Orientation in Objects

Java Concurrency Utilities

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Steffen Schluff
Orientation in Objects GmbH

Veröffentlicht am: 13.2.2008

JAVA CONCURRENCY UTILITIES

Der seit den 70er Jahren vorhandene Trend, dass jede neue Prozessor Generation eine deutliche Steigerung der Taktfrequenz mit sich bringt, beginnt langsam aber sicher abzuebben. Die Hardware Hersteller versuchen nun, ihr Glück in der Erhöhung der verfügbaren Anzahl Prozessoren je Chip zu finden. Dementsprechend wird sich auch die Softwareentwicklung über kurz oder lang an diese neuen Rahmenbedingungen anpassen müssen.

Die Programmiersprache Java hat mit Java 5 in Form der sogenannten Concurrency Utilities eine mächtige neue API dazugewonnen, die es Programmierern erlaubt, mit ganz neuen Voraussetzungen an die Entwicklung von Multithreaded Anwendungen heranzugehen. Der vorliegende Artikel zeigt auf, warum das Themenfeld in Zukunft an Bedeutung gewinnen wird und stellt die wichtigsten Inhalte dieser API in Form eines Tutorial vor.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

HÖHER, SCHNELLER, WEITER

Seit über 30 Jahren ist die Entwicklung neuer Prozessoren untrennbar verbunden mit einem konstanten Anstieg der Leistungsfähigkeit der jeweils neuen Chip Generation. Diese Aussage bezieht sich vor allem auf die Bereiche Takftfrequenz, Ausführungsoptimierung und Cache. Der schöne, oder besser formuliert, praktische Aspekt an diesen drei Bereichen ist, dass sie allesamt einen direkten Performancegewinn für Software bedeuten, sei sie jetzt Single- oder Multithreaded. Dies führt zu gängigen Aussagen der Art "Die Anwendung ist zwar langsam, aber bis wir ausliefern hat auch der Kunde neue Rechner und dann passt das alles!".

Dieses von Herb Sutter scherzhaft als "Free Lunch" [1] bezeichnete regelmäßige Performance Geschenk beginnt aber seit circa 2002 auszubleiben, wie in der nachfolgenden Graphik unschwer erkennbar ist. Ursache für diese Stagnation sind die physikalischen Grenzen, an die man im Chipdesign zu stossen beginnt, vor allem die zu starke Wärmeentwicklung, der hohe Stromverbrauch sowie die vorhandenen Leckströme.

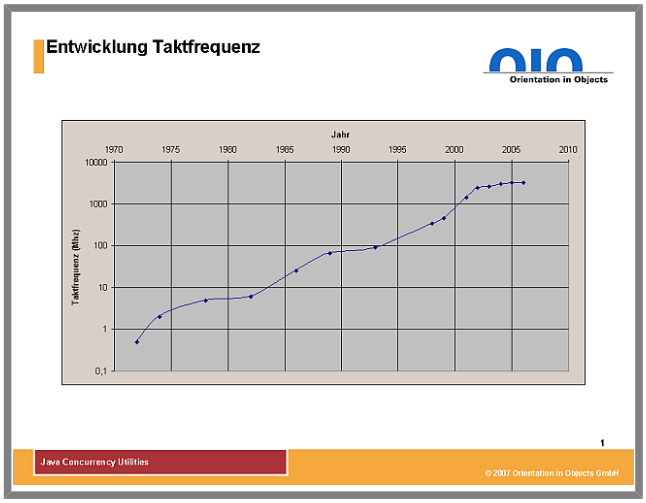

Abbildung 1: Entwicklung Taktfrequenz

Nichtsdestotrotz ist Moore's Law [10] weiterhin ungebrochen, denn es bezieht sich nicht auf die Entwicklung der Taktfrequenz, sondern viel mehr auf die Anzahl Transistoren in einem integrierten Schaltkreis und die wächst auch weiterhin. Der wesentliche Unterschied zu früher ist, dass die Leistungssteigerungen künftiger Chips verstärkt in den Bereichen Hyperthreading, Multicore und Cache zu finden sein werden. Dies hat zur Folge, dass die meisten Anwendungen in Zukunft nicht mehr "von alleine" schneller werden nur weil man neue Hardware kauft.

Die Schlussfolgerung, die sich daraus ergibt ist offensichtlich und lautet, dass man sich als Programmierer in Zukunft mit dem Thema "Concurrent Software" auseinandersetzen muss oder nur noch Bruchteile der Leistungsfähigkeit eines Computers nutzen können wird [1]. Aber was bedeutet eine solche Aussage nun für einen Java Entwickler?

BACK TO THE BASICS

Java unterstützt seit Beginn Multithreading [2], so dass sich die Frage stellt, wozu die ganze Aufregung dient, da Java doch offensichtlich gut für die Zukunft gewappnet ist.

Die Aussage ist sicherlich korrekt, dass mittels der Schlüsselworte `synchronized` und `volatile` sowie den `java.lang.Object` Methoden `wait()`, `notify()` beziehungsweise `notifyAll()` eine vollständige Multithreading Unterstützung vorhanden ist.

Ebenso zutreffend ist allerdings, dass es sich hierbei um komplizierte Konstrukte mit einem niedrigen Abstraktionsgrad handelt, die in ihrer reinen Form in der Praxis eher ungeeignet für die Anwendungsentwicklung sind.

Die Folge davon ist, dass jeder mittels hausgemachter Lösungen sein eigenes Süppchen kocht, wobei der entstehende Code häufig unausgegoren und im schlimmsten Fall weder performant noch testbar ist. Oder wie Sun es in seinem "Java Tuning White Paper" [3] ausdrückt: "Classical multi-threaded programming is very complex and error prone [...]".

Eine Alternative zu Eigenbau Lösungen war lange Zeit die als Open Source verfügbare util.concurrent Klassenbibliothek von Doug Lea [4]. Basierend auf diesen Erfahrungen wurde 2002 unter der Leitung von Doug Lea der JSR 166 ins Leben gerufen mit dem Ziel "[to] provide functionality commonly needed in concurrent programs" [5].

Das Ergebnis dieses JSR sind die sogenannten "Concurrency Utilities", die sich im Package "java.util.concurrent" befinden und die seit Java 5 fester Bestandteil des JDK sind [6]. Diese API stellt eine wesentliche Erweiterung von Java dar, auch wenn sie im Trubel um die ebenfalls in Java 5 vorgestellten neuen Sprachfeatures, allen voran Generics, etwas untergegangen ist.

CONCURRENCY UTILITIES BESTANDTEILE

Die Grundidee der Concurrency Utilities ist, Programmierern fertige Bausteine für die Anwendungsentwicklung zur Verfügung zu stellen, ähnlich wie es zuvor das Collection Framework getan hat. Dabei sollen alle Arten von Entwicklern berücksichtigt werden, vom Anwendungentwickler, der einfache aber "thread-safe" Datenstrukturen benötigt, über den Framework Entwickler, der selbst Threads koordinieren muss bis hin zum Concurrency Experten, der eigene Algorithmen implementieren will [7].

Dementsprechend lassen sich auch die Bestandteile der Concurrency Utilities unterteilen:

- Concurrent Collections

  Spezielle Implementierungen der Collection Framework Interfaces `Map`, `Set`, `List`, `Queue`, `Deque`.

- Executor Framework

  Ein Framework zur Ausführung asynchroner Tasks durch Threadpools.

- Synchronizers

  Diverse Hilfsklassen zur Koordination mehrerer Threads, zum Beispiel Semaphore oder CyclicBarrier.

- Locks und Conditions

  Flexiblere Objektrepräsentation des `synchronized` Schlüsselworts sowie der `java.lang.Object` Monitor Methoden `wait()`, `notify()` und `notifyAll()`.

- Atomic Variables

  Erlauben die atomare Manipulation einzelner Variablen (CAS - Compare And Set) zur Implementierung von Algorithmen ohne Sperren (lock-free algorithms).

Für die gängigen Fälle in der Anwendungsentwicklung besitzen die Concurrent Collections sowie das Executor Framework mit Sicherheit einen hohen Stellenwert und sollen dementsprechend nachfolgend näher vorgestellt werden.

CONCURRENT COLLECTIONS

FRÜHER WAR NICHT ALLES BESSER

Es mag etwas verwundern, dass die Concurrency Utilities eine ganze Reihe an neuen Implementierungen der Collection Framework Interfaces beinhalten, da die Klasse `java.util.Collections` durch die sogenannten "Synchronized Wrapper" in der Lage ist, alle vorhandenen Collection Klassen "thread-safe" zu machen.

Der entscheidende Unterschied ist, dass die genannten "Synchronized Wrapper" ihre Thread-Safety wie der Name schon sagt durch Synchronisierung aller Zugriffsmethoden erreichen. Dieses grob granulare Sperren der ganzen Collection ist zwar "thread-safe" skaliert jedoch ausgesprochen schlecht insbesondere im Hinblick auf die wachsende Zahl der verfügbaren Multicore Prozessoren.

Hinzu kommt, dass sehr häufig noch Synchronisierung durch den Client erforderlich ist, so zum Beispiel bei mehrschrittigen Operationen wie "putIfAbsent" oder auch beim Iterieren über eine Collection, wie die unten stehenden Codebeispiele zeigen.

Das erste Beispiel zeigt eine synchronisierte Liste, über die iteriert wird. Da die Liste während des Iterierens nicht synchronisiert ist, ist es möglich, dass eine `ConcurrentModificationException` geworfen wird.

```
List<String> ls = Collections.synchronizedList(new
ArrayList<String>());
...
// Verhalten kann undeterministisch sein
// Kann ConcurrentModificationException werfen
Iterator<String> i = ls.iterator();
while (i.hasNext()) {
   foo(i.next());
}
...
```

Beispiel 1: Überraschung

Das zweite Beispiel zeigt, wie die Liste während des Iterierens korrekt zu synchronisieren ist, verdeutlicht aber auch sehr anschaulich den entscheidenden Nachteil, nämlich dass die Liste während des Iterierens für andere Zugriffe blockiert ist.

```
List<String> ls = Collections.synchronizedList(new
ArrayList<String>());
...
// Client synchronisieren
// Blockiert anderen Zugriff während Iteration
synchronized(ls) {
   Iterator<String> i = ls.iterator();
   while (i.hasNext()) {
    foo(i.next());
   }
}
...
```

Beispiel 2: Gut Ding will Weile haben

AUS ALT MACH NEU

Im Gegensatz dazu sind die neuen Implemetierungen der klassischen Datenstrukturen nicht nur "thread-safe", sondern auch "concurrent". Das bedeutet, sie erlauben einen nicht blockierenden, und damit skalierbaren, parallelen Zugriff. Darüberhinaus wirft kein Iterator der neuen Concurrent Collections eine `ConcurrentModificationException` und benötigt entsprechend auch keine externe Synchronisierung.

Erkauft wird dieses Verhalten durch leichte semantische Anpassungen der neuen Iteratoren. Sprach man bisher von sogenannten "fail-fast" Iteratoren, werden die neuen als "weakly consistent" bezeichnet. Dies bedeutet, dass sie Veränderungen, die nach ihrer Erzeugung an der zugehörigen Collection vorgenommen wurden, berücksichtigen können aber nicht müssen.

Mit `ConcurrentHashMap` steht eine Map Implementierung zur Verfügung, die mittels eines fein granularen Mechanismus names "Lock Striping" ein paralleles Lesen und Schreiben aus mehreren Threads ermöglicht. Durch Methoden wie `putIfAbsent()` oder `replace()` sind nun auch die gängigsten mehrschrittigen Operationen atomar vorhanden.

Als "concurrent" Ersatz für synchronisierte Listen beziehungsweise Sets dienen `CopyOnWriteArrayList` und `CopyOnWriteArraySet`. Diese Implementierungen basieren auf `ArrayList` und erzeugen bei jeder Änderung ihrer Daten eine Kopie des zu grunde liegenden Arrays. Dementsprechend ist der Einsatz dieser Klassen nur dann sinnvoll, wenn die lesenden die schreibenden Zugriffe deutlich überbieten.

Neben Implementierungen der bekannten Collection Interfaces beinhaltet Java 5 auch ein neues Interface namens `Queue`, welches eine Datenstruktur darstellt, die Elemente zur Verarbeitung bereit hält. Ein typisches Anwendungsszenario hierfür sind Erzeuger-Verbraucher-Probleme und dementsprechend besitzt `Queue` ein Subinterface namens `BlockingQueue`. Letzteres stellt zusätzlich Methoden bereit, die beim Schreiben in eine volle Queue beziehungsweise beim Lesen aus einer leeren Queue blockieren. Für die verschiedenen Implementierungen sei auf die Java 5 API Dokumentation [8] verwiesen.

Die Entwicklung der Concurrent Collections ist mit Java 5 übrigens nicht abgeschlossen. So beinhaltet Java 6 gerade in diesem Bereich weitere Neuerungen [9], etwa "concurrent" Implementierungen von `SortedSet` und `SortedMap`, aber auch neue Interfaces namens `Deque` und `BlockingDeque`. Der Name Deque ist abgeleitet von "Double Ended Queue" und zur Verwendung in sogenannten "Work Stealing" Architekturen gedacht [7].

EXECUTOR FRAMEWORK

THREADPOOLS

Logische Arbeitsblöcke werden als Tasks bezeichnet, die ihrerseits asynchron durch Threads ausgeführt werden können. Die beiden möglichen Extreme bei der Umsetzung, die streng sequentielle Ausführung aller Tasks durch einen einzigen Thread und das Verwenden eines eigenen Threads pro Task, sind beide unbefriedigend. Während der erste Ansatz schlechte Antwortzeiten und niedrigen Durchsatz zur Folge hat, erzeugt der zweite unnötigen Overhead durch die Threaderzeugung und läuft Gefahr, bei unbegrenzter Threadzahl jenseits seiner Leistungsfähigkeit betrieben zu werden.

Eine bessere Lösung ist ein sogenannter Threadpool, der eine bestimmte Anzahl Threads vorrätig hält und diese benutzt, um anfallende Tasks ausführen zu können. Bei sehr hoher Last kann der Threadpool zudem reagieren indem er die Threadzahl erhöht oder aber auch im Überlastfall Tasks abweist.

Die Concurrency Utilities beinhalten eine genau solche Threadpool Implementierung namens Executor Framework. Eine sehr wesentliche Abstraktion innerhalb dieses Frameworks ist die Trennung von Tasks und deren tatsächlicher Ausführung.

Tasks werden wie bisher auch durch Implemtierung des Interfaces `Runnable` beschrieben. Hierbei wird jedoch keinerlei Aussage getätigt, wann oder wie der Task ausgeführt wird (task execution). Neu ist hingegen das Interface `Executor`, welches eine Schnittstelle beschreibt, um Tasks zur Ausführung entgegen nehmen zu können (task submission).

```
public interface Runnable {
   void run();
}
```

Beispiel 3: Runnable Interface

```
public interface Executor {
   void execute(Runnable command);
}
```

Beispiel 4: Executor Interface

Die konkrete Art der Ausführung eines übergebenen Tasks hängt von der jeweiligen `Executor` Implementierung ab. Diese spezifiziert und kapselt die sogenannte "Execution Policy" und bestimmt die wichtigsten Aspekte der Taskausführung, beispielsweise wieviele Tasks parallel ausgeführt werden und in welcher Reihenfolge sie zur Ausführung kommen.

Das Ändern einer "Execution Policy" ist somit nichts weiter als das Verwenden einer anderen `Executor` Implementierung und kann auch sehr leicht konfigurativ erfolgen. Der nachfolgende Code, der nichts anderes darstellt als eine hardcodierte "Ein Thread pro Task Policy", sollte dementsprechend in Zukunft besser durch einen flexibleren `Executor` Aufruf wie unten gezeigt ersetzt werden.

```
...
new Thread(myRunnable).start();
...
```

Beispiel 5: Hard codiertes Starten eines neuen Threads für einen Task

```
...
myExecutor.execute(myRunnable);
...
```

Beispiel 6: Executor Aufruf

Abschliessend sei eine mögliche Implementierung einer "Ein Thread pro Task Policy" mittels eines `Executor` gezeigt, um das bisherige Ausführungsverhalten beizubehalten.

```
public class NewThreadExecutor implements Executor {
   public void execute(Runnable r) {
      new Thread(r).start();
   }
}
```

Beispiel 7: Executor, der einen neuen Thread pro Task startet

STETS ZU DIENSTEN

Neben der Erzeugung eines `Executor` wäre auch die Möglichkeit wünschenswert, einen bereits laufenden zu beenden. Da ein `Executor` letztlich für die meisten Anwendungen einen Dienst zur Verfügung stellt, sollte er sowohl kontrolliert als auch abrupt beendet werden können und dabei auch Rückmeldung geben, welche Tasks durch das Herunterfahren wie betroffen wurden.

Um diesen Lebenszyklus abbilden zu können existiert ein weiteres Interface namens `ExecutorService`, bei dem es sich um ein Subinterface von `Executor` handelt. Dieser `ExecutorService` ermöglicht es dem Entwickler, über Methoden wie `shutdown()`, `shutdownNow()` oder `awaitTermination()` Zustandswechsel an einem laufenden Threadpool vorzunehmen.

Obwohl es, wie zuvor gezeigt, möglich ist selbst `Executor` beziehungsweise `ExecutorService` Implementierungen zu schreiben, empfiehlt es sich auf die bereits in den Concurrency Utilities enthaltenen Klassen zurückzugreifen. Hiermit sind vor allem die bereits vorkonfigurierten Threadpools gemeint, die bequem über die entsprechenden Factory Methoden der Utility Klasse `Executors` erzeugt werden können.

Die folgenden Factory Methoden stehen dabei zur Verfügung:

- `newFixedThreadPool(int nThreads)`

  Threadpool fester Größe, der unerwartet beendete Threads ersetzt.

- `newCachedThreadPool()`

  Threadpool unbegrenzter Größe, der nach Bedarf wachsen und schrumpfen kann.

- `newSingleThreadExecutor()`

  Ein einzelner Arbeitsthread, der übergebene Tasks sequentiell abarbeitet.

- `newScheduledThreadPool(int corePoolSize)`

  Threadpool fester Größe, der verzögerte und periodische Ausführung unterstützt (`java.util.Timer` Ersatz).

```
...
final int NTHREADS = 10;
ExecutorService exec =
Executors.newFixedThreadPool(NTHREADS);
...
```

Beispiel 8: ExecutorService Erzeugung mittels Executors Klasse

Darüberhinaus stellt die Klasse `ThreadPoolExecutor` einen fertigen und fein granular konfigurierbaren `ExecutorService` zur Verfügung, der bei Bedarf auch als Basisklasse für eigene Implementierungen verwendet werden kann.

ALWAYS IN MOTION IS THE FUTURE (YODA, JEDI MASTER)

Die bisher verwendete Repräsentation eines Tasks ist das `Runnable` Interface. Dieses Interface ist zwar schon sehr lange gebräuchlich, besitzt aber zwei entscheidende Nachteile. So kann `Runnable` weder einen Ergebniswert zurück geben, noch eine "Checked Exception" werfen, lediglich das Auslösen von Seiteneffekten ist möglich, etwa Ergebnisse in geteilten Datenstrukturen abzulegen.

Dementsprechend bietet das Executor Framework eine neue Task Abstraktion namens `Callable`, welche die beiden genannten Mängel beseitigt. Es handelt sich hierbei um ein generisches Interface, wobei der Typparameter dem Ergebnistyp des Tasks entspricht.

```
public interface Callable<V> {
   V call() throws Exception;
}
```

Beispiel 9: Callable Interface

Die durch `Callable` beschriebenen Tasks werden üblicherweise asynchron ausgeführt und benötigen zudem eine nicht näher bekannte Zeit zur Ausführung. Somit stellt sich die Frage, wie der Aufrufer über die Beendigung seines Tasks informiert werden kann, beziehungsweise ob und wie er in der Lage ist, einen sehr lange laufenden Task zu unterbrechen.

Für genau diese Fragestellungen existiert ein weiteres Interface namens `Future`, welches das Ergebnis einer asynchronen Berechnung kapselt und zugleich den Lebenszyklus eines Tasks abbildet. Ein Task ist entweder noch nicht gestartet, gerade in Ausführung oder aber bereits beendet. Entsprechend bietet das `Future` Interface nicht nur Methoden um das Taskergebnis abzufragen, sondern auch um den zugehörigen Task zu unterbrechen, beziehungsweise dessen Status abzufragen.

```
public interface Future<V> {
   V get() throws InterruptedException, ExecutionException;
   V get(long timeout, TimeUnit unit) throws
      InterruptedException, ExecutionException,
      TimeoutException;
   boolean cancel(boolean mayInterruptIfRunning);
   boolean isCancelled();
   boolean isDone();
}
```

Beispiel 10: Future Interface

Das Verhalten der `get()` Methode ist wiederum abhängig vom Zustand, in dem sich der zugehörige Task gerade befindet. Ist der Task bereits beendet wird das Ergebnis sofort zurück gegeben anderenfalls blockiert der Aufruf von `get()` bis zum Abschluss der jeweiligen Berechnung.

Der einfachste Weg, ein `Future` Objekt zu erhalten, ist eine der `submit()` Methoden zur Taskaufgabe an einem `ExecutorService` aufzurufen. Diese `submit()` Methoden stehen sowohl für `Callable` als auch `Runnable` Objekte zur Verfügung.

```
...
final int NTHREADS = 10;
ExecutorService exec =
Executors.newFixedThreadPool(NTHREADS);
...
Callable<Integer> task = new MyCallable<Integer>();
Future<Integer> future = exec.submit(task);
try {
   Integer result = future.get();
} catch (InterruptedException e) {
   ...
} catch (ExecutionException e) {
   ...
}
...
```

Beispiel 11: Future Objekt Erzeugung durch Aufruf von submit

Ansonsten kann auch die Klasse `FutureTask`, welche `Future` implementiert, für ein vorhandenes `Callable` oder `Runnable` Objekt explizit instanziiert werden. Da `FutureTask` diverse `protected` Methoden zur Verfügung stellt, kann diese Klasse auch als Basisklasse für eigene Implementierungen genutzt werden.

Der scheinbare Dualismus zwischen `Callable` und `Runnable` wird durch entsprechende Wandlungsmöglichkeiten entschärft. So läßt sich aus jedem `Runnable` sehr leicht ein `Callable` erzeugen, indem eine der `callable()` Wandlungsmethoden der Utility Klasse `Executors` aufgerufen wird. Umgekehrt kann ein `Callable` mittels der gerade erwähnten Klasse `FutureTask` in ein `Runnable` überführt werden, da `FutureTask` ebenfalls das `Runnable` Interface implementiert und somit als entsprechender Adapter dienen kann.

FAZIT

Die seit Java 5 verfügbaren und in Java 6 erweiterten Concurrency Utilities bringen eine deutlich spürbare Arbeitserleichterung bei der Entwicklung von Multithreaded Anwendungen. Viele Problemstellungen, die zuvor nur mühsam oder schwierig zu bearbeiten waren, lassen sich nun einfach mittels einheitlich spezifizierter Klassen beziehungsweise Frameworks lösen.

Durch den momentanen Hardwaretrend in Richtung Multicore Prozessoren wird sich auch die Softwareentwicklung in Zukunft mehr als früher mit dem Thema "Concurrent Software" auseinandersetzen müssen. Dementsprechend sind die Concurrency Utilities eine Klassenbibliothek, die über kurz oder lang zum Rüstzeug eines jeden Java Entwicklers gehören werden.

REFERENZEN

- [1] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
  (http://www.gotw.ca/publications/concurrency-ddj.htm)
- [2] About the Java Technology
  (http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html)
- [3] Java Tuning White Paper
  (http://java.sun.com/performance/reference/whitepapers/tuning.html)
- [4] Concurrent Programming in Java
  Lea, Doug
  (http://gee.cs.oswego.edu/dl/cpj/)
- [5] JSR 166 Homepage
  (http://www.jcp.org/en/jsr/detail?id=166)
- [6] JDK 5.0 Concurrency-related APIs & Developer Guides
  (http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html)
- [7] Java Concurrency In Practice
  Goetz, Brian
  (http://www.javaconcurrencyinpractice.com/)
- [8] Java Platform Standard Edition 5.0 API Specification
  (http://java.sun.com/j2se/1.5.0/docs/api/index.html)
- [9] Collections Framework Change Summary for Java SE 6
  (http://java.sun.com/javase/6/docs/technotes/guides/collections/changes6.html)
- [10] Moore's Law: Made real by Intel innovation
  (http://www.intel.com/technology/mooreslaw/index.htm)