Java Precisely

Version 1.05 of 2000-11-23

Peter Sestoft

 • sestoft@dina.kvl.dk

IT University of Copenhagen, Denmark
and
Royal Veterinary and Agricultural University, Copenhagen, Denmark

This document gives a concise description of the Java programming language, version 1.1 and later. It is a quick reference for the reader who has already learnt (or is learning) Java from a standard textbook and who wants to know the language in more detail.

The document presents general rules (on left-hand pages), and corresponding examples (on right-hand pages). All examples are fragments of legal Java programs. The complete examples are available at the book website; see below.

This document is available at http://www.dina.kvl.dk/~sestoft/javaprecisely/

Contents

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 syntax (grammar) for Java programs, that operators (such as +) are applied to the correct type of operands (such as `5` and `x`), etc. If so, the compiler generates so-called *class files*. Execution then starts by loading the needed 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 program code).

2 Names and reserved names

A legal *name* (of a variable, method, field, parameter, class, or interface) must start with a letter or dollar sign ($) or underscore (_), and continue with zero or more letters or dollar signs or underscores or digits (0–9). Do not use dollar signs in class names. Java is case sensitive: upper case letters are distinguished from lower case letters. A legal name cannot be one of the following *reserved names*:

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

3 Java naming conventions

The following naming conventions are often followed in Java programs, although not enforced by the compiler:

- If a name is composed of several words, then each word (except possibly the first one) begins with an upper case letter. Examples: `setLayout`, `addLayoutComponent`.
- Names of variables, fields, and methods begin with a lower case letter. Examples: `vehicle`, `currentVehicle`.
- Names of classes and interfaces begin with an upper case letter. Examples: `Layout`, `FlowLayout`.
- Named constants (that is, `final` variables and fields) are written entirely in upper case (and the parts of composite names are separated by underscores _). Examples: `CENTER`, `MAX_VALUE`.
- A package name is a sequence of dot-separated lower case names. Example: `java.awt.event`.

4 Comments and program lay-out

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 lay-out 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 is a delimited comment, extending over part of a line */ x = 117;
}
```

Example 2 Program lay-out style

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

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

  int sum(int b) {                      // Multi-line method body
    if (a > 0)                          // If statement
      return a + b;                     // Single statement
    else if (a < 0) {                   // Nested if-else, block statement
      int res = -a + b;
      return res * 117;
    } else { // a == 0                  // Terminal else, block statement
      int sum = 0;
      for (int i=0; i<10; i++)          // For loop
        sum += (b - i) * (b - 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 base type or a reference type.

5.1 Base types

A *base type* is either `boolean`, or one of the *numeric* types `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. The base types, example literals, size in bits, and value range are shown below:

Type	Kind	Example literals	Size	Range
`boolean`	logical	`false`, `true`	1	
`char`	integer	`' '`, `'0'`, `'A'`, ...	16	`\u0000` ... `\uFFFF` (unsigned)
`byte`	integer	`0`, `1`, `-1`, `117`, ...	8	$max = 127$
`short`	integer	`0`, `1`, `-1`, `117`, ...	16	$max = 32767$
`int`	integer	`0`, `1`, `-1`, `117`, ...	32	$max = 2147483647$
`long`	integer	`0L`, `1L`, `-1L`, `117L`, ...	64	$max = 9223372036854775807$
`float`	floating-point	`-1.0f`, `0.499f`, `3E8f`, ...	32	$\pm 10^{-38} \ldots \pm 10^{38}$, sigdig 6–7
`double`	floating-point	`-1.0`, `0.499`, `3E8`, ...	64	$\pm 10^{-308} \ldots \pm 10^{308}$, sigdig 15–16

The integer types are exact within their range. They use signed two'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 above. For special character escape sequences, see page 10.

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

	Base	Distinction	Example integer literals
Decimal	10	No leading `0`	`1234567890`, `127`, `-127`
Octal	8	Leading `0`	`01234567`, `0177`, `-0177`
Hexadecimal	16	Leading `0x`	`0xABCDEF0123`, `0x7F`, `-0x7F`

For all base types there are corresponding classes (reference types), namely `Boolean` and `Character` as well as `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, where the last six have the common superclass `Number`.

5.2 Reference types

A *reference type* is a class type, an interface type, or an array type. A class type is defined by a class declaration (Section 9.1); an interface type is defined by an interface declaration (Section 13.1); array types are discussed in Section 5.3 below.

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 is a reference to an array whose element type is precisely `t` (when `t` is a base type), or is a subtype of `t` (when `t` is a reference type).

5.4 Subtypes and compatibility

A type `t1` may be a *subtype* of a type `t2`, in which case `t2` is a *supertype* of `t1`. Intuitively this means that any value `v1` of type `t1` can be used where a value of type `t2` is expected. When `t1` and `t2` are reference types, `t1` must provide at least the functionality (methods and fields) provided by `t2`. In particular, any value `v1` of type `t1` may be bound to a variable or field or parameter `x2` of type `t2`, e.g. by the assignment `x2 = v1` or by parameter passing. We also say that types `t1` and `t2` are *compatible*. The following rules determine when a type `t1` is a subtype of a type `t2`:

- Every type is a subtype of itself.
- If `t1` is a subtype of `t2` and `t2` is a subtype of `t3`, then `t1` is a subtype of `t3`.
- `char` is a subtype of `int`, `long`, `float`, and `double`.
- `byte` is a subtype of `short`, `int`, `long`, `float`, and `double`.
- `short` is a subtype of `int`, `long`, `float`, and `double`.
- `int` is subtype of `long`, `float`, and `double`.
- `long` is a subtype of `float` and `double`.
- `float` is a subtype of `double`.

Let `t1` and `t2` be reference types.

- If `t1` and `t2` are classes, then `t1` is a subtype of `t2` if `t1` is a subclass of `t2`.
- If `t1` and `t2` are interfaces, then `t1` is a subtype of `t2` if `t1` is a subinterface of `t2`.
- If `t1` is a class and `t2` is an interface, then `t1` is a subtype of `t2` provided that `t1` (is a subclass of a class that) implements `t2` or implements a subinterface of `t2`.
- Array type `t1[]` is a subtype of array type `t2[]` if type `t1` is a subtype of type `t2`.
- Any reference type `t`, including any array type, is also a subtype of predefined class `Object`.

No base type is a subtype of a reference type, and no reference type is a subtype of a base type.

5.5 Signatures and subsumption

A *signature* has form $\mathtt{m}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ where `m` is the name of a method or constructor, and $(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is a list of types; see Example 22. When the method is declared in class `T`, not inherited from a superclass, then its *extended signature* is $\mathtt{m}(\mathtt{T}, \mathtt{t}_1, \ldots, \mathtt{t}_n)$; it is used in method calls (Section 11.10).

We say that a signature $sig_1 = \mathtt{m}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ *subsumes* signature $sig_2 = \mathtt{m}(\mathtt{u}_1, \ldots, \mathtt{u}_n)$ if each $\mathtt{u}_i$ is a subtype of $\mathtt{t}_i$. We also say that sig_2 is *more specific* than sig_1. Note that the method name `m` and the number n of types must be the same in the two signatures. Since every type $\mathtt{t}_i$ is a subtype of itself, every signature subsumes itself. In a collection of signatures there may be one which is subsumed by all others; such as signature is called the *most specific* signature. Examples:

- `m(double,double)` subsumes itself and `m(double,int)` and `m(int,double)` and `m(int,int)`
- `m(double,int)` subsumes itself and `m(int,int)`
- `m(int,double)` subsumes itself and `m(int,int)`
- `m(double,int)` does not subsume `m(int,double)`, nor the other way round
- the collection `m(double,int)`, `m(int,int)` has the most specific signature `m(int,int)`
- the collection `m(double,int)`, `m(int,double)` has no most specific signature

6 Variables, parameters, fields, and scope

A *variable* is declared inside a method, constructor or initializer block, or more generally, inside a block statement (see 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, and only after its declaration.

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 the class, also textually before its declaration.

6.1 Values bound to variables, parameters, or fields

A variable, parameter or field of *base type* always 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 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*`, ... ;`

The *variable-modifier* may be `final`, or absent. If a variable is declared `final`, then it must be initialized or assigned at most once at runtime (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 any expression, or an array initializer (Section 8.3).

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 inner-most 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; see Section 9.1).

Example 3 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 4 Scope of fields, parameters, and variables

```
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`. A string literal is a sequence of characters within double quotes: `"New York"`, `"B52"`, and so on. Internally, a character is stored as a number using the Unicode character encoding, whose character codes 0–127 coincide with the ASCII character 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*
`\udddd`	the character whose character code is the four-digit hexadecimal number *dddd*

Each character escape sequence represents a single character. For example, the letter A has code 65 (decimal), which is written 101 in octal and 0041 in hexadecimal, so the string literal `"A\101\u0041"` is the same as `"AAA"`.

If `s1` and `s2` are expressions of type `String` 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, `false` otherwise.
- `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 `s1.length()`, then the exception `StringIndexOutOfBoundsException` is thrown.
- `s1.concat(s2)` of type `String` is a new string consisting of the characters of `s1` followed by the characters of `s2`.
- `s1.toString()` of type `String` is just `s1` itself.
- `s1 + v` is the same as `s1.concat(Integer.toString(v))` when `v` has type `int`, and similarly for the other base types (Section 5.1).
- `s1 + v` is the same as `s1.concat(v.toString())` when `v` has reference type and `v` is not `null`; and the same as `s1.concat("null")` when `v` is `null`. In particular, `s1 + s2` is the same as `s1.concat(s2)` when `s2` is not `null`. Any class `C` will inherit a default `toString` method from class `Object` (which produces strings of the form `C@265734`), but class `C` may override (redefine) it by declaring a method `public String toString()` to produce more useful 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 character encoding. If `s1` or `s2` is `null`, then the exception `NullPointerException` is thrown.
- more `String` methods are described in the Java class library String section; see Section 18.

Example 5 Equality of strings

```
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)));
```

Example 6 Concatenate all command line arguments

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

Example 7 Count 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 8 Determine 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 9 Using a class that declares a `toString` method

The class `Point` (Example 13) declares a `toString` method which returns a string of the point coordinates. Below 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)
```

8 Arrays

An *array* is a collection of variables, called *elements*. An array has a given *length* ℓ 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 base 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 base type, then 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, then all elements are initialized to `null`.

If `a` has type `u[]` and is a reference to an array with length ℓ and element type `t` then:

- `a.length` is the length ℓ of `a`, that is, the number of elements in `a`.
- the *array access* `a[i]` denotes element number `i` of `a`, counting from 0; it has type `u`. The integer expression `i` is called the *array index*. If the value of `i` is less than 0 or greater or equal to `a.length`, then the exception `ArrayIndexOutOfBoundsException` is thrown.
- when `t` is a reference type, then 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 runtime, but only for reference types.

8.2 Multi-dimensional arrays

The types of multi-dimensional arrays are written `t[][]`, `t[][][]`, etc. 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 multi-dimensional array `a` of type `t[][]` is in fact a one-dimensional array of arrays; its component arrays have type `t[]`. Hence a multi-dimensional array need not be rectangular, and one need not create all the dimensions at once. To create 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].

8.3 Array initializers

A variable or field of array type may be initialized at declaration, using an existing array or an *array initializer*. An array initializer is a comma-separated list of expressions enclosed in braces `{ ... }`. Array initializers can be used only in connection with initialized variable or field declarations. Multi-dimensional arrays can have nested initializers.

Example 10 Creating and using one-dimensional arrays

```
// Roll a die, count frequencies
int[] freq = new int[6];                    // all initialized to 0
for (int i=0; i<1000; i++) {
  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");

// Create an array of the strings "A0", "A1", ..., "A19"
String[] number = new String[20];           // all initialized to null
for (int i=0; i<number.length; i++)
  number[i] = "A" + i;
for (int i=0; i<number.length; i++)
  System.out.println(number[i]);

// Throws ArrayStoreException: Double is not a subtype of 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 11 Using an initialized array
Method `checkdate` below behaves the same as `checkdate` in Example 2.

```
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 12 Multi-dimensional arrays

```
// Create a lower triangular array of the form
// 0.0
// 0.0 0.0
// 0.0 0.0 0.0

final int SIZE = 3;
double[][] a = new double[SIZE][];
for (int i=0; i<SIZE; i++)
  a[i] = new double[i+1];

// Use a nested array initializer to create an array similar to the above
double[][] b = { { 0.0 }, { 0.0, 0.0 }, { 0.0, 0.0, 0.0 } };
```

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*
classbody

A declaration of class `C` introduces a new reference type `C`. The *classbody* may contain declarations of fields, constructors, methods, nested classes, nested interfaces, and initializer blocks. The declarations in a class may appear in any order:

{
field-declarations
constructor-declarations
method-declarations
class-declarations
interface-declarations
initializer-blocks
}

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

The scope of a member is the entire class body, except where shadowed by a variable or parameter or field of a nested class or interface. However, the scope of a static field does not include static initializers preceding its declaration (but does include all non-static initializers), and the scope of a non-static field does not include non-static initializers preceding its declaration.

By *static code* we denote expressions and statements in static field initializers, static initializer blocks, and static methods. By *non-static code* we denote 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`; see Section 11.9. 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 or 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 any object of the inner class will contain a reference to an object of the enclosing class.

For more on nested classes, see 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` and `final`. For a member class, the *class-modifiers* may be a list of `static`, and at most one of `abstract` and `final`, and at most one of `private`, `protected` and `public`. For a local class, the *class-modifiers* is a list of at most one of `abstract` and `final`.

Example 13 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 Example 38.

```
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 14 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`) in the `java.util.Vector` object `allpoints`. The non-static method `getIndex` returns the point's index in the vector. The static method `getSize` returns the number of `SPoints` created so far. The static method `getPoint` returns the `i`'th `SPoint` in the vector. The class is used in Example 45.

```
class SPoint {
  static Vector allpoints = new Vector();
  int x, y;

  SPoint(int x, int y) { allpoints.addElement(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 (SPoint)allpoints.elementAt(i); }
}
```

Example 15 Top-level, member, and local classes

```
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 class in method
  }
}
```

9.4 The class modifiers `public`, `final`, and `abstract`

If a top-level class `C` is declared `public`, then it is accessible outside its package; see Section 17.

If a class `C` is declared `final`, then one cannot declare subclasses of `C`, and hence cannot override any methods declared in `C`. This is useful for preventing rogue subclasses from violating data representation invariants.

If a class `C` is declared `abstract`, then it cannot be instantiated, but non-abstract subclasses of `C` can be instantiated. An abstract class may declare constructors and have initializers, to be executed when instantiating concrete subclasses. An abstract class may declare abstract and non-abstract methods; a non-abstract class cannot declare abstract methods. A class cannot be both `abstract` and `final`, because no objects could be created of that class.

9.5 Subclasses, superclasses, class hierarchy, inheritance and overriding

A class `C` may be declared as a *subclass* of class `B` by an *extends-clause* of the form

```
class C extends B { ... }
```

In this case, `C` is a subclass (and hence subtype, see Section 5.4) of `B` and its superclasses. Class `C` inherits all methods and fields (even private ones, although they are not accessible in class `C`), but not the constructors, from `B`.

Class `B` is called the *immediate superclass* of `C`. A class can have at most one immediate superclass. The predefined class `Object` is a superclass of all other classes; class `Object` has no superclass. Hence the classes form a *class hierarchy* in which every class is a descendant of its immediate superclass, except `Object` which is at the top.

A constructor in subclass `C` may, as its very first action, explicitly call a constructor in the immediate superclass `B`, using the syntax

`super(`*actual-list*`);`

A superclass constructor call `super(...)` may appear only at the very beginning of a constructor.

If a constructor `C(...)` in subclass `C` does not explicitly call `super(...)` as its first action, then it implicitly calls the argumentless default constructor `B()` in superclass `B` as its first action, as if by `super()`. In this case `B` must have a non-private argumentless constructor `B()`. Conversely, if there is no argumentless constructor `B()` in `B`, then `C(...)` in `C` must use `super(...)` to explicitly call some other constructor in `B`.

The declaration of `C` may *override* (redeclare) any non-final method `m` inherited from `B` by declaring a new method `m` with the same signature. The overridden `B`-method `m` can be referred to as `super.m` inside `C`'s constructors, non-static methods, and initializers of non-static fields. The overriding method `m` in `C`:

- must be at least as accessible (Section 9.7) as the overridden method in `B`;
- must have the same returntype as the overridden method in `B`;
- must be static if and only if the overridden method in `B` is static;
- either has no *throws-clause*, or has a *throws-clause* that covers no more exception classes than the *throws-clause* (if any) of the overridden method in `B`.

However, the declaration of a class `C` cannot redeclare a field `f` inherited from `B`, only declare an additional field of the same name; see Section 9.6. The overridden `B`-field can be referred to as `super.f` inside `C`'s constructors, non-static methods, and non-static initializers.

Example 16 Abstract classes, subclasses, and overriding
The abstract class `Vessel` models the notion of a vessel (for holding liquids): it has a field `contents` representing its actual contents, it has an abstract method `capacity` for computing its maximal capacity, and it has a method for filling in more, but only up to its capacity (the excess will be lost). The abstract class has subclasses `Tank` (a rectangular vessel), `Cube` (a cubic vessel, subclass of `Tank`) and `Barrel` (a cylindric vessel).

The subclasses implement the `capacity` method, they inherit the `contents` field and the `fill` method from the superclass, and they override the `toString` method (inherited from class `Object`) to print each vessel object appropriately:

```
abstract class Vessel {
  double contents;
  abstract double capacity();
  void fill(double amount) { contents = Math.min(contents + amount, capacity()); }
}
class Tank extends Vessel {
  double length, width, height;
  Tank(double length, double width, double height)
  { this.length = length; this.width = width; this.height = height; }
  double capacity() { return length * width * height; }
  public String toString()
  { return "tank (" + length + ", " + width + ", " + height + ")"; }
}
class Cube extends Tank {
  Cube(double side) { super(side, side, side); }
  public String toString() { return "cube (" + length + ")"; }
}
class Barrel extends Vessel {
  double radius, height;
  Barrel(double radius, double height) { this.radius = radius; this.height = height; }
  double capacity() { return height * Math.PI * radius * radius; }
  public String toString() { return "barrel (" + radius + ", " + height + ")"; }
}
```

Example 17 Using the vessel hierarchy from Example 16
The call `vs[i].capacity()` is legal only because the method `capacity`, although abstract, is declared in class `Vessel` (Example 16):

```
public static void main(String[] args) {
  Vessel v1 = new Barrel(3, 10);
  Vessel v2 = new Tank(10, 20, 12);
  Vessel v3 = new Cube(4);
  Vessel[] vs = { v1, v2, v3 };
  v1.fill(90); v1.fill(10); v2.fill(100); v3.fill(80);
  double sum = 0;
  for (int i=0; i<vs.length; i++)
    sum += vs[i].capacity();
  System.out.println("Total capacity is " + sum);
  for (int i=0; i<vs.length; i++)
    System.out.println("vessel number " + i + ": " + vs[i]);
}
```

9.6 Field declarations in classes

The purpose of a *field* is to hold a value inside an object (if non-static) or a class (if static). A field must be declared in a class declaration. A *field-declaration* has one of the forms:

> *field-modifiers type fieldname1*, *fieldname2*, ... ;
> *field-modifiers type fieldname1* = *initializer1*, ... ;

The *field-modifiers* is a list of the modifiers `static` and `final`, and at most one of the access modifiers `private`, `protected`, and `public` (see Section 9.7).

If a field `f` in class `C` is declared `static`, then `f` is associated with the class `C` and can be referred to independently of any objects of class `C`. The field can be referred to as `C.f` or `o.f` where `o` is an expression of type `C`, or, in the declaration of `C`, as `f`. If a field `f` in class `C` is not declared `static`, then `f` is associated with an object (instance) of class `C`, and every instance has its own copy of the field. The field can be referred to as `o.f` where `o` is an expression of type `C`, or, in non-static code in the declaration of `C`, as `f`.

If a field `f` in class `C` is declared `final`, then the field cannot be modified after initialization. If `f` has reference type and points to an object or array, then the object's fields or the array's elements may still be modified. The initialization must happen either in the declaration or in an initializer block (Section 9.10), or (if the field is non-static) precisely once in every constructor in class `C`.

A field *initializer* may be any expression, or an array initializer (Section 8.3). A static field initializer can refer only to static members of `C`, and can throw no checked exceptions (Section 14).

A field is given a *default initial value* depending on its declared type `t`. If `t` is a basetype, then the field is 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, then the field is initialized to `null`.

Static fields are initialized when the class is loaded. First all static fields are given their default initial values, then the static initializer blocks (Section 9.10) and static field initializers are executed, in order of appearance in the class declaration.

Non-static fields are initialized when a constructor is called to create an object (instance) of the class, at which time all static fields have been initialized already; see Section 9.9.

If a class `C` declares a non-static field `f`, and `C` is a subclass of a class `B` that has a non-static field `f`, then every object of class `C` has two fields both called `f`: one is the `B`-field `f` declared in the superclass `B`, and one is the `C`-field `f` declared in `C` itself. What field is referred to by a field access `o.f` is determined by the type of `o`; see Section 11.8.

9.7 Member access modifiers: `private`, `protected`, `public`

A member (field or method or nested class or interface) is always accessible in the class in which it is declared, except where shadowed by a variable or parameter or a field (of a nested class). The *access modifiers* `private`, `protected` and `public` determine where else the member is accessible.

If a member is declared `private` in top-level class `C` or a nested class within `C`, then it is accessible in `C` and its nested classes, but not in their subclasses outside `C`, nor in other classes. If a member in class `C` is declared `protected`, then it is accessible in all classes in the same package (see Section 17) as `C`, and in subclasses of `C`, but not in non-subclasses in other packages. If a member in class `C` is not declared `private`, `protected`, or `public`, then it has *package access*, or *default access*, and is accessible only in classes within the same package as `C`, not in classes in other packages. If a member in class `C` is declared `public`, then it is accessible in all classes, including classes in other packages. Thus, in order of increasing accessibility, we have `private` access, package access, `protected` access, and `public` access.

Example 18 Field declarations

The `SPoint` class (Example 14) declares a static field `allpoints` and two non-static fields `x` and `y`.

Example 27 declares a static field `ps` of array type `double[]`. Its field initializer allocates a six-element array and binds it to `ps`, and then the initializer block (Section 9.10) stores some numbers into the array.

The `Barrel` class in Example 73 declares two non-static fields `radius` and `height`. The fields are final, and therefore must be initialized (which is done in the constructor).

Example 19 Several fields with the same name

An object of class `C` below has two non-static fields called `vf`: one declared in the superclass `B`, and one declared in `C` itself. Similarly, an object of class `D` has three non-static fields called `vf`. Class `B` and class `C` each have a static field called `sf`. Class `D` does not declare a static field `sf`, so in class `D` the name `sf` refers to the static field `sf` in the superclass `C`. Example 42 uses these classes.

```
class B                          // one non-static field vf, one static sf
{ int vf; static int sf; B(int i) { vf = i; sf = i+1; } }

class C extends B                // two non-static fields vf, one static sf
{ int vf; static int sf; C(int i) { super(i+20); vf = i; sf = i+2; } }

class D extends C                // three non-static fields vf
{ int vf; D(int i) { super(i+40); vf = i; sf = i+4; } }
```

Example 20 Member access modifiers

The vessel hierarchy in Example 16 is unsatisfactory because everybody can read and modify the fields of a vessel object. Example 73 presents an improved version of the hierarchy in which (1) the `contents` field in `Vessel` is made private to prevent modification, (2) a new public method `getContents` permits reading the field, and (3) the fields of `Tank` and `Barrel` are declared protected to permit access from subclasses declared in other packages.

Since the field `contents` in `Vessel` is private, it is not accessible in the subclasses (`Tank`, `Barrel`, ...), but the subclasses still inherit the field. Thus every vessel subclass object has room for storing the field, but can change and access it only by using the methods `fill` and `getContents` inherited from the abstract superclass.

Example 21 A private member is accessible in the enclosing top-level class

A private member is accessible everywhere inside the enclosing top-level class (and only there):

```
class Access {
  private static int x;
  static class SI {
    private static int y = x;       // access private x from enclosing class
  }
  static void m() {
    int z = SI.y;                   // access private y from nested class
  }
}
```

9.8 Method declarations

A *method* must be declared inside a class. A *method-declaration* declaring method `m` has the form:

method-modifiers returntype `m(`*formal-list*`)` *throws-clause*
method-body

The *formal-list* is a comma-separated list of *formal parameter declarations*, each of the form

parameter-modifier type parametername

The *parameter-modifier* may be `final`, meaning that the parameter cannot be modified inside the method, or absent. The *type* is any type. The *parametername* is any legal name. A formal parameter is similar to an initialized variable; its scope is the *method-body*.

The method name `m` together with the list $t_1, \ldots, t_n$ of declared parameter types in the *formal-list* determine the *method signature* $m(t_1, \ldots, t_n)$. The *returntype* is not part of the method signature.

A class may declare more than one method with the same *methodname*, provided they have different method signatures. This is called *overloading* of the *methodname*.

The *method-body* is a *block-statement* (Section 12.2), and thus may contain statements as well as declarations of variables and local classes.

In particular, the *method-body* may contain `return` statements. If the *returntype* is `void`, then the method does not return a value, and no `return` statement in the *method-body* can have an expression argument. If the *returntype* is not `void`, then the method must return a value: it must not be possible for execution to reach the end of *method-body* without executing a `return` statement. Moreover, every `return` statement must have an expression argument whose type is a subtype of the *returntype*.

The *method-modifiers* is a list of the modifiers `static`, `final`, `abstract`, `synchronized` (Section 15.2), and at most one of the access modifiers `private`, `protected`, or `public` (Section 9.7).

If a method `m` in class `C` is declared `static`, then `m` is associated with the *class* `C`; it can be referred without any object of class `C`. The method may be called as `C.m(...)` or as `o.m(...)` where `o` is an expression whose type is a subtype of `C`, or, inside methods, constructors, field initializers and initializer blocks in `C`, simply as `m(...)`. A static method can refer only to static fields and methods of the class.

If a method `m` in class `C` is not declared `static`, then `m` is associated with an *object* (instance) of class `C`. Outside the class, the method must be called as `o.m(...)` where `o` is an object of class `C` or a subclass, or, inside non-static methods, non-static field initializers and non-static initializer blocks in `C`, simply as `m(...)`. A non-static method can refer to all fields and methods of class `C`, whether they are static or not.

If a method `m` in class `C` is declared `final`, then the method cannot be overridden (redefined) in subclasses of `C`.

If a method `m` in class `C` is declared `abstract`, then class `C` must itself be abstract (and so cannot be instantiated). An abstract method declaration has this form, without a method body:

`abstract` *method-modifiers returntype* `m(`*formal-list*`)` *throws-clause*`;`

The *throws-clause* of a method or constructor has the form

```
throws E1, E2, ...
```

where `E1`, `E2`, ... are the names of exception types covering all the checked exceptions that the method or constructor may throw. More precisely, for every exception `e` that execution may throw, either `e` is an unchecked exception (see Section 14), or it is a checked exception whose class is a subtype of one of `E1`, `E2`, ...

Example 22 Method overloading and signatures
This class declares four overloaded methods `m` whose signatures (Section 5.5) are `m(int)` and `m(boolean)` and `m(int, double)` and `m(double, double)`. Some of the overloaded methods are static, others non-static. The overloaded methods may have different return types, as shown here. Example 47 explains the method calls.

It would be legal to declare an additional method with signature `m(double, int)`, but then the method call `m(10, 20)` would become ambiguous and illegal. Namely, there is no way to determine whether to call `m(int, double)` or `m(double, int)`.

```
class Overloading {
  double m(int i) { return i; }
  boolean m(boolean b) { return !b; }
  static double m(int x, double y) { return x + y + 1; }
  static double m(double x, double y) { return x + y + 3; }

  public static void main(String[] args) {
    System.out.println(m(10, 20));              // Prints 31.0
    System.out.println(m(10, 20.0));            // Prints 31.0
    System.out.println(m(10.0, 20));            // Prints 33.0
    System.out.println(m(10.0, 20.0));          // Prints 33.0
  }
}
```

Example 23 Method overriding
In the vessel hierarchy (Example 16), the classes `Tank` and `Barrel` override the method `toString` inherited from the universal superclass `Object`, and class `Cube` overrides `toString` inherited from class `Tank`.

Example 24 Method overriding and overloading
The class `C1` declares the overloaded method `m1` with signatures `m1(double)` and `m1(int)`, and the method `m2` with signature `m2(int)`. The subclass `C2` hides `C1`'s method `m1(double)` and overloads `m2` by declaring an additional variant. Calls to these methods are shown in Example 48.

```
class C1 {
  static void m1(double d) { System.out.println("11d"); }
  void m1(int i) { System.out.println("11i"); }
  void m2(int i) { System.out.println("12i"); }
}

class C2 extends C1 {
  static void m1(double d) { System.out.println("21d"); }
  void m1(int i) { System.out.println("21i"); }
  void m2(double d) { System.out.println("22d"); }
}
```

9.9 Constructor declarations

The purpose of a constructor in class `C` is to create and initialize new objects (instances) of the class. A *constructor-declaration* in class `C` has the form:

> *constructor-modifiers* `C(`*formal-list*`)` *throws-clause*
> *constructor-body*

The *constructor-modifiers* is a list of at most one of `private`, `protected`, and `public` (Section 9.7); a constructor cannot be `abstract`, `final`, or `static`. The return type of a constructor need not and cannot be specified: by definition, a constructor in class `C` returns an object of class `C`.

Constructors may be overloaded in the same way as methods: the *constructor signature* (a list of the parameter types in *formal-list*) is used to distinguish constructors in the same class. A constructor may call another overloaded constructor in the same class using the syntax:

> `this(`*actual-list*`)`

but a constructor may not call itself, directly or indirectly. A call `this(...)` to another constructor, if present, must be the very first action of a constructor, preceding any declaration, statement, etc.

The *constructor-body* is a *block-statement* (Section 12.2) and so may contain statements as well as declarations of variables and local classes. The *constructor-body* may contain `return` statements, but no `return` statement can take an expression argument.

A class which does not explicitly declare a constructor, implicitly declares a public, argumentless *default constructor* whose only (implicit) action is to call the superclass constructor (Section 9.5):

```
public C() { super(); }
```

The *throws-clause* of the constructor specifies the checked exceptions that may be thrown by the constructor, in the same manner as for methods; see Section 9.8.

When a constructor is called (Section 11.6) to create an object, the following happens: first an object of the class is created in the memory, then the non-static fields are given default initial values according to their type, then some superclass constructor is called (explicitly or implicitly) exactly once, then the non-static field initializers and non-static initializer blocks are executed once, in order of appearance in the class declaration, and finally the constructor body is executed.

9.10 Initializer blocks, field initializers and initializers

In addition to field initializers (Section 9.6), a class may contain zero or more *initializer-blocks*. Initializer blocks are typically used only when field initializers or constructors do not suffice. We use the term *initializer* to denote field initializers as well as initializer blocks. A *static initializer block* has the form

> `static` *block-statement*

The static initializer blocks and field initializers of static fields are executed, in order of appearance in the class declaration, when the class is loaded. A *non-static initializer block* has the form

> *block-statement*

Non-static initializer blocks are executed when an object is created; see Section 9.9 above.

An initializer is not allowed to throw a checked exception (Section 14). If execution of a static initializer throws an (unchecked) exception during class loading, then that exception is discarded and the exception `ExceptionInInitializerError` is thrown instead.

Example 25 Constructor overloading; calling another constructor
We add a new constructor to the `Point` class (Example 13), thus overloading its constructors. The old constructor has signature `Point(int, int)` and the new one `Point(Point)`. The new constructor creates a copy of the point `p` by calling the old constructor using the syntax `this(p.x, p.y)`.

```
class Point {
  int x, y;

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

  Point(Point p)                         // overloaded constructor
  { this(p.x, p.y); }                    // calls the first constructor

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

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

Example 26 Calling a superclass constructor
The constructor in the `ColoredPoint` subclass (Example 65) calls its superclass constructor using the syntax `super(x, y)`.

Example 27 Field initializers and initializer blocks
Below, the static field initializer allocates an array and binds it to field `ps`. The static initializer block fills the array with an increasing sequence of pseudo-random numbers, then scale them so that the last number is 1.0 (this is useful for generating rolls of a random loaded die). This cannot be done using the field initializer alone.

One could delete the two occurrences of `static` to obtain another example, with a non-static field `ps`, a non-static field initializer, and a non-static initializer block. However, it is more common for non-static fields to be initialized in the constructors of the class (none is shown here).

```
class InitializerExample {
  static double[] ps = new double[6];

  static {                               // static initializer block
    double sum = 0;
    for (int i=0; i<ps.length; i++)      // fill with increasing numbers
      ps[i] = sum += Math.random();
    for (int i=0; i<ps.length; i++)      // scale so last ps element is 1.0
      ps[i] /= sum;
  }
  ...
}
```

9.11 Nested classes, member classes, local classes, and inner classes

A non-static nested class, that is, a non-static member class `NMC` or a local class `NLC` in a non-static member, is called an *inner class*. An object of an inner class always contains a reference to an object of the enclosing class `C`, called the *enclosing object*. That object can be referred to as `C.this` (see Example 33), so a non-static member `x` of the enclosing object can be referred to as `C.this.x`. A non-static nested class cannot itself have static members. More precisely, all static fields must also be final, and methods and nested classes in a non-static nested class must be non-static.

A static nested class, that is, a static member class `SMC` or a local class in a static member, has no enclosing object and cannot refer to non-static members of the enclosing class `C`. This is the standard restriction on static members of a class; see Section 9.1. A static nested class may have static as well as non-static members.

If a local class refers to variables or formal parameters in the enclosing method or constructor or initializer, then those variables or parameters must be final.

9.12 Anonymous classes

An *anonymous* class is a special kind of local class; hence it must be declared inside a method or constructor or initializer. An anonymous class can be declared, and an instance created, using the special expression syntax

`new C(`*actual-list*`)`
classbody

where `C` is a class name. This creates an anonymous subclass of class `C`, with the given *classbody* (Section 9.1). Moreover, it creates an object of that anonymous subclass by calling the appropriate `C` constructor with the arguments in *actual-list*, as if by `super(`*actual-list*`)`. An anonymous class cannot declare its own constructors.

When `I` is an interface name, the similar expression syntax

`new I()`
classbody

creates an anonymous local class, with the given *classbody* (Section 9.1), that must implement the interface `I`, and also creates an object of that anonymous class.

Example 28 Member classes and local classes

```
class TLC {                                         // top-level class
  static int sf;
  int nf;
  static class SMC {                                // static member class
    static int ssf = sf + TLC.sf;                   // can have static members
    int snf = sf + TLC.sf;                          // cannot use non-static TLC members
  }
  class NMC {                                       // non-static member class
    int nnf1 = sf + nf;                             // can use non-static TLC members
    int nnf2 = TLC.sf + TLC.this.nf;                // cannot have static members
  }
  void nm() {                                       // non-static method in TLC
    class NLC {                                     // local class in method
      int m(final int p) { return sf+nf+p; }        // can use non-static TLC members
} } }
```

Example 29 An enumeration as a local class
Method `suffixes` returns an object of the local class `SuffixEnumeration` which implements the `Enumeration` interface to enumerate the non-empty suffixes of the string `s`:

```
class LocalInnerClassExample {
   public static void main(String[] args) {
     Enumeration seq = suffixes(args[0]);
     while (seq.hasMoreElements())
       System.out.println(seq.nextElement());
   }

  static Enumeration suffixes(final String s) {
    class SuffixEnumeration implements Enumeration {
      int startindex=0;
      public boolean hasMoreElements() { return startindex < s.length(); }
      public Object nextElement() { return s.substring(startindex++); }
    }
    return new SuffixEnumeration();
} }
```

Example 30 Enumeration as an anonymous local class
Alternatively, we may use an anonymous local class in method `suffixes`:

```
static Enumeration suffixes(final String s) {
  return
    new Enumeration() {
      int startindex=0;
      public boolean hasMoreElements() { return startindex < s.length(); }
      public Object nextElement() { return s.substring(startindex++); }
    };
}
```

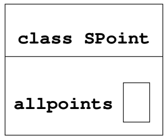
10 Classes and objects in the computer

What is a class?

Conceptually, a class represents a concept, a template for creating instances (objects). In the computer, a class is a chunk of memory, set aside once, when the class is loaded at runtime. A class has the following parts:

- the name of the class;
- room for all the static members of the class.

We can draw a class as a box. The header `class SPoint` gives the class name, and the box itself contains the static members of the class:

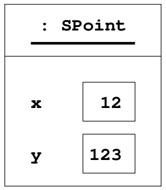


What is an object?

Conceptually, an object is a concrete instance of a concept (a class). In the computer, an object is a chunk of memory, set aside by an object creation expression `new C(...)`; see Section 11.6. Every evaluation of an object creation expression `new C(...)` creates a distinct object, with its own chunk of computer memory. An object has the following parts:

- the *class* `C` of the object; this is the class `C` used when creating the object;
- room for all the non-static members of the object.

We can draw an object as a box. The header `: SPoint` gives the object's class (underlined), and the remainder of the box contains the non-static members of the object:

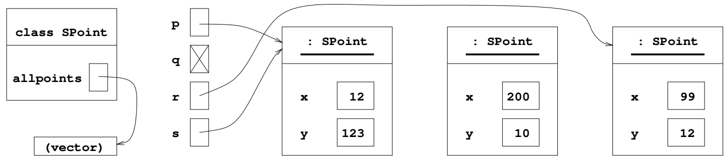


Inner objects

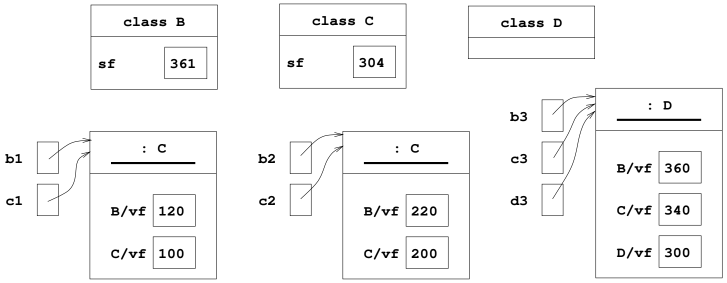
When `NIC` is an inner class (a non-static member class, or a local class in non-static code) in a class `C`, then an object of class `NIC` is an *inner object*. In addition to the object's class and the non-static fields, an inner object will always contain a reference to an *enclosing object*, which is an object of the innermost enclosing class `C`. The outer object reference can be written `C.this` in Java programs.

An object of a static nested class, on the other hand, contains no reference to an enclosing object.

Example 31 Objects and classes
This is the computer memory at the end of the `main` method in Example 45, using the `SPoint` class from Example 14. The variables `p` and `s` refer to the same object, variable `q` is `null`, and variable `r` refers to the right-most object. No variable refers to the middle object.



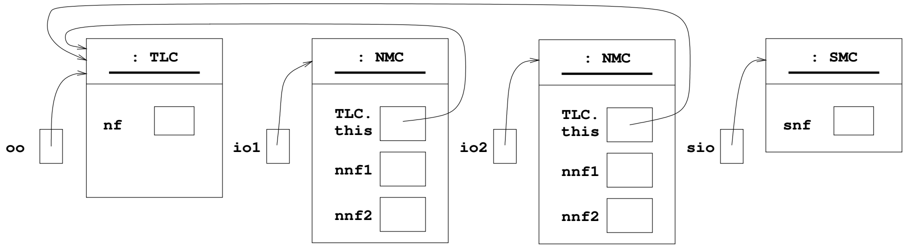
Example 32 Objects with multiple fields of the same name
This is the computer memory at the end of the `main` method in Example 42, using the classes from Example 19. The classes `B` and `C` each have a single static field `sf`; class D has none. The two objects of class `C` each have two non-static fields `vf` (called `B/vf` and `C/fv` in the figure), and the class `D` object has three non-static fields `vf`.



Example 33 Inner objects
Example 28 declares a class `TLC` with non-static member (inner) class `NMC` and static member class `SMC`. If we create a `TLC`-object, two `NMC`-objects, and a `SMC` object:

```
TLC oo = new TLC();
TLC.NMC io1 = oo.new NMC(), io2 = oo.new NMC();
TLC.SMC sio = new TLC.SMC();
```

then the computer memory will contain these objects (the classes are not shown):



11 Expressions

The main purpose of an expression is to compute a value (such as `117`) by evaluation. In addition, evaluation may change the computer's *state*: the value of variables, fields, and array elements, the contents of files, etc. More precisely, evaluation of an expression either:

- terminates normally, producing a value, or
- terminates abnormally by throwing an exception, or
- does not terminate at all (for instance, because it calls a method that does not terminate).

Expressions are built from *literals* (anonymous constants), variables, fields, operators, method calls, array accesses, conditional expressions, the `new` operator, and so on; see the table opposite.

One must distinguish the (compile-time) *type of an expression* from the (runtime) *class of an object*. An expression has a type (Section 5) inferred by the compiler. When this is a reference type `t`, and the value of the expression is an object `o`, then the class of object `o` will be a subtype of `t`, but not necessarily equal to `t`. For instance, the expression `(Number)(new Integer(2))` has type `Number`, but its value is an object whose class is `Integer`, a subclass of `Number`.

Table of expression forms

The table opposite (page 29) shows the form, meaning, associativity, operand (argument) types and result types for Java expressions. The expressions are grouped according to precedence as indicated by the horizontal lines, from high precedence to low precedence. Higher-precedence forms are evaluated before lower precedence forms. Parentheses may be used to emphasize or force a particular order of evaluation.

When an operator (such as `+`) is left associative, then a sequence `e1 + e2 + e3` of operators is evaluated as if parenthesized `(e1 + e2) + e3`. When an operator (such as `=`) is right associative, then a sequence `e1 = e2 = e3` of operators is evaluated as if parenthesized `e1 = (e2 = e3)`.

The table also shows the required operand types and result types. The kind integer stands for any of `char`, `byte`, `short`, `int`, or `long`; and numeric stands for integer or `float` or `double`.

For an operator with one integer or numeric operand, the *promotion type* is `double` if the operand has type `double`; it is `float` if the operand has type `float`; it is `long` if the operand has type `long`; otherwise it is `int` (that is, if the operand has type `byte`, `char`, `short` or `int`).

For an operator with two integer or numeric operands (except the shift operators; Section 11.3), the promotion type is `double` if any operand has type `double`; otherwise, it is `float` if any operand has type `float`; otherwise, it is `long` if any operand has type `long`; otherwise it is `int`.

Before the operation is performed, the operand(s) are promoted, that is, converted to the promotion type by a widening type conversion (page 40).

If the result type is given as numeric also, it equals the promotion type. For example, `10 / 3` has type `int`, whereas `10 / 3.0` has type `double`, and `c + (byte)1` has type `int` when `c` has type `char`.

Table of expression forms

<table>
<tr><th>Expression</th><th>Meaning</th><th>Associativity</th><th>Argument(s)</th><th>Result</th></tr>
<tr><td>a[...]</td><td>array access (Section 8.1)</td><td></td><td>t[], integer</td><td>t</td></tr>
<tr><td>o.f</td><td>field access (Section 11.8)</td><td></td><td>object</td><td></td></tr>
<tr><td>o.m(...)</td><td>method call (Section 11.10)</td><td></td><td>object</td><td></td></tr>
<tr><td>x++</td><td>postincrement</td><td></td><td>numeric</td><td>numeric</td></tr>
<tr><td>x--</td><td>postdecrement</td><td></td><td>numeric</td><td>numeric</td></tr>
<tr><td>++x</td><td>preincrement</td><td></td><td>numeric</td><td>numeric</td></tr>
<tr><td>--x</td><td>predecrement</td><td></td><td>numeric</td><td>numeric</td></tr>
<tr><td>-x</td><td>negation (minus sign)</td><td>right</td><td>numeric</td><td>numeric</td></tr>
<tr><td>~e</td><td>bitwise complement</td><td>right</td><td>integer</td><td>int/long</td></tr>
<tr><td>!e</td><td>logical negation</td><td>right</td><td>boolean</td><td>boolean</td></tr>
<tr><td>new t[...]</td><td>array creation (Section 8.1)</td><td></td><td>type</td><td>t[]</td></tr>
<tr><td>new C(...)</td><td>object creation (Section 11.6)</td><td></td><td>class</td><td>C</td></tr>
<tr><td>(t)e</td><td>type cast (Section 11.11)</td><td></td><td>type, any</td><td>t</td></tr>
<tr><td>e1 * e2</td><td>multiplication</td><td>left</td><td>numeric</td><td>numeric</td></tr>
<tr><td>e1 / e2</td><td>division</td><td>left</td><td>numeric</td><td>numeric</td></tr>
<tr><td>e1 % e2</td><td>remainder</td><td>left</td><td>numeric</td><td>numeric</td></tr>
<tr><td>e1 + e2</td><td>addition</td><td>left</td><td>numeric</td><td>numeric</td></tr>
<tr><td>e1 + e2</td><td>string concatenation</td><td>left</td><td>String, any</td><td>String</td></tr>
<tr><td>e1 + e2</td><td>string concatenation</td><td>left</td><td>any, String</td><td>String</td></tr>
<tr><td>e1 - e2</td><td>subtraction</td><td>left</td><td>numeric</td><td>numeric</td></tr>
<tr><td>e1 << e2</td><td>left shift (Section 11.3)</td><td>left</td><td>integer</td><td>int/long</td></tr>
<tr><td>e1 >> e2</td><td>signed right shift</td><td>left</td><td>integer</td><td>int/long</td></tr>
<tr><td>e1 >>> e2</td><td>unsigned right shift</td><td>left</td><td>integer</td><td>int/long</td></tr>
<tr><td>e1 < e2</td><td>less than</td><td>none</td><td>numeric</td><td>boolean</td></tr>
<tr><td>e1 <= e2</td><td>less than or equal to</td><td>none</td><td>numeric</td><td>boolean</td></tr>
<tr><td>e1 >= e2</td><td>greater than or equal to</td><td>none</td><td>numeric</td><td>boolean</td></tr>
<tr><td>e1 > e2</td><td>greater than</td><td>none</td><td>numeric</td><td>boolean</td></tr>
<tr><td>e instanceof t</td><td>instance test (Section 11.7)</td><td>none</td><td>any, ref. type</td><td>boolean</td></tr>
<tr><td>e1 == e2</td><td>equal</td><td>left</td><td>compatible</td><td>boolean</td></tr>
<tr><td>e1 != e2</td><td>not equal</td><td>left</td><td>compatible</td><td>boolean</td></tr>
<tr><td>e1 & e2</td><td>bitwise and</td><td>left</td><td>integer</td><td>int/long</td></tr>
<tr><td>e1 & e2</td><td>logical strict and</td><td>left</td><td>boolean</td><td>boolean</td></tr>
<tr><td>e1 ^ e2</td><td>bitwise exclusive-or</td><td>left</td><td>integer</td><td>int/long</td></tr>
<tr><td>e1 ^ e2</td><td>logical strict exclusive-or</td><td>left</td><td>boolean</td><td>boolean</td></tr>
<tr><td>e1 | e2</td><td>bitwise or</td><td>left</td><td>integer</td><td>int/long</td></tr>
<tr><td>e1 | e2</td><td>logical strict or</td><td>left</td><td>boolean</td><td>boolean</td></tr>
<tr><td>e1 && e2</td><td>logical and (Section 11.2)</td><td>left</td><td>boolean</td><td>boolean</td></tr>
<tr><td>e1 || e2</td><td>logical or (Section 11.2)</td><td>left</td><td>boolean</td><td>boolean</td></tr>
<tr><td>e1 ? e2 : e3</td><td>conditional (Section 11.5)</td><td>right</td><td>boolean, any, any</td><td>any</td></tr>
<tr><td>x = e</td><td>assignment (Section 11.4)</td><td>right</td><td>e subtype of x</td><td>type of x</td></tr>
<tr><td>x += e</td><td>compound assignment</td><td>right</td><td>compatible</td><td>type of x</td></tr>
</table>

11.1 Arithmetic operators

The value of the postincrement expression `x++` is that of `x`, and its effect is to increment `x` by one; and similarly for postdecrement `x--`.

The value of the preincrement expression `++x` is that of `x+1`, and its effect is to increment `x` by one; and similarly for predecrement `--x`.

Integer division `e1/e2` truncates, that is, rounds towards 0, so `10/3` is 3, and `(-10)/3` is -3. The integer remainder `x%y` equals `x-(x/y)*y` when `y` is non-zero; it has the same sign as `x`. Integer division or remainder by zero throws the exception `ArithmeticException`. Integer overflow does not throw an exception, but wraps around. Thus, in the `int` type, the expression `2147483647+1` evaluates to -2147483648, and the expression `-2147483648-1` evaluates to 2147483647.

The floating-point remainder `x%y` roughly equals `x-(((int)(x/y))*y` when `y` is non-zero. Floating-point division by zero, and floating-point overflow, do not throw exceptions, but produce special values such as `Infinity` or `NaN`, meaning 'not a number'.

11.2 Logical operators

The operators `==` and `!=` require the operand types to be compatible: one must be a subtype of the other. Two values of base type are equal (by `==`) if they represent the same value after conversion to their common supertype. For instance, 10 and 10.0 are equal. Two values of reference type are equal (by `==`) if both are `null`, or both are references to the same object or array, created by the same execution of the `new`-operator. Hence do not use `==` or `!=` to compare strings: two strings `s1` and `s2` may consist of the same sequence of characters (and therefore equal by `s1.equals(s2)`), yet be distinct objects (and therefore unequal by `s1==s2`); see Example 5.

The logical operators `&&` and `||` perform *short-cut evaluation*: if `e1` evaluates to `true` in `e1&&e2`, then `e2` is evaluated to obtain the value of the expression; otherwise `e2` is ignored and the value of the expression is `false`. Conversely, if `e1` evaluates to `false` in `e1||e2`, then `e2` is evaluated to obtain the value of the expression; otherwise `e2` is ignored and the value of the expression is `true`.

By contrast, the operators `&` (logical strict and) and `^` (logical strict exclusive-or) and `|` (logical strict or) always evaluate both operands, regardless of the value of the left-hand operand. Usually the short-cut operators `&&` and `||` are preferable.

11.3 Bitwise operators and shift operators

The operators `~` (bitwise complement) and `&` (bitwise and) and `^` (bitwise exclusive-or) and `|` (bitwise or) may be used on operands of integer type. The operators work in parallel on all bits of the two's complement representation of the operands. Thus `~n` equals `(-n)-1` and also equals `(-1)^n`.

The shift operators `<<` and `>>` and `>>>` shift the bits of the two's complement representation of the first argument. The two operands are promoted (page 28) separately, and the result type is the promotion type (`int` or `long`) of the first argument. Thus the shift operation is always performed on a 32-bit (`int`) or a 64-bit (`long`) value. In the former case, the length of the shift is between 0 and 31 as determined by the 5 least significant bits of the second argument; in the latter case, the length of the shift is between 0 and 63 as determined by the 6 least significant bits of the second argument.

The left shift `n<<s` equals `n*2*2*...*2` where there are `s` multiplications. The signed right shift `n>>s` of a non-negative `n` equals `n/2/2/.../2` where there are `s` divisions; the signed right shift of a negative `n` equals `~((~n)>>s)`. The unsigned right shift `n>>>s` of a non-negative `n` equals `n>>s`; the signed right shift of a negative `n` equals `(n>>s)+(2<<~s)` if `n` has type `int`, and `(n>>s)+(2L<<~s)` if it has type `long`.

Example 34 Arithmetic operators

```
public static void main(String[] args) {
  int max = 2147483647;
  int min = -2147483648;
  println(max+1);                              // Prints -2147483648
  println(min-1);                              // Prints  2147483647
  println(-min);                               // Prints -2147483648
  print(   10/3); println(   10/(-3));         // Prints  3 -3
  print((-10)/3); println((-10)/(-3));         // Prints -3  3
  print(   10%3); println(   10%(-3));         // Prints  1  1
  print((-10)%3); println((-10)%(-3));         // Prints -1 -1
}
static void print(int i)   { System.out.print(i + " "); }
static void println(int i) { System.out.println(i + " "); }
```

Example 35 Logical operators

Due to short-cut evaluation of `&&`, this expression from Example 11 does not evaluate the array access `days[mth-1]` unless $1 \leq$ `mth` ≤ 12, so the index is never out of bounds:

```
(mth >= 1) && (mth <= 12) && (day >= 1) && (day <= days[mth-1]);
```

This method returns `true` if `y` is a leap year, namely, if `y` is a multiple of 4 but not of 100, or is a multiple of 400:

```
static boolean leapyear(int y)
{ return y % 4 == 0 && y % 100 != 0 || y % 400 == 0; }
```

Example 36 Bitwise operators and shift operators

```
class Bitwise {
  public static void main(String[] args) throws Exception {
    int a = 0x3;                          // Bit pattern   0011
    int b = 0x5;                          // Bit pattern   0101
    println4(a);                          // Prints        0011
    println4(b);                          // Prints        0101
    println4(~a);                         // Prints        1100
    println4(~b);                         // Prints        1010
    println4(a & b);                      // Prints        0001
    println4(a ^ b);                      // Prints        0110
    println4(a | b);                      // Prints        0111
  }
  static void println4(int n) {
    for (int i=3; i>=0; i--)
      System.out.print(n >> i & 1);
    System.out.println();
  }
}
```

11.4 Assignment expression

In the *assignment expression* `x = e`, the type of `e` must be a subtype of the type of `x`. The type of the expression is the same as the type of `x`. The assignment is executed by evaluating expression `e` and storing its value in variable `x`, after a widening conversion (Section 11.11) if necessary. When `e` is a compile-time constant of type `byte`, `char`, `short` or `int`, and `x` has type `byte`, `char` or `short`, then a narrowing conversion is done automatically, provided the value of `e` is within the range representable in `x` (Section 5.1). The value of the expression `x = e` is that of `x` after the assignment.

The assignment operator is right associative, so the multiple assignment `x = y = e` has the same meaning as `x = (y = e)`, that is, evaluate the expression `e`, assign its value to `y`, and then to `x`.

When `e` has reference type (object type or array type), then only a reference to the object or array is stored in `x`. Thus the assignment `x = e` does not copy the object or array; see Example 38.

When `x` and `e` have the same type, the compound assignment `x += e` is equivalent to `x = x + e`; however, `x` is evaluated only once, so in `a[i++] += e` the variable `i` is increment only once. When the type of `x` is `t`, different from the type of `e`, then `x += e` is equivalent to `x = (t)(x + e)`, in which the intermediate result `(x + e)` is converted to type `t` (Section 11.11); again `x` is evaluated only once. The other compound assignment operators `-=`, `*=`, and so on, are similar.

Since the value of the expression `x += e` is that of `x` after the assignment, and the assignment operators associate to the right, one can write `ps[i] = sum += e` to first increment `sum` by `e` and then store the result in `ps[i]`; see Example 27.

11.5 Conditional expression

The *conditional expression* `e1 ? e2 : e3` is legal if `e1` has type `boolean`, and `e2` and `e3` both have numeric types, or both have type `boolean`, or both have compatible reference types. The conditional expression is evaluated by first evaluating `e1`. If `e1` evaluates to `true`, then `e2` is evaluated; otherwise `e3` is evaluated. The resulting value is the value of the conditional expression.

11.6 Object creation expression

The *object creation expression*

`new C(`*actual-list*`)`

creates a new object of class `C`, by calling that constructor in class `C` whose signature matches the arguments in *actual-list*.

The *actual-list* is evaluated from left to right to obtain a list of argument values. These argument values are bound to the constructor's parameters, an object of the class is created in the memory, the non-static fields are given default initial values according to their type, a superclass constructor is called (explicitly or implicitly), all non-static field initializers and initializer blocks are executed in order of appearance, and finally the constructor body is executed to initialize the object. The value of the constructor call expression is the newly created object, whose class is `C`.

When `C` is an inner class in class `D`, and `o` evaluates to an object of class `D`, then one may create a `C`-object inside `o` using the syntax `o.new C(`*actual-list*`)`; see Example 33.

11.7 Instance test expression

The *instance test* `e instanceof t` is evaluated by evaluating `e` to a value `v`. If `v` is not `null` and is a reference to an object of class `C`, where `C` is a subtype of `t`, the result is `true`; otherwise `false`.

Example 37 Assignment: widening, narrowing, and truncating compound assignment
The assignment `d = 12` performs a widening of 12 from `int` to `double`. The assignments `b = 123` and `b2 = 123+1` perform an implicit narrowing from `int` to `byte`, because the right-hand sides are compile-time constants. The assignment `b2 = b1+1` would be illegal because `b1+1` is not a compile-time constant. The assignment `b2 = 123+5` would be illegal because, although `123+5` is a compile-time constant, its value is not representable as a `byte` (whose range is $-128..127$).

```
double d;
d = 12;                      // widening conversion from int to double
byte b1 = 123, b2;           // narrowing conversion from int to byte
b2 = 123 + 1;                // legal: 123+1 is a compile-time constant
b2 = (byte)(b1 + 1);         // legal: (byte)(b1 + 1) has type byte
int x = 0;
x += 1.5;                    // equivalent to: x = (int)(x + 1.5); thus adds 1 to x
```

Example 38 Assignment does not copy objects
This example uses the `Point` class from Example 13. Assignment (and parameter passing) copies only the reference, not the object:

```
Point p1 = new Point(10, 20);
System.out.println("p1 is " + p1);       // prints: p1 is (10, 20)
Point p2 = p1;                           // p1 and p2 refer to same object
p2.move(8, 8);
System.out.println("p2 is " + p2);       // prints: p2 is (18, 28)
System.out.println("p1 is " + p1);       // prints: p1 is (18, 28)
```

Example 39 Compound assignment operators
Compute the product of all elements of array `xs`:

```
static double multiply(double[] xs) {
  double prod = 1.0;
  for (int i=0; i<xs.length; i++)
    prod *= xs[i];                       // equivalent to: prod = prod * xs[i]
  return prod;
}
```

Example 40 The conditional expression
Return the absolute value of `x` (always non-negative):

```
static double absolute(double x)
{ return (x >= 0 ? x : -x); }
```

Example 41 Object creation and instance test

```
Number n1 = new Integer(17);
Number n2 = new Double(3.14);
// The following statements print: false, true, false, true
System.out.println("n1 is a Double:   " + (n1 instanceof Double));
System.out.println("n2 is a Double:   " + (n2 instanceof Double));
System.out.println("null is a Double: " + (null instanceof Double));
System.out.println("n2 is a Number:   " + (n2 instanceof Number));
```

11.8 Field access expression

A *field access* must have one of these three forms

```
f
C.f
o.f
```

where `C` is a class and `o` an expression of reference type.

A class may have several fields of the same name `f`; see Section 9.6, Example 19, and Example 42 opposite.

A field access `f` must refer to a static or non-static field declared in or inherited by a class whose declaration encloses the field access expression (not shadowed by a field in a nested enclosing class, or by a variable or parameter of the same name). The class declaring the field is the target class `TC`.

A field access `C.f` must refer to a static field in class `C` or a superclass of `C`. That class is the target class `TC`.

A field access `o.f`, where expression `o` has type `C`, must refer to a static or non-static field in class `C` or a superclass of `C`. That class is the target class `TC`. To evaluate the field access, the expression `o` is evaluated to obtain an object. If the field is static, then the object is ignored, and the value of `o.f` is the `TC`-field `f`. If the field is non-static, then the value of `o` must be non-`null`, and the value of `o.f` is found as the value of the `TC`-field `f` in object `o`.

It is informative to contrast non-static field access and non-static method call (Section 11.10):

- In a non-static field access `o.f`, the field referred to is determined by the (compile-time) *type* of the object expression `o`.
- In a non-static method call `o.m(...)`, the method called is determined by the (runtime) *class* of the target object: the object to which `o` evaluates.

11.9 The current object reference `this`

The name `this` may be used in non-static code to refer to the current object (Section 9.1). When non-static code in a given object is executed, the object reference `this` refers to the object as a whole. Hence, when `f` is a field and `m` is a method (declared in the innermost enclosing class), then `this.f` means the same as `f` (when `f` has not been shadowed by a variable or parameter of the same name), and `this.m(...)` means the same as `m(...)`.

When `C` is an inner class in an enclosing class `D`, then inside `C` the notation `D.this` refers to the `D` object enclosing the inner `C` object. See Example 28 where `TLC.this.nf` refers to field `nf` of the enclosing class `TLC`.

Example 42 Field access
Here we illustrate static and non-static field access in the classes `B`, `C` and `D` from Example 19. Note that the field referred to by an expression of form `o.vf` or `o.sf` is determined by the type of expression `o`, not the class of the object to which `o` evaluates:

```
public static void main(String[] args) {
  C c1 = new C(100);                  // c1 has type C; object has class C
  B b1 = c1;                          // b1 has type B; object has class C
  print(C.sf,  B.sf);                 // Prints 102 121
  print(c1.sf, b1.sf);                // Prints 102 121
  print(c1.vf, b1.vf);                // Prints 100 120
  C c2 = new C(200);                  // c2 has type C; object has class C
  B b2 = c2;                          // b2 has type B; object has class C
  print(c2.sf, b2.sf);                // Prints 202 221
  print(c2.vf, b2.vf);                // Prints 200 220
  print(c1.sf, b1.sf);                // Prints 202 221
  print(c1.vf, b1.vf);                // Prints 100 120
  D d3 = new D(300);                  // d3 has type D; object has class D
  C c3 = d3;                          // c3 has type C; object has class D
  B b3 = d3;                          // b3 has type B; object has class D
  print(D.sf,  C.sf,  B.sf);          // Prints 304 304 361
  print(d3.sf, c3.sf, b3.sf);         // Prints 304 304 361
  print(d3.vf, c3.vf, b3.vf);         // Prints 300 340 360
}
static void print(int x, int y) { System.out.println(x+" "+y); }
static void print(int x, int y, int z) { System.out.println(x+" "+y+" "+z); }
```

Example 43 Using `this` when referring to shadowed fields
A common use of `this` is to refer to fields (`this.x` and `this.y`) that have been shadowed by parameters (`x` and `y`), especially in constructors; see the `Point` class (Example 13):

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

Example 44 Using `this` to pass the current object to a method
In the `SPoint` class (Example 14), the current object reference `this` is used in the constructor to add the newly created object to the vector `allpoints`, and it is used in the method `getIndex` to look up the current object in the vector:

```
class SPoint {
  static Vector allpoints = new Vector();
  int x, y;
  SPoint(int x, int y) { allpoints.addElement(this); this.x = x; this.y = y; }
  int getIndex() { return allpoints.indexOf(this); }
... }
```

11.10 Method call expression

A *method call* expression, or *method invocation*, must have one of these four forms:

> `m(`*actual-list*`)`
> `super.m(`*actual-list*`)`
> `C.m(`*actual-list*`)`
> `o.m(`*actual-list*`)`

where `m` is a method name, `C` is a class name, and `o` is an expression of reference type. The *actual-list* is a possibly empty comma-separated list of expressions, called the *arguments* or *actual parameters*. The *call signature* is $csig = \mathtt{m}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ where $(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is the list of types of the n arguments in the *actual-list*.

Determining what method is actually called by a method call is moderately complicated because (1) method names may be overloaded, each version of the method having a distinct signature; (2) methods may be overridden, that is, re-implemented in subclasses; (3) non-static methods are called by dynamic dispatch, given a target object; and (4) a method call in a nested class may call a method declared in some enclosing class.

Section 11.10.1 describes argument evaluation and parameter passing, assuming the simple case where it is clear which method `m` is being called. Section 11.10.2 then describes how to determine which method is being called in the general case.

11.10.1 Method call: parameter passing

Here we consider the evaluation of a method call `m(`*actual-list*`)` when it is clear which method `m` is called, and focus on the parameter passing mechanism.

The call is evaluated by evaluating the expressions in the *actual-list* from left to right to obtain the argument values. These argument values are then bound to the corresponding parameters in the method's *formal-list*, in order of appearance. A widening conversion (see Section 11.11) occurs if the type of an argument expression is a subtype of the method's corresponding parameter type.

Java uses *call-by-value* to bind argument values to formal parameters, so the formal parameter holds a copy of the argument value. Thus if the method changes the value of a formal parameter, this change does not affect the argument. For an argument of reference type, the parameter holds a copy of the object or array reference, and hence the parameter refers to the same object or array as the actual argument expression. Thus if the method changes that object or array, the changes will be visible after the method returns (see Example 46).

A non-static method must be called with a target object, for example as `o.m(`*actual-list*`)`, where the target object is the value of `o`, or as `m(`*actual-list*`)` where the target object is the current object reference `this`. In either case, during execution of the method body, `this` will be bound to the target object.

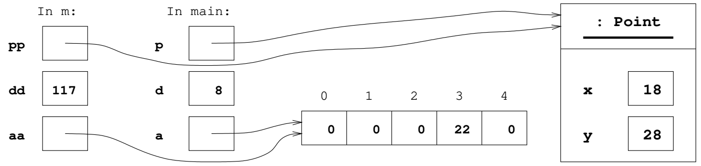
A static method is not called with a target object, and it is illegal to use the identifier `this` inside the method body.

When the argument values have been bound to the formal parameters, then the method body is executed. The value of the method call expression is the value returned by the method if its returntype is non-`void`; otherwise the method call expression has no value. When the method returns, all parameters and local variables in the method are discarded.

Example 45 Calling non-overloaded, non-overridden methods
This program uses the `SPoint` class from Example 14. The static methods `getSize` and `getPoint` may be called by prefixing them with the class name `SPoint` or an expression of type `SPoint`, such as `q`. They may be called before any objects have been created. The non-static method `getIndex` must be called with an object, as in `r.getIndex()`; then the method is executed with the current object reference `this` bound to `r`.

```
public static void main(String[] args) {
  System.out.println("Number of points created: " + SPoint.getSize());
  SPoint p = new SPoint(12, 123);
  SPoint q = new SPoint(200, 10);
  SPoint r = new SPoint(99, 12);
  SPoint s = p;
  q = null;
  System.out.println("Number of points created: " + SPoint.getSize());
  System.out.println("Number of points created: " + q.getSize());
  System.out.println("r is point number " + r.getIndex());
  for (int i=0; i<SPoint.getSize(); i++)
    System.out.println("SPoint number " + i + " is " + SPoint.getPoint(i));
}
```

Example 46 Parameter passing copies references, not objects and arrays
In the method call `m(p, d, a)` below, the object reference held in `p` is copied to parameter `pp` of `m`, so `p` and `pp` refer to the same object, the integer held in `d` is copied to `dd`, and the array reference held in `a` is copied to `aa`. At the end of method `m`, the state of the computer memory is this:



When method `m` returns, its parameters `pp`, `dd` and `aa` are discarded. The variables `p`, `d` and `a` are unmodified, but the object and the array pointed to by `p` and `a` have been modified.

```
public static void main(String[] args) {
  Point p = new Point(10, 20);
  int[] a = new int[5];
  int d = 8;
  System.out.println("p is " + p);            // Prints: p is (10, 20)
  System.out.println("a[3] is " + a[3]);      // Prints: a[3] is 0
  m(p, d, a);
  System.out.println("p is " + p);            // Prints: p is (18, 28)
  System.out.println("d is " + d);            // Prints: d is 8
  System.out.println("a[3] is " + a[3]);      // Prints: a[3] is 22
}
static void m(Point pp, int dd, int[] aa) {
  pp.move(dd, dd);
  dd = 117;
  aa[3] = 22;
}
```

11.10.2 Method call: determining which method is called

In general, methods may be overloaded as well as overridden. The overloading is resolved at compile-time by finding the most specific applicable and accessible method signature for the call. Overriding (for non-static methods) is handled at run-time by searching the class hierarchy starting with the class of the object on which the method is called.

At compile-time: determine the target type and signature

Find the target type `TC`: If the method call has the form `m(`*actual-list*`)` then the target type `TC` is the innermost enclosing class containing a method called `m` that is visible (not shadowed by a method `m` in a nested class). If the method call has the form `super.m(`*actual-list*`)` then the target type `TC` is the superclass of the innermost enclosing class. If the method call has the form `C.m(`*actual-list*`)` then `TC` is `C`. If the method call has the form `o.m(`*actual-list*`)` then `TC` is the type of the expression `o`.

Find the target signature tsig: A method in class `TC` is *applicable* if its signature subsumes the call signature *csig*. Whether a method is *accessible* is determined by its access modifiers; see Section 9.7. Consider the collection of methods in `TC` that are both applicable and accessible. The call is illegal (method unknown) if there is no such method. The call is illegal (ambiguous) if there is more than one method whose extended signature $\mathtt{m}(\mathtt{T}, \mathtt{u}_1, \ldots, \mathtt{u}_n)$ is most specific, that is, one whose extended signature is subsumed by all the others. Thus if the call is legal there is a exactly one most specific extended signature; from that we obtain the target signature $tsig = \mathtt{m}(\mathtt{u}_1, \ldots, \mathtt{u}_n)$.

Determine whether the called method is static: If the method call has the form `C.m(`*actual-list*`)` then the called method must be static. If the method call has the form `m(`*actual-list*`)` or `o.m(`*actual-list*`)` or `super.m(`*actual-list*`)` then we use the target type `TC` and the signature *tsig* to determine whether the called method is static or non-static.

At run-time: determine the target object (if non-static) and execute the method

If the method is static: If the method is static then no target object is needed: the method to call is the method with signature *tsig* in class `TC`. However, when `m` is static in a method call `o.m(`*actual-list*`)` the expression `o` must be evaluated anyway, but its value is ignored.

If the method is non-static, determine the target object: If the method is non-static, then a target object is needed, as it will be bound to the object reference `this` during execution of the called method. In the case of `m(`*actual-list*`)`, the target object is `this` (if `TC` is the innermost class enclosing the method call), or `TC.this` (if `TC` is an outer class containing the method call). In the case `o.m(`*actual-list*`)`, the expression `o` must evaluate to a non-`null` object reference, otherwise the exception `NullPointerException` is thrown; that object is the target object. To determine which method to call, the class hierarchy is searched, starting with the class `RTC` of the target object. If a method with signature *tsig* is not found in class `RTC`, then the immediate superclass of `RTC` is searched, and so on. This procedure is called *dynamic dispatch*.

Evaluate and bind the arguments: See Section 11.10.1.

Example 47 Calling overloaded methods
Here we call the overloaded methods `m` declared in Example 22. The call `m(10, 20)` has call signature `m(int, int)` and thus calls the method with signature `m(int, double)`, because that is the most specific applicable. Hence the first two lines call the method with signature `m(int, double)`, and the last two call the method with signature `m(double, double)`.

```
System.out.println(m(10, 20));              // Prints 31.0
System.out.println(m(10, 20.0));            // Prints 31.0
System.out.println(m(10.0, 20));            // Prints 33.0
System.out.println(m(10.0, 20.0));          // Prints 33.0
```

Example 48 Calling overridden and overloaded methods
Here we use the classes `C1` and `C2` from Example 24. The target type of `c1.m1(i)` is class `C1` which has a non-static method with signature `m1(int)`, so the call is to a non-static method; the target object has class `C2`, so the called method is `m1(int)` in `C2`; and quite similarly for `c2.m1(i)`. The target type for `c1.m1(d)` is the class `C1` which has a static method with signature `m1(double)`, so the call is to a static method, and the object bound to `c1` does not matter. Similarly for `c2.m1(d)` whose target type is `C2`, so it calls `m1(double)` in `C2` which overrides `m1(double)` in `C1`.

The call `c1.m2(i)` has target type `C1` and calls `m2(int)`. However, a call `c2.m2(i)` whose target class is `C2` would be ambiguous and illegal: the applicable extended signatures are `m2(C1,int)` and `m2(C2,double)`, none of which is more specific than the other.

```
int i = 17;
double d = 17.0;
C2 c2 = new C2();                           // Type C2, object class C2
C1 c1 = c2;                                 // Type C1, object class C1
c1.m1(i); c2.m1(i); c1.m1(d); c2.m1(d);     // Prints 21i 21i 11d 21d
c1.m2(i);                                   // Prints 12i
```

11.11 Type cast expression and type conversion

A *type conversion* converts a value from one type to another. A *widening* conversion converts from a type to a supertype. A *narrowing* conversion converts from a type to another type. This requires an explicit *type cast* (except in an assignment `x = e` or initialization where `e` is a compile-time integer constant; see Section 11.4).

Type cast between base types

When `e` is an expression of base type and `t` is a base type, then a *type cast* of `e` to `t` is done using the expression

```
(t)e
```

This expression, when legal, has type `t`. The legal type casts between base types are shown by the table below, where C marks a narrowing conversion which requires a type cast `(t)e`, W marks a widening conversion that preserves the value, and WL marks a widening conversion which may cause a loss of precision:

	To type						
From type	`char`	`byte`	`short`	`int`	`long`	`float`	`double`
`char`	W	C	C	W	W	W	W
`byte`	C	W	W	W	W	W	W
`short`	C	C	W	W	W	W	W
`int`	C	C	C	W	W	WL	W
`long`	C	C	C	C	W	WL	WL
`float`	C	C	C	C	C	W	W
`double`	C	C	C	C	C	C	W

A narrowing integer conversion discards those (most significant) bits which cannot be represented in the smaller integer type. Conversion from an integer type to a floating-point type (`float` or `double`) produces a floating-point approximation of the integer value. Conversion from a floating-point type to an integer type discards the fractional part of the number; that is, it rounds towards zero. When converting a too-large floating-point number to a `long` or `int`, the result is the best approximation (that is, the type's largest positive or the largest negative representable number); conversion to `byte` or `short` or `char` is done by converting to `int` and then to the requested type. The base type `boolean` cannot be cast to any other type. A type cast between base types never fails at runtime.

Type cast between reference types

When `e` is an expression of reference type and `t` is a reference type (class or interface or array type), then a *type cast* of `e` to `t` is done using the expression

```
(t)e
```

This expression has type `t`. It is evaluated by evaluating `e` to a value `v`. If `v` is `null` or is a reference to an object or array whose class is a subtype of `t`, then the type cast succeeds with result `v`; otherwise the exception `ClassCastException` is thrown. The type cast is illegal when it cannot possibly succeed at runtime; for instance, when `e` has type `Double` and `t` is `Boolean`: none of these classes is a subtype of the other.

12 Statements

A *statement* may change the computer's *state*: the value of variables, fields, array elements, the contents of files, etc. More precisely, execution of a statement either

- terminates normally (meaning execution will continue with the next statement, if any), or
- terminates abnormally by throwing an exception, or
- exits by executing a `return` statement (if inside a method or constructor), or
- exits a switch or loop by executing a `break` statement (if inside a switch or loop), or
- exits the current iteration of a loop and starts a new iteration by executing a `continue` statement (if inside a loop), or
- does not terminate at all — for instance, by executing `while (true) {}`.

12.1 Expression statement

An *expression statement* is an *expression* followed by a semicolon:

expression `;`

It is executed by evaluating the *expression* and ignoring its value. The only forms of *expression* that may be legally used in this way are assignment expressions (Section 11.4), increment and decrement expressions (Section 11.1), method call expressions (Section 11.10), and object creation expressions (Section 11.6).

For example, an assignment statement `x=e;` is an assignment expression `x=e` followed by a semicolon.

Similarly, a method call statement is a method call expression followed by semicolon. The value returned by the method, if any, is discarded; the method is executed only for its side effect.

12.2 Block statement

A *block-statement* is a sequence of zero or more *statements* or *variable-declarations* or *class-declarations*, in any order, enclosed in braces:

`{`
  statements
  class-declarations
  variable-declarations
`}`

12.3 The empty statement

The *empty statement* consists of a semicolon only. It is equivalent to the block statement `{ }` that contains no statements or declarations, and has no effect at all:

`;`

It is a common mistake to add an extra semicolon after the header of a `for` or `while` loop, thus introducing an empty loop body; see Example 55.

12.4 Choice statements

12.4.1 The `if` statement

An `if` statement has the form:

> `if (`*condition*`)`
> *truebranch*

The *condition* must have type `boolean`, and *truebranch* is a statement. If the *condition* evaluates to `true`, then the *truebranch* is executed, otherwise not.

12.4.2 The `if-else` statement

An `if-else` statement has the form:

> `if (`*condition*`)`
> *truebranch*
> `else`
> *falsebranch*

The *condition* must have type *boolean*, and *truebranch* and *falsebranch* are statements. If the *condition* evaluates to `true`, then the *truebranch* is executed; otherwise the *falsebranch* is executed.

12.4.3 The `switch` statement

A `switch` statement has the form:

> `switch (`*expression*`) {`
> `case` *constant1*`:` *branch1*
> `case` *constant2*`:` *branch2*
> `...`
> `default:` *branchn*
> `}`

The *expression* must have type `int`, `short`, `char`, or `byte`. Each *constant* must be a *compile-time constant* expression, consisting only of literals, `final` variables, `final` fields declared with explicit field initializers, and operators. No two *constants* may have the same value. The type of each *constant* must be a subtype of the type of *expression*.

Each *branch* is preceded by one or more `case` clauses, and is a possibly empty sequence of statements, usually terminated by `break` or `return` (if inside a method or constructor) or `continue` (inside a loop). The `default` clause may be left out.

The `switch` statement is executed as follows: The *expression* is evaluated to obtain a value `v`. If `v` equals one of the *constants*, then the corresponding *branch* is executed. If `v` does not equal any of the *constants*, then the *branch* following `default` is executed; if there is no `default` clause, nothing is executed. If a *branch* is not exited by `break` or `return` or `continue`, then execution continues with the next *branch* in the switch regardless of the `case` clauses, until a *branch* exits or the switch ends.

Example 49 Block statements
All method bodies and constructor bodies are block statements. In method `sum` from Example 2, the *truebranch* of the second `if` statement is a block statement. Method `m4` in Example 4 contains two block statements, each of which contains a (local) declaration of variable `x`.

Example 50 Single `if-else` statement
This method behaves the same as `absolute` in Example 40:

```
static double absolute(double x) {
  if (x >= 0)
    return x;
  else
    return -x;
}
```

Example 51 A sequence of `if-else` statements
We cannot use a switch here, because a switch can work only on integer types (including `char`):

```
static int wdayno1(String wday) {
  if      (wday.equals("Monday"))    return 1;
  else if (wday.equals("Tuesday"))   return 2;
  else if (wday.equals("Wednesday")) return 3;
  else if (wday.equals("Thursday"))  return 4;
  else if (wday.equals("Friday"))    return 5;
  else if (wday.equals("Saturday"))  return 6;
  else if (wday.equals("Sunday"))    return 7;
  else return -1;                              // Here used to mean 'not found'
}
```

Example 52 `switch` statement
Below we could have used a sequence of `if-else` statements, but a `switch` is both faster and clearer:

```
static String findCountry(int prefix) {
  switch (prefix) {
  case 1:   return "North America";
  case 44:  return "Great Britain";
  case 45:  return "Denmark";
  case 299: return "Greenland";
  case 46:  return "Sweden";
  case 7:   return "Russia";
  case 972: return "Israel";
  default:  return "Unknown";
  }
}
```

12.5 Loop statements

12.5.1 The `for` statement

A `for` statement has the form

`for (`*initialization*`;` *condition*`;` *step*`)`
body

where the *initialization* is a *variable-declaration* (Section 6.2) or an *expression*, *condition* is an *expression* of type `boolean`, *step* is an *expression*, and *body* is a *statement*. More generally, the *initialization* and *step* may also be comma-separated lists of *expressions*; the expressions in each such list are evaluated from left to right. The *initialization*, *condition* and *step* may be empty. An empty *condition* is equivalent to `true`. Thus `for (;;)` *body* means 'forever execute *body*'.

The `for` statement is executed as follows:

1. The *initialization* is executed
2. The *condition* is evaluated. If it is `false`, the loop terminates.
3. If it is `true`, then
   (a) The *body* is executed
   (b) The *step* is executed
   (c) Execution continues at 2.

12.5.2 The `while` statement

A `while` statement has the form

`while (`*condition*`)`
body

where the *condition* is an expression of type `boolean`, and *body* is a statement. The `while` statement is executed as follows:

1. The *condition* is evaluated. If it is `false`, the loop terminates.
2. If it is `true`, then
   (a) The *body* is executed
   (b) Execution continues at 1.

12.5.3 The `do-while` statement

A `do-while` statement has the form

`do`
body
`while (`*condition*`);`

where the *condition* is an expression of type `boolean`, and *body* is a statement. The *body* is executed at least once, because the `do-while` statement is executed as follows:

1. The *body* is executed.
2. The *condition* is evaluated. If it is `false`, the loop terminates.
3. If it is `true`, then execution continues at 1.

Example 53 Nested `for` loops
This program prints a four-line triangle of asterisks (`*`):

```
for (int i=1; i<=4; i++) {
  for (int j=1; j<=i; j++)
    System.out.print("*");
  System.out.println();
}
```

Example 54 Array search using `while` loop
This method behaves the same as `wdayno1` in Example 51:

```
static int wdayno2(String wday) {
  int i=0;
  while (i < wdays.length && ! wday.equals(wdays[i]))
    i++;
  // Now i >= wdays.length or wday equal to wdays[i]
  if (i < wdays.length)
    return i+1;
  else
    return -1;                           // Here used to mean 'not found'
}

static final String[] wdays =
{ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
```

Example 55 Infinite loop because of misplaced semicolon
Here a misplaced semicolon (`;`) creates an empty loop body statement, where the increment `i++` is not part of the loop. Hence it will not terminate, but loop forever:

```
int i=0;
while (i<10);
  i++;
```

Example 56 Using `do-while` (but `while` is usually preferable)
Throw a die and compute sum until 5 or 6 comes up:

```
static int waitsum() {
  int sum = 0, eyes;
  do {
    eyes = (int)(1 + 6 * Math.random());
    sum += eyes;
  } while (eyes < 5);
  return sum;
}
```

12.6 Labelled statements, returns, exits and exceptions

12.6.1 The `return` statement

The simplest form of a `return` statement, without an expression argument, is:

```
return;
```

That form of `return` statement must occur inside the body of a method whose *returntype* is `void`, or inside the body of a constructor. Execution of the `return` statement exits the method or constructor, and continues execution at the place from which it was called.

Alternatively, a `return` statement may have an expression argument:

`return` *expression*`;`

That form of `return` statement must occur inside the body of a method (not constructor) whose *returntype* is a supertype of the type of the *expression*. The `return` statement is executed as follows: First the *expression* is evaluated to some value `v`. Then it exits the method, and continues execution at the method call expression that called the method; the value of that expression will be `v`.

12.6.2 The labelled statement

A labelled statement has the form

label `:` *statement*

where *label* is an identifier. The scope of *label* is *statement*, where it can be used in connection with `break` (Section 12.6.3) and `continue` (Section 12.6.4). It is illegal to re-use the same *label* inside *statement*, unless inside a local class in *statement*.

12.6.3 The `break` statement

A `break` statement is legal only inside a switch or loop, and has one of the forms

`break;`
`break` *label*`;`

Executing `break` exits the inner-most enclosing switch or loop, and continues execution after that switch or loop. Executing `break` *label* exits that enclosing statement which has label *label*, and continues execution after that statement. Such a statement must exist in the inner-most enclosing method or constructor or initializer block.

12.6.4 The `continue` statement

A `continue` statement is legal only inside a loop, and has one of the forms

`continue;`
`continue` *label*`;`

Executing `continue` terminates the current iteration of the inner-most enclosing loop, and continues the execution at the *step* (in `for` loops; see Section 12.5.1), or the *condition* (in `while` and `do-while` loops; see Sections 12.5.2 and 12.5.3). Executing `continue` *label* terminates the current iteration of that enclosing loop which has label *label*, and continues the execution at the *step* or the *condition*. There must be such a loop in the inner-most enclosing method or constructor or initializer block.

Example 57 Using `return` to terminate a loop early
This method behaves the same as `wdayno2` in Example 54:

```
static int wdayno3(String wday) {
  for (int i=0; i < wdays.length; i++)
    if (wday.equals(wdays[i]))
      return i+1;
  return -1;                                    // Here used to mean 'not found'
}
```

Example 58 Using `break` to terminate a loop early

```
double prod = 1.0;
for (int i=0; i<xs.length; i++) {
  prod *= xs[i];
  if (prod == 0.0)
    break;
}
```

Example 59 Using `continue` to start a new iteration
This method decides whether `query` is a substring of `target`. When a mismatch between the strings is found, `continue` starts the next iteration of the outer `for` loop, thus incrementing `j`:

```
static boolean substring1(String query, String target) {
  nextposition:
    for (int j=0; j<=target.length()-query.length(); j++) {
      for (int k=0; k<query.length(); k++)
        if (target.charAt(j+k) != query.charAt(k))
          continue nextposition;
      return true;
    }
  return false;
}
```

Example 60 Using `break` to exit a labelled statement block
This method behaves as `substring1` from Example 59. It uses `break` to exit the entire statement block labelled `thisposition`, thus skipping the first `return` statement and starting a new iteration of the outer `for` loop:

```
static boolean substring2(String query, String target) {
  for (int j=0; j<=target.length()-query.length(); j++)
    thisposition: {
      for (int k=0; k<query.length(); k++)
        if (target.charAt(j+k) != query.charAt(k))
          break thisposition;
      return true;
    }
  return false;
}
```

12.6.5 The `throw` statement

A `throw` statement has the form

    throw *expression*;

where the type of *expression* must be a subtype of class `Throwable` (Section 14). The `throw` statement is executed as follows: The *expression* is evaluated to obtain an exception object `v`. If it is `null`, then the exception `NullPointerException` is thrown. Otherwise, the exception object `v` is thrown. In any case, the enclosing block statement is terminated abnormally; see Section 14. The thrown exception may be caught in a dynamically enclosing `try-catch` statement (Section 12.6.6). If the exception is not caught, then the entire program execution will be aborted, and information from the exception will be printed on the console (for example, at the command prompt, or in the Java Console inside a web browser).

12.6.6 The `try-catch-finally` statement

A `try-catch` statement is used to catch (particular) exceptions thrown by the execution of a block of code. It has the following form:

```
try
  body
catch (E1 x1) catchbody1
catch (E2 x2) catchbody2
...
finally finallybody
```

where `E1`, `E2`, ... are names of exception types, the `x1`, `x2`, ... are variable names, and the *body*, the `catchbody`$_i$ and the *finallybody* are *block-statements* (Section 12.2). There can be zero or more `catch` blocks, and the `finally` clause may be absent, but at least one `catch` or `finally` clause must be present.

We say that `Ei` matches exception type `E` if `E` is a subtype of `Ei` (possibly equal to `Ei`).

The `try-catch-finally` statement is executed by executing the *body*. If the execution of *body* terminates normally, or exits by `return` or `break` or `continue` (when inside a method or constructor or switch or loop), then the `catch` blocks are ignored. If *body* terminates abnormally by throwing exception `e` of class `E`, then the first matching `Ei` (if any) is located, variable `xi` is bound to `e`, and the corresponding *catchbody*$_i$ is executed. If there is no matching `Ei`, then the entire `try-catch` statement terminates abnormally with exception `e`.

If a `finally` clause is present, then the *finallybody* will be executed regardless whether the execution of *body* terminated normally, regardless whether *body* exited by executing `return` or `break` or `continue` (when inside a method or constructor or switch or loop), regardless whether any exception thrown by *body* was caught by the `catch` blocks, and regardless whether any new exception was thrown during the execution of a `catch` body.

Example 61 Throwing an exception to indicate failure
Instead of returning the bogus error value `-1` as in method `wdayno3` above, throw an exception of class `WeekdayException` (Example 67). Note the `throws` clause (Section 9.8) in the method header:

```
static int wdayno4(String wday) throws WeekdayException {
  for (int i=0; i < wdays.length; i++)
    if (wday.equals(wdays[i]))
      return i+1;
  throw new WeekdayException(wday);
}
```

Example 62 A `try-catch` block
This example calls the method `wdayno4` (Example 61) inside a `try-catch` block that handles exceptions of class `WeekdayException` (Example 67) and its superclass `Exception`. The second `catch` clause will be executed (for example) if the array access `args[0]` fails because there is no command line argument (since `ArrayIndexOutOfBoundsException` is a subclass of `Exception`). If an exception is handled, it is bound to the variable `x`, and printed by an implicit call (Section 7) to the exception's `toString`-method:

```
public static void main(String[] args) {
  try {
    System.out.println(args[0] + " is weekday number " + wdayno4(args[0]));
  } catch (WeekdayException x) {
    System.out.println("Weekday problem: " + x);
  } catch (Exception x) {
    System.out.println("Other problem: " + x);
  }
}
```

Example 63 A `try-finally` block
This method attempts to read three lines from a file, each containing a single floating-point number. Regardless whether anything goes wrong during reading (premature end of file, ill-formed number), the `finally` clause will close the readers before the method returns. It would do so even if the `return` statement were inside the `try` block:

```
static double[] readRecord(String filename) throws IOException {
  Reader freader         = new FileReader(filename);
  BufferedReader breader = new BufferedReader(freader);
  double[] res = new double[3];
  try {
    res[0] = new Double(breader.readLine()).doubleValue();
    res[1] = new Double(breader.readLine()).doubleValue();
    res[2] = new Double(breader.readLine()).doubleValue();
  } finally {
    breader.close();
    freader.close();
  }
  return res;
}
```

13 Interfaces

13.1 Interface declarations

An *interface* describes fields and methods, but does not implement them. An *interface-declaration* may contain field descriptions, method descriptions, class declarations, and interface declarations. The declarations in an interface may appear in any order:

> *interface-modifiers* `interface I` *extends-clause* `{`
> *field-descriptions*
> *method-descriptions*
> *class-declarations*
> *interface-declarations*
> `}`

An interface may be declared at top-level or inside a class or interface, but not inside a method or constructor or initializer. At top-level, the *interface-modifiers* may be `public`, or absent. A public interface is accessible also outside its package. Inside a class or interface, the *interface-modifiers* may be `static` (always implicitly understood), and at most one of `public`, `protected`, or `private`.

The *extends-clause* may be absent or have the form

```
extends I1, I2, ...
```

where `I1`, `I2`, ... are interface names. If the *extends-clause* is present, then interface `I` describes all those members described by `I1`, `I2`, ..., and interface `I` is a *subinterface* (and hence subtype) of `I1`, `I2`, ... Interface `I` can describe additional fields and methods, but cannot override inherited member descriptions.

A *field-description* in an interface declares a named constant, and must have the form

> *field-desc-modifiers* *type* `f =` *initializer*`;`

where *field-desc-modifiers* is a list of `static`, `final`, and `public`; all of which are understood and need not be given explicitly. The field *initializer* must be an expression involving only literals and operators, and static members of classes and interfaces.

A *method-description* for method `m` must have the form:

> *method-desc-modifiers* *returntype* `m(`*formal-list*`)` *throws-clause*`;`

where *method-desc-modifiers* is a list of `abstract` and `public`, both of which are understood and need not be given explicitly.

A *class-declaration* inside an interface is always implicitly `static` and `public`.

13.2 Classes implementing interfaces

A class `C` may be declared to implement one or more interfaces by an *implements-clause*:

> `class C implements I1, I2, ...`
> *classbody*

In this case, `C` is a subtype (see Section 5.4) of `I1`, `I2`, and so on. The compiler will check that `C` declares all the methods described by `I1`, `I2`, ..., with exactly the prescribed signatures and return types. A class may implement any number of interfaces. Fields, classes, and interfaces declared in `I1`, `I2`, ... can be used in class `C`.

Example 64 Three interface declarations

```
import java.awt.*;

interface Colored { Color getColor(); }
interface Drawable { void draw(Graphics g); }
interface ColoredDrawable extends Colored, Drawable {}
```

Example 65 Classes implementing interfaces
Note that the methods `getColor` and `draw` must be public because they are implicitly public in the above interfaces.

```
class ColoredPoint extends Point implements Colored {
  Color c;
  ColoredPoint(int x, int y, Color c) { super(x, y); this.c = c; }
  public Color getColor() { return c; }
}

class ColoredDrawablePoint extends ColoredPoint implements ColoredDrawable {
  Color c;
  ColoredDrawablePoint(int x, int y, Color c) { super(x, y, c); }
  public void draw(Graphics g) { g.fillRect(x, y, 1, 1); }
}

class ColoredRectangle implements ColoredDrawable {
  int x1, x2, y1, y2;   // (x1, y1) upper left, (x2, y2) lower right corner
  Color c;

  ColoredRectangle(int x1, int y1, int x2, int y2, Color c)
  { this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2; this.c = c; }
  public Color getColor() { return c; }
  public void draw(Graphics g) { g.drawRect(x1, y1, x2-x1, y2-y1); }
}
```

Example 66 Using interfaces as types

```
static void printcolors(Colored[] cs) {
  for (int i=0; i<cs.length; i++)
    System.out.println(cs[i].getColor().toString());
}

static void draw(Graphics g, ColoredDrawable[] cs) {
  for (int i=0; i<cs.length; i++) {
    g.setColor(cs[i].getColor());
    cs[i].draw(g);
  }
}
```

14 Exceptions

An *exception* is an object of an exception type: a subclass of class `Throwable`. An exception is used to signal and describe an abnormal situation during program execution. The evaluation of an expression or the execution of a statement may terminate abnormally by throwing an exception, either by executing a `throw` statement (Section 12.6.5) or by executing a primitive operation, such as assignment to an array element, that may throw an exception.

A thrown exception may be caught in a dynamically enclosing `try-catch` statement (Section 12.6.6). A caught exception may be re-thrown by a `throw` statement. If the exception is not caught, then the entire program execution will be aborted, and information from the exception will be printed on the console (for example, at the command prompt, or in the Java Console inside a web browser). What is printed on the console is determined by the exception's `toString` method.

Checked and unchecked exception types

There are two kinds of exception types: *checked* (those that must be declared in the *throws-clause* of a method or constructor; Section 9.8) and *unchecked* (those that need not). If the execution of a method body can throw a checked exception of class `E`, then class `E` or a supertype of `E` must be declared in the *throws-clause* of the method.

Some of the most important predefined exception types, and their status (checked or unchecked) are shown below.

```
Throwable
    Error                                           unchecked
        ExceptionInInitializerError                 |
        OutOfMemoryError                            |
        StackOverflowError                          |
    Exception                                       checked
        InterruptedException                        |
        IOException                                 |
        RuntimeException                            unchecked
            ArithmeticException                     |
            ArrayStoreException                     |
            ClassCastException                      |
            IllegalMonitorStateException            |
            IndexOutOfBoundsException               |
                ArrayIndexOutOfBoundsException      |
                StringIndexOutOfBoundsException     |
            NullPointerException                    |
```


Example 67 Declaring a checked exception class
This is the class of exceptions thrown by method `wdayno4` (Example 61). Note the `toString` method which is used when printing an uncaught exception on the console:

```
class WeekdayException extends Exception {
  private String wday;

  public WeekdayException(String wday)
  { this.wday = wday; }

  public String toString()
  { return "Illegal weekday " + wday; }
}
```

15 Threads, concurrent execution, and synchronization

15.1 Threads and concurrent execution

The preceding chapters describe sequential program execution, in which expressions are evaluated and statements are executed one after the other: we have considered only a single thread of execution, where a *thread* is an independent sequential activity. A Java program may execute several threads concurrently, that is, potentially overlapping in time. For instance, one part of a program may continue computing while another part is blocked waiting for input; see Example 68.

Threads are created and manipulated using the `Thread` class and the `Runnable` interface, both of which are part of the Java class library package `java.lang`.

To program a new thread, one must implement the method `public void run()` described by the `Runnable` interface. One can do this by declaring a subclass `U` of class `Thread` (which implements `Runnable`). To create a new thread, create an object `u` of class `U`, and to permit it to run, execute `u.start()`. This enables the new thread, so that it can execute concurrently with the current thread; see Example 68.

Alternatively, declare a class `C` that implements `Runnable`, create an object `o` of that class, create a thread object `u = new Thread(o)` from `o`, and execute `u.start()`; see Example 72.

Threads can communicate with each other only via shared state, namely, by using and assigning static fields, non-static fields, and array elements. By the design of Java, threads cannot use local variables and method parameters for communication.

States and state transitions of a thread

A thread is alive if it has been started and has not died. A thread dies by exiting its `run()` method, either by returning or by throwing an exception. A live thread is in one of the states Enabled (ready to run), Running (actually executing), Sleeping (waiting for a timeout), Joining (waiting for another thread to die), Locking (trying to get the lock on object `o`), or Waiting (for notification on object `o`).

The state transitions of a thread can be summarized by this table and the figure opposite:

From state	To state	Reason for the transition
Enabled	Running	the system schedules the thread for execution
Running	Enabled	the system preempts the thread and schedules another
	Enabled	the thread executes `yield()`
	Waiting	the thread executes `o.wait()`, thus releasing the lock on `o`
	Locking	the thread attempts to execute `synchronized (o) { ... }`
	Sleeping	the thread executes `sleep()`
	Joining	the thread executes `u.join()`
	Running	the thread was interrupted; sets the interrupted status of the thread
	Dead	the thread exited `run()` by returning or by throwing an exception
Sleeping	Enabled	the sleeping period expired
	Enabled	the thread was interrupted; throws `InterruptedException` when run
Joining	Enabled	the thread u being joined died, or the join timed out
	Enabled	the thread was interrupted; throws `InterruptedException` when run
Waiting	Locking	another thread executed `o.notify()` or `o.notifyAll()`
	Locking	the wait for the lock on `o` timed out
	Locking	the thread was interrupted; throws `InterruptedException` when run
Locking	Enabled	the lock on `o` became available and was given to this thread

Example 68 Multiple threads

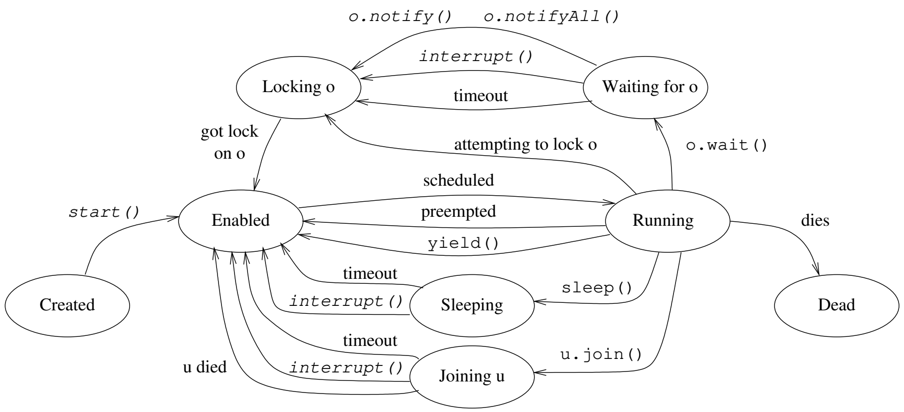
The main program creates a new thread, binds it to `u`, and starts it. Now two threads are executing concurrently: one executes `main`, and another executes `run`. While the `main` method is blocked waiting for keyboard input, the new thread keeps incrementing `i`. The new thread executes `yield()` to make sure that the other thread is allowed to run (when not blocked).

```
class Incrementer extends Thread {
  public int i;
  public void run() {
    for (;;) {                                     // Forever
      i++;                                         //   increment i
      yield();
    }
} }

class ThreadDemo {
  public static void main(String[] args) throws IOException {
    Incrementer u = new Incrementer();
    u.start();
    System.out.println("Repeatedly press Enter to get the current value of i:");
    for (;;) {
      System.in.read();                          // Wait for keyboard input
      System.out.println(u.i);
} } }
```

The states and state transitions of a thread

A thread's transition from one state to another may be caused by a method call performed by the thread itself (shown in the `typewriter` font), by a method call possibly performed by another thread (shown in the *slanted* font); and by timeouts and other actions (shown in the default font):



15.2 Locks and the `synchronized` statement

When multiple concurrent threads access the same fields or array elements, there is considerable risk of creating an inconsistent state; see Example 70. To avoid this, threads may synchronize the access to shared state, such as objects and arrays. A single *lock* is associated with every object, array, and class. A lock can be held by at most one thread at a time.

A thread may explicitly ask for the lock on an object or array by executing a synchronized statement, which has this form:

`synchronized (`*expression*`)`
block-statement

The *expression* must have reference type. The *expression* must evaluate to a non-`null` reference `o`; otherwise a `NullPointerException` is thrown. After the evaluation of the `expression`, the thread becomes Locking on object `o`; see the figure on page 55. When the thread obtains the lock on object `o` (if ever), the thread becomes Enabled, and may become Running so the *block-statement* is executed. When the *block-statement* terminates or is exited by `return` or `break` or `continue` or by throwing an exception, then the lock on `o` is released.

A `synchronized` non-static method declaration (Section 9.8) is a shorthand for a method whose body has the form:

`synchronized (this)`
method-body

That is, the thread will execute the method body only when it has obtained the lock on the current object. It will hold the lock until it leaves the method body, and release it at that time.

A `synchronized` static method declaration (Section 9.8) in class `C` is a shorthand for a method whose body has the form:

`synchronized (C.class)`
method-body

That is, the thread will execute the method body only when it has obtained the lock on the object `C.class`, which is the unique object of class `Class` associated with the class `C`. It will hold the lock until it leaves the method body, and release it at that time.

Constructors and initializers cannot be `synchronized`.

Mutual exclusion is ensured only if *all* threads accessing a shared object lock it before use. For instance, if we add an unsynchronized method `roguetransfer` to a bank object (Example 70), we can no longer be sure that a thread calling the synchronized method `transfer` has exclusive access to the bank object: any number of threads could be executing `roguetransfer` at the same time.

A *monitor* is an object whose fields are private and are manipulated only by synchronized methods of the object, so that all field access is subject to synchronization; see Example 71.

If a thread `u` needs to wait for some condition to become true, or for a resource to become available, it may release its lock on object `o` by calling `o.wait()`. The thread must own the lock on object `o`, otherwise exception `IllegalMonitorStateException` is thrown. The thread `u` will be added to a set of threads waiting for notification on object `o`. This notification must come from another thread which has obtained the lock on `o` and which executes `o.notify()` or `o.notifyAll()`. The notifying thread does not lose the lock on `o`. After being notified, `u` must obtain the lock on `o` again before it can proceed. Thus when the call to `wait` returns, thread `u` will own the lock on `o` just as before the call; see Example 71.

For detailed rules governing the behaviour of unsynchronized Java threads, see the Java Language Specification, Chapter 17.

Example 69 Mutual exclusion
A Printer thread forever prints a (`-`) followed by a (`/`). If we create and run two concurrent printer threads using `new Printer().start()` and `new Printer().start()`, then only one of the threads can hold the lock on object `mutex` at a time, so no other symbols can be printed between (`-`) and (`/`) in one iteration of the `for` loop. Thus the program must print `-/-/-/-/-/-/` and so on. However, if the synchronization is removed, it may print `--//--/-/-//--//` and so on. (The call `pause(n)` pauses the thread for 200 ms, whereas `pause(100,300)` pauses between 100 and 300 ms. This is done only to make the inherent non-determinacy of unsynchronized concurrency more easily observable).

```
class Printer extends Thread {
  static Object mutex = new Object();
  public void run() {
    for (;;) {
      synchronized (mutex) {
        System.out.print("-");
        Util.pause(100,300);
        System.out.print("/");
      }
      Util.pause(200);
} } }
```

Example 70 Synchronized methods in an object
The bank object below has two accounts. Money is repeatedly being transferred from one account to the other by clerks. Clearly the total amount of money should remain constant (at 30 euro). This holds true when the transfer method is declared synchronized, because only one clerk can access the accounts at any one time. If the synchronized declaration is removed, the sum will differ from 30 most of the time, because one clerk is likely to overwrite the other's deposits and withdrawals .

```
class Bank {
  private int account1 = 10, account2 = 20;
  synchronized public void transfer(int amount) {
    int new1 = account1 - amount;
    Util.pause(10);
    account1 = new1; account2 = account2 + amount;
    System.out.println("Sum is " + (account1+account2));
} }

class Clerk extends Thread {
  private Bank bank;
  public Clerk(Bank bank) { this.bank = bank; }
  public void run() {
    for (;;) {                                  // Forever
      bank.transfer(Util.random(-10, 10));      //   transfer money
      Util.pause(200, 300);                     //   then take a break
} } }

... Bank bank = new Bank();
... new Clerk(bank).start(); new Clerk(bank).start();
```

15.3 Operations on threads

The current thread, whose state is Running, may call these methods among others. Further `Thread` methods are described in the Thread section of the Java class library; see Section 18.

- `Thread.yield()` changes the state of the current thread from Running to Enabled, and thereby allows the system to schedule another Enabled thread, if any.
- `Thread.sleep(n)` sleeps for `n` milliseconds: the current thread becomes Sleeping, and after `n` milliseconds becomes Enabled. May throw `InterruptedException` if the thread is interrupted while sleeping.
- `Thread.currentThread()` returns the current thread object.
- `Thread.interrupted()` returns and clears the *interrupted status* of the current thread: `true` if it has been interrupted since the last call to `Thread.interrupted()`; otherwise `false`.

Let `u` be a thread (an object of a subclass of Thread). Then

- `u.start()` changes the state of `u` to Enabled, so that its `run` method will be called when a processor becomes available.
- `u.interrupt()` interrupts the thread `u`: if `u` is Running or Enabled or Locking, then its interrupted status is set to `true`. If `u` is Sleeping or Joining it will become Enabled, and if it is Waiting it will become Locking; in these cases `u` will throw `InterruptedException` when and if it becomes Running.
- `u.isInterrupted()` returns the interrupted status of `u` (and does not clear it).
- `u.join()` waits for thread `u` to die; may throw `InterruptedException` if the current thread is interrupted while waiting.
- `u.join(n)` works as `u.join()` but times out and returns after at most `n` milliseconds. There is no indication whether the call returned because of a timeout or because `u` died.

Operations on locked objects

A thread which owns the lock on an object `o` may call the following methods, inherited by `o` from class `Object` in the Java class library; see Section 18.

- `o.wait()` releases the lock on `o`, changes its own state to Waiting, and adds itself to the set of threads waiting for notification about `o`. When notified (if ever), the thread must obtain the lock on `o`, so when the call to `wait` returns, it again has the lock on `o`. May throw `InterruptedException` if the thread is interrupted while waiting.
- `o.wait(n)` works as `o.wait()` except that the thread will change state to Locking after `n` milliseconds, regardless whether there has been a notification on `o` or not. There is no indication whether the state change was caused by a timeout or because of a notification.
- `o.notify()` chooses an arbitrary thread among the threads waiting for notification about `o` (if any), and changes its state to Locking. The chosen thread cannot actually get the lock on `o` until the current thread has released it.
- `o.notifyAll()` works as `o.notify()`, except that it changes the state to Locking for *all* threads waiting for notification about `o`.

Example 71 Producers and consumers communicating via a monitor
A Buffer has room for one integer, and has a method `put` for storing into the buffer (if empty) and a method `get` for reading from the buffer (if non-empty); it is a monitor (page 56). A thread calling `get` must obtain the lock on the buffer. If it finds that the buffer is empty, it calls `wait` to (release the lock and) wait until something has been put into the buffer. If another thread calls `put` and thus `notify`, then the getting thread will start competing for the buffer lock again, and if it gets it, will continue executing. Here we have used a `synchronized` statement in the method body (instead of making the method `synchronized`, as is normal for a monitor) to emphasize that synchronization, `wait` and `notify` all work on the buffer object `this`:

```
class Buffer {
  private int contents;
  private boolean empty = true;
  public int get() {
    synchronized (this) {
      while (empty)
        try { this.wait(); } catch (InterruptedException x) {};
      empty = true;
      this.notify();
      return contents;
  } }
  public void put(int v) {
    synchronized (this) {
      while (!empty)
        try { this.wait(); } catch (InterruptedException x) {};
      empty = false;
      contents = v;
      this.notify();
  } }
}
```

Example 72 Graphic animation using the Runnable interface
Class AnimatedCanvas below is a subclass of Canvas, and so cannot be a subclass of Thread also. Instead it declares a `run` method and implements the Runnable interface. The constructor creates a Thread object `u` from the AnimatedCanvas object `this`, and then starts the thread. The new thread executes the `run` method, which repeatedly sleeps and repaints, thus creating an animation.

```
class AnimatedCanvas extends Canvas implements Runnable {
  AnimatedCanvas() { Thread u = new Thread(this); u.start(); }

  public void run() {                           // from Runnable
    for (;;) { // forever sleep and repaint
      try { Thread.sleep(100); } catch (InterruptedException e) { }
      ...
      repaint();
    }
  }

  public void paint(Graphics g) { ... }         // from Canvas
  ...
}
```

16 Compilation, source file names, class names, and class files

A *Java program* consists of one or more *source files* (with filename suffix `.java`). A source file may contain one or more class or interface declarations. A source file can contain only one declaration of a public class or interface, which must then have the same name as the file (minus the filename suffix). A source file `source.java` is compiled to Java class files (with filename suffix `.class`) by a Java compiler, such as `jikes` or `javac`:

```
jikes source.java
```

This creates one class file for each class or interface declared in the source file `source.java`. A class or interface `C` declared in a top-level declaration produces a class file called `C.class`. A nested class or interface `D` declared inside class `C` produces a class file called `C$D.class`. A local class `D` declared inside a method in class `C` produces a class file called `C$1$D.class` or similar.

A Java class `C` which declares the method `public static void main(String[] args)` can be executed using the Java runtime system `java` by typing a command line of the form

```
java C arg1 arg2 ...
```

This will execute the body of method `main` with the command line arguments *arg*1, *arg*2, ... bound to the array elements `args[0]`, `args[1]`, ... of type `String` inside the method `main`. The program in Example 6 concatenates all the command line arguments.

17 Packages

Java source files may be organized in *packages*. Every source file belonging to package `p` must begin with the `package` declaration

```
package p;
```

and must be stored in a subdirectory called `p`. A class declared in a source file with no `package` declaration belongs to the anonymous *default package*. A source file not belonging to package `p` may refer to class `C` from package `p` by using the qualified name `p.C`, in which the class name is prefixed by the package name. To use the unqualified class name without the package name prefix, the source file must begin with an `import` declaration (possibly following a `package` declaration):

```
import p.C;
```

Alternatively, it may begin with an `import` declaration of the form:

```
import p.*;
```

after which all accessible class and interface names from package `p` can be used unqualified. The Java class library package `java.lang` is implicitly imported into all source files. Hence all `java.lang` classes, such as `String` and `Math`, can be referred to unqualified, without the package name.

Package names may be composite. For example, the Java class library package `java.util` contains the class `Vector`, which is declared in file `java/util/Vector.java`. The qualified name of that class is `java.util.Vector`; to avoid the package prefix, use one of these declarations:

```
import java.util.Vector;
import java.util.*;
```


Example 73 The vessel hierarchy as a package
The package `vessel` below contains part of the vessel hierarchy (Example 16). The fields in classes `Tank` and `Barrel` are `final`, so they cannot be modified after object creation. They are `protected`, so they are accessible in subclasses declared outside the `vessel` package, as shown in file `Usevessels.java` below (which is in the anonymous default package, not the `vessel` package).

The file `vessel/Vessel.java`

```
package vessel;
public abstract class Vessel {
  private double contents;
  public abstract double capacity();
  public final void fill(double amount)
  { contents = Math.min(contents + amount, capacity()); }
  public final double getContents() { return contents; }
}
```

The file `vessel/Tank.java`

```
package vessel;
public class Tank extends Vessel {
  protected final double length, width, height;
  public Tank(double l, double w, double h) { length = l; width = w; height = h; }
  public double capacity() { return length * width * height; }
  public String toString()
  { return "tank (l,w,h) = (" + length + ", " + width + ", " + height + ")"; }
}
```

The file `vessel/Barrel.java`

```
package vessel;
public class Barrel extends Vessel {
  protected final double radius, height;
  public Barrel(double r, double h) { radius = r; height = h; }
  public double capacity() { return height * Math.PI * radius * radius; }
  public String toString() { return "barrel (r, h) = (" + radius + ", " + height + ")"; }
}
```

The file `Usevessels.java`

Subclass `Cube` of class `Tank` may access the field `length` because that field is declared `protected` in `Tank` above. The `main` method is unmodified from Example 17.

```
import vessel.*;
class Cube extends Tank {
  public Cube(double side) { super(side, side, side); }
  public String toString() { return "cube (s) = (" + length + ")"; }
}
class Usevessels {
   public static void main(String[] args) { ... }
}
```

Notational conventions in this document

Symbol	Meaning
`v`	value of any type
`x`	variable or parameter or field or array element
`e`	expression
`t`	type (base type or reference type)
`s`	expression of type string
`m`	method
`f`	field
`C`	class
`E`	exception type
`I`	interface
`a`	expression or value of array type
`o`	expression or value of object type
sig	signature of method or constructor
`p`	package
`u`	expression or value of thread type

Subjects not covered in this document

Input and output; Garbage collection and finalization; Reflection; Details of IEEE754 floating-point numbers.

18 References

At `http://java.sun.com/docs/` and `http://java.sun.com/j2se/` there is detailed documentation for on-line browsing or downloading. Much documentation is available in print also:

- The authoritative reference on the Java programming language is Gosling, Joy, Steele, Bracha: *The Java Language Specification*, Second Edition, Addison-Wesley, June 2000 (544 pages).

  Browse or download in HTML (573 KB) at `http://java.sun.com/docs/books/jls/`
- An introduction to all aspects of Java programming is Arnold, Gosling, and Holmes: *The Java Programming Language*, Third Edition, Addison-Wesley 2000 (624 pages).
- The Java class libraries (or Java Core API) are described in two volumes: Chan, Lee, and Kramer: *The Java Class Libraries, Second Edition, Volume 1: java.io, java.lang, java.math, java.net, java.text, java.util*, Addison-Wesley 1998 (2050 pages); and Chan and Lee: *The Java Class Libraries, Second Edition, Volume 2: java.applet, java.awt, java.beans*, Addison-Wesley 1997 (1682 pages), plus a supplement: Chan, Lee, and Kramer: *The Java Class Libraries: 1.2 Supplement*, Addison-Wesley 1999 (1157 pages).

  Class library version 1.3 can be browsed at `http://java.sun.com/j2se/1.3/docs/api/` or downloaded at `http://java.sun.com/j2se/1.3/docs.html` (22 MB).

19 Index