Java Precisely

THIRD EDITION

Peter Sestoft

```
public static Stream<IntList> perms(BitSet todo, IntList tail) {
  if (todo.isEmpty())
    return Stream.of(tail);
  else
    return todo.stream().boxed()
      .flatMap(r -> perms(minus(todo, r), new IntList(r, tail)));
}
```

```
public static Stream<IntList> perms(BitSet todo, IntList tail) {
  if (todo.isEmpty())
    return Stream.of(tail);
  else
    return todo.stream().boxed().flatMap(r -> perms(minus(todo, r), new IntList(r, tail)));
}
```

Peter Sestoft

Java Precisely

Third Edition

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times by the author using LaTeX and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data
Names: Sestoft, Peter.
Title: Java precisely / Peter Sestoft.
Description: Third edition. | Cambridge, MA : The MIT Press, [2016] | Includes bibliographical references and index.
Identifiers: LCCN 2015038522 | ISBN 9780262529075 (pbk. : alk. paper)
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J38 S435 2015 | DDC 005.13/3–dc23 LC record available at http://lccn.loc.gov/2015038522

10 9 8 7 6 5 4 3 2 1

Contents

Java Precisely

1 Running Java: Compilation, Loading, and Execution

Before a Java program can be executed, it must be compiled and loaded. The compiler checks that the Java program is *legal:* that the program conforms to the Java syntax, that operators (such as `+`) are applied to operands (such as `5` and `x`) of the correct type, and so on. If so, the compiler generates *class files*. Execution may then be started by loading the class files. Thus running a Java program involves three stages: *compilation* (checks that the program is well-formed), *loading* (loads and initializes classes), and *execution* (runs the code). This holds also for a program run from integrated development environments such as Eclipse or IntelliJ.

2 Names and Reserved Names

A *legal name* (of a variable, method, field, parameter, class, interface or package) starts with a letter or dollar sign (`$`) or underscore (_), and continues with zero or more letters or dollar signs or underscores or digits (`0`–`9`). Avoid dollar signs in class and interface names. Uppercase letters and lowercase letters are considered distinct. A legal name cannot be one of the following *reserved names:*

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

3 Java Naming Conventions

The following naming conventions are often followed, although not enforced by Java:

- If a name is composed of several words, then each word (except possibly the first one) begins with an uppercase letter. Examples: `setLayout`, `addLayoutComponent`.
- Names of variables, fields, and methods begin with a lowercase letter. Examples: `vehicle`, `myVehicle`.
- Names of classes and interfaces begin with an uppercase letter. Examples: `Cube`, `ColorCube`.
- Named constants (such as `final static` fields and enum values) are written entirely in uppercase, and the parts of composite names are separated by underscores (_). Examples: `CENTER`, `MAX_VALUE`.
- Package names are sequences of dot-separated lowercase names. Example: `java.awt.event`. For uniqueness, they are often prefixed with reverse domain names, as in `com.sun.xml.util`.

4 Comments and Program Layout

Comments have no effect on the execution of the program but may be inserted anywhere to help humans understand the program. There are two forms: one-line comments and delimited comments.

Program layout has no effect on the computer's execution of the program but is used to help humans understand the structure of the program.

Example 1 Comments

```
class Comment {
  // This is a one-line comment; it extends to the end of the line.
  /* This is a delimited comment,
     extending over several lines.
  */
  int /* This delimited comment extends over part of a line */ x = 117;
}
```

Example 2 Recommended Program Layout Style
For reasons of space this layout style is not always followed in this book.

```
class Layout {                          // Class declaration
  int x;

  Layout(int x) {
    this.x = x;                         // One-line body
  }

  int sum(int y) {                      // Multi-line body
    if (x > 0) {                        // If statement
      return x + y;                     // Single statement
    } else if (x < 0) {                 // Nested if-else, block statement
      int res = -x + y;
      return res * 117;
    } else { // x == 0                  // Terminal else, block statement
      int sum = 0;
      for (int i=0; i<10; i++) {        // For loop
        sum += (y - i) * (y - i);
      }
      return sum;
    }
  }

  static boolean checkdate(int mth, int day) {
    int length;
    switch (mth) {                      // Switch statement
    case 2:                             // Single case
      length = 28; break;
    case 4: case 6: case 9: case 11:    // Multiple case
      length = 30; break;
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      length = 31; break;
    default:
      return false;
    }
    return (day >= 1) && (day <= length);
  }
}
```

5 Types

A *type* is a set of values and operations on them. A type is either a primitive type or a reference type.

5.1 Primitive Types

A *primitive type* is either `boolean` or one of the *numeric types* `char`, `byte`, `short`, `int`, `long`, `float`, or `double`. The primitive types, example literals (that is, constants), size in bits (where 8 bits equals 1 byte), and value range, are shown in the table opposite. For readability, a number constant may contain underscores (_) anywhere except as the first and last character of the constant.

The integer types are exact within their range. They use signed 2's complement representation (except for `char`), so when the most positive number in a type is *max*, then the most negative number is $-max-1$. The floating-point types are inexact and follow IEEE754, with the number of significant digits indicated by "sigdig" in the table. For character escape sequences such as `\u0000`, see page 10.

5.2 Reference Types

A *reference type* is a class type defined by a class declaration (section 9.1), or an interface type defined by an interface declaration (section 13.1), or an array type (section 5.3), or an enum type (chapter 14).

A value of reference type is either `null` or a reference to an object or array. The special value `null` denotes "no object." The literal `null`, denoting the `null` value, can have any reference type.

5.3 Array Types

An *array type* has the form `t[]`, where `t` is any type. An array type `t[]` is a reference type. Hence a value of array type `t[]` is either `null` or a reference to an array whose element type is precisely `t` (when `t` is a primitive type), or is a subtype of `t` (when `t` is a reference type).

5.4 Boxing: Wrapping Primitive Types as Reference Types

For every primitive type there is a corresponding wrapper class, which is a reference type. The wrapper classes are listed in the table opposite. An object of a wrapper class contains a single value of the corresponding primitive type.

A wrapper class must be used when a value of primitive type is passed to a method that expects a reference type, or is stored in a variable or field of reference type. For instance, to store an `int` in a collection (chapter 22) one must wrap it as an Integer object.

The conversion from primitive type to wrapper class is called *boxing*, and the opposite conversion is called *unboxing*. Boxing and unboxing are performed automatically when needed. Boxing and unboxing may also be performed explicitly using operations such as `new Integer(i)` to box the integer `i`, and `o.intValue()` or `(int)o` to unbox the Integer object `o`. If `o` is `null`, then unboxing of `o` will fail at run-time by throwing NullPointerException. Because of automatic unboxing, a Boolean value may be used in conditional statements (`if`, `for`, `while`, and `do-while`) and in logical operators (such as `!`, `&&`, `?:` and so on); and Integer and other integer type wrapper classes may be used in `switch` statements.

A boxed value can be unboxed only to a value of the boxed type, or to a supertype. Thus an Integer object can be unboxed to an `int` or a `long` because `long` is a supertype of `int`, but not to a `char`, `byte`, or `short`.

The wrapper classes Byte, Short, Integer, Long, Float, and Double have the common superclass Number.

Primitive Types

Type	Kind	Example Literals	Size	Range	Wrapper
`boolean`	logical	`false`, `true`	1		Boolean
`char`	integer	`' '`, `'0'`, `'A'`, ...	16	`\u0000` ... `\uFFFF` (unsigned)	Character
`byte`	integer	`0`, `1`, `-1`, `117`, ...	8	$max = 127$	Byte
`short`	integer	`0`, `1`, `-1`, `117`, `2_117`, ...	16	$max = 32767$	Short
`int`	integer	`0`, `1`, `-1`, `117`, `2_117`, ...	32	$max = 2147483647$	Integer
`long`	integer	`0L`, `1L`, `-1L`, `117L`, `2_117L`, ...	64	$max = 9223372036854775807$	Long
`float`	floating	`-1.0f`, `0.49f`, `3E8f`, ...	32	$\pm 10^{-38} \ldots \pm 10^{38}$, sigdig 6–7	Float
`double`	floating	`-1.0`, `0.49`, `3E8`, ...	64	$\pm 10^{-308} \ldots \pm 10^{308}$, sigdig 15–16	Double

Integer Literals Integer literals (of type `byte`, `char`, `short`, `int`, or `long`) may be written in four bases:

Notation	Base	Distinction	Example Integer Literals
Decimal	10	No leading `0`	`1_234_567_890`, `127`, `-127`
Binary	2	Leading `0b` or `0B`	`0b10`, `0b111_1111`, `-0b111_1111`
Octal	8	Leading `0`	`01234567`, `0177`, `-0177`
Hexadecimal	16	Leading `0x` or `0X`	`0xAB_CDEF_0123`, `0x7F`, `-0x7F`

Example 3 Automatic Boxing and Unboxing

```
Boolean bb1 = false, bb2 = !bb1;    // Boxing to [false] [true]
Integer bi1 = 117;                  // Boxing to [117]
Double bd1 = 1.2;                   // Boxing to [1.2]
boolean b1 = bb1;                   // Unboxing, result false
if (bb1)                            // Unboxing, result false
  System.out.println("Not true");
int i1 = bi1 + 2;                   // Unboxing, result 119
// short s = bi1;                   // Illegal
long l = bi1;                       // Legal: int is subtype of long
Integer bi2 = bi1 + 2;              // Unboxing, boxing, result [119]
Integer[] biarr = { 2, 3, 5, 7, 11 };
int sum = 0;
for (Integer bi : biarr)
  sum += bi;                        // Unboxing in loop body
for (int i : biarr)                 // Unboxing in loop header
  sum += i;
int i = 1934;
Integer bi4 = i, bi5 = i;
// Prints true true true false; bi4==bi5 is a reference comparison:
System.out.format("%b %b %b %b%n", i==i, bi4==i, i==bi5, bi4==bi5);
Boolean bbn = null;
boolean b = bbn;                    // Compiles OK, fails at run-time
if (bbn)                            // Compiles OK, fails at run-time
  System.out.println("Not true");
Integer bin = null;
Integer bi6 = bin + 2;              // Compiles OK, fails at run-time
```

6 Variables, Parameters, Fields, and Scope

A *variable* is declared inside a method, constructor, initializer block, or block statement (section 12.2). The variable can be used only in that block statement (or method or constructor or initializer block), and only after its declaration.

A *parameter* is a special kind of variable: it is declared in the parameter list of a method or constructor, and is given a value when the method or constructor is called. The parameter can be used only in that method or constructor.

A *field* is declared inside a class, but not inside a method or constructor or initializer block of the class. It can be used anywhere in the class, also textually before its declaration.

6.1 Values Bound to Variables, Parameters, or Fields

A variable, parameter, or field of primitive type holds a *value* of that type, such as the boolean `false`, the integer `117`, or the floating-point number `1.7`. A variable, parameter, or field of reference type `t` either has the special value `null` or holds a reference to an object or array. If it is an object, then the run-time class of that object must be `t` or a subclass of `t`.

6.2 Variable Declarations

The purpose of a variable is to hold a value during the execution of a block statement (or method or constructor or initializer block). A *variable-declaration* has one of the forms

> *variable-modifier type varname1*`,` *varname2*`, ... ;`
> *variable-modifier type varname1* `=` *initializer1*`, ... ;`

A *variable-modifier* may be `final` or absent. If a variable is declared `final`, then it must be initialized or assigned at most once at run-time (exactly once if it is ever used): it is a *named constant*. However, if the variable has reference type, then the object or array pointed to by the variable may still be modified. A *variable initializer* may be an expression or an array initializer (section 8.2).

Execution of the variable declaration will reserve space for the variable, then evaluate the initializer, if any, and store the resulting value in the variable. Unlike a field, a variable is not given a default value when declared, but the compiler checks that it has been given a value before it is used.

6.3 Scope of Variables, Parameters, and Fields

The *scope* of a name is that part of the program in which the name is visible. The scope of a variable extends from just after its declaration to the end of the innermost enclosing block statement. The scope of a method or constructor parameter is the entire method or constructor body. For a control variable `x` declared in a `for` statement

```
for (int x = ...; ...; ...) body
```

the scope is the entire `for` statement, including the header and the body.

Within the scope of a variable or parameter `x`, one cannot redeclare `x`. However, one may declare a variable `x` within the scope of a field `x`, thus *shadowing* the field. Hence the scope of a field `x` is the entire class, except where shadowed by a variable or parameter of the same name, and except for initializers preceding the field's declaration (section 9.1).

Example 6 Variable Declarations

```
public static void main(String[] args) {
  int a, b, c;
  int x = 1, y = 2, z = 3;
  int ratio = z/x;
  final double PI = 3.141592653589;
  boolean found = false;
  final int maxyz;
  if (z > y) maxyz = z; else maxyz = y;
}
```

Example 7 Scope of Fields, Parameters, and Variables
This program declares five variables or fields, all called `x`, and shows where each one is in scope (visible). The variables and fields are labeled `#1`, ..., `#5` for reference.

```
class Scope {
  ...                   //
  void m1(int x) {      // Declaration of parameter x (#1)
    ...                 // x #1 in scope
  }                     //
  ...                   //
  void m2(int v2) {     //
    ...                 // x #5 in scope
  }                     //
  ...                   //
  void m3(int v3) {     //
    ...                 // x #5 in scope
    int x;              // Declaration of variable x (#2)
    ...                 // x #2 in scope
  }                     //
  ...                   //
  void m4(int v4) {     //
    ...                 // x #5 in scope
    {                   //
      int x;            // Declaration of variable x (#3)
      ...               // x #3 in scope
    }                   //
    ...                 // x #5 in scope
    {                   //
      int x;            // Declaration of variable x (#4)
      ...               // x #4 in scope
    }                   //
    ...                 // x #5 in scope
  }                     //
  ...                   //
  int x;                // Declaration of field x (#5)
  ...                   // x #5 in scope
}
```

7 Strings

A *string* is an object of the predefined class String. It is immutable: once created it cannot be changed. A string literal is a sequence of characters within double quotes: `"New York"`, `"A38"`, `""`, and so on. Internally, a character is stored as a number using the Unicode [1] character encoding, whose character codes 0–127 coincide with the old ASCII encoding. String literals and character literals may use character *escape sequences*:

Escape Code	Meaning
`\b`	backspace
`\t`	horizontal tab
`\n`	newline
`\f`	form feed (page break)
`\r`	carriage return
`\"`	the double quote character
`\'`	the single quote character
`\\`	the backslash character
`\`*ddd*	the character whose character code is the three-digit octal number *ddd*
`\u`*dddd*	the character whose character code is the four-digit hexadecimal number *dddd*

A character escape sequence represents a single character. Since the letter A has code 65 (decimal), which is written 101 in octal and 0041 in hexadecimal, the string literal `"A\101\u0041"` is the same as `"AAA"`. If `s1` and `s2` are expressions of type `String` and `v` is an expression of any type, then

- `s1.length()` of type `int` is the length of `s1`, that is, the number of characters in `s1`.
- `s1.equals(s2)` of type `boolean` is `true` if `s1` and `s2` contain the same sequence of characters, and `false` otherwise; `equalsIgnoreCase` is similar but does not distinguish lowercase and uppercase.
- `s1.charAt(i)` of type `char` is the character at position `i` in `s1`, counting from 0. If the index `i` is less than 0, or greater than or equal to `s1.length()`, then StringIndexOutOfBoundsException is thrown.
- `s1.toString()` of type `String` is the same object as `s1`.
- `String.valueOf(v)` returns the string representation of `v`, which can have any primitive type (section 5.1) or reference type. When `v` has reference type and is not `null`, then it is converted using `v.toString()`; if it is `null`, then it is converted to the string `"null"`. Any class `C` inherits from Object a default `toString` method that produces strings of the form `C@2a5734`, where `2a5734` is some memory address, but `toString` may be overridden to produce more useful strings.
- `s1 + s2` has the same meaning as `s1.concat(s2)`: it constructs the concatenation of `s1` and `s2`, a new String consisting of the characters of `s1` followed by the characters of `s2`. Both `s1 + v` and `v + s1` are evaluated by converting `v` to a string with `String.valueOf(v)`, thus using `v.toString()` when `v` has reference type, and then concatenating the resulting strings.
- `s1.compareTo(s2)` returns a negative integer, zero, or a positive integer, according as `s1` precedes, equals, or follows `s2` in the usual lexicographical ordering based on the Unicode [1] character encoding. If `s1` or `s2` is `null`, then the exception NullPointerException is thrown. Method `compareToIgnoreCase` is similar but does not distinguish lowercase and uppercase.
- `s1.substring(int i, int j)` returns a new String of the characters from `s1` with indexes `i..(j-1)`. Throws IndexOutOfBoundsException if `i<0` or `i>j` or `j>s1.length`.
- `s1.subSequence(int i, int j)` is like `substring` but returns a CharSequence (section 26.7).
- More `String` methods are described in the Java class library documentation [2].

Example 8 Equality of Strings and the Subtlety of the (+) Operator

```
String s1 = "abc";
String s2 = s1 + "";        // New object, but contains same text as s1
String s3 = s1;             // Same object as s1
String s4 = s1.toString();  // Same object as s1
// The following statements print false, true, true, true, true:
System.out.println("s1 and s2 identical objects: " + (s1 == s2));
System.out.println("s1 and s3 identical objects: " + (s1 == s3));
System.out.println("s1 and s4 identical objects: " + (s1 == s4));
System.out.println("s1 and s2 contain same text: " + (s1.equals(s2)));
System.out.println("s1 and s3 contain same text: " + (s1.equals(s3)));
// These two statements print 35A and A1025 because (+) is left-associative:
System.out.println(10 + 25 + "A");  // Same as (10 + 25) + "A"
System.out.println("A" + 10 + 25);  // Same as ("A" + 10) + 25
```

Example 9 Concatenating All Command Line Arguments
When concatenating many strings, use a string builder instead (chapter 19 and example 104).

```
public static void main(String[] args) {
  String res = "";
  for (int i=0; i<args.length; i++)
    res += args[i];
  System.out.println(res);
}
```

Example 10 Counting the Number of e's in a String

```
static int ecount(String s) {
  int ecount = 0;
  for (int i=0; i<s.length(); i++)
    if (s.charAt(i) == 'e')
      ecount++;
  return ecount;
}
```

Example 11 Determining Whether Strings Occur in Lexicographically Increasing Order

```
static boolean sorted(String[] a) {
  for (int i=1; i<a.length; i++)
    if (a[i-1].compareTo(a[i]) > 0)
      return false;
  return true;
}
```

Example 12 Using a Class That Declares a `toString` Method
The class Point (example 27) declares a `toString` method that returns a string of the point coordinates. The operator (+) calls the `toString` method implicitly to format the Point objects.

```
Point p1 = new Point(10, 20), Point p2 = new Point(30, 40);
System.out.println("p1 is " + p1);      // Prints: p1 is (10, 20)
System.out.println("p2 is " + p2);      // Prints: p2 is (30, 40)
p2.move(7, 7);
System.out.println("p2 is " + p2);      // Prints: p2 is (37, 47)
```

7.1 String Formatting

Formatting of numbers, characters, dates, times, and other data may be done using a formatting string `fmt` containing *formatting specifiers*, using one of these methods:

- `String.format(fmt, v1, ..., vn)` returns a String produced from `fmt` by replacing formatting specifiers with the strings resulting from formatting the values `v1`, ..., `vn`.
- `strm.format(fmt, v1, ..., vn)`, where `strm` is a PrintWriter or PrintStream (section 26.6), constructs a string as above, outputs it to `strm`, and returns `strm`.
- `strm.printf(fmt, v1, ..., vn)` behaves exactly as `strm.format(fmt, v1, ..., vn)`.

These methods exist also in a version that take a Locale object as first argument; see examples 16 and 17. Formatting specifiers are described in sections 7.1.1 and 7.1.2 below. If a value `vi` is of the wrong type for a given formatting specifier, or if the formatting specifier is ill-formed, then a call to the above methods will throw an exception of class IllegalFormatException or one of its subclasses.

7.1.1 Formatting of Numeric, Character, and General Types

A formatting specifier for numeric, character, and general types has this form:

`%[`*index*`$][`*flags*`][`*width*`][.`*precision*`]`*conversion*

The *index* is an integer $1, 2, \ldots$ indicating the value $\mathtt{v}_{index}$ to format; the *conversion* indicates what operation is used to format the value; the *width* indicates the minimum number of characters used to format the value; the *flags* indicate how that width should be used (where "`-`" means left-justification, or padding on the right, and "`0`" means padding with zero); and *precision* limits the output, such as the number of fractional digits. Each of the four parts in brackets `[ ]` is optional; the only mandatory parts are the percent sign (`%`) and the *conversion*.

The documentation for Java API class java.util.Formatter gives the full details of number formatting. The table below shows the legal *conversions* on numbers (I = integers, F = floating-point numbers, IF = both), characters (C), and general types (G). An uppercase conversion such as `X` produces uppercase output.

Format	*conversion*	*flags*	*precision*	Type
Decimal	`d`	`-+ 0,(`		I
Octal	`o`	`-#0`		I
Hexadecimal	`x` or `X`	`-#0`		I
Hexadecimal significand and exponent	`a` or `A`	`-#+ 0`		F
General: scientific or fractional	`g` or `G`	`-#+ 0,(`	Max. significant digits	IF
Fixed-point number	`f`	`-#+ 0,(`	Fractional digits	IF
Scientific notation	`e` or `E`	`-#+ 0,(`	Fractional digits	IF
Unicode character [1]	`c` or `C`	`-`		C
Boolean: `"true"` or `"false"`	`b` or `B`	`-`		Boolean
Hexadecimal hashcode of value, or `"null"`	`h` or `H`	`-`		G
Determined by value's `formatTo` method	`s` or `S`	`-`		G
A percent symbol (`%`)	`%`	(none)		
Platform-specific newline	`n`	(none)		

Example 15 Formatting Dates and Times as Strings
This example prints `2004-09-14 12:09`; months are numbered from 0 in Java's GregorianCalendar class.

```
GregorianCalendar date = new GregorianCalendar(2004, 8, 14, 12, 9, 28);
System.out.format("%1$tF %1$tR%n", date);
```

Example 16 Locale-Specific Formatting of Dates and Times
The formatting of date and time often depends on the locale: language and nationality. For instance, this is the case for the formatting specifier `%tc`. The Locale class is in package `java.util`.

```
Date now = new Date();
System.out.format("%tc%n", now);                       // default locale
System.out.format(Locale.US, "%tc%n", now);            // en_US locale
System.out.format(Locale.GERMANY, "%tc%n", now);       // de_DE locale
```

Example 17 Locale-Specific Formatting of Numbers
Number formatting is locale sensitive: different languages use different decimal separators (point or comma). For instance, this example outputs `1,234,567.90` and `1.234.567,90` and `1 234 567,90` where the spaces in the latter number are special non-breaking spaces (ISO Latin1 character `'\240'`).

```
double d = 1234567.9;
System.out.format(Locale.US,      "%,.2f%n", d);       // en_US locale
System.out.format(Locale.GERMANY, "%,.2f%n", d);       // de_DE locale
System.out.format(Locale.FRANCE,  "%,.2f%n", d);       // fr_FR locale
```

Some Date and Time Formatting Specifiers and Their Effect

Different languages and countries have very different conventions for writing dates, requiring different formatting specifiers for use with `String.format`. In general, the locale mechanism is not sufficient to write locale-specific dates, except when using the `%tc` formatting specifier. To avoid misunderstandings, do give all four digits of the year, and avoid formats such as `03/05/04` that may have different US and UK interpretations.

Formatting Specifier	Result	Locale	Usage
`%tc`	`Fri Mar 05 21:06:07 CET 2004`	`en_US`	US
`%tc`	`Fr Mrz 05 21:06:07 CET 2004`	`de_DE`	Germany
`%tc`	`ven. mars 05 21:06:07 CET 2004`	`fr_FR`	France
`%1$tD`	`03/05/04`	`en_US`	US
`%1$tm/%1$td/%1$ty`	`03/05/04`	`en_US`	US
`%1$tm/%1$td/%1$ty %1$tI:%1$tM %1$tP`	`03/05/04 09:06 PM`	`en_US`	US
`%1$td.%1$tm.%1$tY %1$tH:%1$tM`	`05.03.2004 21:06`	`en_US`	Germany
`%1$td/%1$tm/%1$tY`	`05/03/2004`	`en_US`	UK
`%1$td-%1$tb-%1$ty`	`05-Mar-04`	`en_US`	US/UK
`%1$tB %1$te, %1$tY`	`March 5, 2004`	`en_US`	US
`%1$tA %1$tB %1$te, %1$tY`	`Friday March 5, 2004`	`en_US`	US
`%1$tA, %1$te %1$tB %1$tY`	`Friday, 5 March 2004`	`en_US`	UK
`%1$te. %1$tB %1$tY`	`5. März 2004`	`de_DE`	Germany
`%1$tA %1$te. %1$tB %1$tY`	`Freitag 5. März 2004`	`de_DE`	Germany
`%1$tFT%1$tT`	`2004-03-05T21:06:07`	`en_US`	RFC3339

8 Arrays

An *array* is an indexed collection of variables, called *elements*. An array has a given *length* $\ell \geq 0$ and a given *element type* `t`. The elements are indexed by the integers $0, 1, \ldots, \ell - 1$. The value of an expression of array type `u[]` is either `null` or a reference to an array whose element type `t` is a subtype of `u`. If `u` is a primitive type, then `t` must equal `u`.

8.1 Array Creation and Access

A new array of length ℓ with element type `t` is created (allocated) using an *array creation expression*:

```
new t[ℓ]
```

where ℓ is an expression of type `int`. If type `t` is a primitive type, all elements of the new array are initialized to 0 (when `t` is `byte`, `char`, `short`, `int`, or `long`) or 0.0 (when `t` is `float` or `double`) or `false` (when `t` is `boolean`). If `t` is a reference type, all elements are initialized to `null`.

If ℓ is negative, then the exception NegativeArraySizeException is thrown.

Let `a` be a reference of array type `u[]`, to an array with length ℓ and element type `t`. Then

- `a.length` of type `int` is the length ℓ of `a`, that is, the number of elements in `a`.
- The *array access* expression `a[i]` denotes element number `i` of `a`, counting from 0; this expression has type `u`. The integer expression `i` is called the *array index*. If the value of `i` is less than 0 or greater than or equal to `a.length`, then exception ArrayIndexOutOfBoundsException is thrown.
- When `t` is a reference type, every array element assignment `a[i] = e` checks that the value of `e` is `null` or a reference to an object whose class `C` is a subtype of the element type `t`. If this is not the case, then the exception ArrayStoreException is thrown. This check is made before every array element assignment at run-time, but only for reference types.

8.2 Array Initializers

A variable or field of array type may be initialized at declaration, using an existing array or an *array initializer* for the initial value. An array initializer is a comma-separated list of zero or more expressions enclosed in braces `{ ... }`:

```
t[] x = { expression, ..., expression }
```

The type of each *expression* must be a subtype of `t`. Evaluation of the initializer causes a distinct new array, whose length equals the number of expressions, to be allocated. Then the expressions are evaluated from left to right, their values are stored in the array, and finally the array is bound to `x`. Hence `x` cannot occur in the *expressions:* it has not yet been initialized when they are evaluated.

Array initializers may also be used in connection with array creation expressions:

```
new t[] { expression, ..., expression }
```

Multidimensional arrays can have nested initializers (example 22). Note that there are no array constants: a new distinct array is created every time an array initializer is evaluated.

Example 18 Creating and Using One-Dimensional Arrays
The first half of this example rolls a die 1,000 times, then prints the frequencies of the outcomes. The second half creates and initializes an array of String objects.

```
int[] freq = new int[6];                    // All initialized to 0
for (int i=0; i<1000; i++) {                // Roll dice, count frequencies
  int die = (int)(1 + 6 * Math.random());
  freq[die-1] += 1;
}
for (int c=1; c<=6; c++)
  System.out.println(c + " came up " + freq[c-1] + " times");

String[] number = new String[20];           // Create array of null elements
for (int i=0; i<number.length; i++)         // Fill with strings "A0", ..., "A19"
  number[i] = "A" + i;
for (int i=0; i<number.length; i++)         // Print strings
  System.out.println(number[i]);
```

Example 19 Array Element Assignment Type Check at Run-Time
This program compiles, but at run-time `a[2]=d` throws ArrayStoreException, since the class of the object bound to `d` (that is, Double) is not a subtype of `a`'s element type (that is, Integer).

```
Number[] a = new Integer[10];       // Length 10, element type Integer
Double d = new Double(3.14);        // Type Double,  class Double
Integer i = new Integer(117);       // Type Integer, class Integer
Number n = i;                       // Type Number,  class Integer
a[0] = i;                           // OK, Integer is subtype of Integer
a[1] = n;                           // OK, Integer is subtype of Integer
a[2] = d;                           // No, Double not subtype of Integer
```

Example 20 Using an Initialized Array
Method `checkdate` here behaves the same as `checkdate` in example 2. The array should be declared outside the method, as shown, otherwise a distinct new array is created for every call to the method.

```
static int[] days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
static boolean checkdate(int mth, int day)
{ return (mth >= 1) && (mth <= 12) && (day >= 1) && (day <= days[mth-1]); }
```

Example 21 Creating a String from a Character Array
When replacing character `c1` by character `c2` in a string, the result can be built in a character array because its length is known. This is 50 percent faster than example 105, which uses a string builder.

```
static String replaceCharChar(String s, char c1, char c2) {
  char[] res = new char[s.length()];
  for (int i=0; i<s.length(); i++)
    if (s.charAt(i) == c1)
      res[i] = c2;
    else
      res[i] = s.charAt(i);
  return new String(res);           // A string containing the characters of res
}
```

8.3 Multidimensional Arrays

The types of multidimensional arrays are written `t[][]`, `t[][][]`, and so on. A rectangular n-dimensional array of size $\ell_1 \times \ell_2 \times \cdots \times \ell_n$ is created (allocated) using the array creation expression

new t[ℓ_1][ℓ_2]...[ℓ_n]

A multidimensional array `a` of type `t[][]` is in fact a one-dimensional array of arrays; its component arrays have type `t[]`. Hence a multidimensional array need not be rectangular, and one need not create all the dimensions at once. To create only the first k dimensions of size $\ell_1 \times \ell_2 \times \cdots \times \ell_k$ of an n-dimensional array, leave the $(n-k)$ last brackets empty:

new t[ℓ_1][ℓ_2]...[ℓ_k][]...[]

To access an element of an n-dimensional array `a`, use n index expressions: a[i_1][i_2]...[i_n].

8.4 The Utility Class Arrays

Class Arrays from package `java.util` provides static utility methods to compare, fill, sort, and search arrays, and to create a collection (chapter 22) or stream (chapter 24) from an array. The `binarySearch`, `equals`, `fill`, `parallelSort`, and `sort` methods are overloaded also on arrays with element type `byte`, `char`, `short`, `int`, `long`, `float`, `double`, Object, and generic type parameter T; and the `equals` and `fill` methods also on type `boolean`. The Object versions of `binarySearch` and `sort` use the `compareTo` method of the array elements, unless an explicit Comparator object (section 22.9) is given.

- `static List<T> asList(T... a)` returns a List<T> view (section 22.2) of the elements of parameter array `a`, in index order. The resulting list implements interface RandomAccess.
- `static int binarySearch(byte[] a, byte k)` returns an index `i>=0` for which `a[i]==k`, if any; otherwise returns `i<0` such that `(-i-1)` would be the proper position for `k`. The array `a` must be sorted, as by `sort(a)`, or else the result is undefined.
- `static int binarySearch(Object[] a, Object k)` works like the preceding method but compares array elements using their `compareTo` method (section 22.9 and example 134).
- `static int binarySearch(Object[] a, Object k, Comparator cmp)` works like the preceding method but compares array elements using the method `cmp.compare` (section 22.9).
- `static boolean equals(byte[] a1, byte[] a2)` returns `true` if `a1` and `a2` have the same length and contain the same elements, in the same order.
- `static boolean equals(Object[] a1, Object[] a2)` works like the preceding method but compares array elements using their `equals` method (section 22.9).
- `static void fill(byte[] a, byte v)` sets all elements of `a` to `v`.
- `static void fill(byte[] a, int from, int to, byte v)` sets `a[from..(to-1)]` to `v`.

Example 22 Creating Multidimensional Arrays
Consider this rectangular 3-by-2 array and this two-dimensional “jagged” (lower triangular) array:

```
0.0  0.0          0.0
0.0  0.0          0.0  0.0
0.0  0.0          0.0  0.0  0.0
```

The following program shows two ways (`r1`, `r2`) to create the rectangular array, and three ways (`t1`, `t2`, `t3`) to create the “jagged” array:

```
double[][] r1 = new double[3][2];
double[][] r2 = new double[3][];
for (int i=0; i<3; i++)
  r2[i] = new double[2];

double[][] t1 = new double[3][];
for (int i=0; i<3; i++)
  t1[i] = new double[i+1];
double[][] t2 = { { 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0, 0.0 } };
double[][] t3 = new double[][] { { 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0, 0.0 } };
```

Example 23 Using Multidimensional Arrays
The genetic material of living organisms is held in DNA, conceptually a string AGCTTTTCA of nucleotides A, C, G, and T. A triple of nucleotides, such as AGC, is called a codon; a codon may code for an amino acid. This program counts the frequencies of the $4 \cdot 4 \cdot 4 = 64$ possible codons, using a three-dimensional array `freq`. The auxiliary array `fromNuc` translates from the nucleotide letters (A,C,G,T) to the indexes (0,1,2,3) used in `freq`. The array `toNuc` translates from indexes to nucleotide letters when printing the frequencies.

```
static void codonfreq(String s) {
  int[] fromNuc = new int[128];
  for (int i=0; i<fromNuc.length; i++)
    fromNuc[i] = -1;
  fromNuc['a'] = fromNuc['A'] = 0; fromNuc['c'] = fromNuc['C'] = 1;
  fromNuc['g'] = fromNuc['G'] = 2; fromNuc['t'] = fromNuc['T'] = 3;
  int[][][] freq = new int[4][4][4];
  for (int i=0; i+2<s.length(); i+=3) {
    int nuc1 = fromNuc[s.charAt(i)];
    int nuc2 = fromNuc[s.charAt(i+1)];
    int nuc3 = fromNuc[s.charAt(i+2)];
    freq[nuc1][nuc2][nuc3] += 1;
  }
  final char[] toNuc = { 'A', 'C', 'G', 'T' };
  for (int i=0; i<4; i++)
    for (int j=0; j<4; j++) {
      for (int k=0; k<4; k++)
        System.out.print(" "+toNuc[i]+toNuc[j]+toNuc[k]+": " + freq[i][j][k]);
      System.out.println();
    }
}
```

9 Classes

9.1 Class Declarations and Class Bodies

A *class-declaration* of class `C` has the form

> *class-modifiers* `class C` *extends-clause implements-clause*
>   *class-body*

A declaration of class `C` introduces a new reference type `C`. The *class-body* may contain declarations of fields, constructors, methods, nested classes, nested interfaces, and initializer blocks. A class declaration may take type parameters and be generic; see section 21.4. The declarations in a class may appear in any order:

```
{
  field-declarations
  constructor-declarations
  method-declarations
  class-declarations
  interface-declarations
  enum-type-declaration
  initializer-blocks
}
```

A field, method, nested class, nested interface, or nested enum type is called a *member* of the class. A member may be declared `static`. A non-static member is also called an *instance member*.

The scope of a member is the entire class body, except where shadowed by a variable or parameter or by a member of a nested class or interface. The scope of a (static) field does not include (static) initializers preceding its declaration, but the scope of a static field does include all non-static initializers. There can be no two nested classes, interfaces, or enum types with the same name, and no two fields with the same name, but a field, a method, and a class (or interface or enum type) may have the same name.

By *static code* we mean expressions and statements in static field initializers, static initializer blocks, and static methods. By *non-static code* we mean expressions and statements in constructors, non-static field initializers, non-static initializer blocks, and non-static methods. Non-static code is executed inside a *current object*, which can be referred to as `this` (section 11.10). Static code cannot refer to non-static members or to `this`, only to static members.

9.2 Top-Level Classes, Nested Classes, Member Classes, and Local Classes

A *top-level class* is a class declared outside any other class or interface declaration. A *nested class* is a class declared inside another class or interface. There are two kinds of nested classes: a *local class* is declared inside a method, constructor, or initializer block; a *member class* is not. A non-static member class, or a local class in a non-static member, is called an *inner class*, because an object of the inner class will contain a reference to an object of the enclosing class. See also section 9.11.

9.3 Class Modifiers

For a top-level class, the *class-modifiers* may be a list of `public` and at most one of `abstract` or `final`. For a member class, they may be a list of `static`, at most one of `abstract` or `final`, and at most one of `private`, `protected`, or `public`. For a local class, they may be at most one of `abstract` or `final`.

Example 27 Class Declaration
The Point class is declared to have two non-static fields `x` and `y`, one constructor, and two non-static methods. It is used in examples 12 and 52.

```
class Point {
  int x, y;

  Point(int x, int y) { this.x = x; this.y = y; }

  void move(int dx, int dy) { x += dx; y += dy; }

  public String toString() { return "(" + x + ", " + y + ")"; }
}
```

Example 28 Class with Static and Non-static Members
The SPoint class declares a static field `allpoints` and two non-static fields `x` and `y`. Thus each SPoint object has its own `x` and `y` fields, but all objects share the same `allpoints` field in the SPoint class.

The constructor inserts the new object (`this`) into the ArrayList object `allpoints` (section 22.2). The non-static method `getIndex` returns the point's index in the array list. The static method `getSize` returns the number of SPoints created so far. The static method `getPoint` returns the `i`'th SPoint in the array list. Class SPoint is used in example 59.

```
class SPoint {
  static ArrayList<SPoint> allpoints = new ArrayList<SPoint>();
  int x, y;

  SPoint(int x, int y) { allpoints.add(this); this.x = x; this.y = y; }
  void move(int dx, int dy) { x += dx; y += dy; }
  public String toString() { return "(" + x + ", " + y + ")"; }
  int getIndex() { return allpoints.indexOf(this); }
  static int getSize() { return allpoints.size(); }
  static SPoint getPoint(int i) { return allpoints.get(i); }
}
```

Example 29 Top-Level, Member, and Local Classes
See also examples 42 and 47.

```
class TLC {                                       // Top-level class TLC
  static class SMC { ... }                        // Static member class

  class NMC { ... }                               // Non-static member (inner) class

  void nm() {                                     // Non-static method in TLC
    class NLC { ... }                             // Local (inner) class in method
  }

  static void sm() {                              // Static method in TLC
    class SLC { ... }                             // Local class in method
  }
}
```