Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Het Hoe en Waarom van Generics in Java

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

April 7, 2009

On Java

- Ontwikkeld bij Sun (Gosling et al.)
- Objectgeoriënteerd en imperatief
- Grote meegeleverde bibliotheek van klassen
- “Industry strength”
- Voor onze studenten hun eerste ontmoeting met programmeren tijdens de studie.
- Eerste versie 1.1.4 dateert uit 1997, eerste versie met generics, 1.5.0 uit 2004.

Klassen, velden, methoden en objecten

```
class Voertuig {
  int aantalWielen;
  double waarde;

  public Voertuig(int aantal, double prijs) {
    int aantalWielen = aantal;
    waarde = prijs;
  }

  double geefWaarde() {
    return waarde;
  }
}

Voertuig fiets = new Voertuig(2, 270.0);
```

Subtype-polymorfie in Java

- Java vooral gebaseerd op subtype polymorphism.
  - Ook wel inheritance polymorphism
- Waar `Voertuig` staat, mag ook een `Fiets`.
  - want een `Fiets` is een speciaal soort `Voertuig`
- Andersom werkt het natuurlijk niet.

```
class Fiets extends Voertuig {

  public Fiets(double prijs) {
    super(2, prijs);
  }
}

Fiets fiets = new Fiets(270.0);
```

Generics in Java

- Sinds versie 1.5.0 kent Java ook generics.
  - Ook wel parametric polymorphism
- en daar bleef het toen niet bij:
  - variable arity methods, auto(un)boxing, enums
- Er is redelijk veel kritiek gekomen op de generics:
  - uit de wetenschap: Smith en Cartwright.
  - van programmeerforums: compilers doen raar en foutmelding kloppen niet.

Overloading

- Naast parametric en subtype polymorphism is er nog een andere vorm van polymorfie in Java: ad-hoc polymorphism.
- Voorbeeld: de operator + werkt zowel op strings, integers en floating point getallen.

Vergelijkend warenonderzoek

- Ad-hoc polymorphism: zelfde symbool, per type andere implementatie
- Parametric polymorphism: zelfde symbool, zelfde implementatie
- Subtype polymorphism: zelfde symbool, bij default zelfde implementatie, maar
- het is door overrides mogelijk hiervan af te wijken.
  - Te vaak doen gaat ten koste van de onderhoudbaarheid.

Het programma

- Hoe werken generics in Java nu eigenlijk?
- Waarom werken ze zo?
- Wat is er mis met Java generics?
- Wat is er mis met de Java compilers?
- Wordt daar ook iets aan gedaan?

1. Generics in Java

De oude gang van zaken


```
void bestuur(Voertuig v) { ... }

LinkedList stalling = new LinkedList();
stalling.add(new Fiets()); // Ok
stalling.add(new String()); // Ook ok

.... veeeeeel code ....

Object object1 = stalling.get(0); // De fiets
Object object2 = stalling.get(1); // De string
// bestuur(object1); // Fout, al is het een Fiets
// bestuur(object2); // Mag ook niet
bestuur((Voertuig)object1); // Mag wel
bestuur((Voertuig)object2);
  // Mag ook, maar leidt tot run-time exception
```

Vermijd run-time exceptions


```
Object object1 = stalling.get(0); // De fiets.
Object object2 = stalling.get(1); // De string.
if (object1 instanceof Voertuig) {
    bestuur((Voertuig)object1);
}       // Mag wel, en gebeurt ook
if (object2 instanceof Voertuig) {
    bestuur((Voertuig)object2);
}       // Mag wel, maar wordt niet uitgevoerd.
```

Conclusies over Java voor Generics

- Collectie klassen zijn heterogeen.
  - Alles mag erin, `Objects` mogen er uit.
- Items uit een collectie dienen geregeld te worden gedowncast.
- Dit kan mis gaan, dus
- programmeur moet nagaan dat downcast gaat slagen.
- Flexibiliteit collectieklassen kan handig zijn, maar
- je betaalt er ook een prijs voor als alle elementen hetzelfde type hebben.
- En er zijn geen compile-time garanties.
- Merk op: altijd downcast testen, ook als je (nu) weet dat hij slaagt.
  - Essentieel voor onderhoudbare code.

De oplossing in een notedop

- Sta de programmeur toe preciezer aan te geven wat voor soort objecten in de lijst mogen zitten.
- Java compiler gaat na dat de programmeur zich daar aan houdt.

```
LinkedList<Voertuig> stalling =
                new LinkedList<Voertuig>();
stalling.add(new Fiets()); // Mag wel
stalling.add(new Voertuig()); // Mag wel

// stalling.add(new String()); // Mag niet meer
```

Is dit veilig?

- Zolang je geen gekke dingen doet wel.
- Maar Java staat je altijd toe je eigen graf te graven.

```
stalling.add((Voertuig)(Object)new String());
    // Mag wel maar levert run-time exception.
```

- Er is wel een cruciaal verschil met de eerdere run-time exception. Wat?

Is dit veilig?

- Zolang je geen gekke dingen doet wel.
- Maar Java staat je altijd toe je eigen graf te graven.

```
stalling.add((Voertuig)(Object)new String());
    // Mag wel maar levert run-time exception.
```

- Er is wel een cruciaal verschil met de eerdere run-time exception. Wat?
- De run-time exception krijg je bij het erin stoppen, niet bij het eruit halen.

Voertuig besturen zonder downcasts §1

```
LinkedList<Voertuig> stalling =
                new LinkedList<Voertuig>();
stalling.add(new Fiets());
stalling.add(new Voertuig());
Voertuig voertuig1 = stalling.get(0); // De fiets
Voertuig voertuig2 = stalling.get(1); // Het voertuig

bestuur(voertuig1);
bestuur(voertuig2);

if (voertuig1 instanceof Fiets) {
  plakBand((Fiets)voertuig1); // Verder naar beneden
}
```

Collectie-objecten als methode-argument


```
void vervang(AbstractList<Voertuig> wagenpark,
             Voertuig nieuwVoertuig) { }

vervang(new LinkedList<Voertuig>, new Voertuig());

LinkedList<Fiets> fietsenStalling =
   new LinkedList<Fiets>();
vervang(fietsenStalling, new Voertuig()); // FOUT!!
vervang(fietsenStalling, new Fiets()); // OOK FOUT!!
```

Wat doen we verkeerd?

- Je mag wel `LinkedList<Voertuig>` meegeven aan een parameter van type `AbstractList<Voertuig>`,
- Maar je mag geen `LinkedList<Fiets>` of `LinkedList<Object>` meegeven.
- Collectie-klassen zijn invariant: elementtypen moeten identiek zijn, dwz. geen subtyping toegestaan.
- Let op: een `LinkedList<Voertuig>` mag wel objecten van type `Fiets` bevatten.

Waarom invariantie voor collectie-klasse


```
void vervang(AbstractList<Voertuig> wagenpark,
             Voertuig nieuwVoertuig) { }
```

- Wat zou je in `vervang` verwachten te mogen doen:
  1. willekeurige `Voertuigen` in het `wagenpark` zetten.
  2. `Voertuigen` uit het `wagenpark` halen.
- Als je een lijst van `Fietsen` meegeeft, leidt 1. tot strijdigheid.
- Als je een lijst van `Objecten` meegeeft, leidt 2. tot strijdigheid.
- Conclusie: alleen een lijst van voertuigen voldoet.

Een methode die werkt op alle lijsten

- Door invariantie voldoet

  ```
  int lengte(LinkedList<Object> ll)
  ```

  niet om de lengte van een willekeurig lijst uit te rekenen.
- Maar wat voldoet dan wel?
- `int lengte(LinkedList<?> ll)`
  - `?` noemen we wildcard
- Wat mag je met `LinkedList<?>`?
  - Je mag er niets in stoppen.
  - Je kunt er alleen `Objects` uithalen.
- `?` staat letterlijk voor onbekende.
- Let op: `void wissel(Set<?> s1, Set<?> s2)` zal niet kunnen doen wat je verwacht.
  - Zelfde syntax impliceert niet zelfde type.

Typevariabelen

- Maar hoe definieer je dan zo'n functie als `wissel`?
- `<T> void wissel(Set<T> s1, Set<T> s2)`
- Nu heeft de parameter van de collectie-klasse een naam, `T`.
- En: gelijke naam betekent gelijk type.
  - Binnen de scope van de typevariabele

Maar wacht eens even...

- Waarom schrijven we dan niet:

```
<T> int lengte(LinkedList<T> ll)?
```

- Of betekent dit iets anders?

Maar wacht eens even...

- Waarom schrijven we dan niet:

```
<T> int lengte(LinkedList<T> ll)?
```

- Of betekent dit iets anders?
- Nee, maar het gebruik van `T` zegt dat voor elke legale aanroep een concrete `T` gevonden moet kunnen worden.
- Wildcard `?` staat voor onbekend type, dus daar hoeft dat niet voor.

Bounds: `extends`

- Stel je wilt een methode schrijven die werkt voor alle lijsten waarvan de elementen subtype zijn van `Number`.
  - Zowel `LinkedList<Number>` als `LinkedList<Double>`
- Voldoet `void doeIets(LinkedList<Number>)`?
  Of `void doeIets(LinkedList<Object>)`?
  Of `void doeIets(LinkedList<?>)`?
- Nee, maar wel:
  `void doeIets(LinkedList<? extends Number>)`
  - Je mag er zowel `LinkedList<Integer>` en `LinkedList<Double>` in stoppen.
  - Je mag echter niets in die lijst stoppen.
  - Je kunt er alleen `Numbers` uithalen.

Bounds: `super`

- Ook mogelijk:
  `void doeIets(LinkedList<? super Number>)`
  - `super` geeft een ondergrens aan
- Hier mag je van elementen van elk subtype van `Number` stoppen.
  - Want wat `?` in een concrete aanroep ook is, je weet dat je er een `Number` in mag stoppen.
- En er alleen `Objects` uithalen.
  - Want in het "slechtste" geval is `?` gelijk aan `Object`
- Iets anders is niet veilig, maar downcasten mag.
- Wat is `LinkedList<? super Object>`?

Wildcards zijn niet beperkt tot signaturen §1

```
LinkedList<? extends Voertuig> le =
    new LinkedList<Fiets>();
//le.add(new Fiets); Mag niet
Voertuig v = le.element();
le = new LinkedList<Auto>();

LinkedList<? super Fiets> ls =
    new LinkedList<Voertuig>();
ls.add(new Fiets());
Object o = ls.element();
ls = new LinkedList<Object>();
```

Complicaties met wildcards

- De wildcard ? mag alleen als type parameter.

```
<T> void negeer(T teNegeren) { } // Mag wel
// void negeer(? teNegeren) {} // Mag niet
```

Complicaties met typevariabelen

- Je kent de constructor niet, dus `new T` is verboden.
- Syntactische restricties op bounds.
- Geen type casts, geen instanceof.

```
<T> void zelfde(T t1, T t2) {
 //T x = new T(); // Mag niet
   T x = t1; // Mag
}

void magWel(List<? super Number> l) { }
// <T> void magNiet(List<T super Number> l) { }
<T extends Number> void magWeerWel(List<Tl) { }
```

Capture conversion

- In beperkte mate mag je toch de identiteit van een `?` kennen.
- Zolang die kennis maar niet ontsnapt.
- Standaard voorbeeld: `reverse(LinkedList<?> l)`
- Roept hulpmethode `<T> void rev(List<T> list)` aan.
- Overgang `?` naar `T` heet capture conversion.
- Run-time weten we precies wat `T` is, en informatie over `T` kan niet ontsnappen.
- Ad-hoc: werkt wel op `Set<?>`, maar niet op `Set<Set<?>>`.

2. Wat is er mis en wat is er aan te doen?

Veel ad-hoc regels

- Verschillende soorten constraints (gelijkheid, extends en super). Worden apart behandeld.
- Soms gebruik van assignment context, soms niet.
- Capture conversion.
- Verschil vrije `T` en `?`.
  - Precieze afbakening is mij niet bekend.
- Vooralsnog onverklaarbare restricties op bounds voor typevariabelen.
- Alleen experts kennen de taal echt goed; veel experts zijn er niet.

Generics bestaan niet tijdens executie

- Voordeel: geen run-time kosten.
- Nadeeltje: geen run-time inspectie mogelijk van typeparameters.
- Nadeel: onverwachte naambotsingen tussen methoden.

```
void foo(LinkedList<Number> ln) { }
void foo(LinkedList<Integer> li) { }
```

Java type inferentie schiet tekort

- Smith en Cartwright (OOPSLA '08): Java type inference is broken. Can we fix it?
- De conclusie: bepaalde aspecten van Generics, en dan vooral de wildcard, waren nog niet voldoende goed begrepen.
- Maar het is niet zo broken als het lijkt.
- Oplossing: verder doorgronden, en hopen dat de volgende versie beter is.

Foutmeldingen compilers schieten te kort §2

- Taalspecificatie zegt alleen welke programma's goed zijn en welke niet.
- Compilers moeten ook uitleggen waarom een programma fout is.
- Hierin schieten ze tekort:
  - Foutmeldingen niet informatief.
  - Compilers volgen specificatie niet.
  - Compilers volgen specificatie te nauwlettend.

Foutmeldingen niet informatief


```
<T> void foo(Map<T,T> a){
  Map<Number, Integer> m1 = ...;
  foo(m1);
```

ejc:

```
1. ERROR in Listing1.java (at line 6)
    foo(m1);
The method foo(Map<T,T>) ... is not applicable
for the arguments (Map<Number,Integer>)
```

JAVAC zegt iets vergelijkbaars.

Compilers volgen specificatie niet

§2

Sun's JAVAC accepteert het volgende programma:

```
<T extends Number> void foo(List<? super T> a)
...
List<String> x = ...
foo(x);
```

- Waarom is dit fout?
- `foo` werkt alleen voor types tussen `Number` en `Object`. Dus niet voor `String`.

Compilers volgen specificatie niet

Sun's JAVAC accepteert het volgende programma:

```
<T extends Number> void foo(List<? super T> a)
...
List<String> x = ...
foo(x);
```

- Waarom is dit fout?
- `foo` werkt alleen voor types tussen `Number` en `Object`. Dus niet voor `String`.
- Waarom gaat dit mis?
- Conditie `T extends Number` wordt genegeerd.
- Conclusie: vertrouw niet blindelings op de compiler.

Vreemd gedrag, deel 1


```
<T extends Number>
  void foo(Map<? super T, ? super T> a)
...
Map<String, Number> m = ...;
foo(m);
```

ejc:

```
1. ERROR in Listing5.java (at line 10)
  foo(m);
Bound mismatch: The generic method foo(
 Map<? super T, ? super T>) of type Listing5 is not
 applicable for the arguments (Map<String,Number>).
 The inferred type String is not a valid substitute
 for the bounded parameter <T extends Number>
```

Vreemd gedrag, deel 2


```
<T extends Number>
  void foo(Map<? super T, ? super T> a)
...
Map<Number, String> m = ...;
foo(m);
```

ejc:

```
1. ERROR in Listing5.java (at line 10)
  foo(m);
The method foo(Map<? super T,? super T>) in the
 type Listing5 is not applicable for the arguments
 (Map<Number,String>)
```

De reden: de volgorde waarin EJC constraints oplost bepaalt mede de foutmelding, niet alleen de constraints zelf.

Eigen werk

- Nabil el Boustani en Hage (PEPM '09).
- Verbeteren foutmeldingen voor generieke methode-aanroepen.
- Uitgangspunten
  - Vingers af van het type-checkenproces.
  - Hou meer informatie bij de hand, en langer.
  - Negeer vooraleer alles wat generiek is.
    - Want als dat fout is, faalt typeren te vroeg om een goede foutmelding te geven.
- Implementatie in JastAdd Extensible Java Compiler

Foutmelding informatiever


```
<T> void foo(Map<T,T> a){
  Map<Number, Integer> m1 = ...;
  foo(m1);
```

ours:

```
Listing1.java:6
Method <T>foo(Map<T, T>) of type Listing1 is
  not applicable for the argument of type
  (Map<Number, Integer>), because:
  [*] The type variable T is invariant, but
  - Integer in Map<Number, Integer> on 5:9(5:21)
  - Number in Map<Number, Integer> on 5:9(5:13)
  are not the same type.
```

Verder werk

- Nabil el Boustani en Hage (opgestuurd aan OOPSLA '09)
- Gebruik heuristieken ten einde programmeur suggesties te geven hoe zijn programma te verbeteren.
  - compenseert voor (arbitraire) keuzes in het type-checkenproces,
  - encapsuleert kennis van expert over wat er fout kan zijn,
  - herkent en corrigeert vaakgemaakte fouten.

Compenseren arbitraire keuzes

- Typeringsprobleem is vaak geformuleerd als constraint satisfaction problem:
  - $\texttt{T} = \texttt{Number}, \texttt{T} = \texttt{Integer}, \texttt{T} <: \texttt{Integer}$
- Volgorde waarin constraints worden opgelost bepaalt in sterke mate foutmelding
  - `T` wordt op basis van eerste constraint op `Number` gezet
- Foutmelding gaat uit van die keuze: wat afwijkt is fout.
- Gebruik een heuristiek die een betere afweging maakt:
  - Als $\texttt{T} = \texttt{Integer}$, aan twee constraints voldaan.
  - Als $\texttt{T} = \texttt{Number}$, aan één constraint voldaan.
- Stel voor aan de programmeur om `Number` te vervangen door `Integer`.

Voorbeeldfoutmelding met verbetersuggestie §2

```
<T> List<T> foo(Map<T, ? super T> a){}
...
Map<Number, Integer> m = null;
List<Integer> ret = foo(m);
```

```
cxt_heuristic/Test1.java:6
Method <T>foo(Map<T, ? super T>) of type
Test1 is not applicable for the argument of
type (Map<Number, Integer>), because:
  [*] The type Integer in Map<Number, Integer>
  on 5:9(5:21) is not a supertype of the inferred
  type for T: Number. However, replacing Number
  on 5:13 with Integer may solve the type conflict.
```

Foutpatroon: verwarren `extends` en `super` §2

```
<T> void foo(Map<? extends T, T> a) {}
...
Map<? super Integer, Number> m2 = ...;
foo(m2);
```

```
Test3.java:16
Method <T>foo(Map<? extends T, T>) of type
Test3 is not applicable to the argument of
type(Map<? super Integer, Number>), because:
  [*] The type Map<? extends T, T>, where T
  was inferred to be Number is not a supertype
  of Map<? super Integer, Number> on 15:9.
  However Map<? extends Integer, Number> is a
  subtype of Map<? extends T, T>
```

Foutpatroon: onnodige wildcards


```
<T> void foo(Map<T, T> a) {}
...
Map<? super Object, ? super Object> m = ...;
foo(m);


Test1.java:6
Method <T>foo(Map<T, T>) of type Test1 is not
applicable to the argument of type
(Map<? super Object, ? super Object>), because:
  [*] The type variable T is invariant, but the
  type '? super Object' is not. However, replacing
  - '? super Object' on 5:13
  - '? super Object' on 5:28
  with Object may solve the type conflict.
```

Verder werk?

- Is er niet.
- Waarom laat zich raden, maar
  - iedereen ziet graag dit soort werk gedaan, maar niemand voelt zich geroepen.
  - Typische opmerking: Java-programmeurs maken niet zo vaak typeringsfouten.
    - Klopt redelijk, tot de toevoeging van generics

Wat hebben we gezien?

We bespraken:

- Hoe werken Generics in Java?
  - Maar toch lang niet alles
- Waarom werken ze zo?
- Wat is er allemaal mis?
  - Met de taal en met de tools
- Wat wordt daar aan gedaan?
  - Te weinig