Lösung: ConCrawler – Multithreaded Webcrawler

CrawlCallable

```
class CrawlCallable implements Callable<List<String>> {
    private final String startURL;

    public CrawlCallable(String startURL) {
        this.startURL = startURL;
    }

    @Override
    public List<String> call() throws Exception {
        URL url = new URL(startURL);
        URLConnection conn = url.openConnection();
        conn.setRequestProperty("User-Agent", "ConCrawler/0.1 Mozilla/5.0");
        List<String> result = new LinkedList<String>();

        String contentType = conn.getContentType();
        if (contentType != null && contentType.startsWith("text/html")) {
            BufferedInputStream is = null;
            try {
                is = new BufferedInputStream(conn.getInputStream());
                Document doc = Jsoup.parse(is, null, startURL);
                Elements links = doc.select("a[href]");
                for (Element link : links) {
                    String linkString = link.absUrl("href");
                    if (linkString.startsWith("http")) {
                        result.add(linkString);
                    }
                }
            } finally {
                IOUtils.closeQuietly(is);
            }
        }
        return result;
    }
}
```

Die Klasse CrawlCallable nimmt im Konstruktor als Argument den String startURL und gibt als Resultat der call Methode alle Links zurück, die in der via startURL geladenen HTML Seite gefunden wurden. CrawlCallable ist somit wie eine Funktion von String => List<String> und somit sehr gut wiederverwendbar.

ParCrawler

```
public class ParCrawler implements Crawler {
    private static final int MAX_VISITS = 50;

    @Override
    public List<String> crawl(final String startURL) {
        final ExecutorService ex = Executors.newFixedThreadPool(MAX_VISITS); // (1)
        final CompletionService<List<String>> ecs =
                                new ExecutorCompletionService<List<String>>(ex); // (2)
```

```
        /* Contains all discovered urls. */
        final Set<String> urlsFound = new HashSet<String>();
        /* Contains the urls to be visited. */
        final Queue<String> urlsToVisit = new LinkedList<String>();
        urlsToVisit.add(startURL);

        while ((!urlsToVisit.isEmpty()) && urlsFound.size() < MAX_VISITS) {
            for(String url: urlsToVisit) {  // (3)
                ecs.submit(new CrawlCallable(url));
                urlsFound.add(url);
            }
            urlsToVisit.clear();  // (4)
            try {
                List<String> res = ecs.take().get(); // (5) wait for the first result
                for(String url: res) {
                    if(!urlsFound.contains(url) && !urlsToVisit.contains(url)) {
                        urlsToVisit.add(url);
                    }
                }
                Future<List<String>> fut;
                while((fut = ecs.poll()) != null) { // (6)just poll, don't wait
                    for(String url: fut.get()) {
                        if(!urlsFound.contains(url) && !urlsToVisit.contains(url)) {
                            urlsToVisit.add(url);
                        }
                    }                }
            } catch (Exception e) { e.printStackTrace(); }
        }

        List<String> result = new ArrayList<String>(MAX_VISITS); // (7)
        int i = 0;
        Iterator<String> it = urlsFound.iterator();
        while(i++ < MAX_VISITS && it.hasNext()) { result.add(it.next()); }
        ex.shutdownNow(); // (8) ignore result
        return result;
    }
}
```

Die ParCrawler Klasse implementiert das geforderte Interface. Die Klasse ist threadsafe, denn der einzige Zustand den die Klasse hat, ist die Konstante MAX_VISITS. Die gesamte Funktionalität ist innerhalb der Methode crawl implementiert.
Für jede Anfrage wird ein neuer Cached-ExecuterService erzeugt **(1)** und mit diesem ein CompletionService konstruiert **(2)**. Dann werden analog zum sequentiellen Crawler zwei Collections erstellt – urlsFound für die bereits gefundenen URLs und urlsToVisit für jene, welchen noch nicht gefolgt wurde.
Wenn noch weitere Resultate benötigt werden und noch unbesuchte URLs vorhanden sind, wird für jede unbesuchte URL einen CrawlRunnable erstellt und über den CompletionService ausgeführt **(3)**. Da jetzt für alle URLs einen Job gestartet wurde, kann die urlsToVisit Queue geleert werden **(4)**.
Dann wird auf dem CompletionService blockierend gewartet bis *ein* Resultat vorhanden ist **(5)**. Neue, noch nicht bekannte URLs werden dann in die urlToVisit Queue hinzugefügt, wenn Sie noch nicht drin sind.
Möglicherweise sind bereits weitere Resultate verfügbar. Mittels *poll* kann der CompletionService ohne zu blockieren abgefragt werden **(6)**. Die allfälligen Resultate werden dann gleich behandelt wie zuvor.
Es ist gut möglich, dass mehr als die maximale Anzahl Resultate gefunden wurde. Die gewünschte Anzahl Element wird abgezählt und in eine neue Liste gefüllt **(7)**. Bevor das Resultat zurückgegeben wird, wird der ExecutorService abgewürgt **(8)**. Mit einem Mehraufwand könnte man die laufenden Jobs noch canceln. Dazu müssten die CrawlCallables entsprechend auf Interrupts reagieren.