Java 8 (repeated) function call support

Each function call has a timeout and terminates the execution. After each function call (even after a timeout or an exception) a `TaskResult` is created with one of the following `ResultCode`:

- `ResultCode.OK` The execution is completed without exception and the result can be obtained with `TaskResult#getResult`
- `ResultCode.TIMEOUT` The function call was **NOT** completed, the task was canceled
- `ResultCode.ERROR` An unexpected exception has occurred. The exception can be determined with `TaskResult#getErrorReason`.

Quick start

This library supports `Runnable`, `Callable`, `Consumer` and `Function`. When a timeout occurs, the worker thread is `interrupted`. Be careful to handle the interruption properly, otherwise a thread of an `ExecutorService` (⇒ your function) could end up in an infinite loop (⇒ examples). One way to handle an interruption is as follows:

```
if (Thread.interrupted()) {
  throw new InterruptedException();
}
```

Setup

1. Import the `Library` in your project (see instructions below)
2. Create & store a `WatchdogFactory`. Each factory stores two `ExecutorServices`
3. Use `Watchable.builder(⋯)` to create a watchable and `WatchableOptions.builder(⋯)` to create the options
   optional: add a `ResultProcessor` to the watchable builder **(this callback not monitored)**
4. Create an asynchronous function call with `WatchdogFactory#submitFunctionCall`, a synchronized function call using `WatchdogFactory#waitForCompletion` or create a `RepeatableTask` with `WatchdogFactory#createRepeated`
5. In case of a RepeatableTask call the task with `RepeatableTask#submitFunctionCall` and `RepeatableTask#waitForCompletion`

Import the Library

Gradle

```
repositories {
    maven {
        url = uri("https://maven.pkg.github.com/JanPollmann/Watchdog")
    }
}

dependencies {
    implementation 'de.pollmann.watchdog:watchdog:<version>'
}
```

Maven

Follow the GitHub documentation of the desired package: https://github.com/JanPollmann/Watchdog/packages

Sources

Repo: https://github.com/JanPollmann/Watchdog
Module: https://github.com/JanPollmann/Watchdog/tree/HEAD/Library

Important

CAUTION

- Please remind: `ResultProcessor` (if specified) will be called **after** the monitored function call **without any timeout**. If there is heavy computational work, the call will take longer as specified (or will not terminate if there is an infinite loop)
  - Obviously, you could execute a `RepeatableTask` as `ResultProcessor` :)
- The timeout is specified in milliseconds
  - A timeout of 0 ms will be handled as `no timeout`
- For not repeated function calls, the input of a function/consumer is passed to the builder!
- As soon as the internal worker of the `WatchdogFactory` gets garbage collected, `ExecutorServices#shutdown` is called for every `ExecutorServices` (finalize)
  - A `RepeatableTask` has a reference to the internal worker
  - To terminate the RepeatableTask call `RepeatableTask#terminate`
  - A terminated RepeatableTask will throw a `RepeatableTaskTerminatedException`!

Changelog

v0.1.0

- Introduced `WatchableOptions`, the old API is deprecated (see Javadoc)

Example

Just implement `loop`. Remarks:

- A timeout of 0 ms will be handled as `no timeout`
- both tasks have a `ResultConsumer` registered, but that's an optional feature

```
package de.pollmann.watchdog.tester.app;

import de.pollmann.watchdog.*;
import de.pollmann.watchdog.tasks.Watchable;

import java.util.Objects;

public abstract class FastLoopApp {

  protected static final Integer OK = 0;

  protected final AppContext context;

  private final WatchdogFactory coreFactory;
  /**
   * create the main loop callable. The result is the exit code. The main loop
continues as long as the exit code is {@value OK} and the result code is {@link
ResultCode#OK}
   */
  private final RepeatableTaskWithoutInput<Integer> mainLoop;
  /**
   * will always time out
   */
  private final RepeatableTaskWithoutInput<Integer> timeout;

  public FastLoopApp(AppContext appContext) {
    context = appContext;
    coreFactory = new WatchdogFactory("core");
    // create the main loop callable with enabled statistics
    mainLoop =
coreFactory.createRepeated(WatchableOptions.builder(context.getLoopTimeout())
      // enable statistics
      .enableStatistics().build(),
      Watchable.builder(this::loop)
      // register a loop finished listener
      .withResultConsumer(this::onLoopFinished).build()
    );
    // timeout example
```

```
    timeout = coreFactory.createRepeated(WatchableOptions.builder(10).build(),
Watchable.builder(this::timeout).build());
  }

  /**
   * Start the application
   */
  public final void start() {
    // timeout 0: no timeout, block as long as the task takes
    try {
      coreFactory.waitForCompletion(WatchableOptions.builder(0).build(),
          Watchable.builder(() -> {
            TaskResult<Integer> result;
            boolean stop = false;
            double lastLoopsPerSecond = 0;
            // loop as fast as possible
            do {
              if (Thread.interrupted()) {
                throw new InterruptedException();
              }
              // call the main loop once
              result = mainLoop.waitForCompletion();
              // call timeout
              TaskResult<Integer> timeoutResult = timeout.waitForCompletion();
              if (!timeoutResult.hasError() ||timeoutResult.getCode() !=
ResultCode.TIMEOUT || timeoutResult.getResult() != null ||
timeoutResult.getErrorReason() == null) {
                // if the timeout does not work (it does!) this would stop the main
loop
                // a task in timeout is always in error state and has
ResultCode.TIMEOUT
                // timeoutResult.getResult() is null
                stop = true;
                System.out.println("Timeout does not work :(");
              }
              // print statistics (statistics are enabled for the Repeated Task
"mainLoop"!)
              if (lastLoopsPerSecond != mainLoop.getCallsPerSecond()) {
                lastLoopsPerSecond = mainLoop.getCallsPerSecond();
                // the metrics are only valid after a few seconds, because the arrays
have to fill with data first
                System.out.printf("Current loops per second: %.2f (Call: %.4f ns,
Result: %.4f ns, Overhead: %.4f ns - %.2f %%%n",
                  lastLoopsPerSecond,
                  mainLoop.getAverageApproximatedCallTime(),
                  mainLoop.getAverageApproximatedResultConsumingTime(),
                  mainLoop.getAverageApproximatedOverhead(),
                  mainLoop.getRelativeAverageApproximatedOverhead() * 100
                );
              }
              // The main loop continues as long as this condition is true
```

```
              if (result.getCode() != ResultCode.OK ||
!Objects.equals(result.getResult(), OK)) {
                stop = true;
              }
            } while (!stop);
            return result.getResult();
          })
          // register an exit listener
          .withResultConsumer(this::onExit)
          .build()
      );
    } catch (InterruptedException e) {
      System.out.println("App was interrupted!");
    }
  }

  public Integer timeout() throws Exception {
    int i = 1;
    while (i > 0) {
      // a loop, not responding to interrupts will lead into a never finishing
ExecutorService!
      if (Thread.interrupted()) {
        throw new InterruptedException();
      }
      i++;
      if (i >= 1000) {
        i = 1;
      }
    }
    return i;
  }

  /**
   * The main loop continues as long as the exit code is {@value OK} and no timeout
({@link ResultCode#TIMEOUT}) or error ({@link ResultCode#ERROR}) occurred
   *
   * @return the exit code
   */
  public abstract Integer loop() throws Exception;

  /**
   * The main loop result containing the last exit code of {@link #loop()}
   *
   * @param taskResult the result of the main loop (NOT the taskResult of {@link
#loop()}!)
   */
  public void onExit(TaskResult<Integer> taskResult) throws InterruptedException {
    System.out.println("exit");
  }

  /**
```

```
   * Listener for each execution of {@link #loop()}
   *
   * @param taskResult the taskResult of {@link #loop()}
   */
  public void onLoopFinished(TaskResult<Integer> taskResult) throws
InterruptedException {
    Thread.sleep(1);
  }

}
```

terminate endless loop

```
package de.pollmann.watchdog;

import de.pollmann.watchdog.tasks.ExceptionConsumer;
import de.pollmann.watchdog.tasks.ExceptionRunnable;
import de.pollmann.watchdog.tasks.Watchable;
import de.pollmann.watchdog.tasks.WatchableWithInput;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;

import java.util.concurrent.TimeoutException;
import java.util.concurrent.atomic.AtomicBoolean;

public class SabotageTest {

  /**
   * This runnable will block the ExecutorService, not even a timeout occur (thread
cannot be interrupted!)
   */
  private static class Sabotage implements ExceptionRunnable {
    @Override
    public void run() throws Exception {
      int i = 1;
      while (i > 0) {
        i++;
        if (i >= 1000) {
          i = 1;
        }
      }
    }
  }

  /**
   * The runnable {@link Sabotage} will block the ExecutorService!
   */
  @Test
  @Timeout(2)
```

```
  void
runnable_userTriesEverythingToSabotageTheTimeout_TIMEOUT_threadIsNeverFinished()
throws InterruptedException {
    WatchdogFactory watchdogFactory = new WatchdogFactory(2);
    Watchable<Object> sabotage =  Watchable.builder(new Sabotage()).build();
    AtomicBoolean sabotageStarted = new AtomicBoolean(false);
    AtomicBoolean sabotageStopped = new AtomicBoolean(false);
    AtomicBoolean wasInterrupted = new AtomicBoolean(false);
    // wrapp the call 'sabotage' in a timeout
    //    => interrupt
    //    => cancel the wrapped call but leaves the Thread of the Executor Service in
an infinite loop
    //    => at some point the watchdogFactory does not have any thread remaining
    TaskResult<?> wrappedCall =
watchdogFactory.waitForCompletion(WatchableOptions.builder(1300).build(),
Watchable.builder(() -> {
      sabotageStarted.set(true);
      try {
        // timeout can NOT kill the Thread because the submitted task is not
interruptable
        TaskResult<?> neverCreated =
watchdogFactory.waitForCompletion(WatchableOptions.builder(1000).build(), sabotage);
        // the runnable does not respond to an interrupt => not stopped => infinite
loop
        // unreachable ...
        sabotageStopped.set(true);
      } catch (InterruptedException interruptedException) {
        // But "waitForCompletion" itself is interruptable
        wasInterrupted.set(true);
        throw interruptedException;
      }
    }).build());

    Assertions.assertTrue(wasInterrupted.get());
    Assertions.assertFalse(sabotageStopped.get());
    Assertions.assertTrue(sabotageStarted.get());
    Assertions.assertFalse(sabotage.stopped());
    assertTimeout(wrappedCall);
  }

  /**
   * This runnable will not the ExecutorService because of the {@link
InterruptedException}
   */
  private static class NiceSabotageRunnable implements ExceptionRunnable {

    @Override
    public void run() throws Exception {
      int i = 1;
      while (i > 0) {
        // add this if to your code to stop the runnable if a timeout occurred
```

```
        if (Thread.interrupted()) {
          throw new InterruptedException();
        }
        // e.g. thread sleep does that as well
        Thread.sleep(10);
        i++;
        if (i >= 1000) {
          i = 1;
        }
      }
    }

  }

  /**
   * The runnable {@link NiceSabotageRunnable} will NOT block the ExecutorService!
   */
  @Test
  @Timeout(10)
  void runnable_endlessLoopRespondingToInterrupts_TIMEOUT_threadGetsInterrupted()
throws InterruptedException {
    WatchdogFactory watchdogFactory = new WatchdogFactory();

    for (int i = 0; i < 100; i++) {
      Watchable<Object> niceSabotage =  Watchable.builder(new
NiceSabotageRunnable()).build();

assertTimeout(watchdogFactory.waitForCompletion(WatchableOptions.builder(50).build(),
niceSabotage));
      // the runnable does respond to an interrupt => stopped => no endless loop
      Assertions.assertTrue(niceSabotage.stopped());
    }

  }

  @Test
  @Timeout(10)
  void consumer_endlessLoopRespondingToInterrupts_TIMEOUT_threadGetsInterrupted()
throws InterruptedException {
    WatchdogFactory watchdogFactory = new WatchdogFactory();

    for (int i = 0; i < 100; i++) {
      Watchable<Object> niceSabotage =  Watchable.builder(new
NiceSabotageConsumer()).build();

assertTimeout(watchdogFactory.waitForCompletion(WatchableOptions.builder(50).build(),
niceSabotage));
      // the runnable does respond to interrupts => stopped => no endless loop
      Assertions.assertTrue(niceSabotage.stopped());
    }
```

```
  }

  @Test
  @Timeout(10)
  void
repeatableRunnable_endlessLoopRespondingToInterrupts_TIMEOUT_threadGetsInterrupted()
throws InterruptedException {
    WatchdogFactory watchdogFactory = new WatchdogFactory();
    Watchable<Object> niceSabotage =  Watchable.builder(new NiceSabotageRunnable())
      .withResultConsumer(result ->
Assertions.assertTrue(result.getWatchable().stopped()))
      .build();
    RepeatableTaskWithoutInput<Object> repeatable =
watchdogFactory.createRepeated(WatchableOptions.builder(50).build(), niceSabotage);

    for (int i = 0; i < 100; i++) {
      assertTimeout(repeatable.waitForCompletion());
    }

  }

  @Test
  @Timeout(10)
  void
repeatableConsumer_endlessLoopRespondingToInterrupts_TIMEOUT_threadGetsInterrupted()
throws InterruptedException {
    WatchdogFactory watchdogFactory = new WatchdogFactory();
    WatchableWithInput<Integer, Object> niceSabotage =  Watchable.builder(new
NiceSabotageConsumer())
      .withResultConsumer(result ->
Assertions.assertTrue(result.getWatchable().stopped()))
      .build();
    RepeatableTaskWithInput<Integer, Object> repeatable =
watchdogFactory.createRepeated(WatchableOptions.builder(50).build(), niceSabotage);

    for (int i = 0; i < 100; i++) {
      assertTimeout(repeatable.waitForCompletion(i));
    }

  }

  private void assertTimeout(TaskResult<?> result) {
    Assertions.assertNotNull(result);
    Assertions.assertTrue(result.hasError());
    Assertions.assertEquals(ResultCode.TIMEOUT, result.getCode(),
String.format("Error: %s", result.getErrorReason()));
    Assertions.assertNotNull(result.getErrorReason());
    Assertions.assertNull(result.getResult());
    Assertions.assertTrue(result.getErrorReason() instanceof TimeoutException);
  }
```

```
  /**
   * This runnable will not the ExecutorService because of the {@link
InterruptedException}
   */
  private static class NiceSabotageConsumer implements ExceptionConsumer<Integer> {

    @Override
    public void accept(Integer t) throws Exception {
      int i = 1;
      while (i > 0) {
        // add this if to your code to stop the runnable if a timeout occurred
        if (Thread.interrupted()) {
          throw new InterruptedException();
        }
        // e.g. thread sleep does that as well
        Thread.sleep(10);
        i++;
        if (i >= 1000) {
          i = 1;
        }
      }
    }

  }

}
```

You can find more examples in the UnitTests:
Library/src/test/java/de/pollmann/watchdog/