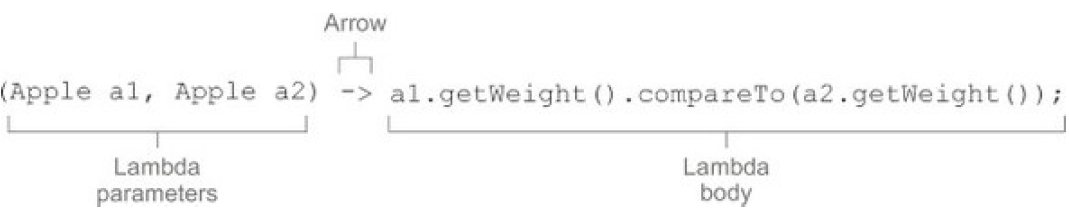
Java 8

**Lambda**

1. The basic syntax of a lambda are:

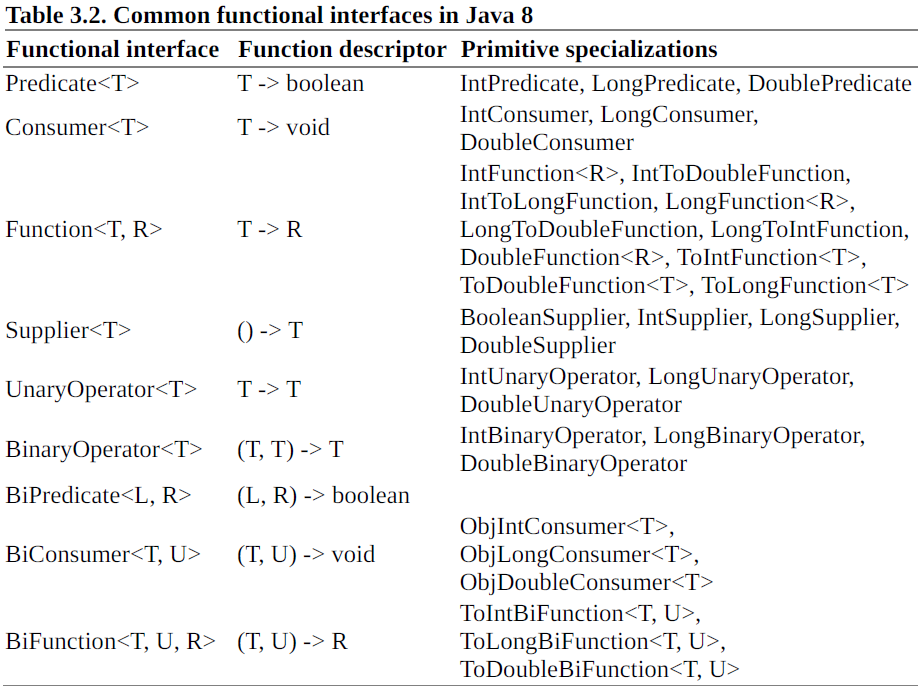
* (parameters) -> expression
* (parameters) -> {statements;} //(note the curly braces for statements)

1. Example:



* A list of parameters.
* An arrow.
* The body of the lambda.

1. Common functional interfaces



Note: none of the functional interfaces allow for a checked exception to be thrown. You have two

options if you need a lambda expression to throw an exception:

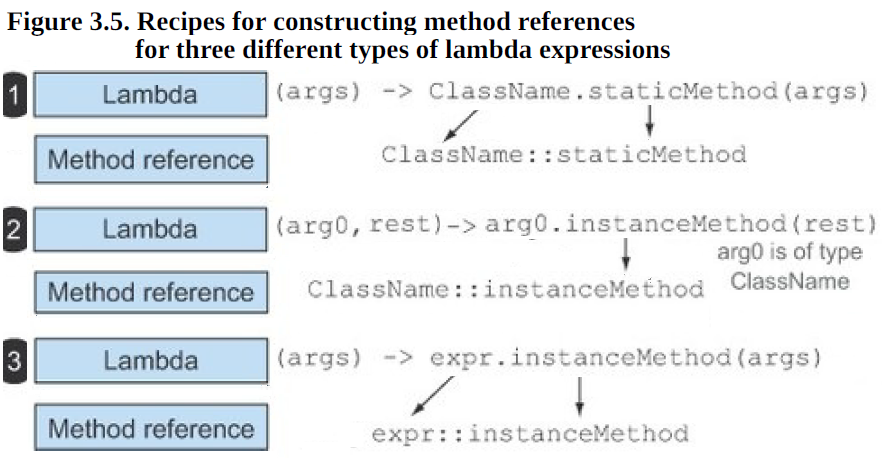
* define your own functional interface that declares the checked exception, or
* wrap the lambda with a try/catch block.

**Method references**

Method references let you reuse existing method definitions and pass them just like lambdas.

There are three main kinds of method references:

1. **static method** (for example, the method parseInt of Integer, written **Integer::parseInt**)
2. **instance method of an arbitrary type** (for example, the method length of a String, written **String::length**)
3. **instance method of an existing object** (for example, suppose you have a local variable expensiveTransaction that holds an object of type Transaction, which supports an instance method getValue; you can write **expensiveTransaction::getValue**)



Constructor references

**Stream**

1. Stream vs collection

* Collections are about data:

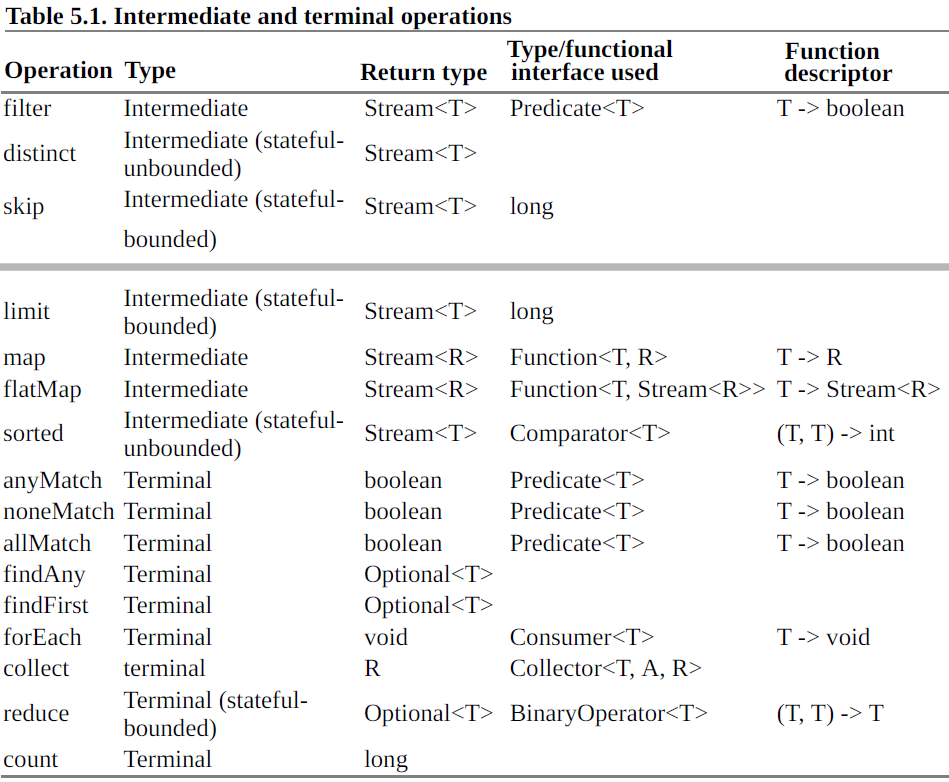
collections are data structures, they’re mostly about storing and accessing elements with specific time/space complexities (for example, an ArrayList vs. a LinkedList).

* Streams are about computations:

streams are about expressing computations such as filter, sorted, and map.

1. Streams consume from a data-providing source such as collections, arrays, or I/O resources.
2. Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on
3. Pipelining— Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline.
4. Internal iteration— In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you.
5. Allows you to exploit parallelism without worrying about locking, provided you embrace stateless behaviors (that is, functions in your stream-processing pipeline don’t interact by one reading from or writing to a variable that’s written by another)
6. you can’t define a stream recursively because a stream can be consumed only once.

5.4. Reducing



**Functional Programming**

the reason for the unexpected variable value problems just discussed is that shared mutable data structures are read and updated by more than one of the methods your

A **side effect** is an action that’s not totally enclosed within the function itself. Here are some examples:

* Modifying a data structure in place, including assigning to any field, apart from initialization inside a constructor (for example, setter methods)
* Throwing an exception
* Doing I/O operations such as writing to a file

**Pure, of no side effects or side-effect free**

A method, which modifies neither the state of its enclosing class nor the state of any other objects and returns its entire results using return, is called pure or side-effect free.

Another way to look at this idea of no side effects is to consider immutable objects. An immutable

object is an object that can’t change its state after it’s instantiated so it can’t be affected by the actions of a function. This means that once immutable objects are instantiated, they can never go into an

unexpected state. You can share them without having to copy them, and they’re thread-safe because

they can’t be modified.

**Declarative programming**

There are two ways of thinking about implementing a system by writing a program:

* how things are done: object-oriented programming, sometimes called imperative programming.
* what’s to be done: declarative programming. The fine detail of how this query is implemented is left to the library. We refer to this idea as internal iteration.

Why functional programming?

Functional programming exemplifies this idea of declarative programming (“just say what you want,

using expressions that don’t interact, and for which the system can choose the implementation”) and

side-effect-free computation explained previously. As we discussed, these two ideas can help you

implement and maintain systems more easily.

What’s functional programming?

functional programming is programming using functions.

What’s a function?

A function corresponds to a mathematical function:

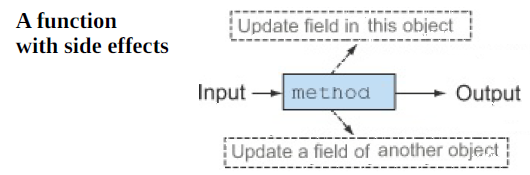
* takes zero or more arguments, gives one or more results, and has no side effects.
* when repeatedly called with the same arguments always return the same results.

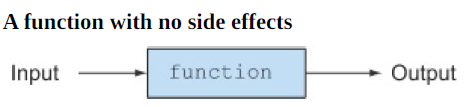
Pure functional programming:

allow a function to do nonfunctional things internally, as long as it doesn’t expose any of these side effects to the rest of the system. Or perform a side effect that can’t be observed by callers.

Functional-style programming:

A function or method can mutate only local variables. In addition, objects it references should be immutable. By this we mean all fields are final, and all fields of reference type refer transitively to other immutable objects.





A function or method shouldn’t throw any exceptions. Use exceptions locally but not expose them via large-scale interfaces.

Your function or method should call only those side-effecting library functions for which you can hide their nonfunctional behavior (that is, ensuring that any mutation they make on data structures is hidden from your caller, perhaps by copying first and by catching any exceptions they might raise

No visible side-effects:

* no mutating structure visible to callers,
* no I/O,
* no exceptions.

**Referential transparency**

A function is referentially transparent if it always returns the same result value when called with the same argument value. Put another way, a function consistently produces the same result given the same input, no matter where and when it’s invoked.

Referential transparency is a great property for program understanding. It also encompasses a save-instead-of-recompute optimization for expensive or long-lived operations, which goes under the name

memorization or caching.

**Currying**

Theoretical definition of currying:

Currying is a technique where a function f of two arguments (x and y, say) is seen instead as a function

g of one argument that returns a function also of one argument. The value returned by the latter function is the same as the value of the original function, that is, f(x,y) = (g(x))(y).

When some but fewer than the full complement of arguments have been passed, we often say the

function is partially applied.

**Persistent data structures**

The use of data structures used in functional-style programs. Also called as functional data structures and immutable data structures.

A functional-style method isn’t allowed to update any global data structure or any structure passed as a parameter.

Destructive updates vs. functional

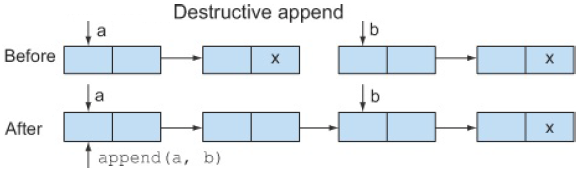
The functional-style approach to this problem is to ban such side-effecting methods. If you need a data

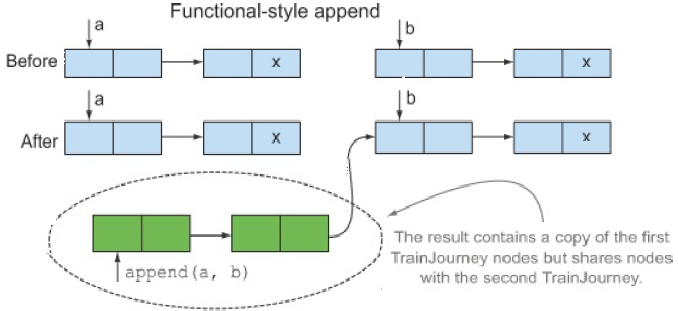
structure to represent the result of a computation, you should make a new one and not mutate an

existing data structure as done previously. This is often best practice in standard object-oriented

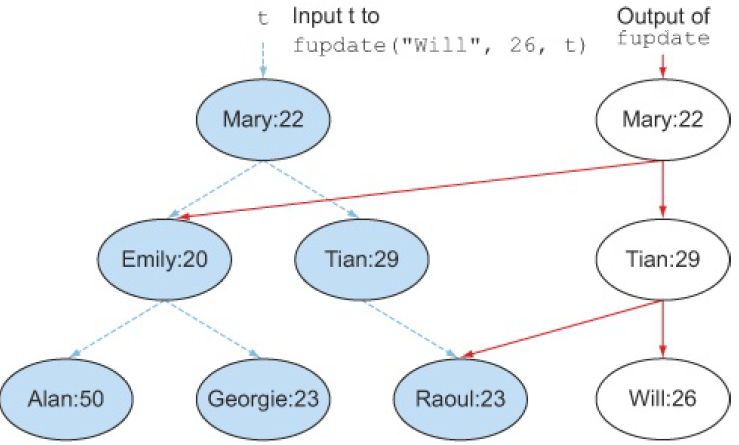
programming too.

Link example:





Tree Example:



Such functional data structures are often called persistent—their values persist and are isolated from

changes happening elsewhere—so as a programmer you’re sure fupdate won’t mutate the data

structures passed as its arguments. There’s just one proviso: the other side of the treaty is you require all users of persistent data structures to follow the do-not-mutate requirement.

**14.3. Lazy evaluation with streams**

you can’t define a stream recursively because a stream can be consumed only once.

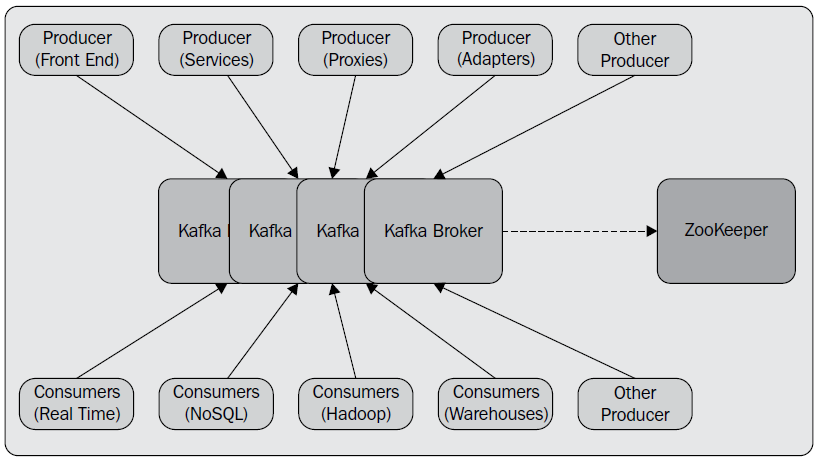
**Kafka**

Apache Kafka is an open source, distributed publish-subscribe messaging system, mainly designed with the following characteristics:

* **Persistent messaging**: To derive the real value from big data, any kind of information loss cannot be afforded. Apache Kafka is designed with **O(1)** disk structures that provide constant-time performance even with very large volumes of stored messages, which is in order of TB.
* **High throughput**: Keeping big data in mind, Kafka is designed to work on commodity hardware and to support millions of messages per second.
* **Distributed**: Apache Kafka explicitly supports messages partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.
* **Multiple client support**: Apache Kafka system supports easy integration of clients from different platforms such as Java, .NET, PHP, Ruby, and Python.
* **Real time**: Messages produced by the producer threads should be immediately visible to consumer threads; this feature is critical to event-based systems such as **Complex Event Processing** (**CEP**) systems.

Kafka provides a real-time publish-subscribe solution, which overcomes the challenges of real-time data usage for consumption, for data volumes that may grow in order of magnitude, larger that the real data. Kafka also supports parallel data loading in the Hadoop systems.

The following diagram shows a typical big data aggregation-and-analysis scenario supported by the Apache Kafka messaging system:



ZooKeeper: A **Distributed Coordination Service** for Distributed Applications.

Storm is a distributed, reliable, fault-tolerant system for processing streams of data.

<http://windowsreport.com/html5-editors-windows-10/>

**Microservices**

Microservices is a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services.

In a microservices architecture, services should be fine-grained and the protocols should be lightweight.

The benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop and test. It also parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.[1] It also allows the architecture of an individual service to emerge through continuous refactoring.[2] Microservices-based architectures enable continuous delivery and deployment