member. Use multiple `@author` tags to indicate multiple authors.
- `@see ...` – adds a hyperlink to another part of the document or another document. This link is placed in a "See also" section. The following formats are allowed:

  `@see "text"` `@see package.class label` `@see package.class#member label` `@see <a href="http://java.sun.com/">javasoft</a>`
- `{@link name label}` – creates an inline hyperlink to `name` labelled `label`. For example:

  ```
  @deprecated As of JDK 1.1, replaced by
              {@link #setBounds(int,int,int,int)}
  ```

Note again that Javadoc is usually compiled into HTML. Therefore, you can use HTML within comments, including `<code>` ... `</code>` to use a typewriter font for inline code examples, `<I>` ... `</I>` for italics, numbered and unnumbered lists, and so on. As in any HTML code, a paragraph mark `<P>` is required in order to begin a new paragraph; simply adding an empty line is not enough. Also, be careful with the special HTML characters `<>&`.

Contents