Peter Sestoft

Java Precisely

Third Edition

Manuscript version 0.93.1 of 2015-07-28

The MIT Press
Cambridge, Massachusetts
London, England

TO BE UPDATED

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in Times by the author using LaTeX.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Sestoft, Peter.
Java precisely / Peter Sestoft — 2nd ed.
p. cm.
Includes bibliographic references and index.
ISBN 0-262-69325-9 (pbk. : alk. paper)
1. Java (Computer program language) I. Title.

QA76.73.J38S435 2005
005.13′3—dc22 2005043349

10 9 8 7 6 5 4 3 2 1

Contents

11.13 Lambda Expressions (Java 8)

A lambda expression evaluates to a function, a value that implements a functional interface (section 23). A lambda expression has one of these three forms, each having zero or more parameters and a lambda body:

```
x -> ebs
(x1, ..., xn) -> ebs
(formal-list) -> ebs
```

Here x is a variable and the lambda body ebs is either an expression or a block statement {...}. The *formal-list* is the same as for methods in section 9.8. In the two first forms, the parameters are implicitly typed (types are inferred), and in the third form they are explicitly typed (types are declared). A lambda expression cannot have a mixture of implicitly and explicitly typed parameters.

A lambda expression (like a method reference expression, section 11.14) can appear only on the right-hand side of an assignment, as an argument in a call, in a cast, or in a method return statement. This is because the lambda expression may have many different types and therefore needs a targeted function type such as Function<String,Integer> from section 23.5. The targeted type is provided by the assignment left-hand side, the parameter type, the cast type, or the method return type.

Evaluation of a lambda expression produces an instance fv of a class implementing a functional interface (section 23), but does not cause evaluation of the lambda body ebs. To cause evaluation of the lambda body, call the single abstract method of the function interface, as in fv.apply(...); see example 65. If execution of the lambda body throws an exception, then that exception will propagate to the call of the function.

In a lambda body that is a block statement, either no return statement has an associated expression, or else all have the form return e and execution cannot "fall through" by reaching the end of the block statement. A return statement returns from the (innermost) enclosing lambda expression, not from any enclosing method.

A variable captured in a lambda body must be declared final or be effectively final, just as for local classes (section 9.11): it must not be the target of reassignment or of pre/post increment/decrement operators.

An occurrence of this or super in a lambda body ebs means the same as in the lambda expression's context, unlike in an anonymous inner class IC, where this refers to the IC instance and super to methods and fields in the base class of IC.

11.14 Method Reference Expressions (Java 8)

A method reference expression has one of these six forms, where t is a type, m a method name, e an expression and C a classname. Example 67 illustrates all of these:

```
t :: m
e :: m
super :: m
t . super :: m
C :: new
t[]...[] :: new
```

The first five of these can further take optional type arguments <t1...tn> between the :: separator and the method name m; the last form cannot. Such type arguments are used to resolve type parameters of the indicated method m or class C constructor. Also, the number n of type parameters is used to limit the search for applicable methods. Note that one cannot specify the method's actual signature (argument types).

Example 64 Lambda Expressions
This example shows the forms of lambda expressions, where the targeted function type is the variable type, such as Function<String,Integer>; see page 125 for the types used. The `fsi1`–`fsi4` are bound to lambda expressions that parse a string as an integer. The `fsis1`–`fsis3` are bound to lambda expressions that return the at most 3-letter substring of `s` starting at `i`. The `concat` is a function that concatenates its two string arguments. The `now` is a function that returns the date and time when `now.get()` is called, not when `now` is defined. The `show1` and `show2` are functions with return type `void`. Example 65 shows how to call these functions. The `fsas1`–`fsas3` are functions that take an array of strings and return their concatenation, with separator `":"`.

In addition to being bound to variables, lambda expressions are often passed as arguments to methods (in examples 165, 171, 176 and others), or are being returned from methods (in example 173).

```
Function<String,Integer>
  fsi1 = s -> Integer.parseInt(s),
  fsi2 = s -> { return Integer.parseInt(s); },
  fsi3 = (String s) -> Integer.parseInt(s),
  fsi4 = (final String s) -> Integer.parseInt(s);
BiFunction<String,Integer,String>
  fsis1 = (s, i) -> s.substring(i, Math.min(i+3, s.length())),
  fsis2 = (s, i) -> { int to = Math.min(i+3, s.length()); return s.substring(i, to); };
BiFunction<String,String,String>
  concat = (s1, s2) -> s1 + s2;
Supplier<String>
  now = () -> new java.util.Date().toString();
Consumer<String>
  show1 = s -> System.out.println(">>>" + s + "<<<"),
  show2 = s -> { System.out.println(">>>" + s + "<<<"); };
Function<String[],String>
  fsas1 = ss -> String.join(":", ss),
  fsas2 = (String[] ss) -> String.join(":", ss),
  fsas3 = (String... ss) -> String.join(":", ss);
```

Example 65 Calling Lambda-Defined Functions
The function values defined in example 64 can be called as follows, using the method names of the respective functional interfaces, shown on page 125. The type of a function determines how it can be called, so `fsas3` above must be called with a single `String[]` argument, not as a variable-arity function; but see example 163.

```
System.out.println(fsi1.apply("004711"));
System.out.println(fsis1.apply("abcdef", 4));
show1.accept(now.get());
System.out.println(fsas1.apply(new String[] { "abc", "DEF" }));
// fsas3.apply("abc", "DEF");                  // Illegal: Must take one String[] argument
```

Example 66 Higher-Order Lambda Expressions
A lambda expression may be higher-order: return another function as result, or take another function as argument, as illustrated by `prefix` and `twice` below. The types look complex but are very descriptive.

```
Function<String,Function<String,String>> prefix = s1 -> s2 -> s1 + s2;
Function<String,String> addDollar = prefix.apply("$");
BiFunction<Function<String,String>,String,String> twice = (f, s) -> f.apply(f.apply(s));
Function<String,String> addTwoDollars = s -> twice.apply(addDollar, s);
prefix.apply("$").apply("100") ... addDollar.apply("100") ... addTwoDollars.apply("100")
```

A method reference expression (like a lambda expression, section 11.13) can appear only on the right-hand side of an assignment, as a method argument, in a cast, or in a method `return` statement. This is because the method reference expression may have many different types and therefore needs a targeted function type such as Function<String,Integer> from section 23.5. The targeted type is provided by the assignment left-hand side, the parameter type, the cast type, or the method return type.

The compile-time processing of a method reference expression `t::m` or `e::m` consists of these steps:

- First, determine which type should be searched for the denoted method. In form `t::m`, type `t` must be a reference type and that is the type to search; in form `e::m`, expression `e` must have a reference type, and that is the type to search.
- Second, search that type for applicable methods, based on both the argument count of the targeted function type, the method name `m`, and if there are optional *type arguments* `<t1...tn>`, the method must have `n` type parameters. When the targeted function type takes `k` *normal arguments* and the method reference expression has form `t::m`, search both for `k`-argument static methods and `(k-1)`-argument instance methods; if the method reference expression has form `e::m`, search only for `k`-argument instance methods.
- Third, if there are any applicable methods, then search among them for static as well as instance methods with appropriate receiver and argument types for the targeted function type. If `t` in `t::e` is a raw type such as ArrayList, this involves finding appropriate type parameters for `t`, to obtain for instance ArrayList<String>. If this search produces a unique method `M`, the search is successful, otherwise the method reference expression is rejected, either because it refers to no methods or to more than one.

A method reference expression of form `C::new` evaluates to an instance constructor with argument count and parameter types determined by the targeted function type.

A method reference expression of form `t[]::new` or `t[][]::new` and so on evaluates to an array allocation function taking a single integer argument, such as `i -> new t[i]` or `i -> new t[i][]` and so on; the argument determines the length of the array's first dimension only. As with normal array instance creation, the element type `t` must not be a generic type instance such as ArrayList<Integer> or a type parameter.

At run-time, a method reference expression must evaluate to a functional value `fv`, through these steps:

- First, in an expression of the form `e::m`, evaluate `e` to a reference `rec`, and if it is null, throw an exception.
- Second, the functional value `fv` is produced. This must be an instance of an internally generated class FC that implements the targeted functional interface and therefore must have a method such as `apply`; see section 23. The value `fv` is produced either by creating a new class instance or by finding and returning an appropriate existing instance. The `apply` method of class FC may perform boxing or unboxing of arguments and result to both implement the functional interface's single abstract method and also pass appropriate arguments to the method `M` previously found during the compile-time method reference search; this is needed for `charat` in example 67. Also, if a reference `rec` was obtained in the first run-time step, then `rec` will be stored in the FC instance and used as the receiver argument of instance method `M`.

The run-time evaluation of a method reference expression to function value `fv` does not involve calling the method `M` found by the compile-time search. Only when `fv` is called, as in `fv.apply(...)`, will `M` be called.

The Java Language Specification [1] gives many more details, especially on the compile-time search.

In addition to the examples opposite, further uses of method reference expressions are illustrated by examples 155, 161, 179, and 180.

Example 67 Method Reference Expressions
The method reference expression bound to `charat` below is a `t::m` reference to an instance method; `parseint` is a `t::m` to a static method; `hex1` is an `e::m` reference giving an explicit receiver `e` (the string) to method `charAt` on class String. The `makeConverter` method shows that the expression part `e` of an `e::m` reference can be complex. Variable `makeC` is bound to the `C(int)` constructor of class C shown further below.

Variable `make1DArray` refers to a method equivalent to `i -> new Double[i]`. In `mkDoubleList`, the type parameter list <Double> is for the generic class ArrayList; in `sorter`, the type parameter list <Double> is for the (static) generic method `sort` on non-generic class Arrays. Class C shows how `this` and `super` can be used to resolve a method reference `e::getVal` to either class C's or superclass B's `getVal` method.

```
BiFunction<String,Integer,Character> charat = String::charAt;              // t::m
Function<String,Integer> parseint = Integer::parseInt;                     // t::m
Function<Integer,Character> hex1 = "0123456789ABCDEF"::charAt;             // e::m
Function<Integer,C> makeC = C::new;                                        // C::new
Function<Integer,Double[]> make1DArray = Double[]::new;                    // t[]::new
Consumer<String> print = System.out::println;                              // e::m
Function<Integer,ArrayList<Double>> mkDoubleList = ArrayList<Double>::new;  // t::new
BiConsumer<Double[],Comparator<Double>> sorter = Arrays::<Double>sort;      // t::<t1>m
private static Function<Integer,Character> makeConverter(boolean uppercase)
{ return (uppercase ? "0123456789ABCDEF" : "0123456789abcdef") :: charAt; }  // e::m
class B {
  protected int val;
  public int getVal() { return val; }
}
class C extends B {
  public C(int val) { this.val = val; }
  public Supplier<Integer> getBVal() { return super::getVal; }             // super::m
  public Supplier<Integer> getCVal() { return this::getVal; }              // this::m
  public int getVal() { return 117 * val; }
}
```

Example 68 A Lambda Expression or Method Reference Expression Needs a Targeted Function Type
A lambda expression or method reference expression has no type in itself and so must appear in a context where it has a targeted function type; four such contexts are shown below. In particular, a method reference expression cannot appear directly as the receiver of a method call as in `Double::toHexString.andThen(...)`:

```
// int len0 = Double::toHexString.andThen(String::length).apply(123.5); // Illegal
Function<Double,String> hexFun;
hexFun = Double::toHexString;                            // Legal: Assignment right-hand side
int len1 = hexFun.andThen(String::length).apply(123.5);
int len2 = applyAndMeasure(Double::toHexString, 123.5);  // Legal: Argument position
// Legal: In cast context:
int len3 = ((Function<Double,String>)Double::toHexString).andThen(String::length).apply(123.5);
int len4 = makeToHex().andThen(String::length).apply(123.5);
static int applyAndMeasure(Function<Double,String> hexFun, double d) {
  return hexFun.andThen(String::length).apply(d);
}
static Function<Double,String> makeToHex() {
  return Double::toHexString;                            // Legal: In return context
}
```

20 Threads, Concurrent Execution, and Synchronization

20.1 Threads and Concurrent Execution

The preceding chapters described sequential program execution, in which expressions are evaluated and statements are executed one after the other: they 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 (example 108). For much more on concurrency in Java, see [4].

A thread is created and controlled using an object of the Thread class found in the package `java.lang`. A thread executes the method `public void run()` in an object of a class implementing the Runnable interface, also found in package `java.lang`. To every thread (independent sequential activity) there is a unique controlling Thread object, so the two are often thought of as being identical.

One way to create and run a thread is to declare a class `U` as a subclass of Thread, overwriting its (trivial) `run` method. Then create an object `u` of class `U` and call `u.start()`. This will enable the thread to execute `u.run()` concurrently with other threads (example 108).

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()`. This will enable the thread to execute `o.run()` concurrently with other threads (example 112).

Threads can communicate with each other via shared state, namely, by using and assigning static fields, non-static fields, array elements, and pipes (section 26.16). By the design of Java, a local variables and method parameters cannot be shared between threads, and hence are always thread-safe.

States and State Transitions of a Thread

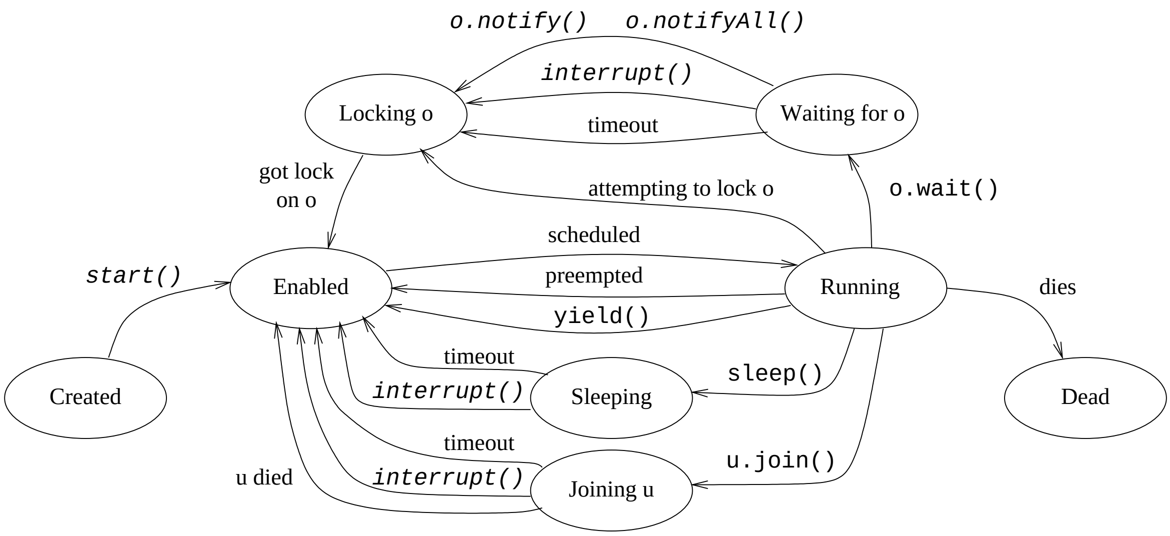
A thread is alive if it has been started and has not died. A thread dies by exiting its `run()` method, either by returning or by throwing an exception. A live thread is in one of the states Enabled (ready to run), Running (actually executing), Sleeping (waiting for a timeout), Joining (waiting for another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

From State	To State	Reason for Transition
Enabled	Running	System schedules thread for execution
Running	Enabled	System preempts thread and schedules another one
	Enabled	Thread executes `yield()`
	Waiting	Thread executes `o.wait()`, releasing lock on `o`
	Locking	Thread attempts to execute `synchronized (o) { ... }`
	Sleeping	Thread executes `sleep()`
	Joining	Thread executes `u.join()`
	Dead	Thread exited `run()` by returning or by throwing an exception
Sleeping	Enabled	Sleeping period expired
	Enabled	Thread was interrupted; throws InterruptedException when run
Joining	Enabled	Thread `u` being joined died, or join timed out
	Enabled	Thread was interrupted; throws InterruptedException when run
Waiting	Locking	Another thread executed `o.notify()` or `o.notifyAll()`
	Locking	Wait for lock on `o` timed out
	Locking	Thread was interrupted; throws InterruptedException when run
Locking	Enabled	Lock on `o` became available and was given to this thread

Example 108 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). The `volatile` modifier on `i` is needed; see section 20.5.1 and example 113.

```
class Incrementer extends Thread {
  public volatile 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);
} } }
```

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 `monospace` font), by a method call possibly performed by another thread (shown in the *`slanted monospace`* font); and by timeouts and other actions.



20.2 Locks and the `synchronized` Statement

Concurrent threads are executed independently. Therefore, when multiple concurrent threads access the same fields or array elements, there is considerable risk of creating an inconsistent state (example 110). 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 request 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 the previous page. 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 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 release the lock when it leaves the method body.

A `synchronized` static method declaration (section 9.8) in class `C` is 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`; see section 27.1. 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 110), 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 (example 111).

If a thread `u` needs to wait for some condition to become true, or for a resource to become available, it may temporarily release its lock on object `o` by calling `o.wait()`. The thread must hold the lock on object `o`, otherwise exception IllegalMonitorStateException is thrown. The thread `u` will be added to the *wait set* of `o`, that is, the set of threads waiting for notification on object `o`. This notification must come from another thread that has obtained the lock on `o` and that executes `o.notify()` or `o.notifyAll()`. The notifying thread does not release its 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 hold the lock on `o` just as before the call (example 111).

For detailed rules governing the behavior of unsynchronized Java threads, see chapter 17 of the Java Language Specification [1].

Example 109 Mutual Exclusion
A Printer thread forever prints a dash (`-`) followed by a slash (`/`). 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 `Util.pause(200)` pauses the thread for 200 ms, whereas `Util.pause(100,300)` pauses it between 100 and 300 ms. This is done only to make the inherent nondeterminacy of unsynchronized concurrency more easily observable.

```
class Printer extends Thread {
  final 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 110 Synchronized Methods in an Object
The Bank object here 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();
```

20.3 Operations on Threads

The current thread, whose state is Running, may call these methods among others. Further Thread methods are described in the Java class library documentation [2].

- `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 there has been no call to `Thread.interrupted()` and no InterruptedException thrown since the last interrupt; 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 (and the interrupted status is set to `false`).
- `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.

20.4 Operations on Locked Objects

A thread that holds the lock on an object `o` may call the following methods, inherited by `o` from class Object.

- `o.wait()` releases the lock on `o`, changes its own state to Waiting, and adds itself to the set of threads waiting for notification on `o`. When notified (if ever), the thread must obtain the lock on `o`, so when the call to `wait` returns, it again holds the lock on `o`. May throw InterruptedException if the thread is interrupted while waiting.
- `o.wait(n)` works like `o.wait()` except that the thread will change state to Locking after `n` milliseconds regardless of whether there has been a notification on `o`. 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 on `o` (if any) and changes its state to Locking. The chosen thread cannot actually obtain the lock on `o` until the current thread has released it.
- `o.notifyAll()` works like `o.notify()`, except that it changes the state to Locking for *all* threads waiting for notification on `o`.

Example 111 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 (section 20.2). 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 `notifyAll`, 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`) to emphasize that synchronization, `wait`, and `notifyAll` all work on the same 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.notifyAll();
      return contents;
  } }
  public void put(int v) {
    synchronized (this) {
      while (!empty)
        try { this.wait(); } catch (InterruptedException x) {};
      empty = false;
      contents = v;
      this.notifyAll();
  } }
}
```

Example 112 Graphic Animation Using the Runnable Interface
Class AnimatedCanvas here 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 interface Runnable
    for (;;) { // Forever sleep and repaint
      try { Thread.sleep(100); } catch (InterruptedException e) { }
      ...
      repaint();
    }
  }

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

20.5 The Java Memory Model and Visibility Across Threads

Concurrent threads in Java communicate via shared mutable memory, for instance in the form of mutable fields accessible to multiple threads. This raises the question of *visibility of writes across threads*: when will a write `x = 42` to shared field `x` performed by thread A be visible to another thread B that reads `x`? Due to optimizations performed by the Java JIT compiler and due to the memory caches of modern multicore processors, the surprising answer may be "never" or "later than you would think"; see examples 113 and 114.

However, the Java Memory Model (since Java 5) guarantees that writes to a shared field `x` or shared array element `a[i]` by thread A are visible to reads performed by thread B in these cases:

- Thread A releases a lock after the write to `x` or `a[i]`, and then thread B acquires the same lock before the read. Hence leaving and then entering `synchronized` methods and blocks enforce visibility.
- Field `x` itself is declared `volatile` and the write in A precedes the read in B in real time.
- Thread A writes to some `volatile` field after the write to `x` or `a[i]`, and then thread B reads the `volatile` field before reading `x` or `a[i]`. Hence the visibility of `x` or `a[i]` may "piggyback" on the visibility effects of writing and then reading any `volatile` field.
- Thread A starts thread B using method `start` from section 20.3: a thread can see every write that its creator thread did.
- Thread A terminates and B awaits the termination of A using `join` from section 20.3; a thread can see every write performed by a thread it knows has terminated.
- Concurrent collection operations from the `java.util.concurrent` package and atomic operations from the `java.util.concurrent.atomic` package also have visibility effects.

20.5.1 The `volatile` Field Modifier

The `volatile` field modifier applied to a field `x` ensures that every write to `x` by a thread A, and every other prior write performed by A, becomes visible to another thread B upon later reading `x`. The `volatile` modifier prevents the Java JIT compiler from performing certain optimizations, and it causes extra work at run-time to make one processor core's writes visible to other processor cores. This may slow down the code; see example 115.

Declaring a field `a` of array type `volatile` does *not* affect the visibility of writes to the array's elements `a[i]`. To ensure visibility of array element writes, one must build on the Java Memory Model guarantees listed above: use locking or `synchronized`; piggyback on writes and subsequent reads of other volatile fields; use atomic operations; and so on.

20.5.2 The `final` Field Modifier

If an instance field `x` is declared `final`, then the value assigned to `x` by a constructor is visible by any thread that obtains the reference returned by the constructor. Since the `final` modifier has visibility effect and also ensures that the field cannot be modified (section 9.6), it can be used to implement thread-safe immutable objects. This is useful in connection with functional programming (section 23) with parallel streams (section 24) and can also be used to avoid locking in some scenarios.

Example 113 Field Writes May Remain Forever Invisible
Without the `volatile` modifier on field `value`, the `mi.set(42)` performed by the main thread may remain forever invisible to the thread executing the `while` loop, which may therefore never terminate.

```
class MutableInteger {
  private /* volatile */ int value = 0;
  public void set(int value) { this.value = value; }
  public int get() { return value; }
}
...
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(new Runnable() { public void run() { while (mi.get() == 0) { } }});
t.start();
mi.set(42);
```

Example 114 Field Writes May Happen in a Surprising Order
Without the `volatile` modifier on the `A` and `B` fields, the hardware memory system may delay the writes to fields `A` and `B` so that both writes appear to happen after both reads of `!B` and `!A`. Thus when executing methods `ThreadA` and `ThreadB` concurrently, one may observe bizarre outcomes such as both `AWon` and `BWon` becoming 1, which does not correspond to any interleaving of the sequential operations performed by the two methods. On a 4-core Intel i7 processor this happens 1% of the time; with the `volatile` modifier it cannot happen.

```
class StoreBufferExample {
  public /* volatile */ boolean A = false, B = false;
  public int AWon = 0, BWon = 0;
  public void ThreadA() {
    A = true;
    if (!B) AWon = 1;
  }
  public void ThreadB() {
    B = true;
    if (!A) BWon = 1;
} }
```

Example 115 The `volatile` Modifier Precludes Some Optimizations
Each iteration of the `for` loop in method `isSorted` seems to read the `array` field three times: once for the array length test, and twice for the array element accesses. Without the `volatile` modifier on `array`, the Java JIT compiler will optimize this code to read the `array` field just once before the loop, and use that reference for the duration of the loop. This enables array bounds check elimination and makes the code run 5 times faster. When the `volatile` modifier is present, such optimizations would be wrong and are not performed.

```
class IntArray {
  private /* volatile */ int[] array;
  public boolean isSorted() {
    for (int i=1; i<array.length; i++)
      if (array[i-1] > array[i])
        return false;
    return true;
  }
}
```

23 Functional Interfaces (Java 8)

Java supports functional programming through functional interfaces (section 23.2), lambda expressions (section 11.13), and method reference expressions (section 11.14).

23.1 Functional Programming

Functional programming has many uses, especially in connection with streams (section 24) and parallel processing of arrays (section 8.4). It goes back to Lisp (1960), and more recent functional languages include ML, Scheme, OCaml, Haskell, F#, Scala and Clojure. Some characteristics of functional programming are:

- Functional programming uses immutable data structures (in which all fields are final and refer only to immutable data) instead of objects with mutable state. In example 154, the call `list1.insert(1, 12)` produces a new list instead of updating the existing `list1`. This may seem to cause excessive allocation of data, but much less than one might think because immutability permits sharing between new and old data. A major advantage is that immutable data structures are automatically thread-safe and require no synchronization when used from multiple threads. See example 176.
- Functional programming performs most iteration using (recursive) function calls instead of `for` loops, `while` loops, and the like. This would lead to deep method call stacks if the implementations did not optimize tail calls, that is, calls performed as the last action of the calling function. Since the Java implementation does not optimize tail calls, it is often better to use loops instead of recursion. In example 154, method `getNodeRecursive` might be replaced by a loop.
- Functional programming often uses higher-order functions, that is, functions that take as argument parameters of function type or return results of function type. In the former case, the higher-order function may embody a general behavior such as data structure traversal, and its function-type argument may represent the specific action to be taken on each data element. Higher-order function `map` in example 154 takes a function argument `f` and applies it to every list element. Higher-order function `less` in example 158 returns a function that can convert numbers less than `limit*limit` into English numerals.
- Some functional programming languages (including ML, Haskell, F# and Scala) use pattern matching to make choices, instead of nested `if` and `switch` statements; this achieves great clarity and some guarantee against missed cases. Java does not support pattern matching.

Many operations on streams (section 24) and parallel operations on arrays (section 8.4) require their function arguments to be side-effect free for the operations to make sense. A function `f` is *side-effect free* or *pure* if it does not modify or rely on any modifiable state; in particular, two calls `f.apply(v1)` and `f.apply(v2)` where `v1` and `v2` are equal values must produce the same result. But note that in an object-oriented setting, where object fields are mutable by default, it is not obvious what "equal values" and "same result" really means. Also, the Java compiler cannot check that a function has no side-effects, so the correctness of functional and parallel programming must rely on care and conventions.

Some operations, such as `reduce` on streams and `parallelPrefix` on arrays, further require their BinaryOperator arguments to be associative. A binary operator `op` is *associative* if for arguments `x`, `y` and `z`, it holds that `op.apply(op.apply(x,y),z)` equals `op.apply(x,op.apply(y,z))`. Writing `op.apply(x,y)` using infix notation as x⊗y, it means that (x⊗y)⊗z must equal x⊗(y⊗z). Typical associative operators are numeric addition (`+`) and multiplication (`*`), `max`, `min`, string concatenation (`+`), logical conjunction (`&&`) and disjunction (`||`), bitwise "and" (`&`), bitwise "or" (`|`), bitwise "xor" (`^`), set union, and set intersection. Non-associative operators include numeric subtraction (`-`), division (`/`) and average, and set difference.

Example 154 Functional Programming with Immutable Lists
Class FunList<T> represents immutable lists of T values, using immutable Node<T> objects. All operations produce a new list instead of updating existing ones, so any number of operations on the same list could proceed concurrently without synchronization. A new list may share nodes with existing ones; `list1`–`list4` all share the three node objects holding 9, 13 and 0. This is harmless because nodes are immutable.

Thanks to recursion and immutability, the methods are simple. Thus `insert(i, item, xs)` says: If `i==0`, put `item` in a new node and let its tail be node list `xs`; otherwise put the first element of `xs` in a new node, and let its tail be the result of inserting `item` at position `i-1` in the rest of list `xs`. Similarly `map(f, xs)` says: If `xs` is the empty list (`null`), the result is an empty list; otherwise create a node to hold the result of applying `f` to the first element of `xs`, and let its tail be the result of mapping `f` over the rest of `xs`.

```
class FunList<T> {
  final Node<T> first;
  protected static class Node<U> {
    public final U item;
    public final Node<U> next;
    public Node(U item, Node<U> next) { this.item = item; this.next = next; }
  }
  public FunList(Node<T> xs) { this.first = xs; }
  public int getCount() { ... }
  public T get(int i) { return getNodeRecursive(i, first).item; }
  protected static <T> Node<T> getNodeRecursive(int i, Node<T> xs) {    // Could use loop instead
    return i == 0 ? xs : getNodeRecursive(i-1, xs.next);
  }
  public static <T> FunList<T> cons(T item, FunList<T> list) { return list.insert(0, item);  }
  public FunList<T> insert(int i, T item) { return new FunList<T>(insert(i, item, this.first)); }
  protected static <T> Node<T> insert(int i, T item, Node<T> xs) {
    return i == 0 ? new Node<T>(item, xs) : new Node<T>(xs.item, insert(i-1, item, xs.next));
  }
  public FunList<T> removeAt(int i) { return new FunList<T>(removeAt(i, this.first)); }
  protected static <T> Node<T> removeAt(int i, Node<T> xs) {
    return i == 0 ? xs.next : new Node<T>(xs.item, removeAt(i-1, xs.next));
  }
  public FunList<T> reverse() { ... }
  public FunList<T> append(FunList<T> ys) { ... }
  public <U> FunList<U> map(Function<T,U> f) { return new FunList<U>(map(f, first)); }
  protected static <T,U> Node<U> map(Function<T,U> f, Node<T> xs) {
    return xs == null ? null : new Node<U>(f.apply(xs.item), map(f, xs.next));
  }
  public <U> U reduce(U x0, BiFunction<U,T,U> op) { ... }
}
...
FunList<Integer> empty = new FunList<>(null),
  list1 = cons(9, cons(13, cons(0, empty))),                // 9 13 0
  list2 = cons(7, list1),                                   // 7 9 13 0
  list3 = cons(8, list1),                                   // 8 9 13 0
  list4 = list1.insert(1, 12),                              // 9 12 13 0
  list5 = list2.removeAt(3),                                // 7 9 13
  list6 = list5.reverse(),                                  // 13 9 7
  list7 = list5.append(list5);                              // 7 9 13 7 9 13
FunList<Double> list8 = list5.map(i -> 2.5 * i);            // 17.5 22.5 32.5
double sum = list8.reduce(0.0, (res, item) -> res + item);  // 72.5
```

23.2 Generic Functional Interfaces

A *functional interface* is an interface that has a single abstract method such as `apply`, `test` or `accept`, and possibly some default and static methods. A concrete instance of a functional interface represents a function, namely the implementation of the interface's single abstract method. Functions are especially useful in connection with stream pipelines (section 24) and parallel processing of arrays (section 8.4).

The generic functional interfaces from package `java.util.function` are listed in the table opposite, along with the corresponding function type notation, as used in many other languages. The arrow (`->`) indicates a function type, and the star (`*`) indicates a product or pair type. More precisely, `T -> R` is the type of a function that takes arguments of type `T` and returns results of type `R`, and `T * U` is the type of a pair of a value of type `T` and a value of type `U`. These can be combined, so `T * U -> R` is the type of a function that takes two arguments of type `T` and `U` and returns a result of type `R`.

Conceptually, the most important functional interface is Function<T,R>, the type of a function that takes an argument of type T and returns a result of type R, corresponding to the function type `T -> R`. The interface's single abstract method is `R apply(T x)`.

Thus Function<String,Integer> is the type of a function that takes a String argument and returns an Integer result. A value of such a function type can be produced in numerous ways, usually from a lambda expression (section 11.13) or a method reference expression (section 11.14) as shown in example 155.

There are other functional interfaces, such as Comparator<T> from package `java.util`; see section 22.10.

23.3 Primitive-Type Specialized Functional Interfaces

The table's list of functional interfaces is very long, but many of them are just *primitive-type specialized* versions of the more generic functional interfaces Function<T,R> and so on, in particular for R and T being Double, Integer, and Long. The primitive-type specialized interfaces exist purely for efficiency reasons. The problem is that Java's generic types, such as Function<T,R>, can take only reference type arguments such as Integer and Long, not primitive type arguments such as a `int` and `long`; see section 21.11. But calling a function represented by an instance of Function<Integer,Long> means calling the method `Long apply(Integer x)`, and this involves checking and unwrapping of the Integer argument and subsequent wrapping of the `long` result as a Long object, which is inefficient. Using instead the `long applyAsLong(int x)` method on an instance of the primitive-type specialized interface IntToLongFunction avoids this run-time overhead. But it also makes the list of functional interfaces bulky and daunting to look at; and it could be even worse, had the Java class library included versions for Byte, Character, Float and Short, which it sensibly does not.

Interface	Sec.	Function Type	Single Abstract Method Signature
One-Argument Functions and Predicates			
Function<T,R>	23.5	T -> R	R apply(T)
UnaryOperator<T>	23.6	T -> T	T apply(T)
Predicate<T>	23.7	T -> boolean	boolean test(T)
Consumer<T>	23.8	T -> void	void accept(T)
Supplier<T>	23.9	void -> T	T get()
Runnable		void -> void	void run()
Two-Argument Functions and Predicates			
BiFunction<T,U,R>	23.10	T * U -> R	R apply(T, U)
BinaryOperator<T>	23.11	T * T -> T	T apply(T, T)
BiPredicate<T,U>	23.7	T * U -> boolean	boolean test(T, U)
BiConsumer<T,U>	23.8	T * U -> void	void accept(T, U)
Primitive-Type Specialized Versions of the Generic Functional Interfaces			
DoubleToIntFunction	23.5	double -> int	int applyAsInt(double)
DoubleToLongFunction	23.5	double -> long	long applyAsLong(double)
IntToDoubleFunction	23.5	int -> double	double applyAsDouble(int)
IntToLongFunction	23.5	int -> long	long applyAsLong(int)
LongToDoubleFunction	23.5	long -> double	double applyAsDouble(long)
LongToIntFunction	23.5	long -> int	int applyAsInt(long)
DoubleFunction<R>	23.5	double -> R	R apply(double)
IntFunction<R>	23.5	int -> R	R apply(int)
LongFunction<R>	23.5	long -> R	R apply(long)
ToDoubleFunction<T>	23.5	T -> double	double applyAsDouble(T)
ToIntFunction<T>	23.5	T -> int	int applyAsInt(T)
ToLongFunction<T>	23.5	T -> long	long applyAsLong(T)
ToDoubleBiFunction<T,U>	23.10	T * U -> double	double applyAsDouble(T, U)
ToIntBiFunction<T,U>	23.10	T * U -> int	int applyAsInt(T, U)
ToLongBiFunction<T,U>	23.10	T * U -> long	long applyAsLong(T, U)
DoubleUnaryOperator	23.6	double -> double	double applyAsDouble(double)
IntUnaryOperator	23.6	int -> int	int applyAsInt(int)
LongUnaryOperator	23.6	long -> long	long applyAsLong(long)
DoubleBinaryOperator	23.11	double * double -> double	double applyAsDouble(double, double)
IntBinaryOperator	23.11	int * int -> int	int applyAsInt(int, int)
LongBinaryOperator	23.11	long * long -> long	long applyAsLong(long, long)
DoublePredicate	23.7	double -> boolean	boolean test(double)
IntPredicate	23.7	int -> boolean	boolean test(int)
LongPredicate	23.7	long -> boolean	boolean test(long)
DoubleConsumer	23.8	double -> void	void accept(double)
IntConsumer	23.8	int -> void	void accept(int)
LongConsumer	23.8	long -> void	void accept(long)
ObjDoubleConsumer<T>	23.8	T * double -> void	void accept(T, double)
ObjIntConsumer<T>	23.8	T * int -> void	void accept(T, int)
ObjLongConsumer<T>	23.8	T * long -> void	void accept(T, long)
BooleanSupplier	23.9	void -> boolean	boolean getAsBoolean()
DoubleSupplier	23.9	void -> double	double getAsDouble()
IntSupplier	23.9	void -> int	int getAsInt()
LongSupplier	23.9	void -> long	long getAsLong()

23.4 Covariance and Contravariance in Functional Interfaces

In general, whenever a method void m(Function<T,R> f) expects an argument f whose type is a functional interface such as Function<T,R>, it is acceptable to provide an actual function f that accepts an argument of a supertype of T and produces a result of a subtype of R. One says that function types are *contravariant* in their argument types and *covariant* in their result type. In Java's type system this flexibility is expressed using wildcard type arguments; see section 21.9 and example 157. Thus the more flexible signature of method m would be described like this: void m(Function<? super T, ? extends R> f)

However, this makes the descriptions of methods that take functional arguments considerably more verbose and harder to read. Hence in most cases this book uses the simple form Function<T,R> rather than the more general Function<? super T, ? extends R>; we do this in particular for the stream methods in section 24.3. Similarly, we write Comparator<T> instead of Comparator<? super T>, write Predicate<T> instead of Predicate<? super T>, and so on. Respecting the method signatures as shown will always work, but the actual Java library implementation is more accommodating.

23.5 Interface Function<T,R>

The functional interface Function<T,R> describes one-argument functions of type T -> R, that is, those that take an argument of type T and return a result of type R. It has a single abstract method apply and some default and static methods:

- abstract R apply(T x) is the function represented by an object implementing the interface.
- default Function<T,V> andThen(Function<R,V> after) returns a function that applies function after to the result of this function; that is, (T x) -> after.apply(this.apply(x)).
- default Function<V,R> compose(Function<V,T> before) returns a function that applies this function to the result of function before, that is, (V x) -> this.apply(before.apply(x)).
- static Function<T,T> identity() returns the identity function (T x) -> x on type T.

The primitive-type specialized interfaces {Double,Long,Int}Function and To{Double,Long,Int}Function and {Double,Long,Int}To{Double,Long,Int}Function have only the abstract methods in page 125. They do not implement the default methods andThen and compose as it would require many overloads; see example 162.

23.6 Interface UnaryOperator<T>

The functional interface UnaryOperator<T> extends Function<T,T> and describes one-argument functions of type T -> T that take an argument of type T and return a result of the same type T. It has the single abstract method apply, the default methods andThen and compose described by Function<T,T>, and a static method:

- abstract T apply(T x) is the unary operator represented by an implementation of the interface.
- default Function<T,V> andThen(Function<T,V> after) returns a function that applies function after to the result of this unary operator; that is, x -> after.apply(this.apply(x)).
- default Function<V,T> compose(Function<V,T> before) returns a function that applies this unary operator to the result of function before, that is, x -> this.apply(before.apply(x)).
- static UnaryOperator<T> identity() returns the identity unary operator x -> x on type T.

There are primitive-type specialized interfaces {Double,Int,Long}UnaryOperator with the same default and static methods and with single abstract methods named applyAs{Double,Int,Long}; see page 125.

Example 155 Some Ways to Obtain a Function<String,Integer>
A value of type Function<String,Integer>, that is, a function from String to Integer, can be obtained in many ways, as shown by the definitions of `fsi1`–`fsi7` below, where `fsi1`–`fsi4` parse a string as an integer, and `fsi5`–`fsi7` return a string's length. The lambda (section 11.13) and method reference (section 11.14) notation are more compact than the anonymous inner class notation used for `fsi7`, but the resulting function values are invoked the same way, as `fsi1.apply("4711")`. Compile-time type inference finds the correct Integer constructor overload in the `fsi4` case and the correct `length` method in the `fsi5` case.

```
Function<String,Integer>
  fsi1 = s -> Integer.parseInt(s),            // lambda with parameter s
  fsi2 = (String s) -> Integer.parseInt(s),   // same, with explicit parameter type
  fsi3 = Integer::parseInt,                   // reference to static method Integer.parseInt
  fsi4 = Integer::new,                        // reference to constructor Integer(String)
  fsi5 = s -> s.length(),                     // lambda with parameter s
  fsi6 = String::length,                      // reference to instance method s.length()
  fsi7 = new Function<String,Integer>() {     // anonymous inner class (Java 1.1)
           public Integer apply(String s) {
             return s.length();
         }};
```

Example 156 Multiple Traversals of a Stream
With function interfaces one can encapsulate general behaviors in methods that take function-type arguments in a type-safe manner. For instance, method `traverse1` below encapsulates the notion of transforming a stream `xs` with element type `T`, first by a function `f` of type `T->U` and then by a function `g` of type `U->V`; the result is a stream with element type `V`.

Function `traverse2` computes exactly the same result, but makes only one "traversal" of the stream, applying the composed function `f.andThen(g)` to each element. Java's stream implementation may in fact automatically fuse the double "traversal" into a single one.

```
public static <T,U,V> Stream<V> traverse1(Stream<T> xs, Function<T,U> f, Function<U,V> g) {
  return xs.map(f).map(g);
}
public static <T,U,V> Stream<V> traverse2(Stream<T> xs, Function<T,U> f, Function<U,V> g) {
  return xs.map(f.andThen(g));
}
```

Example 157 Wildcard Types for a More Accommodating Method Signature
Method `traverse2` from example 156 cannot be applied to a stream `xs` of type Stream<Long>, a function `f` of type Function<Number,String> and a function `g` of type Function<Object,Integer>, because the types do not match. Type Number is different from Long, and type Object is different from String. However, since Number is a supertype of Long, and Object is a supertype of String, the function composition should actually work. Using wildcard types (section 21.9) we can safely give `traverse3` below a more accommodating signature and then the application `traverse3(xs, f, g)` works:

```
public static <T,U,V> Stream<V> traverse3(Stream<T> xs, Function<? super T, ? extends U> f,
                                                         Function<? super U, ? extends V> g) {
  return xs.map(f.andThen(g));
}
// Stream<Double> res = traverse2(xs, f, g);              // Type error!
Stream<Integer> res = traverse3(xs, f, g);
```

23.7 Interfaces Predicate<T> and BiPredicate<T,U>

The functional interface Predicate<T> describes one-argument predicates of type `T -> boolean`, that is, functions that take an argument of type T and return a truth value. It has the single abstract method `test` and some default and static methods:

- `abstract boolean test(T x)` is the predicate represented by an object implementing the interface.
- `default Predicate<T> and(Predicate<T> p)` returns a predicate that is a short-circuiting logical conjunction ("and") of this predicate and `p`; that is, `x -> this.test(x) && p.test(x)`.
- `static <T> Predicate<T> isEqual(Object y)` returns a predicate that tests whether its argument equals `y`; that is, `x -> Objects.equals(y,x)`.
- `default Predicate<T> negate()` returns a predicate that represents the logical negation ("not") of this predicate; that is, `x -> !this.test(x)`.
- `default Predicate<T> or(Predicate<T> other)` returns a predicate that is a short-circuiting logical disjunction ("or") of this predicate and `p`; that is, `x -> this.test(x) || p.test(x)`.

The primitive-type specialized interfaces {Double,Int,Long}Predicate have the same methods except `isEqual`.

The functional interface BiPredicate<T,U> describes two-argument predicates and has a single abstract method (and also the default methods but not the `isEqual` method of Predicate<T>):

- `abstract boolean test(T x, U y)` is the predicate represented by the interface implementation.

23.8 Interfaces Consumer<T> and BiConsumer<T,U>

The functional interface Consumer<T> describes one-argument consumers of type `T -> void`, that is, functions that take an argument of type T and return nothing. It has a single abstract method and a default method:

- `abstract void accept(T x)` is the consumer represented by an object implementing the interface.
- `default Consumer<T> andThen(Consumer<T> after)` returns a Consumer<T> that performs `this.accept(x)` followed by `after.accept(x)`.

The primitive-type specialized interfaces {Double,Int,Long}Consumer have corresponding default methods.

The functional interface BiConsumer<T,U> describes two-argument consumers and has a single abstract method (and also a corresponding default method `andThen`):

- `abstract void accept(T x, U y)` is the consumer represented by the interface implementation.

The primitive-type specialized interfaces Obj{Double,Int,Long}Consumer have corresponding abstract methods; see page 125.

23.9 Interface Supplier<T>

The functional interface Supplier<T> describes one-argument suppliers of type `void -> T`, that is, functions that take no arguments and produce a result of type T. It has the single abstract method `get`:

- `abstract T get()` is the supplier represented by an implementation of the interface.

The primitive-type specialized interfaces {Boolean,Double,Int,Long}Supplier have corresponding abstract methods, named `getAs{Boolean,Double,Int,Long}`; see page 125.

Example 158 Converting Numbers to English Numerals
The conversion of a long integer to an English numeral, such as converting 2,147,483,647 to "two billion one hundred fourty seven million four hundred eighty three thousand six hundred fourty seven", follows a simple pattern, captured by method `less` below. Method `less` returns a function `(n -> ...)` of type LongFunction<String>, and is called four times to define functions `less1K`, ..., `less1G` where the latter handles numbers in the range $\pm 10^{12}$ and is called by method `toEnglish`. To extend the range to $\pm 10^{15}$ or 1000 trillion, simply define an appropriate fifth function `less1T` using `less` and `less1G`, and call `less1T` from `toEnglish`.

```
private static final String[] ones = { "", "one", "two", ..., "nineteen" },
                              tens = { "twenty", "thirty", ..., "ninety" };
private static String after(String s) { return s.equals("") ? "" : " " + s; }
private static String less100(long n) {
  return n<20 ? ones[(int)n] : tens[(int)n/10-2] + after(ones[(int)n%10]);
}
private static LongFunction<String> less(long limit, String unit, LongFunction<String> conv) {
  return n -> n<limit ? conv.apply(n)
                      : conv.apply(n/limit) + " " + unit + after(conv.apply(n%limit));
}
private static final LongFunction<String>
  less1K = less(          100, "hundred",  Numerals::less100),
  less1M = less(        1_000, "thousand", less1K),
  less1B = less(    1_000_000, "million",  less1M),
  less1G = less(1_000_000_000, "billion",  less1B);
public static String toEnglish(long n) {
  return n==0 ? "zero" : n<0 ? "minus " + less1G.apply(-n) : less1G.apply(n);
}
```

Example 159 Consumer Arguments
The `forEach` method on Stream<T> takes as argument a Consumer<T> and applies it to each element of the stream. To print some strings we use method reference `System.out::println` as a Consumer<String>:

```
Stream<String> ss = Stream.of("Hoover", "Roosevelt", "Truman", "Eisenhower", "Kennedy");
ss.forEach(System.out::println);
```

Example 160 Using a Supplier to Generate Infinite Streams of Natural Numbers or Fibonacci Numbers
The stream `nats` of natural numbers $0, 1, \ldots$ may be produced functionally as in example 165. Or use `generate` with an IntSupplier whose state `next` is held inside the anonymous inner class, as in `nats2` below. Or the IntSupplier may be a lambda expression `() -> next[0]++` whose state is in `next[0]` as in `nats3`:

```
IntStream nats2 = IntStream.generate(new IntSupplier() {
    private int next = 0;
    public int getAsInt() { return next++; }
  });
final int[] next = { 0 };                // next is final, its element mutable by the lambda
IntStream nats3 = IntStream.generate(() -> next[0]++);
```

The stateful `generate` approach is particularly useful when the next element depends not just on the preceding element, as in the Fibonacci number sequence. Here the lambda expression is a Supplier<BigInteger>:

```
final BigInteger[] fib = { BigInteger.ZERO, BigInteger.ONE };   // fib is final, its elements mutable
Stream<BigInteger> fibonaccis =
  Stream.generate(() -> { BigInteger f1=fib[1]; fib[1]=fib[0].add(fib[1]); return fib[0]=f1; });
```

23.10 Interface BiFunction<T,U,R>

The functional interface BiFunction<T,U,R> describes two-argument functions of type `T * U -> R`, that is, those that take two arguments of type T and U and return a result of type R. It has a single abstract method `apply` and a default method:

- `abstract R apply(T x, U y)` is the function represented by an object implementing the interface.
- `default BiFunction<T,U,V> andThen(Function<R,V> after)` returns a function that applies function `after` to the result of this function; that is, `(x, y) -> after.apply(this.apply(x,y))`.

The primitive-type specialized interfaces To{Double,Long,Int}BiFunction have only their single abstract methods, named `applyAs{Double,Int,Long}`; see page 125.

23.11 Interface BinaryOperator<T>

The functional interface BinaryOperator<T> extends the interface BiFunction<T,T,T> and describes two-argument functions of type `T * T -> T`, that is, those that take two arguments of type T and return a result of the same type T. It has the single abstract method `apply` and the default method `andThen` described by BiFunction<T,T,T>, and two static methods:

- `abstract T apply(T x, T y)` is the binary operator represented by an implementation of the interface.
- `default BiFunction<T,T,V> andThen(Function<T,V> after)` returns a function that applies function `after` to the result of this function; that is, `(x, y) -> after.apply(this.apply(x,y))`.
- `static <T> BinaryOperator<T> maxBy(Comparator<T> cmp)` returns a BinaryOperator<T> that returns the greatest of two T elements as defined by the comparator `cmp`.
- `static <T> BinaryOperator<T> minBy(Comparator<T> cmp)` returns a BinaryOperator<T> that returns the smallest of two T elements as defined by the comparator `cmp`.

The primitive-type specialized interfaces {Double,Int,Long}BinaryOperator have only their single abstract methods, named `applyAs{Double,Int,Long}`; see page 125. The static methods `max` and `min` from class Math (section 18) may sometimes be used instead of the missing `maxBy` and `minBy`.

Example 161 Some Ways to Obtain a ToIntFunction<String>
Example 155 shows how to create a function of type Function<String,Integer>, but sometimes it is better to use a primitive-type specialized value of type ToIntFunction<String>, which produces an unboxed `int`. Exactly the same definitions can be used, except that the value assigned to `fsi7` must use the correct interface and method name, different from those in example 155—which shows that lambda expressions and method references are easier to use than anonymous inner classes.

```
ToIntFunction<String>
  fsi1 = s -> Integer.parseInt(s),           // lambda with parameter s
  fsi2 = (String s) -> Integer.parseInt(s),   // same, with explicit parameter type
  fsi3 = Integer::parseInt,                   // reference to static method Integer.parseInt
  fsi4 = Integer::new,                        // reference to constructor Integer(String)
  fsi5 = s -> s.length(),                     // lambda with parameter s
  fsi6 = String::length,                      // reference to instance method s.length()
  fsi7 = new ToIntFunction<String>() {        // anonymous inner class (Java 1.1)
           public int applyAsInt(String s) {
             return s.length();
         }};
```

The `fs1`–`fs6` assignments work because a function-value expression such as `s -> Integer.parseInt(s)` does not have a type in itself. Instead, the Java compiler performs type inference and inserts boxing and unboxing operations to match the type of the variable assigned to. Thus the same function-value expression can be assigned to variables of type ToIntFunction<String> as well as Function<String,Integer>. However, there is no subtype relation or conversion between those two types, so the assignment `g = f`, where `f` does have a type in itself, will be rejected by the compiler:

```
Function<String,Integer> f = s -> Integer.parseInt(s);
// ToIntFunction<String> g = f;                // Type error!
```

Example 162 Why No `andThen` Method on Primitive-Type Specialized Interfaces?
Some of the primitive-type specialized interfaces such as {Double,Long,Int}Function do not have methods such as `andThen` and `compose` found on the corresponding generic interface. Presumably the reason is that there would be an excessive number of plausible overloads. For instance, IntFunction<R> might be expected to have these five overloads of `andThen`, and four overloads of `compose`, but has none of them:

```
// Hypothetic methods on IntFunction<R>:
default IntFunction<V> andThen(Function<R,V> after)
default IntUnaryOperator andThen(ToIntFunction<R> after)
default IntToDoubleFunction andThen(ToDoubleFunction<R> after)
default IntToLongFunction andThen(ToLongFunction<R> after)
default IntPredicate andThen(Predicate<R> after)
```

Example 163 A Functional Interface for Variable-Arity Functions
This interface describes functions that take a variable number of arguments via a parameter array (section 9.9), but may not be type-safe. By declaring `fsas1`–`fsas3` from example 64 as VarargFunction<String,String> instead of Function<String[],String>, they can all be called as `fsas1.apply("abc", "DEF")` and so on.

```
interface VarargFunction<T,R> extends Function<T[],R> {
  public abstract R apply(T... xs);
}
```

24 Streams for Bulk Data (Java 8)

A *stream* represents a number of data elements, or bulk data, though usually not explicitly stored in the manner of an array or collection. A stream is typically processed by a pipeline, built from a *stream generator*, several *intermediate stream operations*, and a single *terminal stream operation*. A stream generator produces a stream, an intermediate operation consumes and produces a stream, and a terminal operation only consumes a stream. This approach supports mostly-functional data processing, and in particular enables parallel computation in a painless and safe manner.

A stream is often lazily generated and lazily processed, and the creation and processing of stream elements is driven by the terminal operation's pull (demand) at the end of the pipeline rather than the stream generator's push at the beginning of the pipeline. So only a small fraction of the stream's elements are actually stored in memory at any given time. In fact, a lazily created stream may be infinite, although some operations such as `count()` would never terminate on such a stream. Moreover, some intermediate operations, such as consecutive `map` transformations, may be fused, so that intermediate results are actually never stored at all. This makes functional stream pipelines very fast, sometimes faster than the "obviously best" imperative code; see example 167.

For instance, while it seems that `xs.map(f).map(g)` will perform two "traversals" of the stream `xs`, transforming the elements first by function `f` and then by function `g`, this can be implemented behind the scenes by a single "traversal" `xs.map(f.andThen(g))` applying the composition of `f` and `g`; see example 156.

A stream may be *sequential* or *parallel*. On a sequential stream, intermediate and terminal operations will be performed sequentially, on a single thread. On a parallel stream, intermediate and terminal operations may be performed in parallel on multiple threads.

A stream may be *ordered* or *unordered*. For an ordered stream, intermediate operations such as `filter(p)` and `map(f)` respect the element order and produce a stream with elements in the expected order, even if the stream is parallel. For an unordered stream there is no such guarantee, so some parallel intermediate operations may be more efficient on unordered streams.

Also note that on a parallel stream, even an ordered one, there is no guaranteed order in which the functional arguments `f` and `p` are applied to the stream's elements: the only guarantee is that the resulting stream's elements appear in the expected order.

For example, the result of `IntStream.range(0,5).parallel().map(x -> x*2).toArray()` is guaranteed to be an array containing the elements `[0, 2, 4, 6, 8]`, but the function `(x -> x*2)` may be applied to element `3` before element `2`, or vice versa, or at the exact same time.

Regardless whether the stream is ordered or unordered, the terminal operation `forEach` does not respect the element order for parallel streams.

Since operations on a parallel stream may be evaluated on multiple threads and in an unpredictable order, the functions passed to stream operations should be *stateless*: they must not depend on any state that may change during the pipeline computation, not even state internal to the function. Example 172 uses a predicate that is not stateless, and therefore does not work on parallel streams.

A stream's elements can be consumed only once; trying to use a stream twice will throw IllegalStateException. In this respect Java streams are very different from Haskell's lazy lists, for instance.

When a stream is created from a source such as an array, collection or file, that source must not be modified while the stream is being used; otherwise the results are unpredictable and exceptions may be thrown. The functions passed to stream operations should be *non-interfering*: they must not modify the stream's source.

Example 164 Creating Finite Streams
A finite sequential stream can be created by enumerating its elements, from an array, or from a collection such as a set:

```
IntStream is = IntStream.of(2, 3, 5, 7, 11, 13);

String[] a = { "Hoover", "Roosevelt", "Truman", "Eisenhower", "Kennedy" };
Stream<String> presidents = Arrays.stream(a);

Collection<String> coll = new HashSet<String>();
coll.add("Denmark"); coll.add("Norway"); coll.add("Sweden");
Stream<String> countries = coll.stream();
```

Example 165 Creating an Infinite Stream of Prime Numbers
One can create an infinite sequential stream `nats` of the natural numbers by starting with 0 and adding 1 to each successive element, using the `iterate` method. One can then create an infinite stream `primes` of prime numbers (a natural number divisible only by 1 and itself) by filtering the stream of natural numbers. Simple and efficient.

```
IntStream nats = IntStream.iterate(0, x -> x+1);
IntStream primes = nats.filter(x -> isPrime(x));
```

Example 166 Creating a Finite Stream of Prime Numbers
The simplest way to create a finite sequential stream of the first `n` prime numbers is to create an infinite stream as in example 165 and then `limit` it to the first `n` elements:

```
public static IntStream primes2(int n) {
  return IntStream.iterate(0, x -> x+1).filter(x -> isPrime(x)).limit(n);
}
```

Example 167 Counting Prime Numbers: Sequential Stream, Imperative Loop, and Parallel Stream
One can count the prime numbers less than 10 million by this stream pipeline. It generates the integers between 0 and 10 million, tests whether each of them is a prime, throws away the non-primes, and counts the rest:

```
IntStream.range(0, 10_000_000).filter(i -> isPrime(i)).count()
```

Since the numbers are lazily generated, this uses very little memory, and in fact the above stream expression is just as fast as a classic efficient-looking imperative loop:

```
int count = 0;
for (int i=0; i<10_000_000; i++)
  if (isPrime(i))
    count++;
```

The real advantage of the stream pipeline is that it is trivial to parallelize: just insert `.parallel()`, then the prime number testing and counting will exploit any available parallel processor cores. The parallelized version below is 4 times faster than the imperative loop on a 4-core laptop, and 16 times faster than the imperative loop on a 32-core server. Parallelizing the imperative loop is vastly more cumbersome and not more efficient.

```
IntStream.range(0, 10_000_000).parallel().filter(i -> isPrime(i)).count()
```

24.1 Creating Streams

There are many ways to create a stream:

- By explicit enumeration of elements, using the variable-arity static method `Stream.of(T ...)` that creates a sequential ordered stream, or `Stream.empty()` that creates a stream with no elements.
- From an array, using the static methods `Arrays.stream(T[])` and `Arrays.stream(T[],from,to)` from class Arrays (section 8.4) both of which create a sequential Stream<T>. There are primitive-type specialized overloads `Arrays.stream(double[])` and `Arrays.stream(double[],from,to)` that create a sequential DoubleStream, and similar ones for IntStream and LongStream.
- From a collection, using the Collection<T> default method `coll.stream()` that creates a sequential Stream<T>, or default method `coll.parallelStream()` that creates a possibly parallel Stream<T>.
- By static generator methods on Stream<T> such as `iterate(x0, f)` and `generate(supp)`, or from IntStream's and LongStream's static methods `range(from,to)` and `rangeClosed(from,to)`.
- By imperative generation of stream elements, using a stream builder; see section 24.2.
- From a BitSet (package `java.util`) using method `stream()` which returns an IntStream of the numbers in the set; see examples 175 and 176.
- From a random number generator of class Random, using methods `ints(n)`, `ints()`, `ints(a,b)` or `ints(n,a,b)` that produce an IntStream of `n` random integers, or infinitely many random integers, possibly limited to the range `a..(b-1)`; or using corresponding methods called `doubles` and `longs` to generate a DoubleStream or LongStream.
- From a BufferedReader using the `lines()` method which generates a Stream<String>; see example 170.

24.2 Stream Builders

The Stream.Builder<T> interface from `java.util.stream` extends the Consumer<T> interface (section 23.8) and can be used to build a sequential stream using imperative programming. A stream builder computes the stream elements eagerly, so it cannot create an infinite stream, and it may do more work than necessary in case only some of the generated elements are ever consumed.

The Stream.Builder<T> interface has two abstract methods and a default one:

- `void accept(T x)` adds an element to the stream being built.
- `default Stream.Builder<T> add(T x)` works exactly as `accept(x)` but in addition returns the stream builder to allow chained calls, as in `sb.add(2).add(3).add(5)`.
- `Stream<T> build()` builds the stream and moves the stream builder to the built state. After this call, any call to `accept` or `add` will throw IllegalStateException.

The primitive-type specialized interfaces {Double,Int,Long}Stream.Builder extend the DoubleConsumer, IntConsumer and LongConsumer interfaces (section 23.8) and have the same methods as listed above, with corresponding specialized argument and return types.

Example 168 Using a Stream Builder to Create a Stream of Prime Numbers
One can use a stream builder to create a stream of the first `n` prime numbers as shown below. This will compute the prime numbers eagerly, that is, before anything is consumed from the stream. The functional way to generate such a stream is much more elegant, more efficient, and parallelizable if desired; see example 166.

```
public static IntStream primes4(int n) {
  IntStream.Builder isb = IntStream.builder();
  int p = 2, count = 0;
  while (count < n) {
    if (isPrime(p)) {
      isb.accept(p);
      count++;
    }
    p++;
  }
  return isb.build();
}
```

Example 169 Using a Stream Builder to Collect Pattern Matches
The regular expression `urlPattern` below matches a link `a href="link"` in a webpage. Using a stream builder one can create a stream of Links whose elements are pairs (`url`, `link`) of the webpage `url` and each `link` found in the webpage. The stream is built eagerly: all links must be found before the stream can be used.

```
public static Stream<Link> scanLinks(Webpage page) {
  Matcher urlMatcher = urlPattern.matcher(page.contents);
  Stream.Builder<Link> links = Stream.<Link>builder();
  while (urlMatcher.find()) {
    String link = urlMatcher.group(1);
    links.accept(new Link(page.url, link));
  }
  return links.build();
}
final static Pattern urlPattern = Pattern.compile("a href=\"(\\p{Graph}*)\"");
```

Example 170 Reading a Stream of Lines from a BufferedReader
A stream of the text lines making up a webpage can be obtained by reading the webpage through a BufferedReader (section 26.13) and creating a lazy sequential Stream<String> from the webpage. The consumer of the stream determines how much of the webpage will actually be read via the network. It might be tempting to close the BufferedReader before returning the stream of lines, but this is wrong and likely would throw an UncheckedIOException. The BufferedReader will be in use as long as lines are consumed from the stream, and only when the stream gets closed may the reader and input stream be closed too.

```
public static Stream<String> getPageLines(String url) {
  try {
    InputStreamReader isr = new InputStreamReader(new URL(url).openStream());
    BufferedReader reader = new BufferedReader(isr);
    return reader.lines();
  } catch (IOException exn) {
    return Stream.<String>empty();
  }
}
```

24.3 Methods on Streams

Interface Stream<T> from package java.util.stream has methods for creating, further processing, or consuming streams. In the descriptions below, the elements of a stream are denoted x1, x2, . . . , and in general xi. For brevity we have simplified some types in the method signatures, as explained in section 23.4.

- boolean allMatch(Predicate<T> p) returns true if p.test(xi) is true for all elements, else false.
- boolean anyMatch(Predicate<T> p) returns true if p.test(xi) is true for some element, else false.
- static <T> Stream.Builder<T> builder() returns a builder for a Stream<T>; see section 24.2.
- void close() closes this stream, causing all close handlers for this stream pipeline to be called.
- R collect(Collector<? super T,A,R> coltor) performs a mutable reduction operation on the stream using the collector; see section 24.6. Interfaces {Double,Int,Long}Stream do not have this method.
- R collect(Supplier<R> supp, BiConsumer<R,T> accumulate, BiConsumer<R,R> combine) performs a mutable reduction operation using the collector components; see section 24.6 and example 186.
- static <T> Stream<T> concat(Stream<? extends T> xs, Stream<? extends T> ys) creates a lazy stream whose elements are the elements of xs followed by the elements of ys.
- long count() returns the number of elements in this stream.
- Stream<T> distinct() returns a stream without duplicate elements, as determined by x.equals(y).
- static <T> Stream<T> empty() returns an empty sequential stream.
- Stream<T> filter(Predicate<T> p) returns a stream of the xi for which p.test(xi) is true.
- Optional<T> findAny() returns an Optional containing some element from the stream if non-empty, else returns an empty Optional; see section 25.
- Optional<T> findFirst() returns an Optional containing the first element from the stream if non-empty, else returns an empty Optional.
- <R> Stream<R> flatMap(Function<T,Stream<R>> f) returns a stream whose elements result from computing f.apply(x1), f.apply(x2), . . . to produce a sequence of streams, and flattening the resulting streams into one. The primitive-type specialized methods flatMapTo{Double,Int,Long} correspondingly produce streams of type {Double,Int,Long}Stream.
- void forEach(Consumer<T> cons) performs cons.accept(xi) on the elements xi of this stream.
- void forEachOrdered(Consumer<T> cons) performs cons.accept(xi) on the elements xi of this stream, in encounter order.
- static <T> Stream<T> generate(Supplier<T> supp) returns an infinite sequential unordered stream resulting from the call sequence supp.get(), supp.get(), . . . where supp is possibly stateful. Note that unlike for iterators (section 22.7) there is no way to indicate end of stream.
- boolean isParallel() returns true if this is a parallel stream, otherwise false.
- static <T> Stream<T> iterate(T x0, UnaryOperator<T> f) returns an infinite sequential ordered stream whose elements ri are r0 = x0, r1 = f.apply(r0), r2 = f.apply(r1), ...
- Iterator<T> iterator() returns an iterator for the elements of this stream.
- Stream<T> limit(long n) returns a stream consisting of at most the first n elements of this stream.

Example 171 Using Stream Methods to Find and Print Webpage Links
This example reads webpages from the net, scans the first 200 lines of each webpage for links, discards duplicate links and prints the unique ones, using streams and functional programming to cleanly separate these tasks. Method `getPage` from example 181 returns a Webpage object consisting of a URL and a string holding the first 200 lines of the page contents. Method `scanLinks` from example 169 scans a (partial) webpage for hyperlinks, and returns a stream of Links. The stream method `flatMap` calls `scanLinks` on many webpages to obtain many Link streams and then flattens all those into a single Link stream. The stream method `distinct` discards duplicates from the Link stream. The stream method `forEach` prints the links as they are produced.

```
String[] allUrls = { "http://www.itu.dk", ... };

Stream<String> urls = Stream.<String>of(allUrls);
Stream<Webpage> pages = urls.map(url -> getPage(url, 200));
Stream<Link> links = pages.flatMap(page -> scanLinks(page));
Stream<Link> uniqueLinks = links.distinct();
uniqueLinks.forEach(System.out::println);  // Calls Link.toString()
```

Example 172 Checking Sortedness of a Sequential Stream
To check whether a sequential ordered stream is sorted, one can use the stream method `allMatch` together with a stateful predicate as shown below. Each application of the predicate `x -> { ... }` compares a stream element `x` to its predecessor. The singleton array `last[0]` holds that predecessor; we cannot use a plain `int` variable for that purpose because variables captured in a Java lambda expression must be effectively final, that is, immutable (section 9.11). This method does not work on a parallel stream because the predicate is stateful.

```
static boolean isSorted2(IntStream xs) {
  final int[] last = { Integer.MIN_VALUE };
  return xs.allMatch(x -> { int old = last[0]; last[0] = x; return old <= x; } );
}
```

Example 173 Making a Stream of English Numerals
Using function `toEnglish` from example 158 one can create a (practically) infinite stream of the English numerals “zero”, “one”, “two”, . . . , “thirteen million nine hundred eighty nine thousand four hundred twenty two”, and so on. One can also generate a stream of “logorithms”, where the logorithm of a number `n` is the number of letters in its numeral (I believe this tongue-in-cheek concept is due to Martin Gardner).

```
Stream<String> numerals
  = LongStream.iterate(0, x -> x+1).mapToObj(Numerals::toEnglish);
IntStream logorithms = numerals.mapToInt(String::length);
System.out.println(logorithms.limit(1_000_000).max());
```

Example 174 Versatility of Streams
Streams are very versatile. For instance, if we can lazily generate a stream of solutions to the 8-queens problem (example 176), then we can later decide whether we want to print all solutions, the number of solutions, the first 20 solutions, or an arbitrary solution, as shown below. With a more imperative approach, we would typically have to decide beforehand how to use the results.

```
queens(8).forEach(System.out::println);
System.out.println(queens(8).count());
queens(8).limit(20).forEach(System.out::println);
System.out.println(queens(8).findAny());
```

Methods on interface Stream<T>, continued:

- `Stream<R> map(Function<T,R> f)` returns a stream with elements `f.apply(x1)`, `f.apply(x2)`, ... The primitive-type specialized methods `mapTo{Double,Int,Long}` correspondingly produce streams of type {Double,Int,Long}Stream.
- `Optional<T> max(Comparator<T> cmp)` returns the stream's maximal element according to `cmp`, or an absent Optional if there are no elements.
- `Optional<T> min(Comparator<T> cmp)` returns the stream's minimal element according to `cmp`, or an absent Optional if there are no elements.
- `boolean noneMatch(Predicate<T> p)` returns true if `p.test(xi)` is false for all elements, else false.
- `static <T> Stream<T> of(T... vs)` returns a sequential ordered stream whose elements are the `vs`.
- `static <T> Stream<T> of(T x)` returns a sequential Stream containing the single element `x`.
- `Stream<T> onClose(Runnable handler)` returns a stream with the same elements but an additional close handler. When closing the stream, the close handlers are executed in the order they were added.
- `Stream<T> parallel()` returns a parallel stream with the same elements as this stream.
- `Stream<T> peek(Consumer<T> cons)` returns a stream consisting of the same elements `x1`, `x2`, ... as this stream, additionally performing actions `cons.accept(x1)`, `cons.accept(x2)`, ... as the elements are being consumed from the resulting stream. Use it for debugging purposes only.
- `Optional<T> reduce(BinaryOperator<T> op)` computes the reduction of the stream's elements using the associative operator `op`. More precisely, writing `op.apply(x,y)` as infix `x⊗y`, return an Optional containing the value `x1⊗x2⊗...⊗xn`, computed in some order, if the stream is non-empty, otherwise return an empty Optional.
- `T reduce(T x0, BinaryOperator<T> op)` computes the reduction of `x0` and the stream's elements using the associative operator `op`. More precisely, writing `op.apply(x,y)` as infix `x⊗y`, return `x0⊗x1⊗x2⊗...⊗xn`, computed in some order.
- `U reduce(U r0, BiFunction<U,T,U> op, BinaryOperator<U> comb)` computes the reduction of the stream's elements, using the provided identity `r0`, accumulation function `op`, and combiner `comb`. More precisely, writing `op.apply(r,x)` as infix `r⊗x` and writing the combiner `comp.apply(r,s)` as infix `r⊕s`, return `(r0⊗x11⊗...⊗x1m) ⊕ ... ⊕ (r0⊗xk1⊗...⊗xkm)`, computed in some order, where the `xij` represents some partitioning of the stream's elements into segments. It must hold that `r0⊗x` equals `x`, that `r⊕(r0⊗x)` equals `r⊗x` for all `x` and `r`, and ⊕ must be associative. The primitive-type specialized {Double,Int,Long}Stream do not have this overload.
- `Stream<T> sequential()` returns a sequential stream with the same elements as this stream.
- `Stream<T> skip(long n)` returns a stream of the remaining elements after discarding the `n` first ones.
- `Stream<T> sorted()` returns a stream consisting of the elements of this stream, sorted in natural order.
- `Stream<T> sorted(Comparator<T> cmp)` returns a stream consisting of the elements, sorted by `cmp`.
- `Spliterator<T> spliterator()` returns a spliterator for the elements of this stream.
- `Object[] toArray()` returns an array containing the elements of this stream.
- `T[] toArray(IntFunction<T[]> alloc)` returns an array containing the elements of this stream, using `alloc.apply(n)` to allocate the returned array as well as any intermediate arrays.
- `Stream<T> unordered()` returns an unordered stream with the same elements as this stream.

Example 175 Generating a Stream of Permutations
The stream of all permutations of n numbers $0\ldots(n-1)$ can be generated by maintaining a partially generated permutation as an integer list `tail`, and a set `todo` of the numbers not yet used in the permutation. If `todo` is empty, `tail` is a permutation of all n numbers. Otherwise recursively generate those permutations that can be obtained by removing an element `r` from `todo` and putting it in front of `tail`. To create all n-permutations, start with an empty `tail`, and a `todo` set containing the numbers $0\ldots(n-1)$.

Class IntList represents immutable integer lists; see example 182. The `boxed()` operation turns an IntStream into a Stream<Integer> so one can apply `flatMap` to obtain a Stream<IntList>; the `flatMap` method on IntStream produces only IntStreams. The call `minus(todo, r)` returns a new BitSet with `r` removed.

```
public static Stream<IntList> perms(BitSet todo, IntList tail) {
  if (todo.isEmpty())
    return Stream.of(tail);
  else
    return todo.stream().boxed().flatMap(r -> perms(minus(todo, r), new IntList(r, tail)));
}
public static Stream<IntList> perms(int n) {
  BitSet todo = new BitSet(n); todo.flip(0, n); return perms(todo, null);
}
```

Example 176 Generating a Stream of Solutions to the n-Queens Problem
We now augment the permutation generator (example 175) to generate solutions to the n-queens problem: How to place n queens on an n-by-n chessboard so all queens are safe from each other. A 3-permutation such as $[1,0,2]$ can be considered a safe placement of 3 rooks on a 3-by-3 chessboard, in rows 1, 0 and 2 of columns 0, 1 and 2. So a solution to the n-queens problem is a permutation further constrained by considering diagonals: Filter away those `r` values from `todo` that, if put in front of `tail`, would constitute an unsafe queen's position that could attack some queen in the columns represented by `tail`.

This solution is simple, quite fast, versatile (example 174), and trivial to parallelize because all operations are purely functional. Putting `.parallel()` after `filter` gives a speed-up of 3.5x on a 4-core i7 processor.

```
public static Stream<IntList> queens(BitSet todo, IntList tail) {
  if (todo.isEmpty())
    return Stream.of(tail);
  else
    return todo.stream()
      .filter(r -> safe(r, tail)).boxed()            // could use .parallel() here
      .flatMap(r -> queens(minus(todo, r), new IntList(r, tail)));
}
public static boolean safe(int mid, IntList tail) { return safe(mid+1, mid-1, tail); }
public static boolean safe(int d1, int d2, IntList tail) {
  return tail==null || d1!=tail.item && d2!=tail.item && safe(d1+1, d2-1, tail.next);
}
```

Example 177 A Stream Can Be Consumed Only Once
A stream can be consumed only once, so one cannot find the standard deviation of a DoubleStream `ds` by separately computing its mean and the sum of its squares; instead compute both in one traversal as in example 180.

```
DoubleSummaryStatistics stats = ds.summaryStatistics();
// Fails with IllegalStateException: stream has already been operated upon or closed:
double sqsum = ds.map(x -> x*x).sum();
double sdev = Math.sqrt(sqsum/stats.getCount() - stats.getAverage()*stats.getAverage());
```

24.4 Numeric Streams: DoubleStream, IntStream and LongStream

Numeric streams may be represented by primitive-type specialized interfaces {Double,Int,Long}Stream for efficiency; see section 23.3. They have additional methods `average()`, `max()`, `min()`, `sum()` and `mapToObject`. The argument and result types of their Stream<T> methods are appropriately primitive-type specialized. For instance, the `iterator()` methods return PrimitiveIterator.Of{Double,Int,Long} objects (section 22.7), and the `generate` method's signatures in IntStream respectively Stream<T> look like this:

```
static IntStream generate(IntSupplier supp)
static Stream<T> generate(Supplier<T> supp)
```

In addition to the general stream methods (section 24.3), DoubleStream has these methods:

- `Stream<Double> boxed()` returns a stream of this stream's elements, each boxed as a Double object.
- `DoubleSummaryStatistics summaryStatistics()` returns statistics for this stream; see section 24.5.

In addition to the general stream methods (section 24.3), IntStream has these methods:

- `DoubleStream asDoubleStream()` returns a stream of this stream's elements converted to `double`.
- `LongStream asLongStream()` returns a stream of this stream's elements converted to `long`.
- `Stream<Integer> boxed()` returns a stream of this stream's elements, each boxed as an Integer object.
- `static IntStream range(int a, int b)` returns the `int` stream `[a..(b-1)]`, empty if `a>=b`.
- `static IntStream rangeClosed(int a, int b)` returns the `int` stream `[a..b]`, empty if `a>b`.
- `IntSummaryStatistics summaryStatistics()` returns statistics for this stream; see section 24.5.

In addition to the general stream methods (section 24.3), LongStream has these methods:

- `DoubleStream asDoubleStream()` returns a stream of this stream's elements converted to `double`.
- `Stream<Long> boxed()` returns a stream of this stream's elements, each boxed as a Long object.
- `static LongStream range(long a, long b)` returns the `long` stream `[a..(b-1)]`, empty if `a>=b`.
- `static LongStream rangeClosed(long a, long b)` returns `long` stream `[a..b]`, empty if `a>b`.
- `LongSummaryStatistics summaryStatistics()` returns statistics for this stream; see section 24.5.

24.5 Summary Statistics for Numeric Streams

The classes {Double,Int,Long}SummaryStatistics from package `java.util` represent summary statistics of a numeric stream. The classes have `get` methods that return count, min, max, sum and average (mean); see example 178. The min, max and sum have the same type as the stream elements, except that the sum of an IntStream is a `long`. The average is always a `double`.

Class DoubleSummaryStatistics implements the DoubleConsumer interface and in addition to the `get` methods shown in example 178 have these methods to collect the statistics:

- `void accept(double x)` records value `x` in the summary information.
- `void combine(DoubleSummaryStatistics other)` combines the `other` statistics into this one.

The IntSummaryStatistics and LongSummaryStatistics classes have corresponding methods. These methods can be passed to a stream's `collect` method to compute the statistics (example 179) and they can be overridden to collect more comprehensive statistics (example 180).

Example 178 Summary Statistics for Numeric Streams
The summary statistics for a `double` stream can be computed and printed like this, with the printed output inserted as a comment:

```
DoubleStream ds = DoubleStream.of(2, 4, 4, 4, 5, 5, 7, 9);
DoubleSummaryStatistics stats = ds.summaryStatistics();
System.out.printf("count=%d, min=%g, max=%g, sum=%g, mean=%g%n",
                  stats.getCount(), stats.getMin(), stats.getMax(),
                  stats.getSum(), stats.getAverage());
// count=8, min=2.00000, max=9.00000, sum=40.0000, mean=5.00000
```

Example 179 Computing Summary Statistics Using Collector Functions
The DoubleSummaryStatistics object `stats` computed in example 178 can equivalently be computed like this, using the collector components (section 24.6) of the DoubleSummaryStatistics class:

```
DoubleSummaryStatistics stats
  = ds.collect(DoubleSummaryStatistics::new,
               DoubleSummaryStatistics::accept,
               DoubleSummaryStatistics::combine);
```

Example 180 Extending Double Summary Statistics with Standard Deviation
By creating a subclass BetterDoubleStatistics of the DoubleSummaryStatistics class one can compute also the standard deviation of a stream of `double`s in a single traversal. Note that this cannot be done by multiple traversals (first compute the usual summary statistics, then compute the sum of squares of the stream) because a stream can be consumed only once; see example 177.

The BetterDoubleStatistics class computes the sum of squares in addition to whatever is done by superclass DoubleSummaryStatistics, and adds a method `getSdev` to compute the standard deviation afterwards:

```
class BetterDoubleStatistics extends DoubleSummaryStatistics {
  private double sqsum = 0.0;
  @Override
  public void accept(double d) {
    super.accept(d);
    sqsum += d * d;
  }
  public void combine(BetterDoubleStatistics other) {
    super.combine(other);
    sqsum += other.sqsum;
  }
  public double getSdev() {
    double mean = getAverage();
    return Math.sqrt(sqsum/getCount() - mean*mean);
  }
}
...
DoubleStream ds = DoubleStream.of(2, 4, 4, 4, 5, 5, 7, 9);
BetterDoubleStatistics stats
  = ds.collect(BetterDoubleStatistics::new,
               BetterDoubleStatistics::accept,
               BetterDoubleStatistics::combine);
// count=8, min=2.00000, max=9.00000, sum=40.0000, mean=5.00000, sdev=2.00000
```

24.6 Collectors on Streams

In some cases it is difficult to make the functional stream reduce operations efficient enough. This holds for instance for massive string concatenation (where repeated use of `s1+s2` has quadratic execution time), for creating a collection, list or set from a stream, and for various grouping and binning operations. In those cases, a *mutable reduction operation* using a so-called collector, may be more efficient. However, functional reductions should be preferred where possible, because the mutable reduction operations are easier to get wrong and often see much less speed-up (or even considerable slow-down) on parallel streams than the functional operations.

A collector `coltor` is an instance of interface Collector<T,A,R> that can process a stream `xs` of type Stream<T>, using an internal accumulator of type A, and producing a result of type R.

Stream method `xs.collect(coltor)` applies the collector to the stream `xs`, performing the mutable reduction operation and returning a result of type R. Utility class Collectors in package `java.util.stream` defines many useful collectors, listed below.

Stream method `xs.collect(Supplier<R> supp, BiConsumer<R,T> accu, BiConsumer<R,R> comb)` supports custom mutable reduction operations, producing a final result of type R. Function `supp()` generates a result container; function `accu(rc, x)` is called to process stream element `x` and add it to result container `rc`; and function `comb(rc1,rc2)` is called to combine the state of result container `rc2` into `rc1`; see examples 179, 180 and 186.

Some more advanced features of collectors are not described in this book; see the Java class library documentation.

Below we list the static methods in class Collectors that produce often used collectors. For readability we have simplified some of the wildcard types in the signatures, as described in section 23.4. We use the parameter names `coltor` for collector, `fin` for finisher, `rc` for result container, `comb` for combiner, `cons` for consumer, and `cfier` for classifier.

- `Collector<T,?,Double> averagingDouble(ToDoubleFunction<T> f)` computes the arithmetic mean or average of `f.apply(xi)`. There are similarly named methods for Int and Long.
- `Collector<T,A,RR> collectingAndThen(Collector<T,A,R> coltor, Function<R,RR> fin)` collects by `coltor` and then applies finisher `fin` to the result container.
- `Collector<T,?,Long> counting()` counts the number of elements.
- `Collector<T,?,Map<K,List<T>>> groupingBy(Function<T,K> cfier)` groups elements `xi` into lists by the value of key `cfier.apply(xi)`.
- `Collector<T,?,Map<K,D>> groupingBy(Function<T,K> cfier, Collector<? super T,A,D> coltor)` groups elements `xi` into lists by the value of key `cfier.apply(xi)`, then performs reduction operation `coltor` on the set of values `xi` associated with each key.

  There is also an overload with an additional argument of type `Supplier<Map<K,D>>` to produce the map used. There are concurrent versions of these methods also, named `groupingByConcurrent`; these produce a ConcurrentMap.

  Two `partitioningBy` methods work like the methods above but take a Predicate<T> instead of a Function<T,K> and produce a map whose only keys are `true` and `false`.

Example 181 Using a Collector to Join Lines into a Page

Method `getPageLines` from example 170 produces a lazy stream of the lines of a webpage. We can join the first `maxLines` lines into a single string using the stream methods `limit` and `collect`, where the latter is applied to the predefined `joining` collector that efficiently joins strings.

```
public static Webpage getPage(String url, int maxLines) {
  String contents =
    getPageLines(url).limit(maxLines).collect(Collectors.joining());
  return new Webpage(url, contents);
}
```

Example 182 Using a Collector and IntStream to Print an Integer List

Class IntList is used in examples 175 and 176 to represent immutable integer lists, which we would like to print in the format `[1, 3, 0, 2]` within square brackets and with comma-separated numbers. We could cleverly define `toString` using a StringBuilder, but a simpler and more general idea is to define a method `stream` to convert IntList to IntStream and then use a predefined collector to format the IntStream as a string.

```
class IntList {
  public final int item;
  public final IntList next;
  public IntList(int item, IntList next) { this.item = item; this.next = next; }
  public static IntStream stream(IntList xs) {
    IntStream.Builder sb = IntStream.builder();
    while (xs != null) {
      sb.accept(xs.item);
      xs = xs.next;
    }
    return sb.build();
  }
  public String toString() {
    return stream(this).mapToObj(String::valueOf).collect(Collectors.joining(",", "[", "]"));
  }
}
```

Example 183 Using Stream Functions to Generate a van der Corput Sequence

A van der Corput sequence is an infinite sequence $\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \ldots$ that is dense in the interval $[0,1]$ and evenly distributed over it. The infinite sequence is typically used in (financial) simulations. A 2-billion element approximation of the sequence may be generated lazily using stream functions: For every bit count `b` in range $1 \ldots 31$ and for every `i` in range $2^{b-1} \ldots 2^b - 1$, compute the bit-reversal of integer `i` and divide by 2^b.

Example 186 uses collectors to test that the generated numbers are evenly distributed over $[0,1]$; and example 24 uses array sort to show that they are dense in $[0,1]$.

```
public static DoubleStream vanDerCorput() {
  return IntStream.range(1, 31).asDoubleStream().flatMap(b -> bitReversedRange((int)b));
}

private static DoubleStream bitReversedRange(int b) {
  final long bp = Math.round(Math.pow(2, b));
  return LongStream.range(bp/2, bp).mapToDouble(i -> (double)(bitReverse((int)i) >>> (32-b)) / bp);
}
private static int bitReverse(int i) { ... /* reverse the bits in i */ ...}
```

Static methods on class Collectors that generate collectors, continued:

- `Collector<CharSequence,?,String> joining()` concatenates the input elements `xi` into a string.
- `Collector<CharSequence,?,String> joining(CharSequence delim)` concatenates the input elements `xi`, separated by `delim`, into a string.
- `Collector<CharSequence,?,String> joining(CharSequence delim, CharSequence pre, CharSequence suf)` concatenates the input elements `xi`, separated by `delim`, into a string starting with `pre` and ending with `suf`.
- `Collector<T,?,R> mapping(Function<T,U> f, Collector<? super U,A,R> coltor)` applies `f` to each element `xi` and then uses `coltor` to perform a reduction of the `f.apply(xi)` values.
- `Collector<T,?,Optional<T>> maxBy(Comparator<T> cmp)` produces an optional maximal element according to `cmp`, or absent if no elements.
- `Collector<T,?,Optional<T>> minBy(Comparator<T> cmp)` produces an optional minimal element according to `cmp`, or absent if no elements.
- `Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op)` performs a reduction of the elements using `op`. More precisely, writing `op.apply(x,y)` as infix `x⊗y`, return an Optional containing the value `x1⊗x2⊗...⊗xn` if the stream is non-empty, otherwise return an empty Optional.
- `Collector<T,?,T> reducing(T x0, BinaryOperator<T> op)` performs a reduction of `x0` and the elements using `op`. More precisely, writing `op.apply(x,y)` as infix `x⊗y`, return `x0⊗x1⊗x2⊗...⊗xn`.
- `Collector<T,?,U> reducing(U e, Function<T,U> f, BinaryOperator<U> op)` performs a reduction of transformed elements using `op`. More precisely, writing `op.apply(x,y)` as infix `x⊗y`, return an Optional containing the value `f.apply(x1) ⊗ f.apply(x2) ⊗ ... ⊗ f.apply(xn)` if the stream is non-empty, otherwise return an empty Optional.
- `Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<T> f)` applies function `f` to each element and returns summary statistics for the resulting values; see section 24.5. There are similarly named methods for Int and Long.
- `Collector<T,?,Double> summingDouble(ToDoubleFunction<T> f)` applies function `f` to each element and returns the sum of the values. There are similarly named methods for Int and Long.
- `Collector<T,?,C> toCollection(Supplier<C> collectionFactory)` accumulates the elements into a new collection of type `<C extends Collection<T>>` created by the `collectionFactory`.
- `Collector<T,?,List<T>> toList()` accumulates the elements into a new list.
- `Collector<T,?,Map<K,U>> toMap(Function<T,K> fk, Function<T,U> fv)` accumulates the elements into a new map whose key-value pairs are `(fk.apply(xi), fv.apply(xi))`; throws IllegalStateException unless `fk.apply(xi)` is distinct for all elements `xi`.
- `Collector<T,?,Map<K,U>> toMap(Function<T,K> fk, Function<T,U> fv, BinaryOperator<U> merge)` accumulates the elements into a map whose key-value pairs are `(fk.apply(xi), xival)` where `xiVal` is the result `fv.apply(xi1) ⊗ ... ⊗ fv.apply(xin)` of merging the `fv`-values of all `xij` for which `fk.apply(xij)` equals `fk.apply(xi)`, where `x ⊗ y` is `merge.apply(x,y)`.

  There is also an overload with an additional argument of type `Supplier<Map<K,U>>` to produce the map used. There are also `toConcurrentMap` versions of the two preceding methods that produce concurrent maps, more efficient on parallel streams.
- `Collector<T,?,Set<T>> toSet()` accumulates the elements into a new set.

Example 184 Generating a Stream of Lists of Prime Factors
An infinite stream of lists of prime factors can be generated by mapping as suitable method `factorList` on the infinite stream $[2,3,\ldots]$. This is used in example 185.

```
public static List<Integer> factorList(int p) { ... }
public static Stream<List<Integer>> allFactorLists() {
  return IntStream.iterate(2, x -> x+1).mapToObj(Streams::factorList);
}
```

The computed lists of prime factors for the 11 numbers $2\ldots12$ look like this:

```
[2] [3] [2, 2] [5] [2, 3] [7] [2, 2, 2] [3, 3] [2, 5] [11] [2, 2, 3]
```

Example 185 Collectors for Grouping and Counting
Using a collector one can group the lists of prime factors from example 184 by their length, that is, number `p` of prime factors. The grouping is represented by a map from `p` to the factor lists of length `p`; so group `1` contains the prime numbers, as shown below.

Moreover, instead of storing all factor lists, one can count them using another collector, to obtain a map from the prime factor count `p` to the number of numbers with that many prime factors; so `3=1273` below means that `1273` numbers between 2 and 5001 have `3` prime factors.

```
Map<Integer, List<List<Integer>>> factorGroups =
  allFactorLists().limit(11).collect(Collectors.groupingBy(List::size));
// {1=[[2], [3], [5], [7], [11]], 2=[[2, 2], [2, 3], [3, 3], [2, 5]], 3=[[2, 2, 2], [2, 2, 3]]}
Map<Integer, Long> factorGroupSizes =
  allFactorLists().limit(5000).collect(Collectors.groupingBy(List::size, Collectors.counting()));
// {1=669, 2=1366, 3=1273, 4=832, 5=452, 6=224, 7=104, 8=47, 9=22, 10=7, 11=3, 12=1}
```

Example 186 Grouping and Counting on a DoubleStream
Example 183 shows how to generate a van der Corput sequence. A simple test that the generated numbers are indeed evenly distributed over $[0,1]$ will put the numbers into 10 equally large bins and count them; there should be equally many numbers in each bin. Below we do this by calling the DoubleStream's three-argument `collect` method with a Supplier<`int[]`>, an ObjDoubleConsumer<`int[]`> and BiConsumer<`int[]`,`int[]`> that directly collect the bin counts in a 10-element integer array. Generating and binning the first 100 million van der Corput numbers using this method takes 0.9 seconds using a single CPU core.

Alternatively we could have used `.boxed()` to obtain a Stream<Double> and applied the one-argument `collect` method to a predefined collector, as in example 185. But the boxing of every `double` makes that approach slower by a factor of five.

```
final int bins = 10;
int[] binFrequenciesArray =
  vanDerCorput().limit(100_000_000).
  collect(() -> new int[bins],
          (a, x) -> { a[(int)(bins * x)]++; },
          (a1, a2) -> { for (int i=0; i<a1.length; i++) a1[i] += a2[i]; });
Arrays.stream(binFrequenciesArray).forEach(k -> System.out.printf("%d ", k));
// 10000002 10000001 10000000 10000000 9999997 10000002 10000001 10000000 10000000 9999997
```

25 Class Optional<T> (Java 8)

An instance of class Optional<T> from package `java.util` represents a value that is either *absent* (missing, empty), or is *present* and in that case contains a non-`null` value of type T. Class Optional<T> can be used to make it clearer that an operation may not return a result, instead of letting it silently return `null`. For instance, the `findAny` method on interface Stream<T> has type

```
Optional<T> findAny()
```

which makes it clear that sometimes `findAny` cannot produce a result, namely, when the stream is empty.

Class Optional<T> has these methods:

- `static <T> Optional<T> empty()` returns an absent (empty) Optional.
- `Optional<T> filter(Predicate<T> p)` returns a present Optional containing `v` if a value `v` is present and `p.test(v)` is true, otherwise returns an absent Optional.
- `Optional<U> flatMap(Function<T,Optional<U>> f)` returns `f.apply(v)` if a value `v` is present, otherwise returns an absent Optional.
- `T get()` returns the value if present, otherwise throws NoSuchElementException.
- `void ifPresent(Consumer<T> cons)` invokes `cons.accept(v)` on the value `v` if present, otherwise does nothing.
- `boolean isPresent()` returns true if there is a value present, otherwise false.
- `Optional<U> map(Function<T,U> f)` returns a present Optional containing `res` provided a value `v` is present in this Optional and `res = f.apply(v)` is non-`null`; otherwise returns an absent Optional.
- `static <T> Optional<T> of(T x)` returns a present Optional containing `x` if non-`null`, otherwise throws NullPointerException.
- `static <T> Optional<T> ofNullable(T x)` returns a present Optional containing `x` if `x` is non-`null`, otherwise returns an absent Optional.
- `T orElse(T other)` returns the value if present, otherwise `other`; just like `isPresent() ? get() : other`.
- `T orElseGet(Supplier<T> supp)` returns the value if present, otherwise the result of `supp.get()`.
- `T orElseThrow(Supplier<Throwable> exn)` returns the value if present, otherwise throws the exception created by `exn.get()`.

Note that the methods `empty`, `filter`, `flatMap`, `map` and `of` exist on the Stream<T> interface also, and indeed conceptually an Optional<T> can be thought of as a stream with zero or one element. The `ifPresent` method on an optional is the same as `forEach` on a stream.

There are primitive-type specialized classes OptionalDouble, OptionalInt and OptionalLong for representing results of type `double`, `int` or `long` that may be absent; this is particularly useful since a value of primitive types cannot itself be `null`. These classes have methods `empty`, `getAs{Double,Int,Long}`, `ifPresent`, `isPresent`, `of`, `orElse`, `orElseGet`, and `orElseThrow`, with correspondingly primitive-type specialized argument and result types.

Example 187 Replacing Multiple Kinds of Failure with Optional
Assume we need to (1) read a field `"area"` off a web form, (2) parse the field value as a `double`, (3) compute its square root, and then print either the result or an error message. This illustrates Java's three ways to indicate the absence of a result: (1) returning `null`, if the field is missing from the web form; (2) throwing an exception, if the string cannot be parsed as a `double`; and (3) returning a `NaN`, if taking the square root of a negative number. Code fragment (A) below gives the messy error handling code necessary in this case.

Code fragments (B) and (C) show two ways to do the same using class Optional and its methods. However, this assumes that suitable Option-returning versions of methods `parseDouble` and `sqrt` are available, but the Java class libraries currently have only few of these outside the streams framework.

```
// Alternative (A): Handling three kinds of error indication explicitly:
String areaString = form.get("area"), toPrint = "No value";
if (areaString != null) {
  try {
    double areaValue = Double.parseDouble(areaString);
    double result = Math.sqrt(areaValue);
    if (!Double.isNaN(result))
      toPrint = String.valueOf(result);
  } catch (NumberFormatException exn) { }
}
System.out.println(toPrint);
// Alternative (B): Using Optional, assuming suitable Option-returning methods exist:
Optional<String> areaString = Optional.<String>ofNullable(form.get("area"));
Optional<Double> areaValue = areaString.flatMap(s -> parseDouble(s));
Optional<Double> result = areaValue.flatMap(v -> sqrt(v));
System.out.println(result.map(String::valueOf).orElse("No value"));
// Alternative (C): As (B) but without naming the intermediate results:
String toPrint = Optional.<String>ofNullable(form.get("area"))
                         .flatMap(s -> parseDouble(s))
                         .flatMap(v -> sqrt(v))
                         .map(String::valueOf)
                         .orElse("No value");
System.out.println(toPrint);
```

Example 188 Optional Stream Element
One can use method `findAny` on the stream `queens(n)` of solutions to the *n*-queens problem (example 176), to obtain an arbitrary solution provided there is one. The result is an Optional<IntList>.

The first few lines of output are shown as comments. They show that the *n*-queens problem has no solution for *n* equal to 2 and 3 (so the Optional is empty), but does have a solution for *n* equal to 1, 4 and 5.

```
for (int n=1; n<=17; n++) {
  Optional<IntList> solution = queens(n).findAny();
  System.out.printf("%4d-queens solution: %s%n", n, solution);
}
//  1-queens solution: Optional[[0]]
//  2-queens solution: Optional.empty
//  3-queens solution: Optional.empty
//  4-queens solution: Optional[[1, 3, 0, 2]]
//  5-queens solution: Optional[[4, 1, 3, 0, 2]]
//  ...
```

29 What Is New in Java 8.0

Many new features have been added to the Java programming language in versions 7.0 and 8.0, notably:

- Support for functional programming through the concepts of functional interface (section 23), lambda expressions or anonymous functions (section 11.13) and method reference expressions (section 11.14).
- Further support for function-based data processing through the concept of lazy streams and pipelines (section 24), which also improve modularity and separation of concerns in a way reminiscent of lazy data structures in functional languages; see example 174.
- Support for data parallel programming in the form of functional parallel operations on arrays (section 8.4) and parallel stream processing (section 24). In many cases, a sequential data processing pipeline can be trivially turned into a parallel data processing pipeline, provided the operations are side effect free and stateless. In many cases, this improves throughput considerably by automatically taking advantages of available parallel processor cores.
- More expressive and useful interfaces through the addition of default and static methods on interfaces (section 13.3). In particular, this is use on the Comparator interface (section 22.10), the functional interfaces (section 23), and the stream interfaces (section 24).
- The ability to indicate that an operation may not return a result, and to represent such missing results, also for primitive types via the Optional class (section 25).
- The `switch` statement (section 12.4.3) now works for cases of type String.
- The actual type arguments in a generic object construction may be left out as in `new ArrayList<>()` if they can be inferred from context.
- Integer constants in binary `0b1010`, underscores permitted in number constants `1_000`.

Most of these features are found also in the C# programming language, as shown by this table:

		Java			C#	
Feature	Section	1.4	5.0	8.0	1.1	4.5
Looping over iterators	22.7	−	+	+	+	+
Enum types	14	−	+	+	+	+
Autoboxing simple values	5.4	−	+	+	+	+
Nullable value types/Optional	25	−	−	+	−	+
Try-with-resources	12.7	−	−	+	+	+
Generic types and methods	21	−	+	+	−	+
Run-time type parameter information		−	−	−	−	+
Generic interface co/contravariance		−	−	−	−	+
Wildcard types in generic instances	21.9	−	+	+	−	−
Annotations (metadata, attributes)	28	−	+	+	+	+
Anonymous functions, function types	11.13, 23	−	−	+	−	+
Defining streams or enumerables	24	−	−	+	−	+
Parallel array and stream processing	8.4, 24	−	−	+	−	+
Well-defined memory model	20.5	−	+	+	−	−

Example 216 New Features in Java 7.0 and 8.0
This example illustrates some of the new features. Method `getFunction` illustrates `switch` on strings, the Optional type, a function interface (DoubleUnaryOperator), lambda expressions, and method reference expressions. It may be called as `getFunction("log2").get().applyAsDouble(32.0)`. Method `diamondExample` shows how type arguments can be replaced by <> and inferred from context. Method `getPageAsString` uses the try-with-resources statement to make sure the BufferedReader gets closed eagerly, uses stream operations to read a limited number of lines from a webpage, and uses a collector to join the lines into a string. The useless method `getPageAsStream` illustrates the dangers of combining these features. The try-with-resources closes the BufferedReader eagerly before any part of the lazily generated stream has been consumed. Method `numberConstants` shows a variety of number constant notations; the `i1`–`i4` variables all have the same value.

```
static Optional<DoubleUnaryOperator> getFunction(String name) {
  switch (name) {
  case "ln": case "log": return Optional.of(Math::log);
  case "log2":           return Optional.of(x -> Math.log(x)/Math.log(2));
  case "log10":          return Optional.of(Math::log10);
  default:               return Optional.empty();
  }
}
static void diamondExample() {
  List<String> alist1 = new ArrayList<String>();    // Type argument <String> given
  List<String> alist2 = new ArrayList<>();          // Type argument <String> inferred
  // List<String> alist3 = new ArrayList();         // Raw type, unchecked conversion
  List<Function<String,List<Integer>>> flist = new ArrayList<>();
}
public static String getPageAsString(String url, int maxLines) throws IOException {
  try (BufferedReader in
       = new BufferedReader(new InputStreamReader(new URL(url).openStream()))) {
    Stream<String> lines = in.lines().limit(maxLines);
    return lines.collect(Collectors.joining());
  }
}
public static Stream<String> getPageAsStream(String url) throws IOException {
  try (BufferedReader in // USELESS -- BufferedReader gets closed before the stream is consumed
       = new BufferedReader(new InputStreamReader(new URL(url).openStream()))) {
    return in.lines();
  }
}
static void numberConstants() {
  int i1 = 0b1100_1010_1111_1110;           // Binary
  int i2 = 0xCAFE;                          // Hexadecimal
  int i3 = 0b1_100_101_011_111_110;         // Binary
  int i4 = 0145376;                         // Octal
  int i6 = -2_147_483_648;                  // Decimal
  int i7 = +2_147_483_647;                  // Decimal
  int i8 = 0x8000_0000;                     // Hexadecimal
  int i9 = 0x7FFF_FFFF;                     // Hexadecimal
  double debt = 18_210_520_570_642.0;       // Floating-point
  char c1 = 0b00110001;                     // Character '1'
  char c2 = 0b01000001;                     // Character 'A'
}
```

References

[1] The authoritative reference on the Java programming language is J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley: *Java Language Specification*, version 8, March 2014. Browse or download in HTML or PDF at https://docs.oracle.com/javase/specs/.

[2] The Java class library documentation can be browsed at https://docs.oracle.com/javase/8/docs/api/ or downloaded as a zip-file from http://www.oracle.com/technetwork/java/javase/downloads/

[3] An excellent guide to good programming in Java is Joshua Bloch: *Effective Java*, second edition (Addison-Wesley 2008).

[4] The best text on concurrent programming with Java is Goetz et al: *Java Concurrency in Practice*, Addison-Wesley 2006.

[5] The Java Tutorials from Oracle provide a wealth of general and specialized information on Java. See http://docs.oracle.com/javase/tutorial/

[6] The Unicode character encoding (http://www.unicode.org/) corresponds to part of the Universal Character Set (UCS), which is international standard ISO 10646:2014. The UTF-8 is a variable-length encoding of UCS, in which seven-bit ASCII characters are encoded as themselves.

[7] Floating-point arithmetic is described in the *IEEE Standard for Floating-Point Arithmetic* (IEEE Std 754-2008).

Index