TAMPEREEN
AMMATTIKORKEAKOULU

SÄIKEISTÄMISEN PERUSTEET JAVALLA

Tuomo Martikainen

Opinnäytetyö
Kesäkuu 2018
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

MARTIKAINEN, TUOMO:
Säikeistämisen perusteet Javalla

Opinnäytetyö 69 sivua, joista liitteitä 51 sivua
Kesäkuu 2018

Opinnäytetyön toimeksiantaja oli Tampereen ammattikorkeakoulun ohjelmistotuotannon linja. Työn tavoitteena oli tuottaa verkkosivuille englanninkielistä itseopiskelumateriaalia ohjelmistotuotannon opiskelijoiden osaamisen laajentamiseksi. Prosessorien ytimien lukumäärän ja ohjelmistojen raskauden kasvaessa yhä useampi ohjelmoija joutuu perehtymään siihen, kuinka olemassa olevia resursseja pystytään hyödyntämään ohjelmistoissa paremmin ja turvallisemmin. Tämän vuoksi oli tarpeellista tarjota opiskelijoille helppo mahdollisuus aiheen opiskeluun.

Työhön tarvittavan tiedon hankinta toteutettiin kvalitatiivisena analyysinä. Itse materiaalin teossa hyödynnettiin konstruktiivista lähestymistapaa. Opiskelumateriaali käsittelee säikeistämisen perusteita Java-ohjelmointikielellä ja tarjoaa katsannon erilaisiin kielen tarjoamiin ohjelmointia helpottaviin työvälineisiin. Materiaali sisältää yksinkertaisia osittaisia ja kokonaisia koodiesimerkkejä helpottamaan selitetyn teorian testaamista käytännössä. Sisällön tehokas hyödyntäminen edellyttää Javan perusteiden ja hyvien ohjelmointikäytänteiden osaamisen.

Opinnäytetyön tuloksena tuotettiin verkkosivut, jotka mahdollistavat opiskelijoille aiheen helpon ja itsenäisen opiskelun. Sivujen sisältö on yhtenevä tämän opinnäytetyön liitteisiin lisätyn tekstiversion kanssa, mutta verkkosivujen sisältö on jaettu useammalle erilliselle sivulle ja muokattu paremmin verkkomateriaaliksi soveltuvaksi. Materiaali on englanninkielinen, koska se on tällä alalla yleisesti käytössä oleva kieli.

Opas on suunniteltua laajempi ja kattaa hyvin oleelliset säikeistämisen perusteet. Kaikki mahdollisesti tarpeellinen tieto ei kuitenkaan mahtunut oppaaseen mukaan, joten sisällön laajennus olisi tarpeen kattavamman kokonaiskuvan antamiseksi. Lisäksi useammat esimerkit ja harjoitusten lisääminen auttaisivat asian oppimisessa.

Asiasanat: säie, säikeistys, moniajo, samanaikaisuus, java

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

MARTIKAINEN, TUOMO:
Basics of Multithreading in Java

Bachelor's thesis 69 pages, appendices 51 pages
June 2018

This thesis was commissioned by the Option of Software Development from Tampere University of Applied Sciences. The aim of the thesis was to produce a web-based self-study material about multithreading for the students of Software Production. Lately, there has been a push to increase the core count of computer processors, which also leads to an increased demand for knowledgeable workforce. The self-study material was created to help resolve this issue by providing an easy way to study multithreading.

Qualitative analysis was used to gather information about the subject, and the implementation made use of the constructive approach. The material deals with the basics of multithreading in Java. Simple examples were used to help illustrate the problems of multithreading, and to give readers an easy way test the solutions themselves. Some of the examples are only partial, but others have the complete structure to allow for easier testing. Efficient use of the material requires comprehensive knowledge about the basics of Java and good programming practices.

As the result of the thesis, the self-study material is annexed to it in text form, as well as made available as a website. The content of the website is essentially the same as the one found in the appendix, but it has a different presentation, and may have updates and fixes applied to it. The material is written in English, as it is the default language used in the field.

The finished self-study material is more comprehensive than planned, encompassing the essential basics of multithreading. Unfortunately, the material is not extensive enough to deal with all the information on the subject. This could be mitigated by both extending the scope of the material, and adding exercises and more examples to it.

Key words: thread, threading, multiprocessing, concurrency, java

SISÄLLYS

ERITYISSANASTO tai LYHENTEET JA TERMIT (valitse jompikumpi)

Enkapsulointi	Muille luokille tarpeettomien muuttujien ja metodien piilottaminen (näkyvyys) näiden saatavilta. Ulkopuoliset luokat käsittelevät toisten luokkien sisäistä tilaa hallitusti julkisten metodien kautta. (encapsulation)
Samanaikainen suoritus	Useamman ohjelman sisäisin koodipolun (tai erillisen prosessin) suoritus samanaikaisesti. (concurrency)
Synkronointi	Suojakeino sen varmistamiseksi, että vain yksi säie pääsee kerrallaan käymään läpi samalla lukolla suojattuja koodipolkuja. (synchronization)
Säie	Prosessin (ohjelma) sisäinen koodia suorittava itsenäinen polku, joita voi olla useita. (thread)
Säikeistys	Ohjelmiston suorituspolun jakaminen useampaan mahdollisesti samanaikaisesti suoritettavaan osaan. (multithreading)

1 JOHDANTO

Prosessorien ytimien lukumäärän ja ohjelmistojen raskauden kasvaessa, ohjelmistojen tekijöillä on yhä enenemässä määrin tarvetta jakaa ohjelmien suoritusta useampien ytimien tehtäväksi. Toisin kuin useamman erillisen ohjelman yhtäaikainen suorittaminen eri ytimillä, yhden ohjelmiston kyseessä ollessa tämä ei ole automaattista. Tämän vuoksi ohjelmoijan täytyy nähdä vaivaa saadakseen ohjelmansa tukemaan useamman ytimen turvallista yhtäaikaista käyttöä. Teoriassa ohjelman säikeistäminen on helppo prosessi, mutta käytännössä se voi aiheuttaa arvaamattomia ja vaikeasti paikallistettavia ongelmia. Lisäksi ohjelman suorituskyvyn hyvä skaalautuvuus tarjolla olevien ytimien lukumäärän mukaan voi olla vaikeaa tai mahdotonta saavuttaa.

Koska useiden säikeiden turvallisesti ja tehokkaasti käyttäminen ilman osaamista ja harjoittelua on vaikeaa, opiskelijat tarvitsevat helposti saavutettavia lähteitä perusteiden opettelemiseen. Vaikka kirjat ovat tähän tarkoitukseen hyviä ja kattavia, on niissä oleva tietomäärä myös helposti laajuudeltaan tukahduttava. Tämä materiaali on tarkoitettu opiskelijoille helposti lähestyttäväksi, useampaan eri lähteeseen perustuvaksi kokoelmaksi tietoa säikeistyksen perusteista. Materiaalin kokorajoitusten vuoksi kaikkea mahdollisesti tarpeellista tietoa aiheesta ei ole pystytty ottamaan mukaan, mutta opas on silti suhteellisen kattava verrattuna moniin muihin internetistä vapaasti saatavilla oleviin materiaaleihin. Kattavan kokonaiskuvan saavuttamiseksi aiheesta on kuitenkin suositeltavaa tutustua tietoon useampia erillisiä lähteitä hyödyntäen.

Saadakseen tästä itseopiskelumateriaalista parhaan mahdollisen hyödyn, lukijalla on suotavaa olla Javan perusteet hyvin hallussa. Hyvärakenteisten luokkien tekemisen ja enkapsuloinnin (näkyvyys ja rajapinnat) osaaminen ovat myös tärkeitä taitoja, joista on säikeistystä opiskellessa erittäin suurta hyötyä. Itseopiskelumateriaali on mukana tämän opinnäytetyön liitteissä tekstimuodossa ja nähtävissä verkkomuodossa osoitteessa http://javamultithreading.projects.tamk.fi.

2 SÄIKEISTYS

2.1 Perusajatus

Vain yhtä säiettä käyttävä ohjelma suorittaa toimintansa peräkkäisesti – yksi osa koodia suoritetaan toisen jälkeen. Taustalla toki pyörii ohjelman pääsäikeen lisäksi myös esimerkiksi Java Virtual Machinen (JVM) luomia säikeitä tarpeen mukaan, mutta ne keskittyvät muuhun, kuin ohjelmoijan kirjoittaman koodin suorittamiseen. Tämä kuitenkin jättää mahdollisesti vapaana olevia prosessorin ytimiä hyödyntämättä, vaikka käytössä oleva pääsäie kuormittaisi yhden ytimen sataprosenttisesti. Kun yksittäinen ydin ei enää riitä kunnolla ohjelman koodin suorittamiseen, voidaan käyttöön ottaa lisää säikeitä, mikäli niiden käyttö toimii kyseessä olevaan ongelmaan. Käytössä olevien ytimien lukumäärä ja muut suoritintehoa vievät prosessit toki rajoittavat hyödynnettävissä olevia resursseja.

2.2 Mahdollisuudet

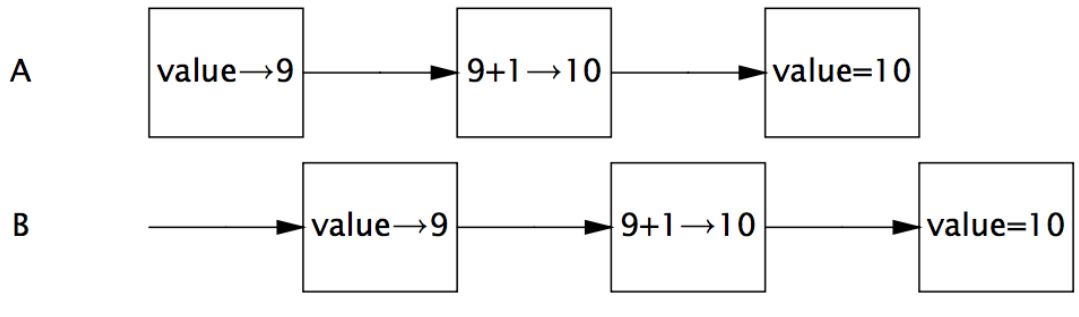
Kun ohjelmoija hyödyntää useampaa säiettä ohjelmakoodin suorittamiseen, on mahdollista suorittaa useita erilaisia koodipolkuja samanaikaisesti. Optimaalisessa tilanteessa ohjelman suorituskyky voi kasvaa lähes lineaarisesti käytössä olevien ytimien mukaan, mutta tämä ei kuitenkaan yleensä ole käytännössä realistista. Ohjelman käyttäjälle muutos voi näkyä esimerkiksi responsiivisempana käyttöjärjestelmänä, pelin kuvataajuuden (frame rate) kasvamisena, tai tiedoston pakkausnopeuden parantumisena.

Esimerkiksi serverit hyötyvät myös suuresti säikeiden käytöstä. Niiden avulla serverille saapuneet kutsut voidaan käsitellä nopeammin, kun eri ytimet saavat käsiteltyä niitä samanaikaisesti. Lisäksi säikeet ovat huomattavasti prosesseja kevyempiä, joten yksittäiselle serverille jää enemmän resursseja käytettäväksi. Kuten muissakin käytöissä, säikeet mahdollistavat myös eri nopeudella valmistuvien tehtävien sujuvan suorituksen. Vaikka käytössä olisi vain yksi prosessoriydin, on jokaisella säikeellä oma aikalohko koodin suorittamiseen, minkä jälkeen toiset säikeet saavat mahdollisuuden suorittaa omaa koodiaan. Tästä johtuen hitaat tehtävät eivät yleensä estä nopeita tehtäviä suoriutumasta järkevässä ajassa.

Koska usean säikeen käyttämisestä seuraa tarve suojata jaetut muuttujat, esimerkiksi synkronoimalla koodia, täytyy ohjelmoijan myös käyttää aikaa luokkien ja niiden vuorovaikutuksen suunnitteluun. Suojaamisen helpottamiseksi on tarpeellista enkapsuloida eri toiminnot hyvin, mikä selkeyttää ohjelman rakennetta. Suuret ja sekavat luokat, joihin on vain lisätty kaikki mahdolliset toiminnot, eivät toimi hyvin useita säikeitä käytettäessä – tai muutenkaan. Edellä mainitut edut eivät kuitenkaan tule aivan ilmaiseksi, vaan implementoinnin ongelmat voivat olla yllättävän suuret.

2.3 Ongelmat

Useamman säikeen turvallinen käyttäminen monimutkaisemmissa ohjelmissa vaatii paljon opiskelua ja harjoittelua. Tämä johtuu monen säikeen kesken jaettujen olioiden ja muuttujien yhtäaikaisen käsittelyn aiheuttamasta mahdollisesta tietojen korruptoitumisesta. Muokkaukset ja lukemiset voivat tapahtua samaan aikaan, jolloin lopputuloskin voi olla yllättävä. Pidempiä operaatioita suoritettaessa on lisäksi mahdollista, että osa toinen säie ehtii muuttaa osaa tiedoista, ennen kuin operaatio on suoritettu loppuun asti. Lisäksi yhdessä säikeessä tapahtuneet muutokset eivät välttämättä näy toisille välittömästi, mistä voi seurata satunnainen vanhan tiedon käyttäminen operaatioissa. Kuvio 1 esittelee huonosti ajoitettujen operaatioiden mahdollisen vaikutuksen yhteisen suojaamattoman muuttujan arvoon (value), kun kaksi säiettä (A, B) kasvattaa sen arvoa limittäin. Kumpikin säie lukee muuttujan arvon (9), suorittaa laskutoimituksen tälle arvolle, ja kirjoittaa tuloksen (10) takaisin muuttujaan. Odotettu arvo (11) jää siis saavuttamatta.



KUVIO 1. Yhtäaikaisen muokkauksen aiheuttama ongelma (Goetz ym. 2010, 6)

Koska edellä mainitut ongelmat ovat hyvin vahvasti aikasidonnaisia, edes testaaminen ei välttämättä paljasta ongelmaa, ellei testaaja tiedä mitä tekee. Yleisenä tapana on käyttää hyvin suurta määrää säikeitä ja toistoja mahdollisten ongelmien havaitsemiseksi, mutta tämä ei ole aina yhtä helppoa miltä se kuulostaa, sillä järkevän testin tekeminen voi olla hyvinkin vaikeaa. Jos jokin säie ei muokkaa suojaamatonta muuttujaa, samalla kun toinen säie muokkaa tai lukee sitä, ei ongelma tule välttämättä koskaan esiin. Satunnaiset ongelmat voivat kuitenkin olla todella haitallisia, varsinkin jos ohjelman tietojen oikeellisuus on hyvinkin tärkeää. Ongelma saatetaan huomata vasta asiakkaalla, mutta satunnaisuuden vuoksi se voi olla tällöin huomattavasti vaikeampi paikallistaa kuin testauksessa.

> Indeed, developing, testing and debugging multithreaded programs can be extremely difficult because concurrency bugs do not manifest themselves predictably. And when they do surface, it is often at the worst possible time—in production, under heavy load. (Goetz ym. 2010, xvii.)

Lisäksi kaikki ohjelmat eivät toimi yhtään paremmin – osa jopa huonommin – vaikka teoreettinen laskentateho kasvaisikin. Kaikki ongelmat eivät hyödy useiden säikeiden käytöstä yhtä paljon. Erilaiset suojamekanismien käytön aiheuttamat koodin suorittamisen hidastumiset laskevatkin säiekohtaista suoritusnopeutta. Koodin synkronointi ja suoritettavien säikeiden vaihto aiheuttavat hidastumista. Lisäksi jotkin Java Virtual Machinen (JVM) käyttämät koodin suorittamista nopeuttavat optimoinnit poistuvat käytöstä turvakeinojen seurauksena. Hyvän suorituskyvyn aikaansaamiseksi tehdyt optimoinnit voivat lisäksi viedä aikaa enemmän kuin itse alkuperäisen koodin kirjoittaminen. Lisäksi optimointien tekeminen vaatii ohjelmoijalta usein vielä parempaa osaamista, varsinkin kun optimointi on usein tasapainottelua nopeuden ja turvallisuuden välillä.

2.4 Huomiot

Useiden säikeiden käyttämisen hyödyt ja haitat riippuvat hyvin vahvasti ohjelman suunnittelusta ja ratkaistavasta ongelmasta. Mitä enemmän jaettuja muuttujia eri säikeet joutuvat käsittelemään, sitä suurempia ongelmia on odotettavissa ohjelmiston kehityksessä, ja sitä suurempi vaikutus näkyy myös suorituskyvyssä. Toisin sanoen, mitä vähemmän säikeet riippuvat toisistaan, sitä helpompaa ohjelmiston suunnittelu ja koodin

kirjoittaminen on. Optimaalisessa tilanteessa koodia suorittava säie vain käynnistetään, ja se suorittaa tehtävänsä ilman tarvetta käsitellä jaettuja muuttujia. Tämä kannustaa ohjelmoijia erottamaan eri tehtävät toisistaan ja muusta sovelluslogiikasta.

Ongelmien vähentämiseksi usean säikeen käyttäminen olisi hyvä ottaa huomioon jo ohjelman ja luokkien suunnitteluvaiheessa. Tällöin kasvanutta laskentatehoa pystytään hyödyntämään paremmin ja turvallisemmin. Lisäksi useiden säikeiden käytössä ollessa tulisi ohjelman dokumentaatioon kiinnittää erityistä huomiota. Eri luokkien käyttö helpottuu huomattavasti, kun turvalliset luokat ja metodit tiedetään, koska tällöin ohjelmoijilla ei ole välttämättä tarvetta tutustua luokkien sisäiseen turvallisuuteen. Tämä auttaa myös koodin ylläpidossa ja vähentää virheiden mahdollisuutta.

Jos yksittäisen koodipolun suoritusnopeus on tarpeeksi hyvä, on parasta pysyä turvallisessa toteutuksessa. Tämä pitää koodin rakenteen hyvänä, helpottaa testausta ja ylläpitoa ja ehkäisee turhat ongelmat. Doug Lean (1999) mukaan joskus kuitenkin satunnaiset ongelmat koodin suorituksessa ovat täysin hyväksyttäviä. Esimerkiksi koordinoimattoman yhtäaikaisen koodin suorittamisen aiheuttama hetkellinen visuaalinen ongelma ei välttämättä haittaa, kunhan tilanne korjaantuu nopeasti ja varmasti. (Lea 1999, 31.)

3 OPPAAN TOTEUTUS

3.1 Suunnitelma

Oppaan ajatuksena oli tarjota suhteellisen kattava materiaali säikeistyksen perusteista ja näin tarjota Ohjelmistotuotannon opiskelijoille helposti lähestyttävä ensikosketus aiheeseen. Tämän toteuttamiseksi suunnitelmana oli tutustua säikeistystä käsittelevään kirjallisuuteen ja verkosta löytyviin resursseihin sekä aiheen opettelemisen lisäksi ottaa mallia aineiston toteuttamiseen. Useisiin lähteisiin tutustumisen jälkeen oli tarkoituksena luoda materiaali, joka sekä yhdistelee lähteissä esitettyjä tietoja että tarkistaa niiden oikeellisuuden – varsinkin verkkomateriaalien kyseessä ollessa. Suunnitelmana oli yhdistää teoriatieto ja yksinkertaiset esimerkit, jotta lukijat voisivat helposti testata esille nostettuja ongelmia.

Materiaalin vaativuus suhteutettiin toista vuotta opiskeleville opiskelijoille sopivaksi, koska itse aihekin vaatii pelkkää peruskurssia paremman osaamisen. Tapauksesta riippuen usean koodipolun samanaikaisen suorituksen turvalliseksi saaminen voi olla joko helppoa tai erittäin haastavaa. Aihe on kokonaisuudessaan hyvin laaja, ja laadukas ohjelmistokehittäminen säikeistettäessä on hyvin vaikeaa ja aiheuttaa ongelmia parhaillekin kehittäjille (Evans & Verburg 2012, 77). Tämän vuoksi tarkoituksena oli luoda lähinnä perusteita käsittelevä opiskelumateriaali, joka mahdollistaa ohjelmien yksinkertaisen säikeistämisen myös opiskelijoille. Hyvien Java-ohjelmoinnin käytänteiden osaaminen oli esioletuksena materiaalin hyödyntämiseen tarvittavalle taitotasolle, koska hyvärakenteinen koodi on erittäin tärkeää enkapsuloinnin onnistumiseksi ja samalla turvallisen koodin kirjoittamiseksi. Lisäksi aiheen ajoittaisen monimutkaisuuden vuoksi oli oletettavaa, että työssä käytetty kieli voi olla paikoitellen vaikeaa ymmärtää, joten suhteellisen hyvät englannin kielen taidot ovat lukijalle tarpeen.

3.2 Toteutus

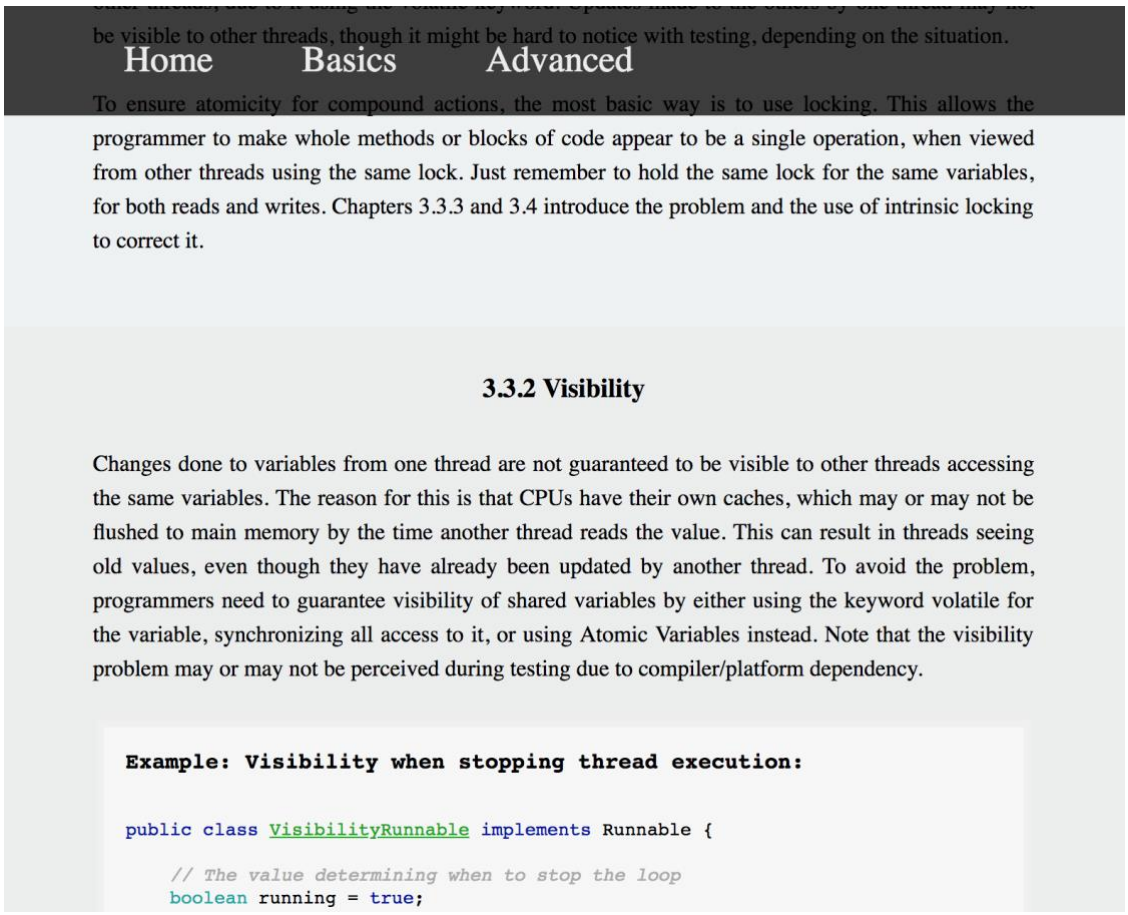
Ennen oppaan teon aloitusta itse aihe täytyi opiskella kunnolla. Aloituksen helpottamiseksi ensimmäiseksi käytössä olivat moninaiset videotutoriaalit sekä Oraclen tarjoama tutoriaali aiheesta (The Java Tutorials. Lesson: Concurrency). Tämän jälkeen

vuorossa olivat kirjojen lukeminen ja tiivistelmien tekeminen, mikä auttoi suuresti itse materiaalin kirjoittamisessa helpottaen useamman tiivistetyn lähdemateriaalin vertailua. Lukemisen yhteydessä oli lisäksi tarpeen testata lukuisia esimerkkikoodeja, joita hyödynnettiin osittain oppaan esimerkkien teossa. Useat lähteissä esitellyt esimerkit olivat tasoltaan liian vaativia sopiakseen hyödynnettäväksi ilman suuria muokkauksia. Javan dokumentaatio auttoi selventämään eri luokkien ja metodien käyttöä. Oppaan sisältö varmistui materiaaleihin tutustumisen yhteydessä.

Oppaan tekeminen alkoi kirjaamalla ylös mukaan tulevat asiat alustavaan sisällysluetteloon. Jokaisessa osiossa hyödynnettiin useita eri lähteitä niin, että olennaiset tiedot saatiin yhdistettyä, ja teksti muotoutui kattamaan materiaaleista saadun tiedon. Tämä auttoi lisäämään osioihin huomioita, joita ei välttämättä esitelty muissa materiaaleissa. Lisäksi useamman materiaalin vertaaminen toisiinsa auttoi välttämään joissakin lähteissä esiin tulleiden virheiden mukaan ottamisen. Esimerkkien teossa käytettiin apuna kirjoja, teoriapohjaista materiaalia, löytyneitä esimerkkejä ja erilaisia internetistä löytyviä oppaita. Näin luodut esimerkit olivat kuitenkin yleensä yksinkertaistettuja ja helposti ymmärrettäviä sekä hyödynnetystä lähdemateriaalista erillisiä. Laajuudeltaan opas on tekstimuotoisena 51-sivuinen.

Materiaalin sisällön, esimerkkien ja rakenteen valmistuttua luotiin verkkosivut, joita hyödynnetään aiheesta kiinnostuneiden opiskelijoiden tavoittamisessa. Sivut ovat rakenteeltaan melko perinteiset ja eri osa-alueet on jaettu omille alisivuilleen (kuva 1). Esimerkkikoodit on esitelty erillisissä laatikoissa, joissa hyödynnetään syntaksin värjäystä koodin lukemisen helpottamisessa. Tekstiin on lisätty mukaan linkkejä auttamaan suorassa siirtymisessä mainittuja aiheita käsitteleviin oppaan osioihin.

Luotu opas löytyy lisäksi tekstimuotoisena tämän opinnäytetyön liitteistä, mikä mahdollistaa materiaalin löytymisen myös mahdollisen verkkosivujen osoitteen vaihtamisen jälkeen. Lukijan on kuitenkin huomioitava, että mahdolliset jatkossa tehdyt muutokset eivät päivity tähän liitteeseen. Tämän vuoksi se voi myös sisältää virheitä, jotka verkkototeutuksessa on jo korjattu.



KUVA 1. Verkkosivujen ulkoasu

3.3 Sisältö

Oppaan ensimmäisessä ja toisessa luvussa esitellään alustavasti säikeistämisen merkitys, hyödyt ja haitat. Esiteltyihin hyötyihin ja haittoihin palataan syventävästi myöhemmässä materiaalissa ja osaa demonstroidaan esimerkkien avulla. Tiivistetysti sanottuna yhtäaikaisen suorittamisen suurimmat hyödyt ovat mahdollinen ohjelman latenssin- ja suorituskyvyn paraneminen, mahdollisuus käytössä olevien resurssien parempaan hyödyntämiseen sekä vaatimus paremmasta koodin rakenteesta. Ongelmat sen sijaan kattavat vaaran tiedon korruptoitumisesta, testaamisen vaikeutumisen, mahdollisesti lisääntyneen määrän ohjelmointivirheitä sekä joissain tapauksissa jopa heikomman suorituskyvyn.

Kolmannessa luvussa käsitellään säikeistyksen perusteet, sekä tutustutaan Javan tarjoamiin perusmekaniikkoja toteuttaviin luokkakirjastoihin ja niiden käyttöön. Osio sisältää säikeiden luomisen, elinkaaren hallinnan ja turvallisen käyttämisen. Toisena suurena kokonaisuutena ovat jaettujen muuttujien turvallisuuteen (thread safety)

liittyvien ongelmien esittely sekä monien Javan tarjoamien apukeinojen esittely ja implementointi käytännössä. Kaikkea mahdollisia apukeinoja ja skenaarioita ei ole pystytty ottamaan mukaan, koska niiden lisääminen olisi kasvattanut oppaan koon moninkertaiseksi. Lisäksi luku sisältää säikeen väliaikaisen pysäyttämisen ja suorituksen pysyvän keskeyttämisen mahdollistavat keinot sekä käsittelee säikeiden välisen kommunikoinnin (wait & notify) ja virhetilanteissa toimimisen.

Neljännessä luvussa tutustutaan lyhyesti koodin suorituskykyyn sekä muutamiin hyödyllisiin kehittyneempiin käytössä oleviin luokkakirjastoihin. Kehittyneempien työkalujen esittely ei ole työn tavoitteiden vuoksi läheskään yhtä kattava kuin perusteissa, vaan kyseessä on enemmänkin pikainen katsaus aiheeseen. Tärkeimpinä asioina ovat katsaus koodin suorituskykyyn, sekä Javan Executors-ohjelmistokehys, joka mahdollistaa muun muassa säikeiden kierrätyksen ja kehittyneemmän hallinnan. Yleisestikin java.util.concurrent-luokkakirjasto tarjoaa monia hyödyllisiä luokkia säikeistyksen toteuttamiseen. Vaikka monet näistä työkaluista ovatkin helppokäyttöisiä, ja jopa monessa tapauksessa paras ratkaisu, on silti suositeltavaa ymmärtää myös yksinkertaisempien välineiden toiminta.

Opas ei itsessään ole kaiken kattava, joten aiheesta kiinnostuneiden on suositeltavaa tutustua myös muihin oppaisiin ja alan kirjallisuuteen. Käsiteltävät luokkakirjastot ja metodit on rajattu aloittelijalle hyödyllisiksi ajateltuihin kokonaisuuksiin, mutta materiaalin laajuus ei riitä kattamaan kaikkia tarpeellisia asioita. Lisäksi käytännön harjoittelu on tarpeen materiaalin sisäistämiseksi, mikä pätee muidenkin ohjelmointiin liittyvien taitojen oppimisessa. Viidennessä luvussa – joka on verkkomateriaalissa siirretty ensimmäiselle sivulle – mainitaan keskeisimmät työssä käytetyt lähteet. Keskeisimpinä työn lähteinä toimivat kirjat Java Concurrency in Practice (Goetz ym. 2010), Concurrent Programming in Java (Lea 1999) ja The Well-Grounded Java Developer (Chapter 4, Modern Concurrency)(Evans & Verburg 2012). Verkkosivuista tärkeimmät olivat The Java Tutorialsin (n.d.) tarjoamat resurssit ja Javan dokumentaatio.

4 POHDINTA

Opiskelumateriaalin tekeminen onnistui tavoitteen mukaisesti. Työhön otettiin mukaan oleelliset perusteet, jättäen harvoin käytettyjä työkaluja esittelemättä, jotta materiaalin laajuus ei kasvaisi liikaa. Tästä huolimatta aineiston koko kasvoi yllättävänkin suureksi, sisältäen myös muutamia hyödyllisiä kehittyneempiä työkaluja. Vaikka esittelemättä jäikin monia uudempia ja monen mielestä parempia välineitä, täytyy myös alkeellisempien työkalujen olla hallinnassa ennen niihin tutustumista. Tämä mahdollistaa tiedon syvällisemmän oppimisen pelkän ulkoa opettelun sijaan.

Koska tekijällä ei ollut aiempaa syvällistä tietoa aiheesta, täytyi ennen kirjoittamisen aloittamista tutustua useisiin erilaisiin lähteisiin ja testata esitettyjä ongelmia omilla kokeiluilla. Vaikka tämä hidastikin työn valmistumista, moniin lähteisiin tutustuminen antoi hyvän kokonaiskuvan aiheesta. Huomioitavaa on, että edes yksittäisen kirja ei riitä esittelemään kaikkien työkalujen kattavaa käyttöä, tai tarjoa jokaisessa kohdassa tarpeeksi selkeitä ja tyhjentäviä esimerkkejä. Lähdemateriaaleja tutkittaessa huomattiin myös, että varsinkin ilmaisissa esimerkeissä oli selkeitä hyvien käytänteiden vastaisia ratkaisuja, tai jopa suoranaisia virheitä. Myös tästä johtuen olisi suotavaa hyödyntää opiskelussa useita erilaisia materiaaleja ja testata koodi varmasti toimivaksi.

Ikävä kyllä tämäkään materiaali ei ole välttämättä täysin virheetön, joten lukijan olisi siihen tutustuessaan hyvä harjoittaa tälläkin alalla tarvittavaa kriittistä lähteiden arviointia. Lisäksi kaikki huomiot eri ongelmista eivät ole kaiken kattavia, joten niitä ei välttämättä pysty suoraan hyödyntämään ongelmien ratkaisuissa. Varsinkin alkupäässä esitellyt esimerkit ovat luonteeltaan hyvin yksinkertaisia ja ne pyrkivät vastaamaan senhetkisiin esitettyihin ongelmiin. Pelkän yksittäisen ratkaisun suora kopioiminen ei ole täten suositeltavaa, vaan lukijan olisi paras ymmärtää itse ongelma ja miettiä mahdollisia ratkaisuja ennen sopivalta vaikuttavan implementointia. Monissa ongelmissa kuitenkin on ohjelmoinnille tyypillisesti useita mahdollisia ratkaisumalleja, joissa voi olla omat etunsa ja haittansa turvallisuuden, nopeuden ja ylläpidon helppouden suhteen. Materiaalista jouduttiin ikävä kyllä rajaamaan pois muutamia työkaluja, jotka olisivat varmasti hyödyllisiä monien ongelmien ratkaisussa.

Materiaalin jatkokehitystä ajatellen tärkeintä olisi ottaa mukaan useampia esimerkkejä, jotka esittelisivät monimutkaisempia ongelmia ja niiden ratkaisuja. Monet säikeistämiseen liittyvät ongelmat eivät ole koodin monimutkaistuessa välttämättä yksinkertaisia, vaan voivat olla vaikeita havaita edes testauksessa. Tästä johtuen myös testauksen kattava esittely olisi tarpeen, varsinkin kun itse testien suunnittelu voi olla jopa vaikeampaa kuin itse turvallisen koodin tekeminen. Lisäksi aihetta tosissaan opiskelevat hyötyisivät varmasti tehtävien ja esimerkkiratkaisujen tarjoamisesta. Tällöin olisi mahdollista myös esitellä paremmin erilaisia säikeistykseen liittyviä ongelmia ja pakottaa lukija miettimään ratkaisuja.

LÄHTEET

Evans, B.J. & Verburg, M. 2012. The Well-Grounded Java Developer. 1. painos. New York: Manning Publications

Goetz, B., Peirtls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. 2010. Java Concurrency in Practice. 9. painos. Upper Saddle River, NJ: Addison-Wesley Professional

Lea, D. 1999. Concurrent Programming in Java: Design Principles and Patterns. 2. painos. Upper Saddle River, NJ: Addison-Wesley

Oracle. N.d. The Java Tutorials. Lesson: Concurrency. Luettu 1.5.2018 https://docs.oracle.com/javase/tutorial/essential/concurrency/

LIITTEET

Liite 1. Basics of Multithreading in Java

Basics of Multithreading in Java

TUOMO MARTIKAINEN

1 What Is Multithreading?

Multithreading allows the separation of program execution into multiple threads. This differs from multiprocessing by threads being lighter weight than processes, and sharing the same memory space. Even single core processors can have multiple processes and threads running, but the number of cores limits their simultaneous execution. Each running process and thread are given their own time slice to execute in, before having to wait for their next time to run.

Even seemingly sequential programs usually have their own UI event thread and possibly other threads running in the background. Java Virtual Machine (JVM), for example, creates threads for garbage collection and finalization in addition to the main thread. Different frameworks may also create threads on behalf of the programmer - sometimes causing unintended trouble. The different threads are given the chance to execute, regardless of the number of CPUs, by the thread scheduler.

2 When to Use Multithreading?

Multithreading can be beneficial and/or detrimental to programs, depending on the thread usage and program architecture. Therefore, the design and implementation of the program determines the result - in addition to the fact that all things cannot be made better with the use of multiple threads.

2.1 Pros

The most obvious benefit of multithreading is potential performance improvement, when system resources have not already been sufficiently utilized. This may mean, for example, a single CPU having prolonged 100% usage while others are idle, or program being unresponsive or unable to execute other things while waiting for I/O to finish. Program responsiveness is often the benefit that end users will first notice.

It is possible to efficiently utilize all system resources with the use of multithreading, but not very many programs can possibly do that, especially for extended periods of time. Optimally program performance can increase nearly linearly when there are enough cores to use, but it is also possible that multithreading can result in negative performance. Still, even simple programs may benefit from it, especially if they have blocking I/O operations. Server applications are one example of multiple threads providing great performance improvements.

Program structure may also benefit from using multithreading. When properly implemented, programmers are forced to encapsulate their code better, which results in better structure. It is also possible to decompose asynchronous workloads into simpler, synchronous workloads that run in separate threads.

2.2 Cons

Using multiple threads may affect thread safety negatively. Without proper synchronization, the results of operations can be highly unpredictable. Modifying unprotected variables that multiple threads have access to can lead to data corruption - interleaving caused by race conditions can result in even basic variable modification causing problems. Due to high dependency on the relative timing of events, interleaving can be easy to miss while testing.

It is also possible for the program to have liveness failures, where the program is permanently unable to make progress in its execution. A thread may hold a resource infinitely, causing other threads to be unable to acquire it. Deadlock, starvation and livelock are all forms of liveness failures. Liveness is also affected by the relative timing of events, so it can be hard to test.

Depending on the situation and program implementation, a multithreaded program may use more resources while being slower than its single-threaded counterpart. Code synchronization and the context switches made by the thread scheduler both take time. Data shared by threads must be protected by synchronization mechanisms, which makes the execution of code slightly slower than unprotected code. When access to a variable is contented by multiple threads, even more time is lost due to some threads having to wait to gain access.

Synchronization may also affect compiler optimizations and forces memory caches to be flushed and invalidated. Compiler optimizations include reordering of operations, where in a seemingly single-threaded environment allows the JVM to execute operations, that do not affect each other, in a more efficient order. For example, operations written in order (A, B, C) might be executed in order (B, A, C), if from a single-threaded point of view, this does not change the result. This can become another problem when multiple threads work together without proper measures taken.

Performance may also suffer from context switches, which happen when an active thread is suspended to allow another thread to run instead. This loses a bit of CPU time, as resources are needed to make the switch, if there are no cores available for all threads. Should threads vastly exceed the number of CPUs available, context switches will become an even greater problem, along with lots of inactive threads hogging a lot of memory.

Testing multithreaded code is also a lot more difficult, thanks to the program now having all the old bugs, but also the sequential ones. Timing-related issues with shared variables make designing the tests more difficult, and there is often the need to test using loops and many more threads than would normally be used. Tracking bugs can also be a lot harder due to many code execution paths running at the same time.

Study and practice is also needed to ensure code safety. Multithreaded programming requires strict adherence to specific protocols, and knowledge is needed for both code optimizations and breaking the rules. More documentation is also needed to make synchronization protocols easier to understand. Otherwise code changes and subclassing can become more difficult and cause unexpected safety problems.

2.3 Concurrency vs parallelism

Multithreading can also be divided into concurrency and parallelism, where concurrency means executing different tasks simultaneously. The tasks may be either loosely or tightly coupled, with entirely separate (the loosest) tasks being the safest and easiest to implement. Parallelism means separating a single larger task into smaller executable units for simultaneous execution, e.g. dividing a large mathematical problem into smaller parts for faster execution. In many cases this approach is neither reasonable nor possible due to the nature of the task.

This study material focuses heavily on concurrency. Both concurrency and parallelism use the same basics, with the difference being the problems which they try to solve. Therefore, it is recommended to first understand the basics before using either for important functions.

3 Thread Basics

This chapter deals with thread basics like visibility, atomicity, thread creation, lifecycle, safety, sleep and wait.

3.1 Creating threads

Basic threads can be created by either extending the class Thread, or by implementing the Runnable interface and passing it to a thread to run. Implementing the runnable interface is the recommended approach due to allowing to extend some other class, and separating the task (Runnable) from the runner (Thread). More advanced tools like the Executors interfaces also work with Runnable.

It is important to remember to start thread execution by calling start(), as calling run() will just run the method in the current thread (which can also sometimes be useful). Threads that have finished their execution cannot be restarted. This also supports the use of Runnable, as it can be passed to a new thread to continue from where the previous left off. Trying to start() a thread again, however, will result in an IllegalThreadStateException to be thrown.

Example: Implementing Runnable

```
public class MyRunnable implements Runnable {
    // Implement the run() method; give to a thread and call start()
    // on it.
    public void run() {
        System.out.println("Thread printing this once");
    }

    public static void main(String[] args) {
        // Create a new thread, giving it a runnable to execute, and
        // start it.
        new Thread(new MyRunnable()).start();

        // Or to save a reference to the thread, assign it to a
        // variable.
        Thread t = new Thread(new MyRunnable());
        // Start the thread
        t.start();
    }
}
```

Example: Extending Thread

```
public class MyThread extends Thread {

    public void run() {
        System.out.println("Thread printing this once");
    }
```

```
    public static void main(String[] args) {
        // Create and start thread without saving the reference
        new MyThread().start();
    }
}
```

3.2 Thread lifecycle

In Java threads have six possible states to be in. These are New, Runnable, Blocked, Waiting, Timed-Waiting and Terminated. Check the java.lang.Thread.State for the states.

When a thread has been created but not started yet, it is in the New state. At this point the thread needs the start() method to begin execution and enter Runnable state. The start() method cannot be called more than once on the same thread without resulting in an IllegalThreadStateException.

When the thread is in Runnable state, it is executing in the Java Virtual Machine. The thread, though, may be either executing code or waiting for system resources to start execution. Thread state may move from this state to Blocked, Waiting and Timed-Waiting and back again, or it may move to Terminated state with no path for return.

The thread state moves to Blocked when waiting to acquire a monitor lock. With intrinsic locking this means getting the lock for a synchronized method or code block. When the lock is acquired, the thread moves back to Runnable state. If the lock is not released by the holding thread and intrinsic locking is used, the blocked thread will wait forever. Explicit locking allows for interruption and time limit to be used for lock acquisition.

When lock.wait() or thread.join() is called, the thread enters Waiting state to wait for the signal to get back to Runnable state. While lock.notify() or lock.notifyAll() (both called on an object) is required to wake up a waiting thread, a joining thread automatically gets its nofityAll() from the thread it is waiting on upon thread termination. Waiting and Timed-Waiting can also be interrupted with thread.interrupt() to exit the state. Note that a thread returning from waiting may move to Blocked state when trying to reacquire the lock.

Timed-Waiting state is entered when the thread waits for a specified amount of time, before returning to Runnable state. Methods like thread.join(long millis), Thread.sleep(long millis) and lock.wait(long millis) trigger this change. Note that while Thread.sleep() is called on the Thread class, the lock.wait() is called on a lock object while synchronized on that lock. Both methods affect the current thread the code is executed on. Interruption also wakes up both waiting and sleep, while lock.notify() and lock.notifyAll() work on waiting but not sleep. It is not recommended to call wait(), notify() and notifyAll() on Thread instances, as terminating threads call this.notifyAll() which would wake up waiting threads.

When a thread has finished its execution, it enters the Terminated state. This happens when the thread has finished the execution of its run() method. A terminated thread cannot be restarted with the start() method, but if a Runnable was used, a new thread may start executing it again.

3.3 Thread safety

Thread safety requires a class to behave correctly when accessed from multiple threads. To ensure safety when more than one thread access a variable, and one of them might write to it, they must all coordinate their access to it. Race conditions might otherwise cause the correctness of a computation to depend on the possibly unfortunate timing or interleaving of multiple threads. Immutable variables, and those inaccessible from other threads, do not require synchronization. The immutable ones never change and thread-local variables are not shared to others. Of course, objects referred to via immutable references might not be immutable themselves, and seemingly local variables might be unexpectedly leaky.

3.3.1 Atomicity

Reads and writes of reference variables and most primitive variables are guaranteed to be atomic, except for long and double, so they do not suffer from interleaving problems resulting from simultaneous reads/writes when assigning new values. What value was written last might be hard to predict, though. When long and double are set as volatile variables, their reads and writes are also guaranteed to be atomic. Note that on 64-bit Java Virtual Machines reads and writes to long and double should be atomic, but it is a poor practice to depend on this due the portability of code suffering from doing so. Also remember that compound actions like value++ are not atomic without the use of proper synchronization.

Examples:

```
int i;
double d;
volatile double vd;
Object o;
...
i = 1;      // assignment atomic
d = 1.1;    // not guaranteed to be atomic
vd = 1.1;   // atomic
o = b;      // atomic (but check safe publication)
i++;        // not atomic
```

Of all the examples above, however, only "vd = 1.1" ensures that the change is immediately visible to other threads, due to it using the volatile keyword. Updates made to the others by one thread may not be visible to other threads, though it might be hard to notice with testing, depending on the situation.

To ensure atomicity for compound actions, the most basic way is to use locking. This allows the programmer to make whole methods or blocks of code appear to be a single operation, when viewed from other threads using the same lock. Just remember to hold the same lock for the same variables, for both reads and writes. Chapters 3.3.3 and 3.4 introduce the problem and the use of intrinsic locking to correct it.

3.3.2 Visibility

Changes done to variables from one thread are not guaranteed to be visible to other threads accessing the same variables. The reason for this is that CPUs have their own caches, which may or may not be flushed to main memory by the time another thread reads the value. This can result in threads seeing old values, even though they have already been updated by another thread. To avoid the problem, programmers need to guarantee visibility of shared variables by either using the keyword volatile for the variable, synchronizing all access to it, or using Atomic Variables instead. Note that the visibility problem may or may not be perceived during testing due to compiler/platform dependency.

Example: Visibility when stopping thread execution:

```
public class VisibilityRunnable implements Runnable {

    // The value determining when to stop the loop
    boolean running = true;
    // Using volatile boolean would fix this potential problem

    public void run() {
        while (running) {
            System.out.println("Running");
        }
        System.out.println("Stop running");
    }

    public static void main(String[] args) {
        VisibilityRunnable r1 = new VisibilityRunnable();
        Thread t1 = new Thread(r1);
        t1.start();         // Start the new thread

        try {

            // Calling thread (main here) sleeps for a while.
            Thread.sleep(10);

        } catch (InterruptedException ie) { }

        System.out.println("Setting to false");
        // Main thread modifies value of the Runnable
        r1.running = false;
    }
}
```

As mentioned above, making the running variable volatile would do fix the problem, but synchronization can also be used for the same effect. Using synchronized setter and getter for the value would ensure visibility of the variable, as synchronization guarantees a happens-before relationship. This means that when a synchronized method or block exits, any changes are guaranteed to be visible to any thread obtaining the same synchronization lock in the future. Writes to a volatile variable does the same to subsequent reads of the same variable.

3.3.3 Race conditions

Race conditions can happen when two or more threads access the same variable at the same time and one or more of them modifies the variable in a non-atomic way. This can result in interleaving of the operations due to unfortunate timing, which can lead to the variable having an unexpected value. For example, if two threads increment a value simultaneously, the result may be different than expected. For long and double, if they are read and written as two 32-bit values, instead of 64-bit (platform etc. dependent, as mentioned before), the read/written value may be a combination of two different written values.

Example: Possible interleaving

```
volatile int a = 1;
// ...
public void incrementA() {
    a++;
}
```

If two threads call incrementA() too close to each other they both can first read the value as 1. After this they will both increment their read values: 1 + 1 and 1 + 1. The results will then be written back to variable a, so that the first thread will assign the value a 2, and the second thread will do the same. The value will therefore not be the 3 as expected. This bug can be hard to detect without proper testing, as is the case with all timing-related problems. This often means using large number of threads and writes to shared variables.

Example: 10 x 10000 interleaving test

```
class Interleaving implements Runnable {

    // Static number for incrementation.
    // Using volatile here would offer a slight increase in result,
    // but does not fix things.
    public static int number = 0;

    public void run() {
        for (int i = 0; i < 10000; i++) {
            number++;
        }
    }

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new Interleaving());
            t.start(); // the threads are started after creation
        }

        try {
            // Main thread waits 100ms so others can finish
            Thread.sleep(100);
        } catch (InterruptedException e) { }
```

```
        System.out.println("10*10000 = " + number);
    }
}
```

The above class Interleaving makes 10 threads that each increment the static variable "number" 10000 times. With no synchronization present, many of the operations will interleave, resulting in an unpredictable number. The latest modification is also not guaranteed to be seen by the printer, or the reading threads for that matter. Test the example with a volatile variable, and different numbers of incrementations and threads to see how the results vary. The threads are not started at the same time, which also affects the results, especially with fewer incrementations. Class java.util.concurrent.CountDownLatch could be used for simultaneous start up.

To avoid interleaving for certain functions without the use of locking, check the java.util.concurrent.atomic package, which offers utility classes like AtomicInteger. Using it would allow the above example to update the value without interleaving problems.

3.4 Locking

Locking can be used to make non-atomic operations atomic. When an atomic operation is performed by a thread, it is guaranteed that other threads do not see the value in an inconsistent state. You should note that guaranteeing this will require the reading method to access the variable using the same synchronization lock that was used for the operations. Everything that a thread did in or before a synchronized block, is also visible to another thread when it executes a synchronized block with the same lock. Also, just combining two atomic operations together does not make the whole atomic (without additional locking), so race conditions still apply.

The synchronization lock is acquired when a thread tries to enter a block of code protected by the lock, and another thread has not already acquired it. Other threads wanting to acquire the same lock will have to wait for their turn, and if multiple threads are waiting, the JVM decides which gets the lock next. The locks are reentrant, so the current owning thread can enter other methods and code blocks guarded by the same lock.

If one thread waits while holding the lock indefinitely (infinite loop, long sleep, deadlock etc.), the other threads waiting for the lock stay blocked. With intrinsic locking there is no way to back up from waiting for a lock. Threads having to wait for their access to the synchronized code is a major cause of slowness for multithreaded code when contention is heavy. Consequently, there are also many ways to mitigate contention and even avoid synchronization altogether. Synchronized blocks may often be made smaller (higher granularity) to minimize the time other threads are blocked. For the same reason, it is a good practice to try to avoid holding locks during lengthy computations or other possibly long tasks like waiting for I/O.

Java has two types of locking: intrinsic locking and explicit locking (see chapter 4.2). Explicit locking has more features available, but also requires more care when used, due to increased complexity.

3.4.1 Intrinsic locking

Intrinsic locking uses the keyword "synchronized" to encompass either methods or blocks of code. The lock can be any representation (instance) of an Object, including the object that has the method itself (this). In the case of using synchronization for a Class, you can use either a static lock object, static synchronized method, or a synchronized block synchronizing on the class itself. Note that using getClass() to select what to synchronize on may surprise you when subclassing. Intrinsic locks also have no way to back up if they are waiting for a lock to be released, so they will either acquire the lock or wait forever.

Examples: synchronization use and syntax:

```
// Method synchronized with current object aka "this" as the lock.
public synchronized void set() {
    // ...
}
```

```
// Block of code synchronized with current object as the lock;
// interchangeable with synchronized method.
public void set() {
    synchronized(this) { ... }
}


// Creating an object to use as a lock. Private final is often
// preferred.
private final Object lock = new Object();
    // ...
public void set() {
    // Block of code synchronized on the object.
    synchronized(lock) {
        // ...
    }
}

// Synchronize method on the class (static method).
public static synchronized void set() {
    // ...
}

public static void set() {
    // Block of code synchronized on the class.
    synchronized(Example.class) {
        //...
    }
}
```

As mentioned before, any object can be used locking instance methods, but it is often preferred to use a private final Object for the job. This is due to the object being unaccessible (private) from outside the class, which could otherwise cause problems if the lock used incorrectly, and preventing the possibility of changing (final) the locking object to ensure correctness. If the locking object is changed while a thread is using the lock, and another thread uses the new object as the lock for the same or another method (which seemingly uses the same lock), the respective blocks of code can be executed simultaneously.

Example: Interleaving corrected with synchronization

```
class Interleaving2 implements Runnable {

    // Static number for incrementation.
    public static int number = 0;

    public void run() {
        for (int i = 0; i < 10000; i++) {
            increment();
        }
    }

    // Synchronize on Interleaving2.class; non-static would sync on
    // the object.
    public static synchronized void increment() {
```

```
        number++;
    }

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new Interleaving2());
            t.start(); // the threads are started after creation
        }

        try {
            // Main thread waits 100ms so others can finish
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // IE must be caught or declared to be thrown
        }

        System.out.println("10*10000 = " + number);
    }
}
```

Note that the code above synchronizes on the class, so the lock is shared by all instances of it. Test removing the static keyword from the method increment() or the value “number”, and you will notice that the result is wrong again. Without the method being static, each Interleaving2 object created in main method starts to use itself as the lock (same as synchronized(this)). Replacing the method with the one below is essentially the same as using the code above:

```
public void increment() {
    // Sync on the class, not object
    synchronized(Interleaving2.class) {
        number++;
    }
}   // The same as static synchronized method
```

To change the method to lock to point to the object itself instead of the class would look like:

```
public void increment() {
    // Synchronizing on this object, not class
    synchronized(this) {
        number++;
    }
}   // Breaks the Interleaving2 again, like with the removal of static

// The above works the same as the following:
public synchronized void increment() {
    // Sync on object, not class
    number++;
}   //Breaks Interleaving2 again
```

There are still problems with Interleaving2, though. The static int number can still be accessed by anyone without using the synchronized increment() method. This would

bypass the synchronization altogether and again cause interleaving due to race conditions, as did using different object instances as locks to modify the same value. Consequently, it is not often good to allow access to shared variables except through controller methods which have been implemented with proper synchronization. Similarly, if there are multiple shared variables that are used together in operations, it may not be safe to access them without using the same lock.

The final printing of value also accesses the number without synchronization, and as the number is not set as volatile, it may see the correct value or some other value written by one of the threads. Using volatile does both fix this and move the result a bit closer to the intended one, as all the losses may not be from interleaving, but also from visibility problems. Using java.util.concurrent.atomic.AtomicInteger here could be used to avoid using synchronization altogether and correct the problems presented above.

3.4.2 Method parameters and local variables

Unlike shared variables and objects, primitive data types passed as method parameters do not require synchronization, as they become thread-local. For example, passing around an int value will create a copy of the value, not point to the same value (while Integer objects are in danger). Object references passed as method parameters, however, point to objects that may be shared by multiple threads, so remember to preserve their thread safety.

Local variables that are generated inside methods, constructors and blocks also do not need to use synchronization, unless they are references shared with other threads. The variables live and die inside their respective blocks and thus are effectively thread-local. Instance and class variables do not have the same visibility, so their access may still need to be restricted or protected.

3.5 Using sleep and join

Sometimes it is necessary to pause thread execution for a time to introduce some time until something happens, or to allow other tasks to use more processing time. In the examples of this material, sleep is often used to help demonstrate different problems that can be caused by multithreading. The two sleep methods available are both static and affect the executing thread. The sleep time is not guaranteed to be accurate and it can also be interrupted by calling interrupt() on the thread, triggering an InterruptedException in response. Any locks held by the sleeping thread are not released, so it is usually recommended to not to sleep while holding locks to avoid liveness problems (often seen as unresponsiveness).

```
Sleep methods for java.lang.Thread:

public static void sleep(long millis) throws InterruptedException

public static void sleep(long millis, int nanos) throws
InterruptedException
```

Example: Using sleep

> This simple example uses sleep to give provide a timeout of approximately 0.1 seconds between each printing. This could be used to help with debugging or allow the user interface to pace the presentation of text. Thread.sleep requires to either catch the possible InterruptedException or declare the "throws InterruptedException" for the method, which causes the thread to crash should there be an exception.

```
class SleepPrinter {

    public static void main(String[] args) throws InterruptedException
{
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            Thread.sleep(100);      // This thread sleeps 100ms
        }
    }
}
```

3.5.1 Sleep and interrupt

The sleep method is a blocking method, which will throw InterruptedException if the thread is interrupted before or during the sleep. This allows other threads to wake up the thread if it is needed, and let it either stop, continue with its execution, or deal with the (un)expected situation. It is up to the programmer to decide how to deal with the InterruptedException, but stopping the operation and returning is the usual approach. It is also usually the expected response to interruption, so be careful when using it In other ways.

Example: Interrupting sleep and returning

The example creates a thread for printing values and has the main thread wait for a while until interrupting the printing thread. Note that the Runnable's run method cannot be declared to throw InterruptedException, so it must use try-catch to catch it.

```
class SleepPrinterInterruption {

    public static void main(String[] args) throws InterruptedException
{

        //Creating thread and the Runnable for it
        Thread t = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 10; i++) {
                    System.out.println(i);
                    try {
                        Thread.sleep(100);      // Thread sleeps 100ms
                    } catch (InterruptedException ie) {
                        System.out.println("Printing interrupted");
                        return;
                    }
                }
            }
        });

        t.start();

        // Main thread sleeps 300ms; now throws IE is declared so no
        // need to catch
        Thread.sleep(300);


        // Interrupt the thread t
        t.interrupt();
    }
}
```

3.5.2 Join

Sometimes a thread needs to wait for another to finish before continuing with its execution. Join method can be called on another thread (thread.join()), which causes the calling thread to suspend its action until the other thread has finished its execution. When the thread terminates, it invokes its this.notifyAll() method, which causes all threads executing join() to awaken. Due to this it is not recommended to use wait(), notify() or notifyAll() on thread instances, as it could disrupt this expected behavior. (Use them on locks instead.)

Like wait() and sleep(), join() is also responsive to interruption which causes it to throw InterruptedException, that either needs to be caught or declared to be thrown by the method. With regard to locks, join() works like sleep() in the way that it is called on the current thread and does not release possible synchronization. Note that wait() is called on a lock object (which is released on wait), that has its synchronized block encompassing the invocation.

```
Join methods:

// Waits until thread dies (join(0) does the same)
public final void join() throws InterruptedException
// Waits until thread dies or time ends
public final void join(long millis) throws InterruptedException
// Waits until thread dies or time ends
public final void join(long millis, int nanos) throws
InterruptedException
```

```
Example: Wait for thread to finish

class JoinerThread extends Thread {

    public void run() {
        System.out.println(this.getName() + " going to sleep");
        try {
            Thread.sleep(1000);     // Wait a sec
        } catch (InterruptedException ie) {
            // Not happening in this example
            System.out.println(this.getName() + " interrupted");
        }
        System.out.println(this.getName() + " ending");
    }
}

class JoinTester {

    public static void main(String[] args) throws InterruptedException
{
        Thread t1 = new JoinerThread();
        t1.start();
        // Wait for t1 to finish until continuing
        t1.join();
        System.out.println("Main thread ending");
    }
}
```

Example: Interrupting joiner and thread ending

In this example two threads are created to demonstrate interrupting thread joining and threads starting and ending at different times. The thread t0 sleeps for 100ms and interrupts the main thread before finishing its execution. The main thread's join is interrupted and it in turn interrupts the t1 thread, which finishes its sleep and also ends.

```
class JoinerThread2 extends Thread {

    private long time;      // Time to sleep
    private Thread main;    // Reference to main thread

    public JoinerThread2(Thread main, long time) {
        this.main = main;
        this.time = time;
    }

    public void run() {
        System.out.println(this.getName() + " going to sleep");
        try {
            Thread.sleep(time);
            System.out.println(this.getName() + " interrupts main");
            main.interrupt();
        } catch (InterruptedException ie) {
            // run() must catch IE, it cannot be declared to throw it
            System.out.println(this.getName() + " interrupted");
        }
        System.out.println(this.getName() + " ending");
    }
}

class JoinTester2 {

    public static void main(String[] args) {
        // Create thread with reference to main thread and 100ms time
        Thread t0 = new JoinerThread2(Thread.currentThread(), 100);
        t0.start();
        Thread t1 = new JoinerThread2(Thread.currentThread(), 2000);
        t1.start();
        try {
            // Wait for the thread that takes longer to finish; join

            // gets interrupted.
            t1.join();
        } catch (InterruptedException ie) {
            System.out.println(Thread.currentThread().getName() +
                               " interrupted on join");
        }
        // Interrupt the sleeping t1
        t1.interrupt();
        System.out.println(Thread.currentThread().getName() +

                           " ending");
    }
}
```

3.6 Stopping threads

Threads are usually required to stop their execution at some point. This can happen either naturally by completing the run() method, by changing the value of a possible loop to finish execution, interrupting the thread, or unexpected termination. If a thread is waiting to acquire an intrinsic lock (trying to enter synchronized block with the lock held by another thread), there is no stopping it until the lock is acquired. Use explicit locking (see chapter 4.2) when wanting to interrupt threads waiting for a lock.

3.6.1 Stopping the loop

The "normal" ways to stop threads, are either letting the run() method run its course and exit, changing the value used for the while loop within run(), or interrupting the thread. Interrupting the thread requires responsiveness to interruption, meaning making the while loop check for interruption status and possibly dealing with InterruptedException.

Example: Stopping thread using a variable

```
public class ThreadStopper implements Runnable {

    // Volatile ensures the visibility of changes made by different
    // threads.
    private volatile boolean running = true;

    public void run() {
        while (running) {
            System.out.println("Running");
        }
    }

    public void shutDown() {
        running = false;
    }

    public static void main(String[] args) {
        ThreadStopper r1 = new ThreadStopper();
        Thread t1 = new Thread(r1);
        t1.start();             //Start the new thread
        try {
            Thread.sleep(10);   // Caller (main) sleeps for a while
        } catch (InterruptedException ie) {}
        r1.shutDown();          // stop the thread
        System.out.println("set to false");
    }
}
```

ThreadStopper is pretty quick in stopping action after the running variable is changed. However, should there be lots of work to do within the loop, it could take a while for the thread to stop. Note that in some cases it is also possible that some thread might change the variable back again before the checking happens.

If there are blocking methods (sleep, wait, join etc) within the loop, it might take even longer for the thread to stop running. In this case it may be preferrable to interrupt the

thread instead of changing the variable, as this can be much faster depending on timing. Interrupting one of these blocking methods will result in an InterruptedException to be thrown.

Notice that Thread.stop() is deprecated and should not be used to stop threads. The method causes the release of locks, which can result in objects being exposed while in an inconsistent state, making data corruption possible. This makes it harder to properly deal with the consequences of using Thread.stop().

3.6.2 Interruption

Interruption is another way to stop thread execution, and sometimes it is the best way to do this. This is due to the possibility of the run() method having blocking methods, which might block thread execution for a long time before the possible while check is made again. When the thread's interrupt status flag is set to true, blocking methods like wait, sleep, and join throw InterruptedException automatically when starting to execute the respective method, or if the execution is already progress. The interrupt can be triggered by using myThread.interrupt() to set the interruption status true for another thread. For the current thread to set the status itself, Thread.currentThread().interrupt() should be used. To set the while loop responsive to interruption, you can use something like while(!Thread.currentThread().isInterrupted()).

InterruptedException can be either declared to be thrown in the method by using "throws InterruptedException", or it can be caught with a try-catch block. The run() method for both Thread and Runnable cannot be declared to throw the exception, as the overwritten run() method does not declare to throw it either. Methods called within run() can “throws” for exception propagation, but calling them forces run() (if not caught earlier) to implement the try-catch block, allowing centralized exception handling for the thread. Remember that setting a thread’s interruption status to true does not automatically throw InterruptedException, but only if there is a method that does so when encountering interruption.

When catching the interruption within methods possibly called by run(), the interruption status should be restored to allow the caller to decide on possible course of action. Restoring the status will again set the interrupt status as true, which may result in a new InterruptedException, should the thread encounter other blocking methods. Remember that interruption may cause shared Objects to be in an inconsistent state, which should be remedied when dealing with the interruption. Also document that the thread uses interruption for stopping it, especially if there is no public stopping method available for use. Otherwise it may be hard to know what the interruption does, at least without taking a closer look at the code, if that is even possible.

Example (incomplete): Catch restores interrupt status

```
// Think that this method is called from inside the run() method of a
thread
public void catchAndRestore() {
    // Calling wait() without synchronization would
    // throw IllegalMonitorStateException.
    synchronized(this) {
```

```
        try {
            // Wait inside while loop.
            while(true) {

                // Calling wait on "this
                wait();
            }
        } catch(InterruptedException ie) {
            System.out.println("IE caught inside XXX");
            // Reset the interruption flag; can be polled inside run()
            // May cause IE to be thrown, or might just be ignored.
            Thread.currentThread().interrupt();
        }
    }
}
```

Example: Allow interrupt through

```
class LetThrough implements Runnable {

    public void run() {
        try {
            sleeperMethod();
        } catch (InterruptedException ie) {
            System.out.println("Exception caught");
        }
    }

    // Declares to throw InterruptedException, so run() has to handle
    // it.
    private void sleeperMethod() throws InterruptedException {
        System.out.println("Go to sleep");
        Thread.sleep(1000);
        System.out.println("Not getting here due to exception");
    }

    public static void main(String[] args) throws InterruptedException
{
        LetThrough r = new LetThrough();
        Thread thread = new Thread(r);
        thread.start();

        Thread.sleep(100);

        thread.interrupt();
    }
}
```

If the thread is stopped by using interruption, the while loop or checks within it can poll for the thread's interrupt status. Checking the status is also a good way to enable exiting before doing long-running or hard-to-reverse-but-must tasks. Thread.currentThread().isInterrupted() checks the interrupt state and returns either true or false, while Thread.currentThread().interrupted() returns the value and sets the original to false.

Example: Polling for interruption

```
public class TheInterruptor implements Runnable {

    public void run() {
        // Note: Interrupting a thread does not throw IE
        // automatically, just sets the status.
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Running " + "interrupted = " +
                               Thread.currentThread().isInterrupted());
        }
        // Calling interrupted() also clears the state
        System.out.println("Thread was interrupted; interrupted = " +
                           Thread.currentThread().interrupted());
    }

    public static void main(String[] args) throws InterruptedException
{
        TheInterruptor r = new TheInterruptor();
        Thread thread = new Thread(r);
        thread.start();

        Thread.sleep(1);

        thread.interrupt();
    }
}
```

3.6.3 Unexpected termination

Unexpected thread termination can cause trouble for unexpecting programmers. If no way to deal with uncaught exceptions have been set, the failing thread will just do so quietly, maybe writing the cause to console. When dealing with multiple threads, it can be difficult to even notice when the thread fails, as the rest of the program chugs away without worry. Unexpected thread termination indicates either programming errors or other unrecoverable problems within the code.

Try-catch can be used to inform about expected problems and try-finally allows knowledge of the failure happening to pass on. This can be achieved by catching a Throwable in the catch block, setting a previously introduced value's pointer to it to preserve the object, and dealing with the problem in the finally block. The finally block is executed even in the case of an uncaught exception, or in the case of an early return, continue, or break. Still, something like the JVM exiting while inside the try-catch block may cause the finally block to fail to execute. Regardless, wrapping everything inside try-catches makes coding life a bit difficult.

Alternatively, uncaught exception handlers can be very useful to indicate thread failure. Implementing the interface java.lang.Thread.UncaughtExceptionHandler and setting it to either a specific Thread, ThreadGroup, or the Virtual Machine will allow the handling of uncaught exceptions. For uncaught exceptions, the Java Virtual Machine will first check the Thread's exception handler, and if one has not been set, it then checks the ThreadGroup's exception handler, going forward to the default uncaught exception handler if none of the previous have been set.

UncaughtExceptionHandler interface:

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

Example (incomplete): Setting UncaughtExceptionHandler for thread

```
Thread thread = new Thread();
thread.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println(t + " throws " + e);
    }
});
```

Example (incomplete): Setting **default** UncaughtExceptionHandler **for** all

```
Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
    public void uncaughtExceptionHandler(Thread t, Throwable e) {
    System.out.println("DEFAULT: " + t " throws " + e);
    }
});
```

Example: Creating handler and setting it to MyThread by default

```
// Create handler class to set to threads
class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {

    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("MUEH caught " + t + " throwing " + e);
        // The handler could for example log the message,
        // throw new RuntimeException, causing the program to crash,
        // try to restart the thread etc
    }
}

// Extending Thread and setting the UEH inside constructor
class MyThread extends Thread {

    public MyThread() {
        setUncaughtExceptionHandler(new MyUncaughtExceptionHandler());
    }

    @Override
    public void run() {
        int[] values = {1, 2, 3};

        // NOTICE: ArrayIndexOutOfBoundsException imminent
        for(int i = 0; i < 4; i++) {
            System.out.println(values[i]);
```

```
        }
    }

    public static void main(String[] args) throws InterruptedException
{
        Thread thread = new MyThread();
        thread.start();
    }
}
```

3.6.4 Dealing with failure

It is important to ensure that shared objects are not left in an inconsistent state when threads are stopped. Checking interruption can be done within code so that the code terminates before large changes have been made. There are multiple ways to deal with exceptions, which include more than just InterruptedException.

Ignoring exceptions and abrupt termination are somewhat harsh ways to deal with failure, but they are often used when testing examples and not caring about the consequences. In real programs this approach may obviously result in inconsistent objects and saved data, so it is important to have some other sort of policy in place when encountering errors. Abrupt termination is still useful for debugging purposes to indicate programming errors, or when the handled objects are just discarded after failure. Dealing with copies of the real objects might also allow to ignore the work already done.

Rolling back, retrying, and rolling forward are also ways to deal with failure. Depending on the changes made prior to failure, it may be possible to roll back the changes so that object states have been reset. Retrying means trying to redo the operation, possibly multiple times with varying pauses between retries. Roll-forward just means completing the operation while trying to ignore the effects of the failure.

It is also possible to provide the user the possibility to choose the course of action on failure. For example, a dialog window may inform of the problem and allow the user to retry or cancel action.

3.7 Guarded methods

When using multiple threads, it is often needed for the threads to coordinate their actions. Using guarded blocks is the most common coordination idiom, and it requires one or more conditions to become true before execution can proceed. Sometimes just throwing an exception when the checked precondition fails can be useful, but it is often preferable to wait for the condition to become true. Methods like wait(), notify(), sleep() and interrupt() can be used to implement this functionality. Both waiting and notifying must be called under synchronization, as they are meant to be used for communication between threads. Failing to do so will result in an IllegalMonitorStateException to be thrown.

3.7.1 Wait and notify

Wait and notify are used for message passing between threads, allowing threads to wait for other threads to do something, before proceeding again. There are three wait methods and two notify methods available for use in the java.lang.Object package. Both are called on objects that are being used as locks, so that wait() calls the lock on current object, while for example lock.wait() works on a self-made lock object. Calling wait() without using multiple threads is obviously a bad idea, because there is nobody to wake up the thread again.

Waiting can be used when a thread needs to wait for other threads to fulfil conditions that are necessary for it to proceed. Wait suspends the thread holding the relevant lock, which is then released for other threads to acquire. All other locks held by the thread are not released, so take care when using nested locking. Waiting should also always be done in a while loop, as otherwise waking up mistakenly, or with notifyAll(), will cause the code to continue regardless of the situation.

Notify is also called on the same object that was used as the lock, so that thread(s) waiting (not blocked) on the lock will wake up and try to reacquire the lock. Method notify() wakes up one waiting thread chosen by the JVM, while notifyAll() wakes up all of them. Remember that the awakened threads will have to contend on the lock like normal, so there is no guarantee on which thread will acquire the lock next.

Wait methods:

```
public final void wait() throws InterruptedException
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws
InterruptedException
```

The wait() method does not have a timeout, so it just waits until a notify() or notifyAll() is called on the lock, or either an InterruptedException or a spurious wakeup happens. Due to possibly mistaken notify, interruption, spurious wakeup, or the condition changing back again before lock acquisition, wait method should always be placed in a while loop to avoid code execution continuing unintentedly. Synchronize the whole loop to keep the value of the condition up-to-date. Timed waits also have the option of waking up without external prompt, should the time limit have elapsed. This allows the

possibility of dealing with unexpected liveness problems by allowing the waiting thread to back out, and therefore avoiding possibly infinite unresponsiveness, while allowing the programmer to decide on how to proceed.

Remember to keep object states consistent when entering wait, as the relevant lock is released when waiting. Therefore there is a possibility of other threads accessing partially changed objects, or changing some (or all) of the values, if there are multiple changes that are not made atomically. In this case it would mean making some changes before the wait and other related changes after it, affecting the result of the operation possibly negatively.

Wait examples (incomplete):

```
// Sample code without catching the possible exception.
// wait() releases the lock it is called on, which is "this" here
public synchronized void waitExample() throws InterruptedException {
    // Use loop to check when condition becomes true
    while (!condition) {
        // Wait indefinitely or for notify etc. to recheck condition.
        // Thread interruption also ends loop
        wait();
    }
    // Code to be executed after condition is true
}

// Sample code with timing, self-made lock and catching the exception
private final Object lock = new Object();
        ...
public void waitExample() {
    synchronized(lock) {
        while (!condition) {
            try {
                // Wait for 100 millisecs, notify to recheck condition
                lock.wait(100);
            } catch (InterruptedException ie) {
                // Handle exception; ignore to redo while-check
            }
        }
        // Code to be executed after condition is true
    }
}
```

Nested lock example; getting stuck:

```
class NestedWait extends Thread {

    // Lock for the whole class
    private static final Object lock = new Object();
    // Instance lock for each object; public not good practice, though
    public final Object instanceLock = new Object();
    private static volatile boolean value = false;

    public void run() {
        // This lock is never released by wait
        synchronized (lock) {
            // This lock gets released
            synchronized (instanceLock) {
```

```
                while (!value) {
                    try {
                        instanceLock.wait();
                    } catch (InterruptedException ie) {
                        System.out.println("Wait interrupted");
                    }
                }
                System.out.println("got out of wait");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException
{
        NestedWait thread = new NestedWait();
        thread.start();
        Thread.sleep(10);

        System.out.println("Trying to acquire static lock");
        synchronized(lock) {
            synchronized(thread.instanceLock) {
                System.out.println("Change value and notify thread's
lock");
                value = true;
                thread.instanceLock.notifyAll();
            }
        }
    }
}
```

```
Notify methods:

public final void notify()
public final void notifyAll()
```

The notify() method wakes up a single thread waiting on the object's monitor (lock), while the notifyAll() does so for all threads waiting on the monitor. The awakened thread will then content with other possible threads for acquiring the lock and continuing code execution. Note that the lock can be acquired by the released thread only after the notifying code has exited the synchronized block. Should many threads be awakened to contend for the lock, and the first one changes their waiting condition back to false, the other threads will acquire their locks one at a time, and go back to waiting. Of course, it is possible that another thread will again change the condition again back to true before they get their turn to acquire the lock.

It is recommended to use notifyAll() (remember to wait in loop) in the beginning and change to use notify() if it is necessary. This makes it easier to verify that code works properly before other functionality and optimization have been made. With notify() it is possible that there will be missed signals, should there be multiple unrelated recipients for the notification, and a wrong one gets awakened. Also, notifyAll() can be quite heavy to use in comparison to notify(), as it may wake up a large number of waiting threads, which will result in context switches.

Notify examples (incomplete):

```
// Sample code changing the condition and possibly waking up one
// waiting thread.
public synchronized void notifyExample() {
    condition = true; // Change the condition to exit while loop above
    notify();         // Wake up one thread waiting on this object
}



// Sample code with self-made lock and waking up all waiting threads
public void notifyExample() {
    synchronized (lock) {
        condition = true;
        // Wake up all waiting threads; race to acquire lock
        lock.notifyAll();
    }
}
```

3.7.2 Wait interruption

If you need to end waiting without proceeding with the code, it is possible to interrupt the waiting thread. Like most blocking operations (sleep, join, etc), waiting also responds to thread interruption by throwing InterruptedException. The most common way to respond to interruption is to terminate the thread, and it is a good practise to do this, even if it is not really required.

Calling Thread.interrupt() sets the thread's interrupt status flag to true, which causes the wait method to throw InterruptedException. If the wait is ongoing, the exception is thrown immediately, but otherwise it is thrown when encountering the call to wait. It can be enough to use break to exit the while loop upon catching the interruption, but sometimes additional clean-up is required to ensure that objects stay consistent. Sometimes it is preferred to not to catch the exception, but to propagate it to higher levels of call stack for handling. Testing interruption status should also be used with non-blocking long-running loops, to add responsiveness to possible interruption.

Note that Runnable's run() method needs to catch InterruptedException, as it is a checked exception, that cannot be declared to be thrown (i.e. throws InterruptedException). In this case, the methods called inside run() can be allowed to throw the exception, and the handling can be centralized to happen via the run() method. If your class is used by others, throwing the InterruptedException instead of catching it, allows the calling code to handle the exception. Even if your code does catch the exception, it is still a good practice to restore the interrupted status by calling Thread.currentThread().interrupt(), to allow calling code to check if interruption has happened. Just beware possible infinite interruption recatching when dealing with loops.

Example: Wait interruption

```
class Waiter implements Runnable {

    public void run() {
        System.out.println("thread running");

        synchronized(this) {
            while (true) {
                try {
                    wait();
                } catch (InterruptedException ie) {     // catch IE
                    System.out.println("Exception caught, exit loop");
                    break;      // break the loop
                }
            }
        }
    }
}

class WaitInterruptTest {
    public static void main(String[] args) throws InterruptedException
{
        Thread t = new Thread(new Waiter());
        t.start();
        // Main thread sleeps for 100ms, may throw IE
        Thread.sleep(100);
        System.out.println("Interrupting t");

        // Interrupt thread t
        t.interrupt();
    }
}
```

Example: Wait interruption propagation

Class Propagator has two methods, one of which catches an InterruptedException that may happen if wait is interrupted. The other declares that the exception may happen and just lets it go through.

PropagatorRunnable uses the Propagator class methods and deals with them appropriately. For the catchAndPropagate() method, it tests the interrupted status of the thread and sets it back to true. The letItGoThrough method's possibility of throwing an uncaught IE forces the use of try-catch to surround the method call in run().

Note that for normal programs the exception may require to deal with the handling of data modifications or other possible repercussions rising from the interruption. Depending on the situation, not catching the InterruptedException may be preferrable, as it forces the user of the class to note the possibility of an exception happening.

```
class Propagator {

    public void catchAndPropagate() {
        synchronized(this) {
            // Test interruption via indefinite wait
            while (true) {
                try {
                    wait();
                } catch (InterruptedException ie) {
                    System.out.println("IE caught in Propagator");
                    // Reset the interruption flag
                    Thread.currentThread().interrupt();
                    // No break? -> an eternal IE loop.
                    break;
                }
            }
        }
    }

    public void letItGoThrough() throws InterruptedException {
        synchronized(this) {
            // Test interruption via indefinite wait
            while (true) {
                wait();
            }
            // Without catching, interruption ends loop due to IE
        }
    }
}

class PropagatorRunnable implements Runnable {

    public void run() {

        Propagator p = new Propagator();

        // Handles IE, so no need to catch here unless rethrown
        p.catchAndPropagate();

        // isInterrupted() checks status, but does not change value
        System.out.println("interruption status "
                           + Thread.currentThread().isInterrupted());

        // interrupted() checks the flag, sets it to false
        System.out.println("interruption status is still "
                           + Thread.currentThread().interrupted() +
                           " but after interrupted() " +
                           Thread.currentThread().isInterrupted());

        try {
            // Can throw uncaught IE, so forces run() to catch it
            p.letItGoThrough();
        } catch (InterruptedException ie) {
            System.out.println("Exception caught in " +
                               "PropagatorRunnable, catch sets flag to " +
            Thread.currentThread().isInterrupted());
        }
    }
}
```

```
class PropagatorTester {


    public static void main(String[] args) throws InterruptedException
{
        Thread t = new Thread(new PropagatorRunnable());
        t.start();
        // Give time for the first method to run, but IE still
        // triggers even if the interruption happens before wait().
        Thread.sleep(10);

        // Interrupts while in catchAndPropagate infinite loop
        t.interrupt();
        // Give time for second method to run before interruption
        Thread.sleep(10);
        // Interrupts while in letItGoThrough infinite loop
        t.interrupt();
    }
}
```

3.8 Immutability

When using variables that do not need changing, or you wish to avoid locking, immutable variables may be able to help. Immutable variables and objects cannot be modified after initialization, and can be shared across threads without worry (but objects need safe publishing), as the visibility and atomicity are guaranteed. This allows for easier reasoning about the state of the program, and can reduce the complexity by reducing synchronization. The use of "final" keyword indicates immutability, though with references it is still possible that the reference points to a mutable object. Using immutability with primitives (int, double, etc.) is easy, as they cannot be modified later, but objects can be a bit more complicated.

3.8.1 Immutable objects

To be immutable, an object's state must not be modifiable after construction. All fields should be declared final to prevent unintended changes to them, and the object must also be properly constructed. This requires safe publication, so that the "this" reference cannot escape during construction. Otherwise it is possible for other threads to access the partially constructed object, seeing some or all values as the default ones. Starting a thread from within the constructor is especially dangerous and should be avoided.

According to Oracle's concurrency tutorial there are four rules for creating a simple strategy for immutable objects (https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html). These rules are the easiest way to make objects immutable, but not all are necessarily needed in all cases. First rule is not providing setter methods, so that modification of fields and objects is not possible. Second is making all fields final and private. Third disables the possibility of subclasses overriding methods, which is easiest to achieve by declaring the class as final. Fourth is that if instance fields refer to mutable objects, then those objects should not be allowed to be modified. This includes both not allowing methods to modify the objects, and not sharing references to the mutable objects with others, though passing copies (remember deep copy) of them is still allowed.

Although immutable references might stay the same, the referenced objects might not be immutable and thus be subject to change. It is also possible to point to an immutable object with a mutable reference, so that the reference can be pointed to a new object when change is needed. When properly used, these features can help to achieve thread safety, while providing efficient access to objects by reducing unnecessary synchronization. However, creating new objects every time change (or copy of a mutable object) is needed takes time and resources, so without sufficient contention on the locks that were replaced, performance can become even worse. The effect to performance is hard to ensure without proper testing with realistic loads.

Examples: Immutable object

```
// Without stored fields in the class, there is no danger of change.
// Beware subclassing, though.
class ImmutableClass {
    public int getMultiplied(int a, int b) {
        return a * b;
```

```
    }
}

// Making the class final makes subclassing impossible.
final class ImmutableClass2 {
    private final int a;

    public ImmutableClass2(int a) {
        this.a = a;
    }

    public int getA() {
        return a;
    }
}
```

3.9 Safe publication

Safe publication means ensuring that objects that are shared with other threads are not seen in an inconsistent state by said threads. Publishing partially constructed objects can cause some or values to be seen in a wrong state. Note that safe publication deals with only the state of the object when published, and possible later changes can still cause trouble.

3.9.1 Publishing immutable objects

Immutable objects are safe to publish to other threads, if they are properly constructed, have unmodifiable state, and all their fields are final. Proper construction here means that the object constructor is not shared with other threads from inside the constructor. In this case, immutable fields (and the possible objects referred to within) cannot be seen in an inconsistent state.

3.9.2 Publishing mutable objects

With mutable objects there are other ways to ensure safe publication. These include guarding the object state properly with a lock, using a volatile field (or an AtomicReference) to refer to the object, using a static initializer to publish the object, or using thread-safe collections for sharing. Synchronization on creation and access is an easy way to ensure safety, but remember that constructors cannot be synchronized. However, without proper measures, changes made to an object's fields even after safe publishing may or may not be visible to other threads.

3.9.3 Leaking object within constructor

When constructing objects, it is important to remember not to leak a reference to the object while inside constructor. For example, adding the object to a list accessible to other threads, or registering it as a listener can share the object with other threads before construction has finished. Even if the sharing is done on the final line of constructor, and the compiler has not optimized code by rearranging operations, possible subclasses will still need to call super() on the first line of the constructor (or it is called automatically, if there is a no-args constructor available), making the object partially constructed while the reference has already leaked.

3.10 Documentation

Proper documentation is an important part of creating thread-safe classes. Besides allowing programmers to know if the class it thread-safe, it also allows easier maintenance and modification. Remembering what is protecting what becomes a lot easier with documentation. Using clean and easily understandable code is also an important way to reduce problems and avoid unnecessarily extensive documentation.

Documenting a class's synchronization policy is also important for subclassing, so that it is possible to use the same policy with the superclass if needed. It should also be documented whether the policy has been locked so that subclasses do not need to worry about changes. Without knowledge of this, it may be preferable to implement your own synchronization policy for your subclasses, so that the code does not just silently break when changes are made to superclass.

4 Advanced Topics

4.1 Liveness & performance

Even if thread-safety has been achieved for a program, there can still be concerns for liveness. The execution of the program should be able to make progress in a timely manner, but sometimes it is slowed down either by bad design, programming errors, or excessive (or wrongly implemented) safety measures. Too much locking, long waits, deadlocks, etc. can cause the program, or parts of it, to slow down to a crawl. Deadlock, livelock, starvation, nested monitor lockout, missing signals, too much locking etc. can all cause programs to slow down, fail or freeze.

Due to this, many changes and optimizations may be required for the program to work more efficiently. However, this can take a lot of time and endanger thread-safety, while also making maintenance harder. Due to this, programs should be first made to work right and leave optimization for later, if it is necessary at all. Many programs do not need performance tuning, even if they use multiple threads, as the real-world impact of slow choices may be negligible.

4.1.1 Thread liveness

Deadlock happens when two or more threads require the use of same locks at the same time, with each holding some while needing those that others have. For example, a thread holds lock A while trying to acquire B, and another thread holds lock B while trying to acquire A. Neither thread can make progress while having to wait to acquire the other lock infinitely. Note that deadlock is also a timing-related problem, so it might not be easy to notice without proper testing.

Livelock happens when two threads respond to each other's action with another action, which causes both to be unable to make progress. In this case neither thread is blocked, but they only have time to respond to each other's actions. For example, two AI opponents in a game may respond to each other's move by moving in the same direction, neither being able to execute other actions.

Starvation is the result of one or more threads hogging a resource often and usually for long times, so that other threads are unable to get use of it. Synchronized methods that take too long to execute may cause this kind of problem to happen. A denial of service attack may also cause starvation by preventing the real clients from accessing resources.

Nested monitor lockout happens when the waiting thread holds multiple locks upon entering wait, with only one being released, and the awakening thread needs both to wake it up. See the example NestedWait in chapter 3.7.1. Missed signals are also a wait-related problem, where the notify to wake up is issued only once, and the waiting thread starts waiting only after the signal has been sent.

4.1.2 Optimization

Using multiple threads usually means trading slightly poorer efficiency with better latency and scalability. This means that it is possible for single-threaded code to achieve better performance than its multithreaded counterpart in many situations. Of course, many programs benefit substantially from using multiple threads, for example, games and artificial intelligence.

As threading is often used for performance reasons, it may be tempting to try to optimize code from the get go, due to synchronization costing resources. Although optimizing code from the start might seem tempting, it is not recommended to do so for safety reasons. It is a good practice to first implement the code according to good practices and make optimizations later, if they are really necessary. Subclassing thread-safe classes may even normally be harder than normal, but proper structure is necessary for it to work.

Assumptions about the performance impact of threadsafe code can be hard to reason about. Due to this, it is advisable to test the code with realistic load and only make optimizations if necessary. When simplicity is compromised, testing becomes harder and there is a bigger chance for programming errors. Making code maintenance harder is also not a good thing to do, unless you really want to make things harder in the future.

Regardless, in some cases the performance benefits of even knowingly allowing errors can be allowed, if the results are not too harmful. For example, showing an old position for a moving object in a gps app may not hurt much, as long as the app has not updated one coordinate and not the other, so that the spot is entirely wrong. Also, small graphical or UI errors, that only appear for a short period of time, may sometimes neither impact user experience nor affect any important calculations.

Some basic optimization methods include reducing lock contention by using smaller synchronized blocks and changing to use non-blocking methods where possible. Immutability, read-write locks, and atomic variables can also be used to reduce synchronization. Also check java.util.concurrent package for more tools.

4.2 Explicit locking

The package java.util.concurrent.locks includes interfaces for Lock, ReadWriteLock and Condition, in addition to a bunch other of classes. This chapter introduces the Lock interface using the class ReentrantLock. This lock provides additional features that are not available for intrinsic locking, but it is not meant to replace intrinsic locks. Use explicit locks when you need their additional features, but otherwise prefer intrinsic locks (synchronized) for their simplicity.

Locks allow programmers to use unconditional, polled, timed, and interruptible lock acquisition mechanisms. Methods lock() and unlock() handle the acquisition and release of the locks. It is recommended to lock the lock just before the try-finally block, and to unlock the lock in the finally block to ensure lock release. This should release the lock in practically every case, with the exception of, for example, the JVM exiting unexpectedly.

Explicit lock usage:

```
Lock lock = ...;
lock.lock();
try {
    // ...
} finally {
    lock.unlock();
}
```

Lock interface:

```
public interface Lock {
    // Acquires the lock
    void lock();
    // Acquires lock unless the thread is interrupted
    void lockInterruptibly();
    // Returns a new Condition, bound to this Lock instance
    Condition newCondition();
    // Acquires the lock if it is free when trying
    boolean tryLock();
    // Acquires the lock if it is free within the time limit and the
current
    // thread has not been interrupted.
    boolean tryLock(long time, TimeUnit unit);
    // Releases the lock
    void unlock();
}
```

Remember not to use the intrinsic lock of objects that implement Lock and/or condition. Doing so will not lock on the same object as when using the lock() method, which will obviously cause problems. Also remember to release the lock when it is no longer needed - something that can be forgotten due to intrinsic locks releasing automatically.

4.2.1 ReentrantLock

Acquiring and releasing a ReentrantLock (java.util.concurrent.locks.ReentrantLock) works essentially like entering and exiting a synchronized block. However, with ReentrantLock it is possible to interrupt lock acquisition and back out if the lock is not available on time. There is also no need to release the lock in the same block of code. It is also possible to give the lock an additional fairness parameter to favour the longest-waiting thread on acquisition; though another thread will still acquire it first, if the lock happens to be free when requested.

Example: Basic usage of ReentrantLock

```
Lock lock = new ReentrantLock();
// ...
lock.lock();
try {
    // ...
} catch (IOException e) {
    // Catch what you need; catch can often be optional
} finally {
    lock.unlock();
}
```

Example: Timed acquisition of ReentrantLock

TimedAcquisition demonstrates timed tryLock(long time, TimeUnit unit) with ten threads (with DoIts as their Runnables), each trying to call increment(). The threads will block until they either get the lock, interruption happens (last one interrupted in this example), or their waiting time ends. The increment() method waits for a while to get some of the threads time out on tryLock. Note that the catch(InterruptedException ie) before the finally block is not necessary here, as the outer catch encompassing the lock.tryLock as a whole, would catch it.

```
import java.util.concurrent.locks.*;
import java.util.concurrent.TimeUnit;

public class TimedAcquisition {

    private int counter = 0;
    // Create the lock
    private final Lock lock = new ReentrantLock();

    public void increment() {
        try {
            if(lock.tryLock(10, TimeUnit.MILLISECONDS)) {
                try {
                    counter++;
                    System.out.println("Incremente to " + counter +
                    " by " + Thread.currentThread());
                    // Sleeps to decrease time for others
                    Thread.sleep(2);
                } catch (InterruptedException ie) {
                    // In some cases it may be needed to roll back
                    // (or otherwise handle) the state changes, if
```

```
                    // interrupted during work, to prevent
                    // inconsistent object state
                    // (only part of work done).
                    System.out.println("sleep interrupted");
                } finally {
                    // Finally is entered even in case of an uncaught
                    // exception.
                    // Unlock in the finally block!
                    lock.unlock();
                }
            } else {
                // Deal with time ending here
                System.out.println("lock not acquired within time");
            }
        } catch (InterruptedException ie) {
            // Deal with possible tryLock interruption: retry? etc.
            System.out.println("tryLock interrupted for " +
                               Thread.currentThread());
        }
    }

    public static void main(String[] args) {
        TimedAcquisition ta = new TimedAcquisition();
        ta.go();
        // Main thread ends here, but the others still finish.
    }

    private void go() {
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new DoIt(this));
            t.start();
            if (i == 9) {
                // Testing interruption on timed tryLock
                t.interrupt();
            }
        }
    }
}

// Class for the Runnables; works just as fine as an inner class
public class DoIt implements Runnable {
    TimedAcquisition master;

    public DoIt(TimedAcquisition master) {
        this.master = master;
    }

    public void run() {
        master.increment();
    }
}
```

4.3 Blocking Queue

Blocking queues can be used for implementing the producer-consumer pattern, allowing some thread(s) to produce tasks for other threads to run. This is usually used with thread pools (see chapter 4.4.3), but can also be implemented via the use of basic Runnables, along with the java.util.concurrent.BlockingQueue framework. The basic idea of blocking queues is putting tasks into the queue, possibly blocking if there is no more room, and taking those tasks from the queue to execute, again possibly blocking if no tasks are available. Whether the add/take blocks or fails is up to the methods used and their implementation.

The BlockingQueue interface has methods like put(E e), offer(E) and offer(E e, long timeout, TimeUnit unit) for offering tasks to the queue, along with take() and poll(long timeout, TimeUnit unit) for retrieving tasks from it. See the other methods from Java documentation, along with the different implementations of interface (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) . The BlockingQueue implementations are thread-safe, with the bulk Collection operations possibly being non-atomic.

Example: Using ArrayBlockingQueue with producer and consumer threads

```
// The class produces numbers and puts them into the given queue
import java.util.concurrent.BlockingQueue;

class ProducerRunnable implements Runnable {
    private BlockingQueue<Integer> queue;

    public ProducerRunnable(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                queue.put(i);
                System.out.println("put " + i);
                Thread.sleep(300);
            } catch (InterruptedException e) {}
        }
    }
}

// This class takes numbers from given queue and prints them.
import java.util.concurrent.BlockingQueue;

class ConsumerRunnable implements Runnable {
    private BlockingQueue<Integer> queue;

    public ConsumerRunnable(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for(int i = 0; i < 5; i++) {
            try {
```

```
                int a = queue.take();
                System.out.println("taken " + a);
                Thread.sleep(500);
            } catch (InterruptedException e) {}
        }
    }
}

// Creates BlockingQueue, producer and consumer
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;

class ProducerConsumerTest {

    public static void main(String[] args) {
        BlockingQueue<Integer> queue =
            new ArrayBlockingQueue<Integer>(5);

        new Thread(new ProducerRunnable(queue)).start();
        new Thread(new ConsumerRunnable(queue)).start();
    }
}
```

4.4 The Executor Framework

Executors allow programmers to handle thread management with different frameworks. In addition to Executors handling Runnable (another reason to support the use of Runnable), tasks called Callable can also be handed over. Unlike Runnable, Callable allows the return of values, which can be retrieved by using Future objects.

4.4.1 Executors

The Executors interfaces Executor, ExecutorService and ScheduledExecutorService can be found from the java.util.concurrent package. The Executor interface handles Runnable objects, the ExecutorService interface both Runnable and Callable objects, and the ScheduledExecutorService interface extends ExecutorService by supporting the scheduling of tasks. All of the interfaces have ready implementations made in the java.util.concurrent package, but it is also possible to make your own.

4.4.2 The Executor interface

Executor interface is the simplest one and has the method execute(Runnable command), which just basically replaces the “new Thread()” call. The implementation of the executor, however, decides on how to run the thread. The given Runnable might, for example, be run in a new thread, an existing worker thread, the current thread, or be put in a queue to await execution. Due to this, changing execution policy by changing the executor in use is also easy.

Example: Implementing simple Executor

```
import java.util.concurrent.Executor;

class MyExecutor implements Executor {

    @Override
    public void execute(Runnable r) {
        // This executor just starts a new thread for each task.
        new Thread(r).start();
    }

    public static void main(String[] args) {
        MyExecutor e = new MyExecutor();
        e.execute(new Runnable() {
            public void run() {
                System.out.println("Run via Executor");
            }
        });
    }
}
```

4.4.3 The ExecutorService interface

The ExecutorService interface extends the functionality of Executor and is the more advanced version, providing methods for lifetime management and allowing the use of Future in addition to Runnable. The ExecutorService has three submit methods (and

invoke variants for multiple tasks), which are used to give it tasks, and methods shutdown() and shutdownNow() for stopping the service, along with other utility methods.

Remember that the service should be shut down after it's no longer needed, so that the resources can be released. Failing to do so may prevent the JVM from exiting due to live threads. The method shutdownNow() also requires for the tasks to be responsive to interruption to ensure task termination. Submitting new tasks to an ExecutorService that has been ordered to shut down can cause RejectedExecutionException to be thrown. Using the method awaitTermination() will cause the current thread to wait for the ExecutorService to finish, but isTerminated() can also be repeatedly polled for the same effect, with the exception of easily setting a time limit for the wait.

```
Some of the methods (see
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executo
rService.html ):
// Submits a Callable, which returns a Future for getting the result
submit(Callable<T> task)
// Submits a Runnable, which returns a Future for getting the result.
// For example, you can pass the result reference to Runnable
// constructor to modify the value from within the Runnable.
submit(Runnable task,T result)
// Submits a Runnable, returning a Future returning a null on
//successful completion.
submit(Runnable task)
// Allows previously submitted to be executed, not accepting new ones,
// and shuts down.
shutdown()
// Attempts to stop executing tasks, halts waiting tasks, returns a
// list of awaiting tasks.
// Does not guarantee stopping tasks; e.g. not responsive to
// interrupt? may not terminate.
shutdownNow()
// Blocks until tasks have completed execution, the timeout passes, or
// current thread is interrupted. Does not replace shutdown() or
// shutdownNow().
awaitTermination(long timeout, TimeUnit unit) throws
InterruptedException
```

4.4.4 Callable and Future

Callable is a task that can be run by ExecutorService interface implementations by calling submit(task) on the executor service, which causes the Callable method call() to be executed when given the chance by the executor. Unlike Runnable's run(), the method call() is declared to throw an Exception, which lets the programmer propagate the exception itself back to caller. Callable also allows to return a result, which can then be retrieved with the use of Future.

Callable methods:

```
// Returns the value type and can throw exceptions of your choice
V call() throws Exception
```

A Future can be used to get the result of a computation. The interface has methods for getting the result, cancelling the task execution and checking the task status. When cancelling tasks while using thread pools, remember that you should not interrupt a thread itself, but cancel the task you wish to stop via the task's Future. When directly interrupting a pool thread, it is not known what task is running at the time.

Future methods:

```
// Attempts to cancel task execution, but requires responsiveness to
// interruption if the task is already running. May be set to try
interruption
// (true) or cancel only if not started.
boolean cancel(boolean mayInterruptIfRunning)
// Waits for completion and retrieves the result
V get()
// Waits until result comes or timeout expires, and gets the possible
result
V get(long timeout, TimeUnit unit)
// Returns true if task was cancelled before completion
boolean isCancelled()
// Returns true if task has completed
boolean isDone()
```

Example: Simple use of Callable and Future

```
import java.util.concurrent.Callable;

class MyCallable implements Callable<Integer> {

    int a;
    int b;

    public MyCallable(int a, int b) {
        this.a = a;
        this.b = b;
    }
    // Returns Integer and can throw any Exception unchecked.
    @Override
    public Integer call() throws Exception {
        return a * b;
    }
}

import java.util.concurrent.Future;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ExecutionException;

class CallableFutureTest {

    public static void main(String[] args) throws
InterruptedException,
```

```
                                                        ExecutionException {
        CallableRunnableTest crt = new CallableRunnableTest();

        ExecutorService exe = Executors.newSingleThreadExecutor();
        // Construct the callable
        MyCallable mc = new MyCallable(5, 6);
        // Submit task and return value to Future f; IE & EE possible
        Future<Integer> f = exe.submit(mc);
        // Get the value, block until finished;
        int a = f.get().intValue();
        System.out.println(a);
        // Remember to shut down the pool
        exe.shutdown();
    }
}
```

4.4.5 The ScheduledExecutorService interface

This interface is an extension of the ExecutorService interface and allows the use of delay for executing given Runnable and Callable tasks. Repeated task execution at fixed rate or delay is also possible with the use of scheduleAtFixedRate and scheduleWithFixedDelay methods. If you wish to replace java.util.Timer, which is used in many apps that need basic timing functionality, using this interface's implementation ScheduledThreadPoolExecutor (also extends ThreadPoolExecutor) allows you to create a more versatile version with support for multiple threads. This is especially useful with frequent and long-running tasks due to Timer limitations.

4.4.6 Thread Pools

Thread pools, which use worker threads to execute tasks, are the most common implementation of executors. The java.util.concurrent package holds multiple ready-made thread pool implementations, with different thread management policies. The basic functions of thread pools are assigning tasks for pooled threads, recycling these worker threads to minimize thread creation overhead, and defining the number of threads usable simultaneously.

Examples: thread pool creation

```
// Create a thread pool with 1 thread, and unbounded queue for tasks.
// If the thread dies before shutdown, a new one will be created for
new tasks.
ExecutorService executor = Executors.newSingleThreadExecutor();
// Create a thread pool with 5 threads; has an unbounded queue for
tasks.
// With all the threads running simultaneously, if more tasks are
// submitted, they will be queued while waiting for a free thread.
ExecutorService executor = Executors.newFixedThreadPool(5);
// Creates new threads for tasks when needed, but also reuses free
ones.
// Kills threads not used for 60 secs; can get heavy fast with lots of
tasks.
ExecutorService executor = Executors.newCachedThreadPool();
```

Choosing which thread pool to use can have a large effect on performance. If lots of tasks are submitted and a cached thread pool is used, the threads will spend a lot of time for context switches, and too many threads may cause the application to crash. Limiting the number of threads and queueing the additional tasks will be lighter, but even so it may eventually be required to reject new tasks with heavy continuous loads. The number of available cores (and other tasks) affects the optimum number of threads to use, and it is possible to tune the thread pool to accommodate that number.

Example: Simple thread pool test

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

class SingleThreadExecutorTest {

    // Throws IE is for the awaitTermination method
    public static void main(String[] args) throws InterruptedException
{
        ExecutorService executor =
Executors.newSingleThreadExecutor();
        // ExecutorService executor = Executors.newCachedThreadPool();
        // ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 10; i++) {
            executor.submit(new Runnable() {
                public void run() {
                    for (int i = 0; i < 10; i++) {

System.out.println(Thread.currentThread().getName() +
                                                    " " + i);
                    }
                }
            });
        }
        executor.shutdown();
        System.out.println("Allow previously submitted tasks to
finish");
        // executor.shutdownNow();
        // System.out.println(" sdN prevents new starts, try to stop
runners");
        executor.awaitTermination(5, TimeUnit.MILLISECONDS);
        System.out.println("Main thread ends");
    }
}
```

5 Resources Used for This Study Material

This material has been created by using multiple books, web material, and my own testing on the different aspects of multithreading in Java. The following resources had the largest impact on this material, and I recommend buying and reading the books in question for more information on the subject.

Java Concurrency in Practice by Brian Goetz

Concurrent Programming in Java: Design Principles and Patterns (2nd edition) by Doug Lea

The Well-Grounded Java Developer (Chapter 4, Modern Concurrency), by Benjamin J. Evans, Martijn Verburg

Java Documentation

Oracle. The Java Tutorials: Concurrency
https://docs.oracle.com/javase/tutorial/essential/concurrency/

Carnegie Mellon University, Software Engineering Institute, Confluence pages related to Java multithreading (Rules 8-12):
https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java