

Cancellation and Shutdown

Alles hat ein Ende.....

...nur die Wurst hat zwei.

Task Cancellation

Es gibt 5 Gründe für das Beenden von Programmen:

- Errors
- Shutdown
- Time-limited activities
- Application events
- User-requested cancellation

Ein Ansatz zum Beenden eines Threads ist das Einführen einer eigenen booleschen Variable:

```
1. @Threadsafe
2. public class PrimeGenerator implements Runnable {
3.     @GuardedBy("this")
4.     private final List<BigInteger> primes = new ArrayList<BigInteger>();
5.     private volatile boolean cancelled;
6.
7.     public void run() {
8.         BigInteger p = BigInteger.ONE;
9.         while (!cancelled) {
10.             p = p.nextProbablePrime();
11.             synchronized (this) {
12.                 primes.add(p);
13.             }
14.         }
15.     }
16.
17.     public void cancel() {cancelled = true;}
18.
19.     public synchronized List<BigInteger> get() {
20.         return new ArrayList<BigInteger>(primes);
21.     }
22. }
```

```
1. List<BigInteger> aSecondOfPrimes() throws InterruptedException {
2.     PrimeGenerator generator = new PrimeGenerator();
3.     new Thread(generator).start();
4.     try {
5.         SECONDS.sleep(1);
6.     } finally {
7.         generator.cancel();
8.     }
9.     return generator.get();
10. }
```

```
1. class BrokenPrimeProducer extends Thread {
2.     private final BlockingQueue<BigInteger> queue;
3.     private volatile boolean cancelled = false;
4.
5.     BrokenPrimeProducer(BlockingQueue<BigInteger> queue) {
6.         this.queue = queue;
7.     }
8.
9.     public void run() {
10.        try {
11.            BigInteger p = BigInteger.ONE;
12.            while (!cancelled)
13.                queue.put(p = new p.nextProbablePrime());
14.        } catch (InterruptedException consumed) { }
15.    }
16.
17.    public void cancel() {cancelled = true;}
18.}
19.
20. void consumePrimes() throws InterruptedException {
21.     // entnimmt Elemente aus der queue
22.}
```

Interruption

- Jeder Thread hat eine eingebaute boolesche Variable, die man mit folgenden Methoden anfragen bzw. setzen kann:

1. `public class Thread {`
2. `public void interrupt() { ... }`
3. `public boolean isInterrupted() { ... }`
4. `public static boolean interrupted() { ... }`
5. `...`
6. `}`

- Die statische Methode `interrupted()` gibt den Wert von `currentThread().isInterrupted()` zurück und setzt den `interrupted`-Status auf `false`

```
1. class PrimeProducer extends Thread {
2.     private final BlockingQueue<BigInteger> queue;
3.
4.     PrimeProducer(BlockingQueue<BigInteger> queue) {
5.         this.queue = queue;
6.     }
7.
8.     public void run() {
9.         try {
10.             BigInteger p = BigInteger.ONE;
11.             while (!Thread.currentThread().isInterrupted())
12.                 queue.put(p = new p.nextProbablePrime());
13.         } catch (InterruptedException consumed) {
14.             /* Allow thread to exit */
15.         }
16.     }
17.
18.     public void cancel() {interrupt();}
19. }
```

Interruption Policies

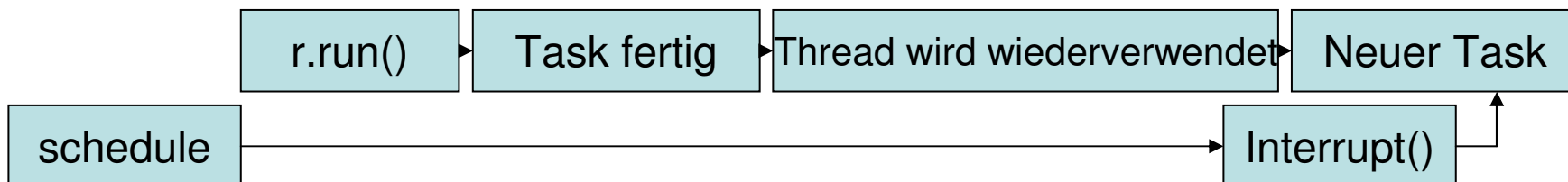
- Möglichst schnell, aber definiert beenden, „aufräumen“ und den „besitzenden Code“ benachrichtigen
- Meistens ist es am Besten, den Interrupted-Status zu erhalten
- Man sollte nur eigenen/bekanntem Code interrupten

Responding to Interruption

- Entweder die Exception weiterleiten oder nicht fangen, so dass der übergeordnete Code darauf reagieren kann
- Interrupted Status wiederherstellen d.h. `interrupt()` aufrufen
- Nur Code, der die Interruption-Regeln eines Threads implementiert, darf u.U. die Exception verschlucken

```
1. private static final ScheduledExecutorService cancelExec = ...;
2.
3. public static void timedRun(Runnable r, long timeout, TimeUnit unit) {
4.     final Thread taskThread = Thread.currentThread();
5.     cancelExec.schedule(new Runnable() {
6.         public void run() { taskThread.interrupt(); }
7.     }, timeout, unit);
8.     r.run();
9. }
```

Ein möglicher Ablauf:



```
1. public static void timedRun (final Runnable r, long timeout, TimeUnit unit)
2.     throws InterruptedException {
3.     class RethrowableTask implements Runnable {
4.         private volatile Throwable t;
5.         public void run(); {
6.             try { r.run(); }
7.             catch (Throwable t) { this.t = t; }
8.         }
9.         void rethrow() {
10.            if (t != null)
11.                throw launderThrowable(t);
12.        }
13.    }
14.
15.    RethrowableTask task = new RethrowableTask();
16.    final Thread taskThread = new Thread(task);
17.    taskThread.start();
18.    cancelExec.schedule(new Runnable() { public void run() {
19.        taskThread.interrupt(); } }, timeout, unit);
20.    taskThread.join(unit.toMillis(timeout));
21.    task.rethrow();
22.}
```

Cancellation via Future

- `ExecutorService.submit()`: Gibt ein Future Objekt zurück, das die Aufgabe beschreibt
- `Future.cancel()`: Beendet die Aufgabe sofort
- wenn man einen Standard Executor verwendet, kann man ohne Gefahr `Future.cancel()` aufrufen

```
1. public static void timedRun(Runnable r, long timeout, TimeUnit unit)
2.     throws InterruptedException {
3.     Future<?> task = taskExec.submit(r);
4.     try {
5.         task.get(timeout, unit);
6.     } catch (TimeoutException e) {
7.         // task will be cancelled below
8.     } catch (ExecutionException e) {
9.         // exception thrown in task; rethrow
10.        throw launderThrowable(e.getCause());
11.    } finally {
12.        // Harmless if task already completed
13.        task.cancel(true); // interrupt if running
14.    }
15. }
```

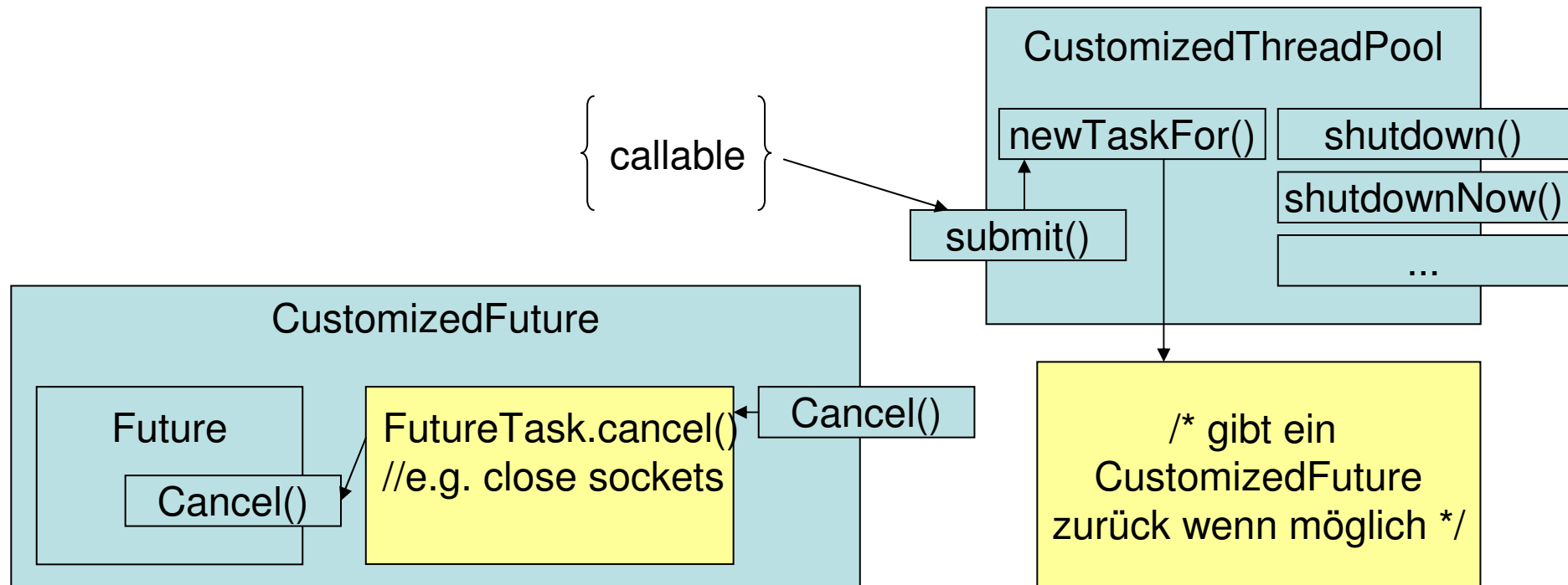
Dealing with non-interruptable blocking

- `interrupt()` Methode muss überschrieben werden, wenn ein non-interruptable Block schnell aufgelöst werden soll
- Beispiele
 - Lock Erwerb
 - I/O mit sockets

```
1. public class ReaderThread extends Thread {
2.     private final Socket socket;
3.     private final InputStream in;
4.     public ReaderThread(Socket socket) throws IOException {
5.         this.socket = socket;
6.         this.in = socket.getInputStream();
7.     }
8.
9.     public void interrupt() {
10.        try {
11.            socket.close();
12.        }
13.        catch (IOException ignored) { }
14.        finally {
15.            super.interrupt();
16.        }
17.    }
18.    public void run() {
19.        try {
20.            byte[ ] buf = new byte[BUFSZ];
21.            while (true) {
22.                int count = in.read(buf);
23.                if (count < 0)
24.                    break;
25.                else if (count > 0)
26.                    processBuffer(buf, count);
27.            }
28.        } catch (IOException e) { /*Allow thread to exit */}
29.    }
30. }
```

Encapsulating nonstandard cancellation with newTaskFor

- Man kapselt Future so in ein Objekt ein, dass man bestehende Methoden wie cancel() erweitern kann



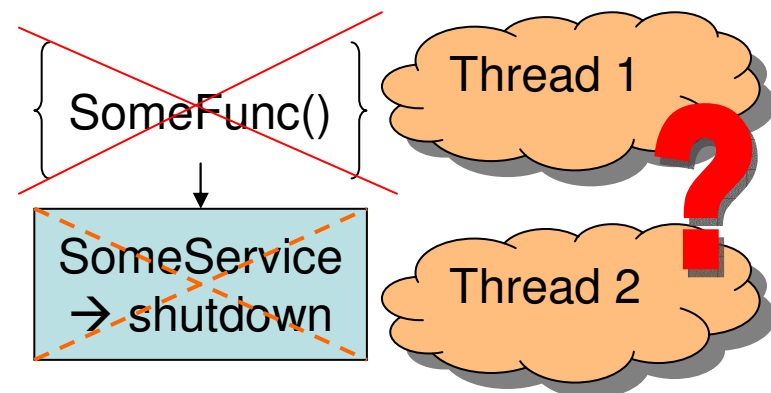
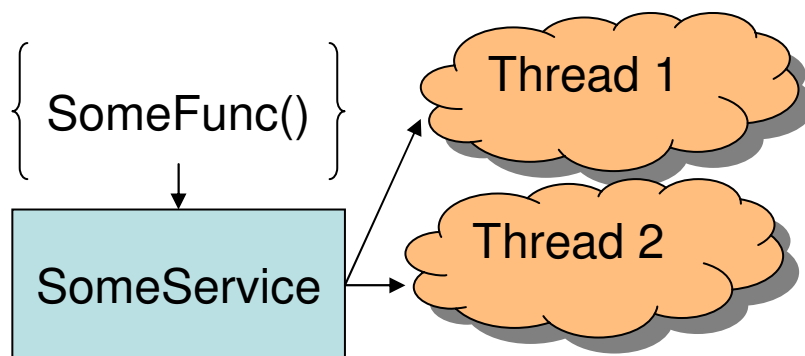
```

1. public interface CancellableTask<T> extends Callable<T> {
2.     void cancel();
3.     RunnableFuture<T> newTask();
4. }
5. public class CancellingExecutor extends ThreadPoolExecutor {
6.     ...
7.     protected<T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
8.         if (callable instanceof CancellableTask)
9.             return ((CancellableTask<T>) callable).newTask();
10.        else
11.            return super.newTaskFor(callable);
12.    }
13. }
14. public abstract class SocketUsingTask<T> implements CancellableTask<T> {
15.     protected synchronized void setSocket(Socket s) { socket = s; }
16.     public synchronized void cancel() {
17.         try {
18.             if (socket != null)
19.                 socket.close();
20.         } catch (IOException ignored) { /* ignorieren wir */ }
21.     }
22.
23.     public RunnableFuture<T> newTask() {
24.         return new FutureTask<T>(this) {
25.             public boolean cancel(boolean mayInterruptIfRunning) {
26.                 try {
27.                     SocketUsingTask.this.cancel();
28.                 } finally {
29.                     return super.cancel(mayInterruptIfRunning);
30.                 }
             }
         }
     }

```

Stopping a thread-based service

- Anwendungen erstellen Dienste und Dienste wiederum besitzen Threads
- Anwendungen haben eine geringere Lebenszeit als Threads
- die Anwendung wird beendet → Threads müssen beendet werden, aber wer ist dafür zuständig?



```
1. public class Logwriter {
2.     private final BlockingQueue<String> queue;
3.     private final LoggerThread logger;
4.
5.     public Logwriter(Writer writer) {
6.         this.queue = new LinkedBlockingQueue<String>(CAPACITY);
7.         this.logger = new LoggerThread(writer);
8.     }
9.
10.    public void start() { logger.start(); }
11.
12.    public void log(String msg) throws InterruptedException {queue.put(msg);}
13.
14.    private class LoggerTread extends Thread {
15.        private final PrintWriter writer;
16.        ...
17.        public void run() {
18.            try {
19.                while (true)
20.                    writer.println(queue.take());
21.            } catch(InterruptedException ignored) {
22.            } finally {
23.                writer.close();
24.            }
25.        }
26.    }
27. }
```

```

1. public class LogService {
2.     private final BlockingQueue<String> queue;
3.     private final LoggerThread loggerThread;
4.     private final PrintWriter writer;
5.     private boolean isShutdown;
6.     private int reservations;
7.     public void start() { loggerThread.start(); }
8.     public void stop() {
9.         synchronized (this) { isShutdown= true; }
10.        loggerThread.interrupt();
11.    }
12.    public void log(String msg) throws InterruptedException {
13.        synchronized (this) {
14.            if (isShutdown) { throw new IllegalStateException(...); }
15.            ++reservations;
16.        } queue.put(msg);
17.    }
18.    private class LoggerThread extends Thread {
19.        public void run() {
20.            try {
21.                while (true) {
22.                    try {
23.                        synchronized (this) {
24.                            if (isShutdown && reservations == 0)
25.                                break;
26.                        }
27.                        String msg = queue.take();
28.                        synchronized (this) { --reservations; }
29.                        writer.println(msg);
30.                    } catch (InterruptedException e) { /*retry*/ }
31.                }
32.            } finally {
33.                writer.close();
34.            }
34.    }

```

Poison Pills

- Eine „Poison Pill“ ist ein Objekt, dass dem Consumer signalisiert, dass er sich beenden soll („Rote Karte“ = Platzverweis)
- benötigt:
 - Bekannte Producer-Anzahl
 - Bekannte Consumer-Anzahl
 - (FIFO queue)

```
1. public class IndexingService {
2.     private static final File POISON = new File("");;
3.     private final IndexerThread consumer = new IndexerThread();
4.     private final CrawlerThread producer = new CrawlerThread();
5.     private final BlockingQueue<File> queue;
6.     private final FileFilter fileFilter;
7.     private final File root;
8.
9.     class CrawlerThread extends Thread { /* nächste Folie */ }
10.    class IndexerThread extends Thread { /* übernächste Folie */ }
11.
12.    public void start() {
13.        producer.start();
14.        consumer.start();
15.    }
16.
17.    public void stop() { producer.interrupt(); }
18.
19.    public void awaitTermination() throws InterruptedException {
20.        consumer.join();
21.    }
22. }
```

```
1. public class CrawlerThread extends Thread {
2.     public void run() {
3.         try {
4.             crawl(root);
5.         } catch (InterruptedException e) { /* fall through */ }
6.         finally {
7.             while (true) {
8.                 try {
9.                     queue.put(POISON);
10.                    break;
11.                } catch(InterruptedException e1){/*retry*/}
12.            }
13.        }
14.    }
15.
16.    private void crawl(File root) throws InterruptedException {
17.        ...
18.    }
19.}
```

```
1. public class IndexerThread extends Thread {
2.     public void run() {
3.         try {
4.             while (true) {
5.                 File file = queue.take();
6.                 if (file == POISON)
7.                     break;
8.                 else
9.                     indexFile(file);
10.            }
11.        } catch (InterruptedException consumed) { }
12.    }
13. }
```

ExecutorService Shutdown

- ExecutorService bietet die Methoden `shutdown()` und `shutdownNow()`
- einfache Programme starten und beenden ExecutorServices aus der `main()`-Methode heraus, komplexere Programme überlassen dies übergeordneten Service-Klassen
- durch das Einkapseln des ExecutorServices lässt sich dieser besser verwalten

Limitations of ShutdownNow

- Problem: Wir wollen wissen, welche Aufgaben liefen, als `ShutdownNow()` aufgerufen wurde
- Lösung: Der TrackingExecutor fügt seinen Task zu einem `synchronizedSet` hinzu, wenn
 - `isShutdown() true` ist
 - und der aktuelle Thread interrupted ist

```
1. public class TrackingExecutor extends AbstractExecutorService {
2.     private final ExecutorService exec;
3.     private final Set<Runnable> tasksCancelledAtShutdown =
4.         Collections.synchronizedSet(new HashSet<Runnable>());
5.     ...
6.     public List<Runnable> getCancelledTasks() {
7.         if (!exec.isTerminated())
8.             throw new IllegalStateException(...);
9.         return new ArrayList<Runnable>(tasksCancelledAtShutdown);
10.    }
11.
12.    public void execute(final Runnable runnable) {
13.        exec.execute(new Runnable() {
14.            public void run() {
15.                try {
16.                    runnable.run();
17.                } finally {
18.                    if (isShutdown() &&
19.                        Thread.currentThread().isInterrupted())
20.                        tasksCancelledAtShutdown.add(runnable);
21.                }
22.            }
23.        });
24.    }
25.    //delegate other ExecutorService methods to exec
26. }
```

Handling abnormal thread Termination

- Traue niemals unbekanntem Code
- Fange alle Exceptions von fremdem Code

```
1. public void run() {  
2.     Throwable thrown = null;  
3.     try {  
4.         while (!isInterrupted())  
5.             runTask(getTaskFromWorkQueue());  
6.     } catch (Throwable e) {  
7.         thrown = e;  
8.     } finally {  
9.         threadExited(this, thrown);  
10.    }  
11.}
```

Uncaught Exception Handlers

- das Interface `UncaughtExceptionHandler` kann mit der Methode

`uncaughtException(Thread t, Throwable e)`

auf nicht gefangene Exceptions reagieren.

```
1. public interface UncaughtExceptionHandler {  
2.     void uncaughtException(Thread t, Throwable e);  
3. }
```

```
1. public class UEHLogger implements Thread.UncaughtExceptionHandler {  
2.     public void uncaughtException(Thread t, Throwable e) {  
3.         Logger logger = Logger.getAnonymousLogger();  
4.         logger.log(Level.SEVERE,  
5.             "Thread terminated with exception: " + t.getName(), e);  
6.     }  
7. }  
8.  
9. // UEH = UnununcultuntuExceptionuHandler
```

Shutdown Hooks

- Shutdown Hooks sind Threads, die beim Shutdown der JVM automatisch gestartet werden

```
Runtime.getRuntime().addShutdownHook(Thread t)
```

ermöglicht es, eigene Aufräumprotokolle hinzuzufügen

Daemon Threads

- Daemon Threads sind Threads, auf die beim Shutdown der JVM nicht gewartet wird, bis sie beendet werden
- Sie eignen sich für „Aufräum“ Aufgaben, wie z.B. ein Thread, der in regelmäßigen Abständen veraltete, unnötige Daten löscht

Finalizers

- Der Garbage Collector versucht – falls vorhanden – die `finalize()` Methode von zu recycelnden Objekten aufzurufen
 - Der Aufruf von `finalize()` ist nicht garantiert
 - Verbraucht viele Ressourcen
- Finalizer am besten vermeiden

Das Ende naht...

- Noch Fragen?