Java

Modificatori di classe

Parola chiave	Significato
abstract	Deve essere estesa da un'implementazione.
final	Non può essere estesa da un'implementazione.
strictfp	Il codice della classe usa semantica FP

Modificatori di metodo

Parola chiave	Significato
abstract	Deve essere implementato da una classe di estensione.
static	Legato alla classe e non ad una istanza.
final	Non può essere reimplementato da una classe di estensione.
native	Implementato da una libreria nativa.
strictfp	Il codice del metodo usa semantica FP restrittiva.
synchronized	Il metodo può essere usato da un solo thread per volta.

Visibilità

Modificatore	Classe	Package	Sottoclasse	Universo
Public	V	V	V	V
Protected	V	X	V	X
default	V	V	X	X
Private	V	X	X	X

Eccezioni

Tutte le eccezioni derivano dalla classe **Throwable**.

- **Exception**: gli errori nonostante i quali il programma dovrebbe essere in grado di proseguire.
- **Error**: gli errori dai quali il programma non è in grado di proseguire.
- **RuntimeException**: rappresenta ogni errore che può avvenire durante la normale valutazione di espressioni. Sono dette unchecked exceptions
- Tutte le altre sono dette checked exception e devono essere dichiarate nella definizione di un metodo.

Static nested classes

Una classe static non ha un accesso privilegiato ai membri (statici o meno) della classe ospite.

Inner classes

Ha lo stesso ciclo di vita, ed ha un riferimento privilegiato all'oggetto ospitante, non può dichiarare membri static ma solo membri di istanza.

Inizializzazione

- Inizializzatori statici
- Supercostruttore
- Inizializzatori di istanza

Interfaccia

Alle interfacce viene permesso di avere dei default method, cioè dei metodi implementati che si comportano in modo simile a quello delle superclassi.

Per mezzo dei default method una interfaccia può essere modificata con nuovi metodi senza che le implementazioni debbano essere toccate; se il metodo nuovo non è gestito dalla classe, viene usato quanto dichiarato nell'interfaccia.

Il Diamond Problem viene rilevato al momento della compilazione: se la gerarchia di ereditarietà ed implementazioni di una classe porta ad una ambiguità nella selezione di un metodo, il compilatore segnala un errore, che può essere risolto solo modificando il codice.

Annotazioni

Il loro scopo è arricchire di metadati la struttura a cui sono applicate, in modo da consentirne la rilevazione e l'uso durante la compilazione o l'esecuzione.

Le annotazioni possono avere parametri (solo di determinati tipi) e anche metodi (che possono ritornare lo stesso limitato insieme di tipi). Il compilatore può eseguire degli Annotation Processors che, durante la compilazione, possono produrre nuovi file (nuovi sorgenti, nuove classi, o qualsiasi altro tipo di file) a partire da annotazioni presenti nel codice.

Lambda expression

```
(<lista-parametri>) -> istruzione
```

Espressione Switch

Non c'è fall-through. L'elenco dei casi deve essere esaustivo.

Try with resources

Permette di dichiarare delle variabili. queste devono implementare l'interfaccia AutoClosable, e verranno automaticamente , e con certezza, "chiuse". In questa forma le clausole catch e finally sono entrambe opzionali.

Streams

A molto interfacce è stato aggiunto il metodo stream() che permette di trattare le collezioni con questa metafora.

Le operazioni sugli Stream vengono composte in sequenza, in una pipeline, fino ad arrivare ad una operazione detta terminale che produce il risultato.

Le operazioni intermedie sugli stream di dividono in stateful e stateless. Il loro uso influenza la costruzione e l'efficienza della pipeline che le contiene.

Il codice che implementa la pipeline ha ampie libertà su come riordinare e disporre l'esecuzione delle operazioni intermedie. Queste ultime devono:

- non interferire, cioè non modificare gli elementi dello stream
- (nella maggior parte dei casi) non avere uno stato interno

Programmazione concorrente

Thread

```
/**
* Allocates a new Thread object so that it has target as
* its run object, has the specified name as its name, and
* belongs to the thread group referred to by group, and
* has the specified stack size.
*
*/
public Thread(ThreadGroup group, Runnable target, String name, long stackSize);

/**
Il principale metodo è start(), che avvia un nuovo
percorso di esecuzione (similmente ad una fork) che
lavora all'interno della stessa JVM, condividendo lo
stesso heap e quindi lo stesso stato complessivo.

* Causes this thread to begin execution; the Java Virtual
* Machine calls the run method of this thread.
*/
void start();

/**
* Causes the currently executing thread to sleep
* (temporarily cease execution) for the specified number of
* milliseconds, subject to the precision and accuracy of
* system timers and schedulers
*
* @param millis the length of time to sleep in milliseconds
*
*/
public static void sleep(long millis)
throws InterruptedException

/**
* The Runnable interface should be implemented by any
* class whose instances are intended to be executed
* by a thread.
*/
@FunctionalInterface
public interface Runnable
{
    /**
    * The general contract of the method run is that
    * it may take any action whatsoever.
    */
    void run();
}

/**
* Returns this thread's name.
*/
```

```
public String getName();

/**
* Tests if this thread is alive.
*/
public boolean isAlive();

/**
* Returns a reference to the currently executing thread
* object.
*
*/
public static Thread currentThread();

/**
* Interrupts this thread.
*
*/
public void interrupt();

/**
* Set the handler invoked when this thread abruptly
* terminates due to an uncaught exception.
*/
public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh);
```

Executor

```
/**
* Executes the given command at some time in the future.
* The command may execute in a new thread, in a pooled
* thread, or in the calling thread, at the discretion
* of the Executor implementation.
*
* @param command the runnable task
*
*/
void execute(Runnable command);
```

Tipo	Funzionamento
CachedThreadPool	Riusa thread già creati, ne crea nuovi se necessario
FixedThreadPool	Riusa un insieme di thread di dimensione fissa
ScheduledThreadPool	Esegue i compiti con una temporizzazione
SingleThreadExecutor	Usa un solo thread per tutti

Callables

```
/**
* A task that returns a result and may throw an exception.
*/
@FunctionalInterface
```

```
public interface Callable < V >
{
    /**
    * Computes a result, or throws an exception if unable
    * to do so.
    *
    * @return computed result
    * @throws Exception - if unable to compute a result
    */
    V call() throws Exception;
}

/**
* An Executor that provides methods to manage termination
* and methods that can produce a Future for tracking
* progress of one or more asynchronous tasks.
*
*/
public interface ExecutorService extends Executor;

/**
* Submits a value-returning task for execution and
* returns a Future representing the pending results
* of the task.
*
* @param T - the type of the task's result
* @param task - the task to submit
* @return a Future representing pending completion
* of the task
*
*/
<T> Future<T> submit(Callable<T> task);

/**
* A Future represents the result of an asynchronous
* computation. Methods are provided to check if the
* computation is complete, to wait for its completion,
* and to retrieve the result of the computation.
*
*/
public interface Future< V >;

/**
* Waits if necessary for the computation to complete,
* and then retrieves its result.
*
*/
T get();

/**
* Returns true if this task completed.
*
*/
boolean isDone();
```

ExecutorService

```
/**
* Executes the given tasks, returning the
* result of one that has completed successfully
* (i.e. without throwing an exception), if any do.
*
*/
< T > T invokeAny(Collection < ? extends Callable< T > > tasks);

/**
* Executes the given tasks, returning a list of Futures
* holding their status and results when all complete.
* Future.isDone() is true for each element of the
* returned list.
*/
< T > List< Future< T > > invokeAll(Collection< ? extends Callable< T > > tasks);

/**
* Initiates an orderly shutdown in which previously
* submitted tasks are executed, but no new tasks will
* be accepted.
*
*/
void shutdown();

/**
* Blocks until all tasks have completed execution after a
* shutdown request, or the timeout occurs, or the current
* thread is interrupted, whichever happens first.
*
*/
boolean awaitTermination(long timeout, TimeUnit unit);

/**
* Returns true if all tasks have completed following
* shut down.
*
*/
boolean isTerminated();

/**
* Attempts to stop all actively executing tasks, halts
* the processing of waiting tasks, and returns a list
* of the tasks that were awaiting execution.
*
*/
List< Runnable > shutdownNow()
```

Dati thread-safe

Una struttura dati non thread-safe non consente a più thread di operare contemporaneamente.

- **nel migliore dei casi** lancia una java.util.ConcurrentModificationException
- **nel caso intermedio** lo stato diventa inconsistente
- **nel peggiore dei casi** ottengo un'altra eccezione

Atomic variables

La funzionalità richiede la disponibilità del supporto dell'hardware attraverso istruzioni CAS (Compare-and-swap).

In mancanza di queste l'implementazione ripiega su metodi più convenzionali (meno efficienti, che bloccano il thread).

Tipo	Array
Integer	AtomicInteger
Long	AtomicLong
Object	AtomicReference
Integer[]	AtomicIntegerArray
Long[]	AtomicLongArray
Object[]	AtomicReferenceArray

Concurrent data structures

ConcurrentMap

```
/**
* A Map providing thread safety and atomicity
* guarantees.
*/
public interface ConcurrentMap< K,V > extends Map< K,V >;

/**
* If the specified key is not already associated with a
* value, associate it with the given value.
* The action is performed atomically.
*/
V putIfAbsent(K key, V Value);

/**
* Replaces the entry for a key only if currently mapped
* to some value.
* The action is performed atomically.
*/
V replace(K key, V Value);

/**
* Replaces the entry for a key only if currently mapped
* to a given value.
* The action is performed atomically.
*/
V replace(K key, V oldValue, V newValue);

/**
* Returns the result of accumulating the given
* transformation of all (key, value) pairs using the
* given reducer to combine values, or null if none.
*
* @param the elements needed to switch to parallel
```

```
* @param the transformation for an element
* @param a commutative associative combining function
*/
public < U > U reduce(long parallelismThreshold, BiFunction< ? super K,? super
V,? extends U> transformer, BiFunction< ? super U,? super U,? extends U>
reducer);

/**
* Returns a non-null result from applying the given
* search function on each (key, value), or null if none.
* Upon success, further element processing is suppressed.
*
* @param the elements needed to switch to parallel
* @param a search function, that returns non-null on
*success
*/
public < U > U search(long parallelismThreshold, BiFunction< ? super K,? super
V,? extends U> searchFunction);

/**
* Performs the given action for each (key, value).
*
* @param the elements needed to switch to parallel
* @param the action (can have side-effects)
*/
public void forEach(long parallelismThreshold, BiConsumer< ? super K,? super V>
action);
```

BlockingQueue

Metodo	Risultato negativo
add(e)	eccezione
offer(e)	false
put(e)	attesa
offer(e, time, unit)	attesa limitata
remove()	eccezione
poll()	null
take()	attesa
poll(time, unit)	attesa limitata
element()	eccezione
peek()	null

```
/**
* Removes all available elements from this
* queue and adds them to the given collection.
* This operation may be more efficient than
* repeatedly polling this queue.
*
* @param c the collection to transfer elements into
* @return the number of elements transferred
*/
int drainTo(Collection< ? super E > c);
```

Varianti:

- **TransferQueue**: interfaccia per una coda in cui i produttori aspettano i consumatori
- **BlockingDeque**: interfaccia che permette di prendere un elemento dalla coda o dalla testa
- **ArrayBlockingQueue**: implementazione basata su array, con possibilità di fairness
- **LinkedBlockingDeque**, **LinkedBlockingQueue**, **LinkedTransferQueue**: implementazioni basate su liste collegate
- **PriorityBlockingQueue**: coda ordinata per priorità
- **DelayQueue**: un elemento non può essere preso prima di un ritardo impostato
- **SynchronousQueue**: ogni produttore deve attendere un consumatore (capacità nulla)

Thread local

```
/**
* These variables differ from their normal counterparts
* in that each thread that accesses one (via its get
* or set method) has its own, independently initialized
* copy of the variable.
*/
public class ThreadLocal< T >;

/**
* Returns the value in the current thread's copy of this
* thread-local variable. If the variable has no value
* for the current thread, it is first initialized to the
* value returned by an invocation of the initialValue()
* method.
*/
public T get();

/**
* Removes the current thread's value for this thread-local
* variable.
*/
public void remove();

/**
* Sets the current thread's copy of this thread-local
* variable to the specified value.
*/
public void set(T value);

/**
* Returns the current thread's "initial value" for
* this thread-local variable.
```

```
*
*/
protected T initialValue();
```

Synchronized

Tutti i blocchi sincronizzati di un oggetto condividono lo stesso monitor lock o intrinsic lock. Può decorare due tipi di raggruppamenti di istruzioni:

- un blocco di istruzioni semplice { ... }
- un metodo
- un oggetto

Wait

Ci costringe a gestire lo stato del blocco. Un Thread può farlo:

- eseguendo un metodo synchronized dell'oggetto
- eseguendo un blocco synchronized all'interno dell'oggetto
- se l'oggetto è una Class, eseguendone un metodo synchronized static

```
/**
* Causes the current thread to wait until another
* thread invokes the notify() method or the notifyAll()
* method for this object.
*/
void wait() throws InterruptedException;

/**
* Wakes up a single thread that is waiting on this
* object's monitor.
*/
void notify();

/**
* Wakes up all threads that are waiting on this
* object's monitor.
*/
void notifyAll();
```

Lock

```
/**
* Lock implementations provide more extensive locking
* operations than can be obtained using synchronized
* methods and statements.
*
*/
public interface Lock;

/**
* Acquires the lock.
*
*/
```

```
void lock();

/**
* Questa chiamata ha l'effetto di sbloccare un thread (tipicamente, uno a caso)
fra quelli
* che attendevano di acquisire il lock.
* Releases the lock.
*
*/
void unlock();

/**
* Acquires the lock only if it is free at the time of invocation.
*
*/
boolean tryLock();
```

ReentrantLock

```
/**
* Creates an instance of ReentrantLock with the given
* fairness policy.
*
*/
public ReentrantLock(boolean fair);
```

Conditions

Una Condition permette di separare l'accodamento in attesa dal possesso del lock che controlla l'attesa.
Lo scopo è da poter gestire, su di un solo lock, di più condizioni di attesa distinte.

```
/**
* una Condition ci viene fornita dal lock su cui deve sussistere. Ciascuna
Condition di uno
* stesso lock consente di gestire un insieme distinto di Thread in attesa.
* Returns a new Condition instance that is bound to this
* Lock instance.
*
*/
public Condition newCondition();

/**
* Causes the current thread to wait until it is signalled
* or interrupted.
*
*/
public void await();

/**
* Wakes up one waiting thread.
*
*/
public void signal();
```

```
/**
* Wakes up all waiting threads.
*
*/
public void signalAll();
```

Semafori

Il valore iniziale del semaforo non è un limite: può essere superato, e può essere anche negativo inizialmente. Mantenere la coerenza semantica sta all'utilizzatore.

La maggior parte dei metodi di Semaphore:

- può lanciare InterruptedException se il thread viene interrotto durante l'attesa
- lancia IllegalArgumentException se il parametro è negativo

```
/**
* Creates a Semaphore with the given number of permits and
* the given fairness setting.
*
*/
public Semaphore(int permits, boolean fair);

/**
* Acquires a permit from this semaphore, blocking until one
* is available, or the thread is interrupted.
*
*/
public void acquire();

/**
* Acquires the given number of permits from this semaphore,
* blocking until all are available, or the thread is
* interrupted.
*
* @param the number of permits to acquire
*/
public void acquire(int permits);

/**
* Releases a permit, returning it to the semaphore.
*
*/
public void release();

/**
* Releases the given number of permits, returning them to
* the semaphore.
*
* @param the number of permits to release
*/
public void release(int permits);

/**
* Shrinks the number of available permits by the
* indicated reduction.
```

```
*
*/
protected void reducePermits(int reduction);

/**
* Acquires a permit from this semaphore, only if one is
* available at the time of invocation.
*
*/
public boolean tryAcquire();

/**
* Acquires the given number of permits from this semaphore,
* if all become available within the given waiting time and
* the current thread has not been interrupted.
* tryAcquire ritorna immediatamente, con risultato falso se non ha ottenuto un
permesso.
* E' in grado di violare la fairness del semaforo.
*/
public boolean tryAcquire(int permits, long timeout, TimeUnit unit);
```

Programmazione distribuita

Sockets

```
package java.net;
/**
* This class implements client sockets (also
* called just "sockets"). A socket is an
* endpoint for communication between two machines.
**/
public class Socket implements Closeable
{
    /**
    * Creates a socket and connects it to the specified
    * remote address on the specified remote port.
    * The Socket will also bind() to the local address
    * and port supplied.
    *
    * @param address the remote address
    * @param port the remote port
    * @param localAddr the local address or null for anyLocal
    * @param localPort the local port, o zero for arbitrary
    **/
    public Socket(InetAddress address, int port,
    InetAddress localAddr, int localPort)
    throws IOException;

    /**
    * Returns an input stream for this socket.
    *
    **/
    public InputStream getInputStream()
    throws IOException;

    /**
    * Returns an output stream for this socket.
    *
    **/
    public OutputStream getOutputStream()
    throws IOException;

    /**
    * Closes this socket. Any thread currently blocked in
    * accept() will throw a SocketException.
    **/
    public void close() throws IOException;
}
```

ServerSockets

```
package java.net;
```

```
/**
* A server socket waits for requests to come
* in over the network.
**/
public class ServerSocket implements Closeable
{
    /**
    * Create a server with the specified port, listen backlog,
    * and local IP address to bind to.
    *
    * @param port the local port, o zero for arbitrary
    * @param backlog maximum length of the queue of incoming
    * connections
    * @param bindAddr the local InetAddress the server
    * will bind to
    **/
    public ServerSocket(int port, int backlog,
    InetAddress bindAddr)
    throws IOException;

    /**
    * Listens for a connection to be made to this socket and
    * accepts it. The method blocks until a connection is made.
    *
    **/
    public Socket accept() throws IOException;

     /**
    * Returns an input stream for this socket.
    *
    **/
    public InputStream getInputStream()
    throws IOException;

    /**
    * Returns an output stream for this socket.
    *
    **/
    public OutputStream getOutputStream()
    throws IOException;
}
```

Datagrams

```
package java.net;
/**
* Datagram packets are used to implement a connectionless
* packet delivery service.
**/
public final class DatagramPacket
{
    /**
    * Constructs a DatagramPacket for receiving packets.
    *
    * @param buf buffer for holding the incoming datagram
    * @param length the number of bytes to read.
    **/
```

```
    public DatagramPacket(byte[] buf, int length);

    /**
    * Constructs a DatagramPacket for receiving packets.
    *
    * @param buf the packet data
    * @param length the packet length
    * @param address the destination address
    * @param the destination port number
    **/
    public DatagramPacket(byte[] buf, int length,
    InetAddress address, int port);
}
```

MTU

Protocollo	MTU (bytes)
IPv4 (link)	68
IPv4 (host)	576
IPv4 (Ethernet)	1500
IPv6	1280
802.11	2304

DatagramSocket

```
package java.net;

public class DatagramSocket
{
    /**
    * Constructs a datagram socket and binds it to the
    * specified port on the local host machine.
    *
    * @param port the port to use
    **/
    public DatagramSocket(int port) throws SocketException;

    /**
    * Connects the socket to a remote address for this
    * socket. When a socket is connected to a remote address,
    * packets may only be sent to or received from that
    * address. By default a datagram socket is not connected.
    *
    * @param address the remote address for the socket
    * @param port the remote port for the socket.
    **/
    public void connect(InetAddress address, int port);

    /**
    * Sends a datagram packet from this socket. The
    * DatagramPacket includes information indicating the data
```

```
    * to be sent, its length, the IP address of the remote host,
    * and the port number on the remote host.
    *
    * @param p the DatagramPacket to be sent
    **/
    public void send(DatagramPacket p) throws IOException;

    /**
    * blocca fino alla ricezione del messaggio. Se il messaggio è più lungo del
buffer, viene troncato.
    *
    * Receives a datagram packet from this socket. When this
    * method returns, the DatagramPacket's buffer is filled
    * with the data received. The datagram packet also
    * contains the sender's IP address, and the port number
    * on the sender's machine.
    *
    * @param p the DatagramPacket to be sent
    **/
    public void receive(DatagramPacket p) throws IOException;

    /**
    * Closes this datagram socket.
    *
    * Any thread currently blocked in receive() upon this
    * socket will throw a SocketException.
    *
    * @param p the DatagramPacket to be sent
    **/
    public void close();
}
```

URL

```
package java.net;
/**
* Class URL represents a Uniform Resource Locator, a
* pointer to a "resource" on the World Wide Web.
**/
public final class URL
{
    /**
    * Creates a URL object from the String representation.
    **/
    public URL(String spec) throws MalformedURLException;

    /**
    * GET
    * Opens a connection to this URL and returns an
    * InputStream for reading from that connection.
    **/
    public InputStream openStream() throws IOException;

    /**
    * POST
    * Returns a URLConnection instance that represents a
    * connection to the remote object referred to by the URL.
```

```
    **/
    public URLConnection openConnection() throws IOException;
}
```

URLConnection

```
package java.net;
/**
* The abstract class URLConnection is the superclass of
* all classes that represent a communications link
* between the application and a URL.
**/
public abstract class URLConnection
{
    /**
    * Sets the value of the doOutput field for this
    * URLConnection to the specified value.
    **/
    public void setDoOutput(boolean dooutput);

    /**
    * Returns an output stream that writes to this connection.
    **/
    public OutputStream getOutputStream() throws IOException;


}
```

Channel

```
/**
* A nexus for I/O operations.
*
**/
public interface Channel extends Closeable
{

}
```

NetworkChannel

Un NetworkChannel rappresenta una comunicazione su di una rete. Può:

- esssere legato (con l'operazione bind) ad un'indirizzo
- dichiarare le opzioni che supporta.

```
/**
* A channel to a network socket.
*
**/
public interface NetworkChannel extends Closeable
{

}
```

AsynchronousServerSocketChannel

Un AsynchronousServerSocketChannel è un canale asincrono basato su di una server socket. Ci permette, in modo asincrono, di accettare connessioni e gestirle.

```
/**
* An asynchronous channel for stream-oriented
* listening sockets.
*
**/
public abstract class AsynchronousServerSocketChannel
implements AsynchronousChannel, NetworkChannel
{
    /**
    * (from AsynchronousServerSocketChannel)
    * Accepts a connection.
    *
    * @param A The type of the attachment
    * @param attachment The object to attach to the I/O
    * operation; can be null
    * @param handler The handler for consuming the result
    **/
    public abstract < A > void accept(A attachment,
    CompletionHandler< AsynchronousSocketChannel,? super A >
    handler);

    /**
    * (from AsynchronousSocketChannel)
    * Reads a sequence of bytes from this channel into the given
    * buffer.
    *
    * @param A The type of the attachment
    * @param dst The buffer into which bytes are to be
    * transferred
    * @param attachment The object to attach to the I/O op.
    * @param handler The completion handler
    **/
    public final < A > void read(ByteBuffer dst, A attachment,
    CompletionHandler< Integer,? super A > handler);

    /**
    * (from AsynchronousSocketChannel)
    * Writes a sequence of bytes to this channel from the given
    * buffer.
    *
    * @param A The type of the attachment
    * @param src The buffer from which bytes are to be
    * retrieved
    * @param attachment The object to attach to the I/O op.
    * @param handler The completion handler object
    **/
    public final < A > void write(ByteBuffer src, A attachment,
    CompletionHandler< Integer,? super A > handler);

    /**
    * (from AsynchronousServerSocketChannel)
    * Accepts a connection.
```

```
     *
     * @return a Future object representing the pending result
     **/
     public abstract Future< AsynchronousSocketChannel > accept();

     /**
     * (from AsynchronousSocketChannel)
     * Reads a sequence of bytes from this channel into the given
     * buffer.
     *
     * @param dst The buffer into which bytes are to be
     * transferred
     * @return A Future representing the result of the operation
     **/
     public abstract Future< Integer > read(ByteBuffer dst);

     /**
     * (from AsynchronousSocketChannel)
     * Writes a sequence of bytes to this channel from the given
     * buffer.
     *
     * @param src The buffer from which bytes are to be
     * retrieved
     * @return A Future representing the result of the operation
     **/
     public abstract Future< Integer > write(ByteBuffer src);
}
```

CompletionHandler

CompletionHandler è l'interfaccia che deve implementare un oggetto che gestisce la ricezione di un'operazione di I/O asincrona. Essendo la modalità asincrona, questa interfaccia ci permette di indicare al sistema l'azione da effettuare al completamento della successiva interazione.

Implementando un CompletionHandler possiamo esprimere il comportamento del server alla prossima interazione, in maniera asincrona.

Il parametro generico attachment ci permette di far circolare le informazioni di contesto riguardo allo stato della conversazione.

Vari handler potrebbero essere chiamati da Thread diversi, in momenti imprevedibili; da qui la necessità di gestire esplicitamente il passaggio del contesto.

La gestione delle operazioni di I/O richiede quindi di pecificare sempre l'attachment da far circolare ed il CompletitionHandler che gestisce il completamento.

Un'alternativa all'uso di un CompletionHandler è data dalla versione dei metodi che ritorna un Future.La principale differenza è che in questo caso, se il blocco di codice è unico per tutta la conversazione, il thread che la gestisce è unico e rimane allocato per l'intera durata della conversazione.

```
/**
* A handler for consuming the result of an asynchronous
* I/O operation.
*
* @param V The result type of the I/O operation
* @param A The type of the object attached to the
```

```
* I/O operation
**/
interface CompletionHandler< V,A >
{
    /**
    * Il compito del metodo completed è gestire l'interazione
    * relativa ai dati ricevuti, ed eventualmente
    * predisporre l'operazione successiva.
    *
    * Invoked when an operation has completed.
    *
    * @param result The result of the I/O operation
    * @param attachment The type of the object attached to
    * the I/O operation when it was initiated
    **/
    void completed(V result, A attachment);

    /**
    * Il compito del metodo failed è, ovviamente, gestire
    * il caso in cui un'interazione ha incontrato una eccezione.
    *
    * Invoked when an operation fails.
    *
    * @param exc The exception to indicate why the I/O
    * operation failed
    * @param attachment The type of the object attached to
    * the I/O operation when it was initiated
    **/
    void failed(Throwable exc, A attachment);
}
```

Reattività

È una implementazione dell'Observer pattern done right. Un Observable in Rx è un oggetto concettualmente simile ad uno stream, che emette nel tempo una sequenza di valori.

E' possibile osservare i valori emessi da un Observable fornendo il comportamento da adottare in caso di:

- valore ricevuto
- eccezione lanciata da un precedente componente
- termine del flusso di dati

Evento	Iterable	Observable
successivo	T next()	onNext(T)
errore	lancia Exception	onError(E)
completamento	!hasNext()	onCompleted()

L'Observable ribalta il funzionamento dell'Iterable, secondo il paradigma dello Stream. Aggiunge però, rispetto alla libreria standard, la gestione esplicita di errori e completamento dello stream.

Il risultato è:

- una semantica più ricca
- maggiore regolarità nella composizione
- indipendenza dal modello di esecuzione (sincrono/asincrono)

La maggior parte degli operatori sugli Observable accettano uno Scheduler come parametro. Ogni operatore può così essere reso concorrente; lo Scheduler scelto permette di indicare il tipo di concorrenza desiderato.

Un Subscriber rappresenta un ascoltatore di un Observable: fornisce il codice che reagisce agli eventi per ottenere il risultato finale dalla catena di elaborazione.

Un Subject può consumare uno o più Observable, per poi comportarsi esso stesso da Observable e quindi introdurre modifiche sostanziali nel flusso degli eventi.

Reactive streams

```
public interface Publisher< T >
{
    public void subscribe(Subscriber< ? super T > s);
}

public interface Subscriber< T >
{
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

```
public interface Subscription
{
    public void request(long n);
    public void cancel();
}

public interface Processor< T, R >
{
    extends Subscriber< T >, Publisher< R > {
}
```

Operatori

Map

Trasforma gli elementi di uno stream, ottenendo uno stream di elementi trasformati.

Flatmap

Trasforma gli elementi di uno stream, concatenando i risultati in un solo stream.

Filter

Emette uno stream contenente solo gli elementi che soddisfano un predicato.

Skip

Emette uno stream saltando i primi N elementi della sorgente.

Zip

Emette uno stream combinando a coppie elementi di due stream in ingresso.

Debounce

Emette un elemento solo se è passato un lasso di tempo dall'ultimo elemento della sorgente.

Window

Emette uno stream di partizioni dello stream sorgente.

Parallelismo

L'asincronia nell'esecuzione dei vari operatori è definita dallo Schedulatore usato per osservare un Observable o definire un operatore di uno stream.

Metodo	Schedulatore
.io()	Per stream legati alle operazioni d IO
.single()	Usa un singolo thread
.computation()	Per operatori legati al calcolo
.from(ex)	Usa l'Executor fornito

L'operatore parallel() permette di indicare che uno stream, da un certo punto in poi. In questa modalità, solo alcuni operatori sono consentiti ed è necessario specificare lo schedulatore da usare con il metodo .runOn(scheduler).

Il metodo sequential() indica che da quel punto in poi la pipeline di elaborazione va nuovamente intesa come sequenziale.

A differenza degli Stream della libreria di base, è possibile indicare una precisa sezione della pipeline che viene configurata parallelamente.

Streams

Flags

La sorgente di uno Stream può dichiarare alcune caratteristiche che gli operatori intermedi possono verificare e che l'operazione terminale usa per prendere decisioni sulla esecuzione.

L'operazione terminale ha piena visibilità, prima di cominciare a lavorare, di quali sono le caratteristiche della pipeline di esecuzione, e può quindi prendere decisioni in merito.

Flag	Caratteristica
CONCURRENT	Parallelizzabile
DISTINCT	Elementi distinti (equals())
IMMUTABLE	Immutabile durante il consumo
NONNULL	Elementi non nulli
ORDERED	Elementi ordinati (Comparator)
SIZED	Dimensione nota
SORTED	Ordinamento definito
SUBSIZED	Suddivisioni di dimensione nota

Operazioni

```
/**
* Returns an equivalent stream that is parallel.
* May return itself, either because the stream
* was already parallel, or because the underlying
* stream state was modified to be parallel.
*/
S parallel();

/**
* Returns an equivalent stream that is sequential.
* May return itself, either because the stream
* was already sequential, or because the underlying
* stream state was modified to be sequential.
*/
S sequential();

/**
* Returns whether any elements of this stream match
* the provided predicate. May not evaluate the
* predicate on all elements if not necessary for
* determining the result.
*/
boolean anyMatch(Predicate< ? super T > predicate);
```

```
/**
* Returns an Optional describing some element of the
* stream, or an empty Optional if the stream is empty.
* The behavior of this operation is explicitly
* nondeterministic; it is free to select any element
* in the stream.
*/
Optional< T > findAny();

/**
* Returns a stream consisting of the elements of this
* stream, truncated to be no longer than maxSize in
* length.
*/
Stream< T > limit(long maxSize);
```

SplitIterator

```
/**
* An object for traversing and partitioning elements
* of a source.
*
*/
public interface Spliterator< T >;

/**
* Il metodo di avanzamento diventa tryAdvance(), che ribalta il
* funzionamento dell'iterator: non è l'utilizzatore che ottiene il
* nuovo elemento, ma è lo stream che fornisce l'elemento al codice che
* deveoperarci sopra.
*
* If a remaining element exists, performs the given
* action on it, returning true; else returns false.
*
*/
boolean tryAdvance(Consumer< ? super T > action);

/**
* Returns an estimate of the number of elements that
* would be encountered by a forEachRemaining()
* traversal, or returns Long.MAX_VALUE if infinite,
* unknown, or too expensive to compute.
*
* @return the estimated size
*/
long estimateSize();

/**
* notare il tipo di ritorno. Ci si attende che ogni chiamata a questo
* metodo dia lo stesso risultato; eventualmente, il risultato può cambiare dopo
la separazione.
* Returns a set of characteristics of this
* Spliterator and its elements.
*
* @return a representation of characteristics
```

```
*/
int characteristics();

/**
* Performs the given action for each remaining element,
* sequentially in the current thread, until all elements
* have been processed or the action throws an exception.
*
*/
default void forEachRemaining(Consumer< ? super T > action);

/**
* If this spliterator can be partitioned, returns a
* Spliterator covering elements, that will, upon return
* from this method, not be covered by this Spliterator.
*
* @return a Spliterator covering some portion of the
* elements, or null if this spliterator cannot
* be split
*
*/
Spliterator< T > trySplit();
```

Collector

La classe Collectors permette di produrre dei Collector a partire dagli elementi di base:

- un Supplier< A > del contenitore di risultato
- un BiConsumer< A,T > che accumula un elemento nel contenitore
- un BinaryOperator< A > che combina due contenitori parziali
- un Function< A,R > che dal contenitore ottiene il risultato finale

```
/**
* Performs a reduction on the elements of this stream,
* using the provided identity value and an associative
* accumulation function, and returns the reduced value.
*
* @param identity the identity value for the accumulating
* function
* @param accumulator an associative, non-interfering,
* stateless function for combining
* two values
* @result the result of the reduction
*
*/
T reduce(T identity, BinaryOperator< T > accumulator);

/**
* Performs a mutable reduction operation on the elements
* of this stream using a Collector.
*
* @R the type of the result
* @A the intermediate accumulation type of the Collector
* @param the Collector describing the reduction
* @result the result of the reduction
*/
< R,A > R collect(Collector< ? super T,A,R > collector);
```

```
/**
* A mutable reduction operation that accumulates input
* elements into a mutable result container.
*
* @T the type of input elements to the reduction
* operation
* @A the mutable accumulation type of the
* reduction operation
* @R the result type of the reduction operation
*/
interface Collector< T,A,R >;
```

Vertx

Le sue principali caratteristiche sono:

- modularità e compattezza delle dipendenze
- event-driven, fortemente asincrono
- poliglotta
- altamente scalabile
- approccio non-standard
- "unopinionated"
- semplice e pragmatico
- open source (Apache 2)
- supportato dalla fondazione Eclipse

L'approccio del framework Vert.X è tipico dei framework asincroni: ci viene fornito un ambiente all'interno del quale possiamo specificare gli handler che verranno richiamati in seguito a eventi definiti dal framework.

Le conseguenze sono:

- con un po' di disciplina, possiamo mantenere il codice che interagisce con il framework all'interno dell'handler
- gli handler compiono effetti collaterali, non sono quindi funzioni pure
- la struttura dichiarativa costringe ad esternalizzare l'organizzazione del codice
- è in carico a noi la gestione coerente dello stato condiviso

Astrazione

L'astrazione che il modulo vertx-web ci mette a disposizione è un router che ci consente di associare ad una URL una lambda.

Questa può modificare la risposta impostando esito e contenuto. Il modo in cui il nostro codice interagisce con il framework è definito da questa astrazione.

Perchè usare un framework

Un framework fornisce un ambiente all'interno del quale un insieme di casi d'uso fondamentali è reso facile, efficiente e sicuro da implementare.

Un framework per applicazioni web, per esempio, rende facile:

- specificare le rotte a cui rispondere
- costruire le risposte

Lo stesso framework web, cercherà di rendere trasparente, ed eventualmente configurabile:

- la gestione dei dettagli del protocollo
- la sicurezza nel trattamento della comunicazione
- la suddivisione delle risorse fra le varie parti del sistema

Gli autori del framework scelgono le loro priorità fra semplicità d'uso, sicurezza ed efficienza. Come utenti, noi otteniamo automaticamente ogni incremento su ciascuna di queste dimensioni.

Perchè non usare un framework

Un caso d'uso che non è fra quelli prescelti come importanti dagli autori del framework, può essere molto, molto complesso da implementare.

Un framework per applicazioni web, per esempio, può rendere difficile:

- rispondere a richieste esotiche
- controllare finemente l'erogazione della risposta

Lo stesso framework web, potrebbe rendere oscuro, oppure inaspettatamente:

- fallire l'esecuzione di una richiesta a causa di un baco nella sua implementazione
- aprire una falla di sicurezza a causa di un default errato
- permettere il sovraccarico del sistema a causa di una mancata limitazione delle risorse

Gli autori del framework scelgono le loro priorità fra semplicità d'uso, sicurezza ed efficienza.

Come utenti, dipendiamo completamente da loro per l'ordine con cui queste caratteristiche sono implementate, e dobbiamo sopportare i costi di aggiornamento se l'interfaccia fra il framework ed il nostro codice cambia.

Se il framework prende una direzione che non si allinea alle nostre esigenze, oppure cambia radicalmente interfacce esposte richiedendo manutenzione del nostro codice, o ancora peggio se viene abbandonato, possiamo incorrere in problemi anche gravi. La scelta del framework richiede quindi una conoscenza non solo della tecnica, ma anche del mercato e dell'ecosistema che lo circonda.