UNIVERSITETET I OSLO
Institutt for informatikk

Sammenligning av generiske mekanismer i Java og C#

Masteroppgave
(30 studiepoeng)

Haakon Peder Haugsten

23. mai 2006

Forord

Først vil jeg takke min veileder, Birger Møller-Pedersen, for verdifull veiledning.

Jeg ønsker også å takke min arbeidsgiver Transact AS og spesielt Knut Jonassen for fleksibel arbeidstid slik at det har vært mulig å gjennomføre denne oppgaven.

Til slutt vil jeg takke Agata og resten av min familie for støtte, tålmodighet og hjelp med oppgaven. Nå er den endelig ferdig.

Oslo, 23. mai 2006

Haakon Peder Haugsten

Innhold

1 Innledning **1**

1.1 Om oppgaven 2

1.2 Språket, eksemplene og koden 2

2 Om generiske mekanismer **4**

2.1 Generiske mekanismer 4

2.2 Generiske pakker og funksjoner i Ada 5

2.3 Templates i C++ 6

2.4 Virtuelle klasser 7

2.5 Det generiske idiom 8

2.6 Implementasjon av generiske mekanismer 10

2.7 Generiske problemområder 11

2.7.1 Problemer med generisk varians 11

2.7.2 Rekursivt avhengige klasser 13

3 Generiske mekanismer i Java og C# **14**

3.1 Java 14

3.1.1 Generiske mekanismer i Java 5 15

3.1.2 Beskrankning i Java 16

3.1.3 Jokernotasjon 17

3.1.4 Raw types 18

3.1.5 Utvidelser og endringer av Javas klassebiblioteket . . 18

3.1.6 Implementasjon 19

3.2 C# 20

3.2.1 Generiske mekanismer i C# 21
3.2.2 *Where condition* 21
3.2.3 Alias 22
3.2.4 Klassebiblioteket 22
3.2.5 Implementasjon av generiske mekanismer i C# 23

4 Sammenlikning **24**
4.1 MyGenCollection 24
4.2 Ulikheter og problemer 25
4.2.1 Generiske arrayer 25
4.2.2 Programmering med statiske variable og metoder . . 27
4.2.3 Nestede og indre klasser 28
4.2.4 Aritmetiske operasjoner 30
4.2.5 Generisk varians 31
4.2.6 Typesikkert? 32
4.2.7 Kodens lesbarhet 33
4.3 Testing av effektivitet 34
4.3.1 Resultater 35

5 Oppsummering **37**
5.1 Videre arbeid 39

Bibliografi **39**

Tillegg **43**

Figurer

2.1 Eksempel på generisk kode 5
2.2 Generisk funksjon i Ada 6
2.3 *Templates* i C++ 7
2.4 Virtuelle typer i Beta 8
2.5 Stakk i Java, *det generiske idiom* 9
2.6 Kovarians 12
2.7 Generiske typer med flere parametere 13

3.1 Generiske mekanismer i Java 15
3.2 Beskrankning i Java 16
3.3 Jokernotasjon 17
3.4 *Raw type* 18
3.5 *Type erasure* i Java 20
3.6 Generiske mekanismer i C# 21
3.7 Beskrankning i C# 22
3.8 Alias i C# 22
3.9 Instansiering av generiske klasser i C# 23

4.1 Oversikt over *MyGenCollection* 25
4.2 Generiske arrayer i Java 26
4.3 Arrayer av generiske typer i Java 26
4.4 Generisk klasse med en statisk variabel i C# 28
4.5 Nestede og indre klasser i Java 29
4.6 Nestede klasser i C# 30
4.7 Summering av en liste med tall i Java 31

4.8 Metode med kovariante typeparametere 31
4.9 Typesikker? 32
4.10 *Where*-betingelser i C# 33
4.11 Metode i C# for hastighetstest av generisk stakk med *int* . . 34

Tabeller

4.1 Generisk vs. generisk idiom i C# (ms) 35
4.2 Generisk vs. generisk idiom i Java (ms) 36
4.3 Generisk stakk ArrayList vs vanlig array i Java (ms) 36

5.1 Oppsummering 37

Kapittel 1

Innledning

> "As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem."
>
> — *Edsger W. Dijkstra 1972* [Dij72]

Programmeringsspråkene vi bruker er i stadig utvikling. Det dukker hele tiden opp nye språk, mens andre forsvinner. De eksisterende programmeringsspråkene blir oppdatert og utviklet videre. Bak denne utviklingen ligger det et ønske om å gjøre oss i stand til å løse stadig mer kompliserte problemer raskere, enklere og mer kostnadseffektivt.

Endringer av eksisterende programmeringsspråk kan få store konsekvenser, ikke minst økonomisk. Det gjelder spesielt de mest brukte språkene. Java og C# er uten tvil to av de viktigste programmerimgsspråkene i dag, og det er knyttet store økonomiske interesser til dem. Både Java og C# har opplevd store endringer de siste årene. Det er blant annet innført generiske mekanismer i begge språkene. Dette har skapt debatt. Særlig har de generiske mekanismene i Java vært heftig diskutert i ulike fora. I tillegg er dette to språk som er svært like, og som konkurrerer mot hverandre. En sammenligning av språkene og deres endringer vil kunne belyse heldige og uheldige konsekvenser av de nye mekanismene. I denne oppgaven vil jeg sammenligne de generiske mekanismene som er innført i Java og C#.

1.1 Om oppgaven

I oppgaven ønsker jeg å sammenligne de generiske mekanismene i Java og C#. Jeg ønsker å se på:

1. Uttrykkskraft
   - Er det mulig å uttrykke det samme i de to språkene?
2. Lesbarhet
   - Hvor lesbar og oversiktlig er den nye syntaksen?
   - Passer den med resten av språket?
3. Integrasjon
   - Hvor godt passer de generiske mekanismene inn i språket?
   - Tilfører de nye begreper som bryter med eksisterende begreper?
   - Fungerer de generiske mekanismene slik man forventer ut fra tidligere kunnskap og erfaring med språket?
4. Effektivitet
   - Gir de nye generiske mekanismene en hastighetsgevinst?

Oppgaven er bygget opp på følgende måte. I kapittel 2 beskriver jeg generiske mekanismer og viser eksempler på generiske mekanismer i forskjellige språk. Jeg presenterer også noen problemområder knyttet til de forskjellige formene for generiske mekanismer. I kapittel 3 beskriver jeg de generiske mekanismene i Java 5.0 og C# 2.0. I kapittel 4 presenterer jeg et større eksempel programmert i både Java og C#, og diskuterer forskjeller og problemer relatert til disse. Kapittel 5 er en oppsummering av oppgaven.

1.2 Språket, eksemplene og koden

Oppgaven er skrevet på norsk, og norske ord er brukt så langt som mulig. Noen steder har det likevel vært naturlig å bruke engelske uttrykk. Når

man bruker engelske uttrykk må man velge om man skal bøye uttrykket med norsk eller engelsk endelse. Jeg har valgt å bruke norske endelser de fleste stedene i oppgaven, men noen få uttrykk har jeg valgt å bøye på engelsk, f.eks *casts*.

I koden som brukes i eksemplene i oppgaven og i programmeringseksemplene bruker jeg engelske variabel-, klasse- og metodenavn. Det er derfor naturlig at også kommentarene i koden er skrevet på engelsk. Dette kan se unaturlig ut i oppgaveteksten, men jeg mener likevel at det er det ryddigste alternativet.

De fleste eksemplene i oppgaven er hentet fra eksemplene jeg har programmert i forbindelse med arbeidet med oppgaven. Mange av eksemplene er forkortet for ikke å ta unødvendig mye plass.

De få stedene jeg bruker eksempler som er hentet fra andre steder, er dette kommentert.

Kapittel 2

Om generiske mekanismer

2.1 Generiske mekanismer

Generiske mekanismer i programmeringsspråk er en form for polymorfisme. Polymorfisme er en måte å definere programenheter som kan ha flere former, eller som kan brukes i flere ulike sammenhenger. Med generiske mekanismer ønsker man å kunne skrive kode så generell som mulig for senere å kunne bruke koden i flere sammenhenger. Man ønsker for eksempel å lage en generisk liste for å kunne lage lister av objekter av ulike typer i forskjellige sammenhenger.

En av de viktigste hensiktene med generiske mekanismer er å gjøre det enklere å skrive gjenbrukbar og sikker kode i statisk typete språk. I dynamiske språk som f. eks. Python og Php kan slik kode skrives uten egne mekanismer.

Den vanligeste formen for generisk mekanisme er parametriserte typer. Ada, C++ og Eifel er eksempler på språk med parametriserte typer. Mekanismen går ut på at man skriver kode med et symbol for en typeparameter som kan erstattes med en type. Figur 2.1 på neste side viser et eksempel på bruk av en slik mekanisme. Figuren viser en generisk klasse med en formel typeparameter `T`. I linje 9 ser vi hvordan instansieringen av et objekt av den generiske klassen krever at man gir `T` en type, i dette tilfellet typen `ClassName`. På denne måten kan klassen `A` skrives uavhengig av hvilken

```
1 class A<T>{
2       T varName;
3
4       T getVarName(){
5               return varName;
6        }
7 }
8
9 A<ClassName> objectRef = new A<ClassName>();
10 A<Double> objectRef = new A<Double>();
```

Figur 2.1: Eksempel på generisk kode

type `varName` har. I mange språk finnes det også mekanismer for å avgrense hvilke typer parameteren `T` kan ha. Man kunne f.eks. sagt at typen til typeparameteren til klassen `A` måtte være en subtype av en bestemt type.

I objektorienterte språk som Java og C# kan man oppnå mye av denne formen for polymorfisme uten generiske mekanismer. Dette gjøres ved at man skriver generelle klasser og metoder med parametre og returverdier av typen `Object`[1]. Eventuelle returverdier av typen `Object` må konverteres til riktig type. Denne formen for programmering kalles ofte *det generiske idiom*. *Det generiske idiom* har flere svakheter i forhold til generiske mekanismer. Jeg ser nærmere på dette senere i kapitlet (2.5).

De vanligste bruksområder for generiske mekanismer i objektorienterte språk er i datastrukturer som stakker, lister og hash-tabeller.

2.2 Generiske pakker og funksjoner i Ada

Ada var et av de første programmeringsspråkene med parametriserte typer.

Figur 2.2 på neste side viser et eksempel på en generisk funksjon i Ada. Eksemplet er hentet fra "Genericity versus Inheritanc" [Mey86] og viser en generisk funksjon som bytter verdien på to variable. Istedenfor å gi variabelen en type brukes et symbol, T, for typeparameteren. Før funksjonen

[1] I Java og C# er alle klasser direkte eller indirekte subklasser av klassen `Object`.

```
1 generic
2         type T is private;
3 procedure swap(x, y: in out T) is
4         t: T
5 begin
6         t:=x; x:=y; y:=t;
7 end swap
8
9 procedure int_swap is new swap(INTEGER);
10 procedure str_swap is new swap(STRING);
11
12 int_swap(i,j);
13 str_swap(s,t);
```

Figur 2.2: Generisk funksjon i Ada

kan brukes må den konstrueres[2], linje 9 og 10 i Figur 2.2 viser eksempel på dette. I linje 12 og 13 brukes funksjonen. Når en funksjon eller pakke blir konstruert opprettes det en fysisk kopi der den formelle typeparameteren byttes ut med den aktuelle. Det blir altså mange kopier av den generiske koden i binærkoden.

2.3 Templates i C++

Et eksemplel på generiske mekanismer som mange kjenner til, er *templates* i C++. *Templates* ble innført for å gjøre det enklere å lage generiske datastrukturer i C++. C++ har ikke et klassehierarki der alle klasser er subklasser av en felles klasse og egner seg derfor ikke så godt til generisk programmering ved hjelp av *det generiske idiom*. Store deler av *the Standard C++ Library* er programmert med templates.

Kritikken mot templates i C++ går ofte ut på at de er kompliserte og at det er vanskelig å få en god forståelse av hvordan de fungerer. Det kan ha sammenheng med at det tradisjonelt har vært dårlige feilmeldinger og varierende støtte for *templates* i de forskjellige C++ kompilatorene.

Figur 2.3 på neste side viser et eksempel på en generisk stakk skrevet med *templates* i C++.

[2]I Ada er det vanlig å kalle dette instansiering, men i oppgaven har jeg valgt å kalle det "konstruksjon" uavhengig av hvilket språk jeg omtaler.

```
1 template <class T> class MyGenStack{
2  protected:
3         T* stack; int top; int length;
4
5  public:
6         MyGenStack(int n);
7         void push(T object);
8         T pop(T &object);
9 };
10
11 template <class T> MyGenStack<T>::MyGenStack(int n){
12         length = n;
13         stack = new T[n];
14 }
15
16 template <class T> void MyGenStack<T>::push(T object){
17   stack[top] = object;
18   top++;
19 }
20
21 template <class T> T MyGenStack<T>::pop(T &object){
22   top--;
23   object = stack[top];
24   return stack[top];
25 }
```

Figur 2.3: *Templates* i C++

2.4 Virtuelle klasser

En annen form for generiske mekanismer er virtuelle klasser, som vi finner f.eks. i Beta. I Beta har ikke klasser typeparametere, men klasser kan ha virtuelle klassedeklarasjoner. Istedenfor å konstruere den virtuelle klassen med en typeparameter definerer man en subklasse av den virtuelle klassen. I denne subklassen definerer man hvilken type den virtuelle typen skal ha.

Figur 2.4 på neste side viser hvordan man kan skrive en generisk stakk i Beta. Figuren viser også hvordan man lager en subklasse for en bestemt type.

```
1 Point: (# x,y; @integer #)
2
3 Stack:
4   (# type:< Object;
5      push:< (# e: ^type enter e[] do (* ... *) #)
6      (* ... *)
7   #)
8
9 PointStack: Stack
10  (# type::< Point;
11     (* ... *)
12  #)
```

Figur 2.4: Virtuelle typer i Beta

2.5 Det generiske idiom

I mange programmeringsspråk er det mulig å oppnå mye av det samme som vi ønsker å oppnå med generiske mekanismer uten å bruke generiske mekanismer, som f. eks. parametriserte typer eller virtuelle klasser. I mange objektorienterte språk er det vanlig å skrive f.eks. datastrukturer med referanser til supertyper, som alle typene som skal lagres er subtyper av. Figur 2.5 på neste side er et eksempel på hvordan man kan skrive generisk kode ved hjelp av objektorienterte mekanismer. Istedenfor å bruke en typeparameter som i C++ eksemplet, lagrer stakken referanser til objekter av klassen `Object`. Når et objekt hentes ut av stakken må man legge inn en eksplisitt konvertering av referansetypen, se linje 26 i figuren. En slik typekonvertering kalles et *cast*.

Programmering ved hjelp av det generiske idiom har som nevnt tidligere i kapitlet, flere svakheter. Koden kan ikke typesjekkes ved kompilering, blir mer uoversiktlig pga. *casts* og fungerer ikke for primitive typer.

Siden kompilatoren ikke er i stand til å sjekke typene under kompileringen, er det mulig at objektene som hentes ut, ikke er av den forventede typen. Slike feil vil først bli oppdaget under kjøring. Man vil få en `Exception` under kjøring hvis man prøver å angi feil type ved *casting* av en referanse. Hvis dette ikke er tatt høyde for, fører en slik feil til at programmet stopper. Linje 30 i figur 2.5 på neste side viser et eksempel på en typefeil som ikke oppdages av kompilatoren, men som gir en feil under kjøring. Typesjek-

```
1 class MyStack{
2
3   private Object stack[];
4   private int top;
5
6   public MyStack(int n){
7     this.top = 0;
8     this.stack = new Object[n];
9   }
10
11  public void push(Object o){
12    stack[this.top++] = o;
13  }
14
15  public Object pop(){
16    return this.stack[--this.top];
17  }
18 }
19
20 public class MyStackApp{
21   public static void main(String []args){
22     MyStack stringStack = new MyStack(10);
23
24     stringStack.push("Dette er en tekst.");
25
26     String tmp = (String) stringStack.pop();
27     System.out.println(tmp);
28
29     stringStack.push(new Object()); // No error
30     // tmp = (String) stringStack.pop(); // ClassCastException
31   }
32 }
```

Figur 2.5: Stakk i Java, *det generiske idiom*

king under kjøring er også noe som går utover kjørehastigheten.

Typesjekking ved kompilering er også nyttig av andre grunner. Koden blir enklere å lese og det kan genereres mer effektiv kode [MMMP90].

Generiske metoder og klasser programmert ved hjelp av *det generiske idiom* kan kun brukes med referansetyper. En referansetype inneholder ikke de faktiske data, men en referanse til en annen plass i minnet. Ønsker man å bruke primitive typer må de pakkes inn i objekter. I noen språk pakkes primitive typer automatisk inn i tilfeller der det er påkrevd, slik at programmereren ikke trenger å gjøre det eksplisitt. Dette kalles gjerne *autoboxing*. Dette går utover eksekveringshastigheten. Kode programmert ved

hjelp av *det generiske idiom* er derfor mindre egnet til programmering der eksekveringshastighet er kritisk enn kode skrevet ved hjelp av generiske mekanismer med støtte for primitive typer. Eksempler på programmering der eksekveringshastighet er kritisk kan være simuleringer og beregninger av store datasett. I slike sammenhenger er f.eks. C++ *templates* mye brukt.

De nødvendige *castene* gjør koden mer uoversiktlig. De kan også være til irritasjon når programmereren vet at *castingen* er "unødvendig". Bruk av generiske mekanismer kan dessuten gi mer tydelig kode.

```
Stack<Integer> intStack = new Stack<Integer>();
```

Her vises det tydelig at dette er en stakk som lagrer referanser til objekter av typen `Integer`.

2.6 Implementasjon av generiske mekanismer

Generiske mekanismer kan implementeres på flere måter. Det er vanlig å skille mellom heterogen og homogen implementasjon.

Heterogen implementasjon betyr at den kompilerte koden som beskriver den generiske mekanismen blir duplisert for hver gang den generiske koden blir konstruert med en ny typeparameter. Det er slik *templates* er implementert i C++. I Java ville man med en slik tilnærming endt opp med en class-fil for hver gang man bruker en generisk klasse med en ny typeparameter. Dette resulterer i unødvendig mye kompilert kode. Den kompilerte koden bruker derfor unødvendig mye lagringsplass, noe som også kan føre til at det tar lang tid å overføre koden. Dette kan være problematisk hvis den kompilerte koden overføres over et nettverk eller liknende før den eksekveres.

Fordelen med heterogen implementering er at det gir mulighet for optimalisering av koden under kompilering. Dette er mindre viktig hvis alle typeparameterene er referansetyper, noe som betyr at de har samme størrelse i den kompilerte koden. Heterogen implementering gir også mulighet for refleksjon, noe som betyr at det finnes typeinformasjon om typeparameterne under kjøring. Denne typeinformasjonen kan brukes til typesjekking under kjøring.

Ved homogen implementasjon av generiske mekanismer er det bare en representasjon av den generiske kodebiten i den kompilerte koden. Den kompilerte kodebiten må være så generell at den fungerer for alle lovlige parametertyper. Man er derfor avhengig av at den kompilerte koden blir knyttet opp mot konteksten etter kompilering. Dette kan føre til ekstra eksekveringstid. Fordelen med denne formen for kompilering er at man slipper at den kompilerte koden tar unødvendig mye plass.

Det er også mulig å kombinere heterogen og homogen implementering av generiske mekanismer. Hensikten er å utnytte fordelene ved både heterogen og homogen implementasjon. Man kan f.eks. representere generisk kode som er konstruert med referansetyper med en felles kodebit, mens generisk kode som er konstruert med primitive typer representeres forskjellig i den kompilerte koden.

Ada og C++ er eksempler på språk med heterogen representasjon av generiske typene i kompilert kode, mens Eiffel er et eksempel på et språk som representerer de generiske typene homogent.

2.7 Generiske problemområder

Selv om generiske mekanismer ofte er både fleksible og kraftige, finnes det problemområder. Flere av problemene oppstår i møte mellom generiske mekanismer og objektorienterte språk. Mange av problemene har blitt drøftet i den akademiske debatten rundt innføringen av generiske mekanismer i Java [AFM97, SA98, CGLS98, Tho97, TT99].

2.7.1 Problemer med generisk varians

Et problemene med generiske mekanismer henger sammen spørsmålet om generiske typer skal være kovariante og/eller kontravariante. Kovariante og kontravariante generiske typer kan defineres på følgende måte:

- Generiske typer er *kovariante* hvis det er slik at `List<Integer>` er en subtype av `List<Number>` hvis og bare hvis `Integer` er en subtype av `Number`.

- Generiske typer er *kontravariante* hvis det er slik at `List<Integer>` er en subtype av `List<Number>` hvis og bare hvis `Number` er en subtype av `Integer`.

Arrayer kan også være kovariante eller kontravariante. I figur 2.6 viser a) et eksempel på at kovariante arrayer ikke er statisk typersikre, og b) viser et tilsvarende eksempel med kovariante generiske typer.

a)

```
1 Object[] array = new String[10];
2 array[0] = new Object(); // ArrayStoreException
```

b)

```
1 class A<T>{
2     private T varName;
3
4     void set(T varName){ this.varName = varName; }
5
6     T get(){ return this.varName; }
7 }
8
9 A<Object> aObject = new A<String>(); // generic covariance
10 aObject.set(new Object()); // runtime error
```

Figur 2.6: Kovarians

Selv om kovariante arrayer ikke kan typesjekkes under kompilering er arrayer i både Java og C# kovariante. Grunnen til det er at det gjør språket mer fleksibelt. Det er på grunn av at arrayer er kovariante det er mulig å skrive stakken i figur 2.5 på side 9 med en array for å lagre elementene.

Problemet med ko- og kontra-variante generiske typer avhenger av om man ønsker å hente ute eller oppdatere noe i den generiske typen. Som vi ser i figur 2.6 i b) oppstår det problemer når man prøver å bruke typeparameteren som argument i en metode, det er derimot trygt å bruke typeparameteren som returverdi. Med kontravariante generiske typer er det omvendt, det er alltid trygt å bruke typeparameteren som argument til en metode, men ikke som returverdi.

På grunn av disse problemene er generiske typer i mange språk *invariante*, det betyr at det ikke er noe sub/superklasseforhold mellom generiske ty-

per selv om det er det mellom typeparameterene. Dette er tilfellet i C++. Men det finnes også andre alternativer. En av mulighetene er å bruke enten kontravarians eller kovarians alt ettersom typeparameteren brukes i returverdier eller metodeargumenter i en generisk klasse. En slik løsning ble foreslått i "A Proposal for Making Eiffel Type-safe" [Coo89].

En annen løsning på problemet er å ha en spesiell syntaks for å spesifisere om en type skal være kovariant eller kontravariant. En slik mekanisme er beskrevet av Thorup og Torgersen [TT99]. I artikkelen innfører de to versjoner av en generisk klasse `List<E>`, nemlig den invariante `List<Number>` og den kovariante `List<+Number>`.

Variante generiske typer kan også være problematisk i språk der det er tillatt med flere typeparametre. Hvordan vil subtypeforholdet mellom typene i figur 2.7 være? Spørsmålet stilles blandt annet av Kresten Krab Thorup i "Genericity in Java with Virtual Types" [Tho97].

```
List<Number, Integer>
List<Integer, Object>
List<Double, Double>
```

Figur 2.7: Generiske typer med flere parametere

2.7.2 Rekursivt avhengige klasser

Et annet "generisk" problemområde er rekursivt avhengige klasser. Problemet oppstår når en klasse må referere til seg selv. Dette blir fort komplisert å utrykke med parametriserte typer. I artikkelen "A Statically Safe Alternative to Virtual Types" [BOW98] vises det hvordan en alternerende liste relativt enkelt kan skrives med virtuelle typer, mens det samme er mer komplisert med parametriserte typer. Dette er også konklusjonen i "Genericity in Java with Virtual Types" [Tho97].

Kapittel 3

Generiske mekanismer i Java og C#

3.1 Java

Java er et relativt nytt programmeringsspråk. Språket er utviklet av Sun Microsystems på 1990-tallet. Java kjennetegnes ved at det er objektorientert, plattformuavhengig og at det har et stort klassebibliotek.

Java har mye felles med språk som C++, Pascal og Algol 60. Det har også mange av de samme objektorienterte prinsippene som Simula. Mange vil si at Java er et enklere og "renere" språk enn f.eks. C++. Om det er tilfelle også etter innføringen av generiske mekanismer er en av problemstillingene i oppgaven.

Java er ikke et 100% statisk typet språk, det vil si at kompilatoren ikke alltid er i stand til å garantere at koden er typesikker. Selv om det er lagt opp til at mest mulig av typesjekkingen skal gjøres under kompilering er det enkelte konstruksjoner som må sjekkes under kjøring.

For at kompilert Java skal være plattformuavhengig kompileres ikke koden til maskinkode som f.eks. i C++. Kompilatoren omgjør koden til *bytecode* som eksekveres ved hjelp av et kjøresystem (JVM, Java Virtual Machine). Dette fører til at kode skrevet i Java ofte vil være noe tregere enn tilsvarende kode skrevet i f.eks. C++.

Java har et plattformuavhengig klassebibliotek. Det betyr at programmering av operasjoner som er tett knyttet til operativsystemet kan gjøres uavhengig av operativsystemet programmet skal kjøre på. Dette gjelder operasjoner som f.eks. nettverksprogrammering, programmering av grafisk brukergrensesnitt og trådprogrammering. I mange språk er slike bibliotek tettere knyttet opp mot ett bestemt operativsystem. Program skrevet i slike språk må ikke bare kompileres for hvert operativsystem det skal kjøre på, ofte må de også skrives om for å kunne gjøre de samme operasjonene på forskjellige plattformer.

3.1.1 Generiske mekanismer i Java 5

Prosessen med å implementere generiske mekanismer i Java har vært lang. Det har vært mye debatt rundt hvordan dette bør gjøres. Mange har gjennom artikler og studier kommet med forslag til hvordan de generiske mekanismene bør fungere og hvordan de kan implementeres i Java. Mange har foreslått parametriserte typer [BOSW98, AFM97]. Andre har foreslått at virtuelle typer som ligner de virtuelle klassene i Beta er en bedre mekanisme for et objektorientert språk [Tho97]. Det har også vært foreslått en kombinasjon av disse for å kunne benytte seg av de sterke sidene i begge mekanismene [TT99, BOW98].

```
1 // Generic class
2 class A<T>{
3     A varName;
4 }
5
6 // Create object
7 A<Integer> a = new A<Integer>();
8
9 // Generic method
10 static void <T> methodName(T arg){
11     //...
12 }
```

Figur 3.1: Generiske mekanismer i Java

De nye generiske mekanismene i Java [Bra04] er parametriserte typer med syntaks som ligner på *templates* fra C++. Figur 3.1 viser et enkelt eksempel på en generisk klasse og en generisk metode i Java.

Selv om syntaksen likner på *templates* fra C++ er det mange forskjeller. Den største forskjellen er hvordan den generiske koden kompileres. Mens templates i C++ egentlig bare er makroer som kompileres til ny maskinkode for hver gang de brukes, kompileres en generisk konstruksjon i Java kun til en instans i den kompilerte koden.

En begrensning i forhold til *templates* i C++ er at primitive typer ikke kan brukes som typeparametere.

Mange Java-tilhengere vil si at de generiske mekanismene i Java har uttrykkraften til C++ templates uten problemene, kritikerne mener at mekanismene kun er en *auto-casting feature*[1] som gjør språket mer komplisert og mindre forutsigbart.

3.1.2 Beskrankning i Java

I Java benyttes nøkkelordet *extends* for å angi begrensninger for typeparameterene, se figur 3.2. Begrensningene angir at den aktuelle typeparameteren må være en subklasse av en klasse eller at parameteren må implementere et *interface*, som også kan være generisk. Typeparameteren kan også brukes om betingelse, såkalt *f-bound polymorphism*.

```
1 static <T extends Comparable<T>> T max(IMyGenCollection<T> c){
2     //...
3 }
```

Figur 3.2: Beskrankning i Java

Syntaksen for beskrankning av parametertyper bryter med syntaksen for klasser og *interface* der man bruker *implements* når en klasse implementerer et *interface*. Dette kunne også vært brukt for typeparametere som i forslaget til Agesen, Freund og Mitchell [AFM97]. Det kan argumenteres med at det er unødvendig å eksplisitt uttrykke om det er et *interface* som må være implementert eller en klasse som må være arvet, men dette er et konsept programmereren kjenner. På den annen side vil det å bare bruke *extends* redusere lengden på generiske klassedeklarasjoner, noe som gir mer oversiktlig kode.

[1] Jeg sikter ikke til *auto-boxing* som også er innført i Java 5

Figur 3.2 på forrige side viser en metode der typeparameteren må kunne sammenlignes med seg selv, eller sagt på en annen måte må typeparameterens type implementere det generiske interfacet `Comparable<T>`.

3.1.3 Jokernotasjon

Jokernotasjon er en ny konstruksjon som er innført for å gjøre det objektorienterte typesystemet med parametriserte typer mer fleksibelt [THE+04], og adresserer noen av problemene diskutert i 2.7 på side 11. Generiske typer i Java er ikke kovariante. En referanse til typen `List<Integer>` kan derfor ikke erstattes med en referanse til typen `List<Number>`. Jokernotasjon er en måte å omgå dette. Ved å bruke et spesielt typeargument '?' kan man referere til generiske typer med en hvilken som helst typeparameter. En referanse typet med `List<?>` eller `List<? Extends Number>` kan referere til objekter av typen `List<Integer>`.

a)

```
1 static double sum(IMyGenCollection<? extends Number> c){
2     //...
3 }
```

b)

```
1 static <T extends Number> double sum(IMyGenCollection<T> c){
2     //...
3 }
```

c)

```
1 static <T extends Comparable<? super T>
2              void sort(MyGenList<T> l){
3     //...
4 }
```

Figur 3.3: Jokernotasjon

I figur 3.3 er a) et eksempel på bruk av jokernotasjon. Metoden `sum` summerer en liste med objekter av typer som er subklasser av `Number`. I figur 3.3 viser b) den samme metoden skrevet uten bruk av jokernotasjon.

Man kan også definere en nedre grense for typen. I figur 3.3 på forrige side er c) et eksempel på en metode med en nedre grense. Metoden `sort` sorterer en liste med typer som kan sammenlignes med seg selv eller en supertype.

3.1.4 Raw types

For at de nye generiske klassene skal være kompatible med gammel kode kan man bruke generiske klasser uten å oppgi noe typeargument, som vist i linje 3 figur 3.4. En slik type kalles *raw type* og må ikke forveksles med typer med jokernotasjon. I figur 3.4 betyr ikke `A` det samme som `A<?>` men `A<Object>`.

```
1 class A<T>{ }
2
3 A a = new A(); // raw type
4
5 a = new A<String>(); // legal
6 A<String> c = new A() // warning: unchecked conversion
```

Figur 3.4: *Raw type*

Det er tillat å bruke en *raw type* referanse til å referere til en type med en bestemt typeparameter (linje 5). Det er også mulig å bruke en referanse med en bestemt typeparameter til å referere til en *raw type* (linje 6), men dette vil gi en *unchecked*-advarsel[2] fra kompilatoren.

3.1.5 Utvidelser og endringer av Javas klassebiblioteket

Store deler av klassebiblioteket er fra og med Java 5 generiske bygget opp med generiske typer. Dette gjelder blant annet klasser og *interface* for datastrukturer f.eks. `ArrayList<E>`, `HashMap<K,V>` og `Iterator<E>`. Klassene kan fortsatt brukes med *raw types*, men man vil da få en *unchecked*-advarsel fra kompilatoren.

Men kunne tenkt seg at man istedenfor å endre det eksisterende klassebiblioteket kunne laget nye generiske klasser. Et problem relatert til dette er

[2] "Note: xxx.java uses unchecked or unsafe operations."

navning av de nye klassene, siden det er lov med *raw type* kan ikke generiske klasser ha samme navn som de ikke generiske. Det at det ikke finnes noe alternativ til de generiske klassene i klassebiblioteket fører også til at programmerere raskere må begynne å bruke de nye generiske mekanismene.

3.1.6 Implementasjon

På grunn av at Java kompileres til bytekode som kjøres ved hjelp av JVM vil endringer av bytekoden føre til at ny kode ikke kan kjøre på eldre versjoner av kjøresystemet. Det ble derfor bestemt at det det ikke skulle gjøres endringer i kjøresystemet eller *bytecoden* ved innføringen av parametriserte typer i Java. Det finnes av den grunn ikke støtte for de nye generiske mekanismene i kjøresystemet. Dette har lagt store begrensninger for innføringen av de generiske mekanismene i Java. De generiske mekanismene er derfor kun en abstraksjon ved kompilering. Kompilatoren sjekker at typene som brukes, stemmer med typeparameterene til de generiske klassene. Deretter fjernes type informasjonen og referansene blir byttet ut med referanser til den mest generelle typen typeparameteren kan ha. Dette kalles gjerne *type erasure*. Figur 3.5 på neste side viser hvordan koden oversettes under kompilering.

Type erasure fører til at det ikke finnes typeinformasjon om typeparameteren under kjøring. Dette gir flere vesentlige restriksjoner. Operasjoner som er avhengig av typeinformasjon under kjøring, kan derfor ikke utføres på typeparametere. Eksempler på operasjoner som ikke er tilgjengelige for typeparametere, er:

- *new T()*
- *new T[size]*
- type *cast*[3]
- *instanceof*

[3]Det er lovlig å bruke type parameteren til *cast* og noen ganger helt nødvendig, men man vil få en advarsel.

```
1 // Before type erasure
2 class List<T>{
3     T head;
4     List<T> tail;
5     //...
6 }
7
8 List<Integer> intList = new List<Integer>();
9 Integer i = intList.getHead();
10
11 // After type erasure
12 class List{
13     Object head;
14     List tail;
15     //...
16 }
17
18 List intList = new List();
19 Integer i = (Integer) intList.getHead();
```

Figur 3.5: *Type erasure* i Java

3.2 C#

C# er et objektorientert programmeringsspråk utviklet av Microsoft og lansert i 2000. C# ligner svært mye på Java. C# er en viktig del av Microsofts .NET-plattform [Cor01]. Språkene på .NET-plattformen kompileres til midlertidig kode i *Common Intermediate Language* (CIL) som kjøres ved hjelp av et kjøresystem *Common Language Runtime* (CLR).

.NET-plattformen tilbyr programmerere å kombinere kode skrevet i flere programmeringsspråk. Objekter skrevet i f.eks. C# kan brukes fra andre språk som er implementert for plattformen. Visual Basic, C++, Java, COBOL, Python og Perl er foruten C# noen av språkene som kan kompileres og kjøres på .NET-plattformen.

Det finnes også alternative implementasjoner av .NET-plattformen. En av disse er Mono, en åpen kildekodeimplementasjon av .NET som utvikles med støtte fra Novell. Mono gjør det mulig å skrive og kjøre .NET- og C#-applikasjoner på blant annet Linux og OS X.

3.2.1 Generiske mekanismer i C#

De generiske mekanismene som er innført for .NET-plattformen og C# 2.0, minner mye om de generiske mekanismene i Java. Syntaksen er nokså lik, selv om det er noen forskjeller. De største forskjellene mellom de generiske mekanismene ligger i hvordan mekanismene er implementert, og ulikhetene det medfører. De generiske mekanismene i Java er implementert homogent, mens det i C# er gjort en blanding. En mer detaljert beskrivelse av implementasjonen kommer senere i denne delen.

Figur 3.6 viser et enkelt eksempel på en generisk klasse og en generisk metode i C#. Koden i eksemplet er svært lik kode i Java (se figur 3.1 på side 15).

```
1 // Generic class
2 class A<T>
3 {
4     A varName;
5 }
6
7 // Create object
8 A<int> a = new A<int>();
9
10 // Generic method
11 static void methodName<T>(T arg)
12 {
13     //...
14 }
```

Figur 3.6: Generiske mekanismer i C#

I motsetning til Java er det i C# mulig å ha primitive datatyper som aktuelle typeparametere. Dette er noe av det Anders Hejlsberg trekker frem i et intervju om de generiske mekanismene i C# med Bill Venners og Bruce Eckel [VE04]. Anders Hejlsberg er sjefsarkitekten bak C#. Han trekker også frem at C# i motsetning til Java har gjort endringer i kjøresystemet for å gi støtte for de generiske mekanismene under kjøring.

3.2.2 *Where condition*

I C# brukes *where condition* for å begrense lovlige typer av typeparametere. Betingelsen kan være klasser, *interface*, strukter, nøkkelordet `new()` eller

nøkkelordet `class`. Nøkkelordet `new()` gjør det mulig å opprette objekter av typeparameteren, mens *class* betyr at den aktuelle typeparameteren må være en klasse.

Figur 3.7 viser hvordan en metode tilsvarende metoden i figur 3.2 på side 16 vil se ut i C#. Her kommer betingelsen etter selve metodedefinisjonen.

```
1 public static T max<T>(IMyGenCollection<T> c)
2              where T : IComparale<T>
3 {
4
5 }
```

Figur 3.7: Beskrankning i C#

3.2.3 Alias

Med `using` kan man i C# lage et alias for en generisk type med bestemte typeparametere. I figur 3.8 er det et eksempel på bruk av alias. Bruk av alias kan gi mer oversiktlig kode, spesielt hvis man kan utnytte det til å forkorte lange uttrykk. På den andre siden kan noe av lesbarheten mistes ved at man ikke lenger ser hva aliaset egentlig står for.

```
1 using WordDescriptions = MyGenHashMap<string, Description>;
2
3 WordDescriptions wordDescriptions = new WordDescriptions();
```

Figur 3.8: Alias i C#

3.2.4 Klassebiblioteket

I C# er ikke det eksisterende klassebiblioteket endret slik som i Java. I C# er det lagt til et nytt *namespace* `System.Collections.Generic` som inneholder generiske datastrukturer. Det er derfor mulig å bruke C# uten å bruke de generiske mekanismene.

3.2.5 Implementasjon av generiske mekanismer i C#

Microsoft valgte en annen strategi enn Sun Microsystems da de implementerte de generiske mekanismene i C# [KS01]. De valgte å gjøre endringer i kjøresystemet slik at de generiske mekanismene ikke bare er støttet under kompilering, men også under kjøring.

Når generisk C# kode kompileres lagres den i en midlertidig form med den generiske informasjonen. Når programmet blir eksekvert og de generiske klassene blir brukt blir den midlertidige koden brukt med de aktuelle typeparameterne.

Når en generisk type blir konstruert med en verditype som aktuell typeparameter lages det en kopi av den generiske typen i den kompilerte koden. Dersom den samme generiske typen konstrueres med en annen verditype lages det en ny kopi, se figur 3.9.

```
List<int> list;   // new copy of List with ints
List<float> list; // new copy of List with floats
List<Cat> list;   // new copy of List with references
List<Dog> list;   // no new copy List
```

Figur 3.9: Instansiering av generiske klasser i C#

Dette er ikke tilfelle for referansetyper. Første gang en generisk type blir konstruert med en referansetype, blir det laget en spesiell generisk type med en referanse til parametertypen. Denne blir gjenbrukt hver gang det konstrueres en generisk type med en referansetype som aktuell typeparameter. Det blir opprettet en peker i minnet til et område med størrelsen av typen referansen refererer til.

De generiske mekanismene er implementert for hele .Net-plattformen, slik at det er støtte for generiske mekanismer også i f.eks. Eiffel, Ada og ML. Det betyr også at alle språkene kan utvides med generiske mekanismer.

Kapittel 4

Sammenlikning

For bedre å forstå forskjellene mellom de generiske mekanismene i Java og C# har jeg programmert noen eksempler med generiske mekanismer i både Java og C#. Det overordnede målet med programmeringen av eksemplene var å få innsikt til å kunne besvare spørsmålene og problemstillingene presentert i kapittel 1.

All kildekode finnes på en vedlagt cd. En kort beskrivelse av filene på cd'en finnes i tillegget på side 43.

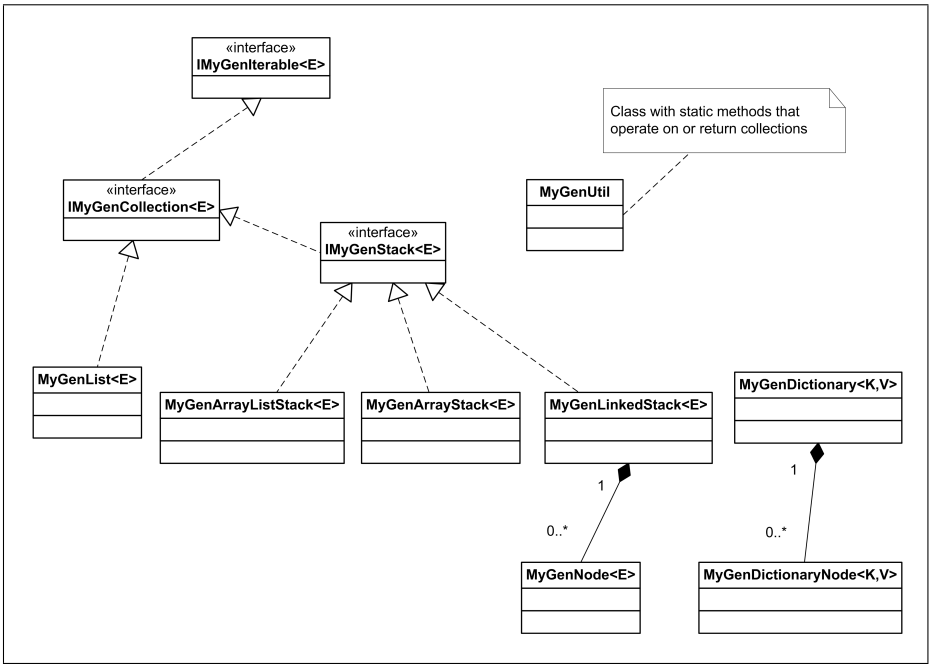
4.1 MyGenCollection

De fleste av eksemplene er en del av pakken `MyGenCollection`[1], som er en pakke med generiske datastrukturer. Pakken består også av verktøymetoder som kan brukes sammen med datastrukturene.

En av datastrukturene i `MyGenCollection` er en generisk stakk. Dette er en generisk versjon av stakken fra figur 2.5 på side 9. Hensikten med disse er å undersøke om de generiske mekanismene adresserer svakhetene med *det generiske idiom*, som jeg presenterte i kapittel 2.

Figur 4.1 på neste side viser en oversikt over datastrukturene og metodene i *MyGenCollection*.

[1] I Java er det vanlig å bruke småbokstaver i pakkenavn, jeg har valgt å bryte denne konvensjonen for å bruke det samme navnet både i C# og Java.



Figur 4.1: Oversikt over *MyGenCollection*

4.2 Ulikheter og problemer

I kapittel 3 så vi at det er forskjeller mellom de generiske mekanismene i Java og C#. Her vil jeg presentere og diskutere forskjellene og problemene ved implementeringen av programmeringseksemplene beskrevet over.

4.2.1 Generiske arrayer

Et av de første problemene man møter på når man skal bruke eller lage egne generiske klasser i Java er de generiske arrayene, eller mangelen på dem. Det er nemlig ikke tillatt å opprette en array av en typeparameter. Det er heller ikke tillatt å opprette en array av en parametrisert type. Figur 4.2 på neste side gir eksempler på hva som er lov og ikke. I C# har man ikke disse begrensningene.

For å forstå hvorfor det ikke er tillatt å opprette generiske arrayer må vi forstå hvordan arrayer i Java fungerer. Arrayer i Java er kovariante. Det betyr at hvis klassen Student er en subklasse av klassen Person, så er `Student[]`

```
1 class MyGenStack<E>{
2
3     // Error: generic array creation
4     // E[] stack = new E[10];
5
6     // Warning: unchecked operation
7     E[] stack = (E[]) new Object[10];
8
9     // Recomended
10    ArrayList<E> alStack = new ArrayList<E>(10);
11 }
12
13 // Error: generic array creation
14 // A<Integer>[] a = new A<Integer>[10];
```

Figur 4.2: Generiske arrayer i Java

en subtype av `Person[]`. Man kan derfor bruke en referanse av typen `Person[]` til å referere til array av typen `Student`. Arrayer må derfor i mange tilfeller typesjekkes dynamisk. La oss se hva som kunne ha skjedd hvis det var tillatt å opprette arrayer av generiske typer. Figur 4.3 viser et tenkt eksempel. På grunn av at arrayer er kovariante er tilordningen i linje 7 lovlig (`A<String>[]` er en subtype av `A<Object>[]`). Siden det i Java ikke finnes noen informasjon om typen på den generiske parameteren under kjøring (pga. *erasure*) kan det under kjøring ikke sjekkes at et arrayelement tilordnes riktig type, se linje 8 figur 4.3.

```
1 class A<T>{
2     T varName;
3 }
4
5 A<String>[] aStringArray = new A<String>[3];
6 A<Object>[] aObjectArray = aStringArray;
7 aObjectArray[0] = new A<Integer>(); // cannot check type
8 String str = aStringArray[0].varName; // ClassCastException
```

Figur 4.3: Arrayer av generiske typer i Java

Grunnen til at det ikke er tillatt å opprette arrayer av typeparametere er at det ikke finnes typeinformasjon under kjøring. I figur 4.2 på linje 7, er det et eksempel på en *workaround* som gjør det mulig å opprette en array av en typeparameter. Denne er ikke spesielt pen. Grunnen til at den fungerer er at typeinformasjonen fjernes under kompilering slik at linjen blir:

```
Object[] stack = (Object[]) new Object[10];
```

Det kan argumenteres for at man bør bruke `ArrayList` og ikke vanlige arrayer, slik Peter Ahé[2] gjør i sin weblog [vdA05]. I mange tilfeller er det et svært godt alternativ. En generisk `ArrayList` er statisk typesikker og fleksibel, men er ikke alltid et tilfredsstillende alternativ. I enkelte situasjoner er `ArrayList` tregere enn vanlige arrayer. Dette diskuteres nærmere når jeg beskriver hastighetstestene jeg har utført.

Et interessant poeng er at det i Javas klassebibliotek, og spesielt i klassene for datastrukturer, er en utstrakt bruk av arrayer av typeparametere med forskjellige *workaround*[Bra04]. Det genereres derfor advarsler når klassebiblioteket kompileres, men dette er ikke synlig for brukere av de ferdig kompilerte pakkene. Dette trenger ikke å bety at det vil kunne oppstå typefeil under kjøring.

Uansett hvordan det argumenteres for alternativer til vanlige arrayer er mangelen på muligheten til å bruke vanlige arrayer problemstisk. For en programmerer virker ikke språket konsistent når en så sentral del som arrayer ikke fungerer på samme måte og som forventet i alle situasjoner. Arrayer er dessuten en konstruksjon som vi finner igjen med ganske lik syntaks i mange forskjellige språk. Dette er etter min mening en av det største svakhetene ved de generiske mekanismene i Java.

4.2.2 Programmering med statiske variable og metoder

I Java er statiske elementer felles for alle objekter av en klasse, uavhengig av typeparametere. Det er derfor ikke støtte for å bruke typeparametere i statiske deklarasjoner. I C#, derimot, er de statiske elementene kun felles for klasser som er bundet til samme typeparameter, f.eks. deler objekter av typen `A<int>` én statisk variabel mens objekter av typen `A<string>` deler én statisk variabel, se figur 4.4 på neste side. Dersom man ønsker å aksessere en statisk metode eller en variabel fra utsiden, må man bruke klassenavnet og typeparameteren. Dersom man prøver å gjøre det samme i Java, får man kompileringsfeil.

[2]Peter Ahé jobber for Sun Microsystems med utvikling av Javakompilatoren. Han var med på prosjektet som innførte jokernotasjon i Java.

```
1 class A<T>
2 {
3     static int staticVarName;
4     //...
5 }
6
7 A<int>.staticVarName = 10;
8 A<string>.staticVarName = 20;
9
10 System.Console.WriteLine(A<int>.staticVarName); // 10
11 System.Console.WriteLine(A<string>.staticVarName); // 20
```

Figur 4.4: Generisk klasse med en statisk variabel i C#

Det er også forskjell på hvordan de statiske elementene i en generisk klasse kan brukes. I C# er det mulig å referere til klassen A<int> og A<string>. Statiske metoder og variable kan derfor brukes sammen med type parameteren. Dette gjør at begrepene flyter godt over i hverandre. Men det er også noe merkelig med måten de statiske metodene og variablene fungerer. Det statiske begrepet i C# har endret seg, en statisk variabel eller metode er ikke lenger felles for alle objekter av en klasse. Dette indikerer kanskje at statiske metoder og variable ikke passer så godt inn i det språk med generiske klasser.

4.2.3 Nestede og indre klasser

Hvilken relasjon er det mellom typeparametere og nestede klasser? Nestede klasser oppfører seg i utgangspunktet forskjellig i Java og C#. I Java har objekter av ikke statiske nestede klasser, også kalt indre klasser, tilgang til variable og metoder i det omsluttende objektet. Dette er ikke tilfellet i C# der objekter av den nestede klassen ikke har tilgang til metoder og variable i objekter av den ytre klassen.

Figur 4.5 på neste side viser relasjonen mellom indre klasser og deres typeparametere i Java. Relasjonene er naturlige og forventede, men kode med indre klasser kan lett bli uoversiktlig. En indre klasse uten typeparametere er generisk med samme typeparametere som den omsluttende klassen. Den indre klassen `AA` i figur 4.5 på neste side viser det eksempel på dette. Slik bruk fører til minst forvirring og misforståelser, og er å anbefale frem-

```
1 class A<T>{
2
3     T v;
4
5     class AA{ // generic on T
6         T v1 = v; // ok
7     }
8
9     class AB<T>{ // generic on T, local T hides outer T
10        // T v1 = v; // error: cannot convert from T to T
11        T v2 = (T) v; // warning: unchecked cast
12    }
13
14    class AC<S>{ // generic on T and S
15        S v2 = (T) v; // warning: unchecked cast
16        S v2 = (S) v; // warning: unchecked cast
17    }
18
19    class AC<S extends T>{ // generic on T and S
20        S s;
21        T v1 = s; // ok
22    }
23
24    static class AD{
25        // T v1; // error: class T is non-static
26    }
27 }
```

Figur 4.5: Nestede og indre klasser i Java

for å gi den indre klassene egne typeparametere.

Figur 4.6 på neste side viser forholdet mellom nestede klasser i C#. I C# har objekter av den nestede klassen ikke tilgang til variable og metoder i objektet av den ytre klassen, men den nestede klassen er likevel generisk med den ytre klassens typeparametere. Det bryter med det nestede klasse-begrepet i C#. Alternativet er at den nestede klassen eksplisitt må gjøres generisk. Dette passer bedre med det man forventer, men kan virke unødvendig tungvint og blir fort uoversiktlig.

I 4.2.2 så vi at det bare var i C# at statiske elementer kan kombineres med typeparametere. Dette har også betydning for hvordan nestede klasser fungerer i generiske klasser. I Java er ikke en nestet statisk klasse generisk med typeparameteren til den ytre klassen. Det er den derimot i C#.

```
1 class A<T>
2 {
3     T v;
4
5     // generic on T
6     class AA
7     {
8         // T v2 = v // error: object reference required
9     }
10
11    // generic on T
12    static class AB
13    {
14        static T w1;
15    }
16 }
```

Figur 4.6: Nestede klasser i C#

4.2.4 Aritmetiske operasjoner

Verken i Java eller C# er det lov å gjøre aritmetiske operasjoner på typeparametere. Dette vil spesielt C++ programmerere reagere på. I C++ er det vanlig å bruke *templates* for å lage numeriske bibliotek. Dette kan gjøres i Java og C# ved å lage egne typer og definere operasjoner i et *interface*. Dette er enklere i Java enn i C#. I Java er de numeriske typeparameterene referanser til klasser som inneholder tallverdier. Disse klassene er subklasser av klassen `Number` og inneholder metoder for å hente ut tallverdiene. Figur 4.7 på neste side viser hvordan man kan utnytte dette til å skrive en metode som summerer alle tallene i en liste.

Å skrive en tilsvarende metode i C# er ikke like enkelt. I C# er det ikke mulig å bruke beskrankning for å begrense lovlige typeparametere til å være primitive typer. For å omgå dette har man flere muligheter. Man kan f.eks. lage egne numeriske klasser. Ulempene med å lage egne numeriske klasser er at man ikke lenger får utnyttet primitive typer som typeparametere, og man får dermed ikke den mulige hastighetsfordelen dette gir.

Eric Gunnerson har i sin weblog skissert en annen løsning inspirert av Anders Hejlsberg [Gun03]. Løsningen går ut på at man lager en abstrakt generisk kalkulatorklasse og uttnytter at det i C# er tillatt å overskrive en generisk metode med en ikke-generisk metode i en subklasse.

```
1 static double sum(IMyGenCollection<? extends Number> c){
2
3     double sum = 0;
4
5     IMyGenIterator<? extends Number> it = c.iterator();
6
7     while(it.hasNext()){
8         sum += it.next().doubleValue();
9     }
10
11    return sum;
12 }
```

Figur 4.7: Summering av en liste med tall i Java

4.2.5 Generisk varians

Jokernotasjon gir mulighet for kovariante og kontravariante generiske typer i Java. Dette gjør de generiske mekanismene mer fleksible. Det er spesielt i generiske metoder dette gir en fordel. Figur 3.3 på side 17 viste at det i noen tilfeller er mulig å skrive om kovariante typeparametere ved eksplisitt å bruke en typeparameter. Ifølge Gilad Bracha [Bra04] er imidlertid metodesignaturer med jokernotasjon å foretrekke fordi de er tydeligere og mer konsise.

Figur 4.8 viser et eksempel på bruk av jokernotasjon som ikke er like enkelt å utrykke uten støtte for kovarians.

```
1 public class MyGenUtil {
2     public static <T> MyGenLinkedStack<T>
3             merge2(IMyGenCollection<? extends T> l1,
4                    IMyGenCollection<? extends T> l2){
5         //...
6     }
7 }
8
9 MyGenLinkedStack<?> stack3 = MyGenUtil.merge2(stack1, stack2);
```

Figur 4.8: Metode med kovariante typeparametere

Som vi har sett er det i noen tilfeller mulig å simulere bruk av jokernotasjon i metodesignaturer, men jokernotasjon kan også brukes utenfor metodesignaturer. Linje 9 i figur 4.8 viser et eksempel på dette.

Et viktig poeng i forbindelse med jokernotasjon er at konseptet kan være vanskelig å forstå. Det konfronterer programmereren med begreper som både er uvante og kompliserte. Det synes likevel klart at den ekstra fleksibiliteten og uttrykkskraften jokernotasjon gir er nyttig, spesielt for ekspertprogrammerere som lager bibliotek. Spørsmålet blir da om denne kompleksiteten er så viktig at det kan forsvares å ha den med. Eksemplet under er et eksempel på en komplisert metodesignatur med jokernotasjon med øvre og nedre beskrankning hentet fra Javas Api.

```
public static <T extends Object & Comparable<? super T>>
        T max(Collection<? extends T> coll)
```

4.2.6 Typesikkert?

Som vi har sett tidligere i oppgaven lar Java-kompilatoren i noen tilfeller generisk kode som ikke er statisk typesikker kompilere uten feil, kun med en advarsel. Her ligger det et potensielt hull i de statisk sikre generiske mekanismene. Godtar man en av disse advarselene er det lett å overse eventuelt nye advarsler. Det er dessuten mulig å gi kompilatoren beskjed om at man ikke ønsker å se slike advarsler. Hvis man ikke er oppmerksom på dette kan de generiske mekanismene i Java gi falsk trygghet.

```
1 public static <T> MyGenLinkedStack merge(/*...*/){
2
3 }
4
5 public static <T> MyGenLinkedStack<T> merge2(/*...*/){
6
7 }
```

Figur 4.9: Typesikker?

Et annet poeng er at bruk av *raw type* ikke alltid gir advarsler eller feil der de er deklarert, men først når de brukes. Figur 4.9 viser to metoder som kompilerer uten feil eller advarsel. Når man prøver å bruke metoden `merge` får man en advarsel, som kan være vanskelig å forstå. Slike feil kan være spesielt vanskelige å finne hvis det er lenge siden koden ble skrevet, eller hvis koden er skrevet av andre.

4.2.7 Kodens lesbarhet

Som vi har sett er det mange likheter mellom syntaksen på de generiske mekanismene i Java og C#. De største forskjellene ligger imidlertid i syntaksen for beskrankning. I C# kommer betingelsene etter selve klasse og metodesignaturen, mens den i Java står i forbindelse med typeparameteren. Javas syntaks for beskrankning er enkel og oversiktlig når signaturene ikke blir for lange. Når det er flere typeparametere og hver av disse har flere beskrankninger, kan det derimot bli uoversiktlig. Det kan også være vanskelig å finne naturlige steder å bryte signaturen i flere linjer. I C# er det enklere å bryte slike linjer. Det kan gjøres ved å sette *where*-betingelsene under hverandre. Dette kan gi ganske oversiktlig kode, se figur 4.10.

```
1 public class VeryLongClassName<S,T,U>
2     where S : IComparable<S>, new()
3     where T : ClassA, ClassB
4     where U : class, new()
5 {
6     //...
7 }
```

Figur 4.10: *Where*-betingelser i C#

En annen fordel i C# er at det er mulig å opprette et alias for en klasse bundet med en typeparameter. Ved fornuftig bruk kan et alias gi mer oversiktlig kode. Den viktigste effekten av dette er nok at det er med på å gjøre lange uttrykk kortere. Faren er at man mister noe av fordelen av en eksplisitt typeparameter.

4.3 Testing av effektivitet

I hovedsak var det to tester jeg ønsket å utføre:

Test 1 Hvilken effekt har de generiske mekanismene på eksekveringshastigheten til de to språkene?

Test 2 Hvilken effekt har bruk av `ArrayList` kontra bruk av vanlig array?

For å besvare spørsmålet i test 1, utførte jeg de samme operasjonene på en generisk stakk og en stakk programmert med *det generiske idiom*. Figur 4.11 viser eksempel på koden som er brukt for å teste hastigheten til en generisk stakk. Testene ble utført både med tall (`int`, `Integer`) og tekststrenger (`String`). I tillegg utførte jeg den samme testen med to ikke generiske stakker som lagret elementene i en `int[]` array og i en `String[]` array.

```
1 private int numOfIterations = 1000000, numOfTests = 50;
2
3 private void testMyGenArrayStackInt()
4 {
5     int element = 1234;
6     int element2;
7
8     MyGenArrayStack<int> stack = new MyGenArrayStack<int>();
9     for (int j = 0; j < this.numOfIterations; j++)
10    {
11        stack.push(element);
12    }
13
14    MyGenArrayStack<int> stack2 = new MyGenArrayStack<int>();
15    for (int j = 0; j < this.numOfIterations; j++)
16    {
17        element2 = stack.pop();
18        stack2.push(element2);
19    }
20 }
```

Figur 4.11: Metode i C# for hastighetstest av generisk stakk med `int`

I C# utførte jeg i tillegg en test mellom den generiske listen `List<E>` og den ikke generiske `ArrayList` fra C#s klassebibliotek. Denne testen ble også utført med både tall og tekststrenger.

For å besvare spørsmålet i test 2, utførte jeg den samme testen, men mellom en generisk stakk med `ArrayList` og en generisk stakk med en vanlig array.

Testene ble kjørt 50 ganger og resultatene i 4.3.1 er gjennomsnittet av kjøringene. Testene er ikke utført for å måle hvorvidt Java eller C# er raskest, men for å teste hvilket språk som har fått mest ut av de generiske mekanismene når det gjelder effektivitet.

4.3.1 Resultater

Tabell 4.1 viser C#s resultater fra test 1. Testen viste ikke uventet en vesentlig hastighetsforbedring ved bruk av primitive typer som aktuelle typeparametere i C#. Den generiske stakken var mer enn 6 ganger raskere enn stakken programmert med *det generiske idiom*. Ved bruk av referanestype var det så godt som ingen hastighetsforskjell mellom stakkene, men den generiske listen `List<E>` var noe tregere enn den ikke generiske listen `ArrayList`.

Tabell 4.1: Generisk vs. generisk idiom i C# (ms)

	`int`	`string`
Generisk stakk	36	69
Stakk (*generiske idiom*)	394	67
Ikke generisk stakk	26	67
List<E>	43	112
ArrayList	387	80

Tabell 4.2 på neste side viser resultatene fra test 1 med Java. I Java var det ikke noen hastighetsgevinst ved bruk av generiske mekanismer og numeriske verdier. Det var heller ikke noen forskjell mellom den generiske stakken og stakken programmert med *det generiske idiom*, ved bruk av referansetyper.

Tabell 4.3 på neste side viser resultatene fra test 2. Når man ikke spesifiserer lengden på `ArrayListen`, er denne betydelig tregere enn stakken med

Tabell 4.2: Generisk vs. generisk idiom i Java (ms)

	`Integer`[1]	`String`
Generisk stakk	531	101
Stakk (*generisk idiom*)	527	104
Ikke generisk stakk	54	100

[1] Den ikke generiske stakken bruker `int[]`

den vanlige arrayen. Med spesifisert lengde var derimot forskjellen mye mindre.

Tabell 4.3: Generisk stakk ArrayList vs vanlig array i Java (ms)

	`Integer`	`String`
Vanlig array	531	101
ArrayList<E>	890	465
ArrayList<E> (spesifisert lengde)	566	147

Testene som ble utført, kan være noe ekstreme i forhold til normal bruk. Ved mer normale forhold vil forskjellene sannsynligvis være mindre. Trolig vil det kun være forskjellen mellom *det generiske idiom* og generisk kode med primitive typer i C# som har stor praktisk betydning. Ved bruk av generiske mekanismer og referansetyper vil det kanskje være andre forhold som har større betydning for kjørehastigheten.

Kapittel 5

Oppsummering

I denne oppsummeringen vil jeg forsøke å svare kort på spørsmålene jeg stilte i innledningen. Tabell 5.1 oppsummerer noen av de viktigste forskjellene mellom de generiske mekanismene i Java og C#.

Beskrivelse	**Java**	**C#**
Primitive typer som aktuelle typeparametere	Nei	Ja
Type informasjon om typeparametere under kjøring	Nei	Ja
Mulighet for å lage objekter av typeparametere	Nei	Ja
Mulighet for å lage arrayer av typeparametere	Nei	Ja
Typeparametere i statiske metoder og variable	Nei	Ja
Statisk typesjekking	Ja	Ja
Jokernotasjon	Ja	Nei

Tabell 5.1: Oppsummering

Uttrykkskraft

En av forskjellene i uttrykkskraft mellom C# og Java ligger i muligheten for bruk av jokernotasjon i Java. Dette gjør Javas typesystem mer fleksibelt og uttrykksfullt, men det er også med på å øke språkets kompleksitet.

Type erasure i Java fører til en del begrensninger. De mest betydningsfulle begrensningene er at det ikke er mulig å opprette objekter eller arrayer av typeparametere. På grunn av begrensningene må man i noen tilfeller bruke

mye tid og ressurser på å uttrykke relativt enkle ting med de generiske mekanismene i Java.

Lesbarhet

Både C# og Java har sine svakheter og styrker når det gjelder lesbarhet av kode. Hvilket språk som oppfattes som mest oversiktlig vil avhenge av hvem som svarer. I C# kan alias brukes til å forkorte lange uttrykk, noe som kan gi mer lesbar kode ved gjennomtenkt bruk. På den annen side kan jokernotasjon i Java i noen tilfeller gjøre kompliserte metodesignaturer mer oversiktlige og konsise.

Syntaksen passer relativt godt inn i begge språkene.

Integrasjon

I Java og C# har man valgt forskjellige tilnærminger til de generiske mekanismene. Det kan se ut som om de generiske mekanismene i C# passer best inn i det eksisterende språket. Språket fungerer i de fleste tilfeller slik man forventer, kanskje med unntak av nestede klasser og programmering med statiske metoder og variable. I C# er det ikke noe problem å bruke språket uten å bruke de generiske mekanismene.

I Java er det derimot flere uventede og ved første øyekast uforståelige begrensninger. De fleste begrensningene skyldes *type erasure*. Grunnet det endrede klassebiblioteket i Java, finnes det ikke innebygde alternativer til bibliotekets generiske datastrukturer. Disse kan brukes med *raw type*, noe som kan gi problemer ved bruk sammen med generiske klasser eller metoder.

Effektivitet

Det er ingen tvil om at primitive typer som typeparametere gir de generiske mekanismene i C# en hastighetsgevinst i forhold til programmering med *det generiske idiom*. Fordelen forringes derimot noe, siden det ikke er mulig å gjøre aritmetiske operasjoner direkte på typeparametere.

Som vist i oppgaven er det ikke problemfritt å innføre nye mekanismer og begrep i eksisterende programmeringsspråk. Det er derfor viktig å studere endringene av de eksisterende språkene for å sikre heldige utfall ved innføring av nye endringer.

5.1 Videre arbeid

For videre arbeid finnes det mange interessante problemstillinger.

Først og fremst vil det være interessant å studere hvordan de generiske mekanismene i Java og C# fungerer sammenlignet med virtuelle typer. En sammenligning med NextGen, et av forslagene til generiske mekanismer i Java som likner mye på C#, vil også være av interesse.

En studie av hvordan de generiske mekanismene fungerer på tvers av språkene på .NET-plattformen, og en sammenligning av de generiske mekanismene i disse vil også være interessant.

Videre kan det også være vel verdt å studere hvor godt egnet C# egentlig er til tyngre simuleringer og beregninger ved bruk av generiske mekanismer.

Bibliografi

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 49–65, New York, NY, USA, 1997. ACM Press.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 523–549, London, UK, 1998. Springer-Verlag.

[Bra04] Gilad Bracha. Generics in the java programming language, July 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[CGLS98] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with run-time types for the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 201–215, New York, NY, USA, 1998. ACM Press.

[Coo89] W. R. Cook. A proposal for making eiffel type-safe. *Comput. J.*, 32(4):305–311, 1989.

[Cor01] Microsoft Corporation. The .NET Common Language Runtime. Website, 2001. http://msdn.microsoft.com/net/ (2006-05-15).

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[Gun03] Eric Gunnerson. Generics algorithms. Weblog, 2003. http://blogs.msdn.com/ericgu/archive/2003/11/14/52852.aspx (2006-05-20).

[KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[Mey86] Bertrand Meyer. Genericity versus inheritance. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 391–405, New York, NY, USA, 1986. ACM Press.

[MMMP90] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 140–150, New York, NY, USA, 1990. ACM Press.

[SA98] Jose H. Solorzano and Suad Alagić. Parametric polymorphism for java: a reflective solution. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 216–225, New York, NY, USA, 1998. ACM Press.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wild-

cards to the java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.

[Tho97] Kresten Krab Thorup. Genericity in java with virtual types. *Lecture Notes in Computer Science*, 1241:444–471, 1997.

[TT99] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 186–204, London, UK, 1999. Springer-Verlag.

[vdA05] Peter von der Ahé. Considered harmful, Peter Ahé's weblog. Weblog, July 2005. http://blogs.sun.com/roller/page/ahe?entry=considered_harmful (2006-05-15).

[VE04] Bill Venners and Bruce Eckel. Generics in C#, Java, and C++, A Conversation with Anders Hejlsberg, part vii. Website, January 2004. http://www.artima.com/intv/generics.html (2006-05-15).

Tillegg

Oversikt over vedlagt cd

Dette er en oversikt over filene på den vedlagte cd'en. Mappene *CSharpExamples* og *JavaExamples*, som inneholder kildekoden, er i utgangspunktet nesten identiske. I tillegg til kildekode finnes dokumentasjon av koden i mappen *Doc*.

MyGenCollection

Pakken `MyGenCollection` består av noen generiske datastrukturer og metoder for å manipulere disse.

Navnene på klassene er laget så beskrivende som mulig. Navnet på stakkene beskriver også hvordan objektene lagres i stakken. Interface har en 'I' i begynnelsen av navnet for å skille dem fra klassene.

MyCollection

Pakken `MyCollection` består av 3 ikke generiske stakker som er brukt i hastighetstesten.

PerformanceTest

Pakken `PerformanceTest` består av de to klassene `Timer` og `Tests` som er brukt i hastighetstesten.

Programfiler

Java-filene *TestMyGenCollection.java* og *PerformanceTests.java*, som inneholder main-metoder, ligger direkte i mappen *JavaExamples*. I mappen *CSharpExamples* finnes klassene som inneholder main-metoder i pakkene *PerformanceTest* og *TestMyGenCollection*.