Samenvatting Java Advanced

# Generieken

Generieke klassen zijn klassen die een datatype gebruiken dat aanvankelijk onbepaald is, maar deze worden concreet ingevuld op het moment dat de klasse geïnstantieerd wordt. De lijst van type-parameters voegen we achteraan tussen driehoekige haakjes toe. Het voordeel van generieken is dat we niet meer hoeven te casten.

public class KLASSENAAM<E> {  
 // CODE  
}

Het is niet nodig om de type-parameter ook op te geven bij de instantiatie.

// Overbodige parameter bij instantiatie  
Duo<String> sd = new **Duo<String>**();

// Correcter gebruik  
Duo<String> sd = new **Duo<>**();

Een klasse heeft een onbeperkt aantal types.

public class KLASSENAAM<E, T> {  
 // CODE  
}

## Inperking

We kunnen ook zeggen dat het type van de klasse zelf of een subklasse moet zijn.

<E extends SUPERKLASSE>

We kunnen om het even welke klasse toelaten door het jokenteken (?) of de wildcard te gebruiken.

We kunnen het ook inperken doormiddel van over te erven van een klasse en een interface, dit kan door het ampersand (&).

Het is ook mogelijk ervoor te zorgen dat een generieke alleen haar klasse en superklassen accepteert. Maar niet haar subklassen. Dit kan door super.

<E super KLASSE>

## Subklassen

We kunnen een subklasse van een generieke klassen maken. Dit doen we als we code van de generieke klasse willen overnemen.

public class KLASSENAAM extends SUPERKLASSE<E> {  
 // CODE  
}

Het is ook mogelijk het datatype over te nemen, dan blijft de subklasse ook een generieke klasse.

public class KLASSENAAM<E> extends SUPERKLASSE<E> {  
 // CODE  
}

Het is ook mogelijk een generieke interface te implementeren.

public class KLASSENAAM implements INTERFACENAAM<E> {  
 // CODE  
}

Een generieke klassen kan ook een generieke interface interpreteren.

public class KLASSENAAM<E> implements INTERFACENAAM<E> {  
 // CODE  
}

Als we objecten willen opslaan die met elkaar vergeleken kunnen worden, dan dienen we de interface Comparable te implementeren.

## Generieke methoden

De klasse van een generieke methode moet niet perse zelf generiek zijn.

public static void METHODENAAM(GENERIEKEKLASSE<E> var) {  
 // CODE  
}

Indien we gebruik willen maken van het type van de generieke klasse, moeten we dit als volgt definiëren.

public static **<E>** void METHODENAAM(GENERIEKEKLASSE<E> var) {  
 // CODE  
}

In dit voorbeeld kunnen we vooraan het type inperken en moeten we dit niet dubbel schrijven bij meerdere parameters.

## Achter de schermen

1. Er bestaat slechts 1 klassenbestand van een generieke klasse. Dit klassenbestand wordt gedeeld door alle instanties van de klasse. Dit impliceert tevens dat het niet toegelaten is de typeparameter van de klasse te gebruiken in statische members.
2. In het gecompileerde klassenbestand is er geen spoor te vinden van generieke datatypes. Tijdens de compilatie doet de compiler de nodige controles en vertaalt hij de nodige datatypes. Nadien worden alle sporen van de generieken uitgewist. Het gebruik van generieken is dus louter een zaak van de compiler en heeft niet in het minst betrekking op de virtuele machine.
3. Casten is overbodig en werkt het basisprincipe van generieken tegen.
4. De operator instanceof kan niet gebruikt worden om te testen of een bepaald object van een bepaalde klasse is. Deze controle gebeurd namelijk at runtime en op dat moment is er geen verdere informatie meer beschikbaar over het specifieke datatype.

## Arrays

Het juiste datatype kan nooit gegarandeerd worden bij arrays. Daarom is het gebruik van specifieke types in reeksen niet toegestaan.

## Oudere code

Men kan een object maken van een generieke klasse, dit wordt door de compiler beschouwd als een “raw type”. De compiler genereerd een waarschuwing.

# Geneste en anonieme klassen

## UML

Een geneste klasse wordt in een UML-schema zoals hieronder weergegeven.



De volledige klassenaam is outerClass.innerClass. Het klassenbestand zal dit bevatten: OuterClass$InnerClass.class

## Gewone geneste klassen

outerClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.new InnerClass();

We gebruiken hier de new-operator op bases van een reeds bestaand object van de nestende klasse. De geneste klasse heeft toegang tot alle members van de nestende klasse, ook de private members. Dit is meestal de reden waarom men gebruik maakt van geneste klassen.

Om een verwijzing naar het object van de nestende klasse te krijgen, gebruikt men OuterClass.this.

## Local Inner Classes

Deze zijn binnen een methode gedefinieerd. De objecten van de lokale geneste klassen hebben enkel toegang tot de constante final variabelen.

public class OuterClass {  
 public void method() {  
 final int CONST = 5;  
 int var = 7;  
 var = 8;  
 class LocalInnerClass {  
 // CODE  
 }  
 LocalInnerClass inner = new LocalInnerClass();  
 }  
}

## Anonymous Inner Classes

Een anonieme klasse is een subklasse van een bepaalde superklasse of een klasse die een bepaalde interface implementeert. Het is de bedoeling dat de anonieme klasse methoden van de superklasse of interface vervangt of implementeert. Men kan nieuwe methoden toevoegen maar die zijn ontoegankelijk. Tevens kan men bij anonieme klassen enkel gebruikmaken van de standaard constructor zonder parameters.

## Static Nested Classes

public class OuterClass {  
 public static class InnerClass {  
 // CODE  
 }  
}

Opsommingstypes (enums) worden soms gedefinieerd binnen een bestaande klasse. Dit is evenwaardig aan een static nested class. Dan mag static weggelaten worden.

Een veel gebruikte toepassing is de iterator. Een iterator is een object dat het mogelijk maakt een aantal objecten in een verzameling te overlopen.

# Lambda expressies

Als we een functionele interface maken, kunnen we beroep doen op de aanroeper van de methode om ons het juiste criterium te geven. Dit is een callback method. Een functionele interface is een interface met slechts één methode. We kunnen boven deze interface de annotatie “@FunctionalInterface” zetten om dit duidelijk te maken. Dus ze mogen maar 1 abstracte methode hebben maar mogen wel meerdere default en static methoden hebben.

Een lambda expressie is een uitdrukking die op een zeer compacte wijze een object maakt van een anonieme geneste klasse die een functionele interface implementeert. Een voorbeeld hiervan is: “(String s) -> s.Contains(“e”)”. Eerst hebben we dus een lijst van formele parameters. Als er 1 parameter is, kan het type en de haken weggelaten worden. Na het pijltje komt de inhoud of de body. Dit ‘doet’ iets met de parameters.

De gebruikte variabelen moeten uniek zijn, ze vallen namelijk binnen dezelfde scope als de omgevende code. Er wordt geen nieuwe scope gemaakt dus ‘this’ en ‘super’ hebben dezelfde betekenis.

Reference by method is een referentie doorgeven naar de methode of constructor. Een voorbeeld hiervan is “TextUtil::quote”. De syntax is dus qualifier::identifier.

|  |  |
| --- | --- |
| Soort referentie | Syntax |
| Statische methodereferentie | className::staticMethodName |
| Gebonden methodereferentie | objectName::methodName |
| Ongebonden methodereferentie | parameterType::methodName |
| Constructorreferentie | className::new |

## Predicate

De interface Predicate<T> is een generieke interface met abstracte methode: public boolean test(T t) De interface heeft ook nog and, or en negate als default-methoden.

De interface Function<T, R> is een generieke interface met abstracte methode: public R apply(T t) De interface heeft ook nog andThen en compose als default-methoden.

De interface Consumer<T> is een generieke interface met abstracte methode: public void accept(T t) De interface heeft ook nog andThen als default-methode.

# Streaming API

Één manier om elementen te overlopen is doormiddel van een lus of iterator, dit is een externe interatie. Dit is ook mogelijk met een stream, dit is een interne iteratie. We kunnen een stream maken door: “Stream.of(lijst).

Een stream is een stroom van objecten en worden voorgesteld door de interface BaseStream. Deze interface heeft de volgende afgeleide interfaces: Stream<T>, IntStream, LongStream en DoubleStream. Mogelijke bronnen voor een stream zijn een verzameling van elementen, een generator of een I/O-kanaal.

De interface Stream beschikt over generate() waarbij we oneindige lussen kunnen genereren zoals “Stream.generate(() -> “Hallo!”). De andere interface hebben enkele handige methodes zoals range() en rangeClosed(). Met de methode iterate() kunnen we een stream van getallen maken waarbij we een initeel getal en een bewerking meegeven die we herhaaldelijk toepassen zoals “IntStream.iterate(0, i -> i + 1)”.

Tussenliggende bewerkingen resulteren in een nieuwe stream terwijl afsluitende bewerkingen resulteren in een of andere eindresultaat. Eindebewerkingen kunnen in de volgende dingen resulteren: Niets (consumerend), één enkel nieuw gegeven (reducerend) of een nieuwe verzameling (collecterend). Bewerkingen die geen resultaat geven, zijn consumerende bewerkingen zoals forEach().

De stroom van gegevens kan gereduceerd worden door sum() of max(). Bij max kan het zijn dat er geen maximum is omdat de stream geen elementen bevat. Daarom geeft deze een OptionalInt terug. We moeten eerst een if-check doen om te kijken of max.isPresent() is.

Streams geven ook de mogelijkheid om een eigen reductiemethode te maken met reduce(). Deze heeft een startwaarde en een accumulatiefunctie nodig. Een voorbeeld hiervan is: “stream.reduce(“\*”, (acc, el) -> acc + el + “\*”).

We kunnen ook een nieuwe verzameling creëren door collect() op te roepen. Dit kan zijn om een stream naar een String-array te doen zoals “.toArray(String[]::new)”.

Tussenliggende bewerkingen maken een nieuwe stream van elementen. Een voorbeeld hiervan zijn filters. Zij houden elementen tegen die niet aan het criterium voldoen, dit kan met de methode filter(). Omdat we verschillende filter-functies aaneen kunnen rijgen, spreken we van aggregatiefuncties. Er zijn ook enkele ingebouwde filter-functies die duplicaten filtert (distinct()), het aantal elementen beperkt (limit()) of een aantal elementen overslaat (skip()).

We kunnen ook een omzetting doen. Dit is het aanmaken van een nieuwe stream doormiddel van de bewerkte elementen. Dit kan van hetzelfde of een ander datatype zijn. Enkele methodes zijn: mapToInt(), mapToLong(), mapToDouble() en map().

We kunnen ook elementen sorteren (nieuwe stream aanmaken) door middel van sorted().

# Collections

Een collection is handig om elementen toe te voegen, te verwijderen, te controleren of een element aanwezig is, op te vragen of te overlopen. Het Collections Framework bestaat uit de klassen Collection en Map.

Een list is een geordende sequentiële verzameling waarbij duplicaten mogelijk zijn.

Een arraylist kan zijn grootte dynamisch aanpassen maar is niet snel (door lengte altijd aan te passen). Er zijn duplicaten toegelaten. Autoboxing zorgt voor het automatisch omzetten naar het juiste type. Deze bevat ook een methode toArray() om het terug naar een array om te zetten. We moeten hierbij een parameter meegeven. Als we een array met de juiste grootte meegeven, noemen we dit een anonieme array. Meestal doen we dit: “numbers.toArray(new Integer[0]).

Een linkedlist is een dubbel gekoppelde lijst. Ieder element verwijst naar het element ervoor en erachter. Dit maakt het mogelijk de lijst in 2 richtingen te doorlopen. Het is snel in toevoegen, invoegen en verwijderen op willekeurige plaatsen. Het is niet zo snel voor willekeurige toegang of iteratie.

Een set heeft alleen unieke elementen, er komen dus geen duplicaten voor. Als men een duplicaat toevoegt, zal de set het slechts éénmaal hebben. De elementen hebben geen vaste positie of index.

Een hashset is een ongeordende en ongesorteerde lijst. De volgorde ervan is niet gegarandeerd. Deze is zeer snel in opzoeken, toevoegen en verwijderen van elementen. Als men 2 StringBuilder-objecten met dezelfde inhoud toevoegt, zal het er 2 keer in staan omdat alleen de referenties en niet de inhoud wordt doorgegeven.

Een linkedhashset is een geordende en ongesorteerde lijst. Het is een dubbel gekoppelde lijst waarin de volgorde van de elementen bijgehouden kan worden. Het heeft snelle toegang, geen duplicaten en vaste volgorde.

Een sortedset is geordend en gesorteerd.

Een navigableset heeft de mogelijkheid het dichtst bijzijnde element op te vragen.

Een treeset is een geordende en gestructureerde verzameling waar automatisch of live gesorteerd wordt. Het is niet zo snel.

Een queue dient om elementen één voor één op te vragen te verwijderen.

Een linkedlist is een queue die volgens het FIFO-principe werkt. peek() zorgt ervoor dat je kan kijken, terwijl poll() het ook echt verwijderd.

Een priorityqueue sorteerd volgens natuurlijke volgorde of een comparator.

Een deque is een double ended queue die aan twee zijden benaderd kan worden.

Een intrinsieke sortering is een verzameling die zelf zorgt voor zijn sortering. Dit kan via natuurlijke volgorde of een comparator (“Comparator.comparingInt(b -> b.getWidth())”). Dit kan uitgebreid worden door er “.thenComparingInt(b -> b.getHeight())” aan toe te voegen. We kunnen ook gewoon Comparator.naturalOrder() aanroepen.

Een map is een verzameling met sleutels en waarden. Dit is een ongeordende en ongesorteerde map, null-waarden zijn eenmaal als sleutel toegestaan en meermaals als waarde. Een map heef geen vaste volgorde waarbij sleutels uniek moeten zijn.

Een linkedhashmap is een geordende ongesorteerde map die dezelfde volgorde hebben zoals ze zijn toegevoegd.

Een sortedmap is een geordende en gesorteerde map.

Een navigablemap maakt het mogelijk om de dichtst bijzijnde sleutel op te vragen.

Een treemap is een sortedmap en een navigablemap die live gesorteerd wordt.

# Lezen en Schrijven (I/O)

TODO

# Commandolijn

Er bestaan programma’s om code te compileren, documentatie te genereren etc. Deze bevinden zich in de map ‘bin’. Er is een omgevingsvariabele ‘JAVA\_HOME’ die naar de plaats van de geïnstalleerde JDK wijst.

Compileren van code wordt met het programma javac gedaan. De map src bevat de gecompileerde bronbestanden. We kunnen dit dus compileren door: ‘javac -d bin src\eu\...\\*.java’. Alle .java bestanden worden hierdoor .class.

We kunnen een programma uitvoeren door java.exe, dit start een VM op. Dit kan door: ‘java -cp bin eu.App’.

JAR-bestanden zijn Java ARchive bestanden met het ZIP-formaat die een aantal voordelen hebben:

* Beveiliging (digitale handtekening)
* Snelheid (downloaden van 1 bestand)
* Compressie (minder ruimte innemen)
* Verzegeling (juiste versies – alles uit 1 pakket)
* Overdraagbaarheid (standaardonderdeel van Java – API’s beschikbaar)

Met de Java Archive Tool kunnen we enkele dingen doen:

* Een JAR-bestand maken jar cf jarfile files
* Bestanden toevoegen jar uf jarfile files
* Inhoud bekijken jar tf jarfile
* Inhoud extraheren jar xf jarfile
* Specifieke bestanden extraheren jar xf jarfile files

Als we JAR-bestanden als applicatie willen gebruiken, moeten we informatie over de hoofdklasse toevoegen. Dit gebeurt in het manifest-bestand.

We kunnen bestanden lezen via 2 methoden, ‘getClass().getResourceAsStream()’ en ‘getClass().getResource()’.

# Systeembronnen

Toegang tot systeembronnen wordt geregeld door de Java Runtime Environment die aangeboden wordt via de klassen System en Runtime. De System-klasse is platformONafhankelijk. Runtime daarintegen niet.

Van de System-klasse kunnen geen objecten of subklassen gemaakt worden. Deze is final en heeft alleen statische methoden.

De standaard-inputstream is toegankelijk via ‘System.in’. Dit is een byte stream. InputStreamReader is een processing stream die dit kan omzetten naar een char stream. Als we volledige strings willen, gebruiken we een BufferedReader. Dit kunnen we doen door ‘

InputStreamReader in = new InputStreamReader(System.in);  
BufferedReader keyboard = new BufferedReader(in);

‘. Dit kan eenvoudiger door een nieuw Scanner-object aan te maken.

De standaard-outputstream is toegankelijk via ‘System.out’. Deze heeft als methoden, print(), println(), printf(), format() etc.

We kunnen de standaard-streams vervangen door ‘setIn()’, ‘setOut()’ en ‘setErr()’. Om de errors naar een log te schrijven, kunnen we het volgende doen: ‘

PrintStream logstream = new PrintStream(new FileOutputStream(“logfile.txt”, true));  
System.setErr(logstream);

‘. De klasse Console kan opgevraagd worden door ‘System.console()’. We moeten altijd nakijken of deze niet null is. Deze klasse heeft de volgende methoden: ‘readLine()’, ‘readPassword()’, ‘printf()’ en ‘format()’.

Indien je een programma opstart vanaf een IDE, is de Console doorgaans niet beschikbaar omdat een IDE de inputstream en outputstream van het programma omleidt naar zijn eigen vensters.

In de System-klasse zijn ook enkele properties beschikbaar:

* Scheidingsteken file.separator
* Java-classpath java.class.path
* Java-class-versienummer java.class.version
* Java installatiemap java.home
* Java leverancier java.vendor
* URL van leverancier java.vendor.url
* Java-versienummer java.version
* Lijnscheidingsteken line.separator
* Architectuur OS os.arch
* Naam OS os.name
* Versie OS os.version
* Padscheidingsteken path.separator
* Huidige map user.dir
* Home map user.home
* Gebruikersnaam user.name

We kunnen deze properties ophalen door de methoden ‘getProperties()’ of ‘getProperty()’. Hier bestaan ook setters van. We kunnen een property veranderen door ‘java -Duser.name=Jan MyApp’.

Er wordt altijd de methode ‘finalize()’ aangeroepen voor een object opgeruimd wordt. We kunnen deze overriden maar dan is het wel belangrijk dat we daarna de superklasse opnieuw aanroepen door ‘super.finalize()’.

We kunnen een vriendelijk verzoek aan de Garbage Collector doen door ‘System.gc()’.

De System-klasse beschikt nog over enkele andere methoden:

* Kopiëren van reeks naar reeks arraycopy()
* Huidige tijd in milliseconden currentTimeMillis()
* Huidige tijd in nanoseconden nanoTime()
* Beëindigt de Java Runtime Environment exit()
* Laadt een externe module in loadLibrary()

# Multithreading

Bij multiprocessing hebben we meerdere programma’s die onafhankelijk naast elkaar worden uitgevoerd. Bij multithreading gaat het om meerdere threads binnen hetzelfde programma of process. Deze threads horen tot hetzelfde programma en delen dezelfde geheugenruimte. Voor multithreading moeten er dus speciale voorzieningen getroffen worden.

We kunnen een nieuwe thread maken door ‘Thread thread = new Thread();’ en deze kunnen we starten door ‘start()’. De code die dan wordt uitgevoerd, is de code die in ‘run()’ staat van de interface Runnable. Standaard is deze leeg. We kunnen dus ofwel een subklasse maken van Thread of de interface Runnable implementeren.

We onthouden dat onze main() ook altijd een thread is dus als we 1 thread aanmaken, zijn er 2 threads actief (main() en extra thread).

Een subklasse van een thread maken we door ‘extends Thread’ toe te voegen en ‘@Override public void run() {}’ toe te voegen.

We kunnen ook de interface runnable implementeren door ‘implements Runnable’ en dan de methode ‘public void run() {}’ toe te voegen. Dan moeten we in het hoofdprogramma een nieuw object van die klasse maken. Daarna moeten we een nieuwe thread maken met als parameter de instantie van die klasse. Daarna kunnen we de thread starten.

We kunnen ook een thread maken zonder een tussenklasse. Dit kan door de implementatie van run() bij het aanmaken van de thread al te definiëren. Dit kan door ‘Thread thread = new Thread(() -> print(100));’. De methode print staat in de main().

Zodra een thread gemaakt is door middel van zijn constructor, bevindt hij zich in de NEW toestand. Als je de thread start(), kan hij in 2 toestanden zijn: RUNNING of READY. Wanneer een thread klaar is, komt hij in de toestand TERMINATED. Deze kan niet opnieuw opgestart worden. Met ‘isAlive()’ of ‘getState()’ kunnen we de toestand controleren. getState geeft een waarde van de Enum State terug (NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING of TERMINATED).

De scheduler bepaald welke thread actief is. Dit is een afzonderlijke taak die zich enkel bezig houdt met de taakverdeling van de processor. Degene met de hoogste prioriteit wordt in RUNNING gezet. Anders is dit willekeurig. Een nieuwe thread erft de prioriteit van zijn parent. Met setPriority() kunnen we zelf een prioriteit zetten (MIN\_PRIORITY, NORM\_PRIORITY of MAX\_PRIORITY).

Preëmptieve multitasking is dat een thread op elk moment onderbroken kan worden door een thread met een hogere prioriteit.

Coöperatieve multitasking is dat andere moeten wachten tot de huidige threads gedaan zijn. Dit kan leiden dat andere threads nooit aan bod komen. We kunnen dan in onze applicatie ‘Thread.yield()’ oproepen om andere threads ook een kans te geven. Sommige systemen gebruiken ook time slicing waar na een bepaalde periode, een thread onderbroken wordt.

Men kan het aantal cores opvragen door: ‘Runtime.getRuntime().availableProcessors()’.

Een deamon thread is een thread die niet beëindigd hoeft te zijn opdat de volledige applicatie zou kunnen afsluiten. Dit kunnen we doen door ‘setDeamon(true)’. Een applicatie kan door ‘System.exit()’ afgesloten worden en dan worden alle threads onvoorwaardelijk beëindigd.

Threads kunnen ook in de wachttoestand gezet worden. Dit betekent dat ze uit RUNNABLE worden gehaald. Enkel actieve threads die RUNNING zijn, kunnen naar wachttoestand gaan.

Een thread kan zichzelf in slaap brengen door ‘Thread.sleep()’. Dan komt deze in de toestand TIMED\_WAITING. Deze thread kan terug wakker gemaakt worden als de tijd verstreken is of door de methode ‘interrupt()’. Deze keert terug naar de READY-toestand en wordt ten vroegste na deze periode terug actief. Sleep() kan een uitzondering genereren (InterruptedException).

We kunnen een naam instellen door ‘setName()’ en ophalen door ‘Thread.currentThread().getName()’.

Een thread kan wachten tot een andere thread gedaan is. Dit kan met de methode ‘join()’ en komt dan in de toestand WAITING. Men kan ook een maximaal aantal seconden meegeven aan de join(). Deze komt dan niet in WAITING maar in TIMED\_WAITING.

Als threads dezelfde bronnen gebruiken, moeten we rekening houden met synchronisatie of object locking.We kunnen synchronised rond een reeks van bewerkingen zetten: ‘synchronised (monitor) { bewerkingen; }’. Elk object kan dienst doen als monitor. Het is belangrijk dat 2 zones dezelfde monitor hebben. Indien het slot reeds in gebruik is, moet de thread wachten tot het weer beschikbaar is. De thread komt dan in de toestand BLOCKED.

We kunnen ook het teller object als monitor gebruiken. We zetten dan gewoon ‘this’ in de plaats van monitor. We kunnen dit nog vereenvoudigen door aan de methode synchronised toe te voegen: ‘public synchronised void naam() {}’. Indien we synchronised bij een statische methode toepassen zal het monitor object de klasseninstantie zijn. Bij constructors is dit niet nodig. JVM beschermd deze automatisch al.

We gebruiken de Timer en TimerTask klassen om iets te doen op een bepaald tijdstip of interval. De kunnen een klasse CustomTask maken die extend van TimerTask. Dan kunnen we in onze main() hetvolgende doen:  
CustomTask task = new CustomTask();  
Timer timer = new Timer(true);  
timer.schedule(task, 1000 \* 10);

We kunnen het concurrency framework gebruiken om Java een deel van het synchroniseren af te handelen. We kunnen gebruik maken van de volgende methoden:

* synchronisedCollection
* synchronisedList
* synchronisedNavigableMap
* synchronisedNavigableSet
* synchronisedSet
* synchronisedSortedMap
* synchronisedSortedSet

We kunnen deze gebruiken alsvolgt:  
List<Integer> list = new ArrayList<>();  
List<Integer> slist = Collections.synchronisedList(list);

Automaire objecten vormen één geheel en mogen niet door een andere thread onderbroken worden. Er bestaan volgende klassen:

* AtomicBoolean
* AtomicInteger
* AtomicIntegerArray
* AtomicLong
* AtomicLongArray
* AtomicReference  
  AtomicReferenceArray

We kunnen een klasse PrimeCalculator maken die ‘implements Callable<List<Long>>’ doet. We kunnen dan een methode ‘public List<Long> call() { CODE }’ maken die priemgetallen berekend. We kunnen dan in onze main hetvolgende doen:  
PrimeCalculator pc = new PrimeCalculator(1000000);  
ExecutorService es = Executors.newSingleThreadExecutor();  
Future<List<Long>> future = es.submit(pc);

while (!future.isDone()) {  
 // WAITING  
}

List<Long> primes = future.get();  
es.shutdown();

Deze executor service is in staat een taak (Callable) uit te voeren met de methode submit(). Het resultaat van deze methode is van het type Future. Om de executor service af te sluiten gebruiken we shutdown().

We kunnen stream parallel laten lopen door ‘Collection.parallelStream(). Dit kan door de methode parallel(). We kunnen dan de .parallel() in onze stream zetten.