„Type Erasure“ in Java 5

Helmi Jouini
Institut für Theoretische Informatik
Universität Karlsruhe

Warum Generics?

- Containerklassen in Java 1.4 sind generisch nutzbar aber typunsicher.

```
Public class LinkedList
{
    Public class Node
     {
        Object entry;
        Node next;
     }

    private Node root;
    void add(Object obj){…}
    Object getAt(int pos){…}
}
```

```
LinkedList l = new LinkedList();
l.add(new Interger(5));
Integer i0 = (Integer)l.getAt(0);
String s = new String(„string“);
l.add(s);
Integer i1 = (Integer)l.getAt(1); //Laufzeit
```

Mit Generics

```
public class LinkedList<T>
{
    protected class Node
    {
        T entry;
        Node next;
    }

    Node root;
    void add(T obj){…}
    T getAt(int pos){…}
}
```

```
LinkedList<Integer> l = new
    LinkedList<Interger>();
l.add(new Interger(5));
Integer i = l.getAt(0);
String s = new String(„string“);
l.add(s); //Übersetzerfehler
String str = (String)l.getAt(0); //Über.
```

Anforderung an Java 5

- Typsicherheit zur Übersetzerzeit.
- Kompatibilität mit Java 1.4.

Typsicherheit

- Wenn sich ein Programm fehler- und warnungsfrei übersetzen lässt, dann ist eine unerwartete ClassCastException zur Laufzeit ausgeschlossen.

- Diese Garantie macht eine Programmiersprache typsicher.

- Eine "unerwartete" ClassCastException wäre eine, die ohne einen entsprechenden Cast im Sourcecode entsteht.

Implementierung von Generics(1)

- Type Erasure = Entfernung der Informationen über den parametrisierten Typ.
- Eine Übersetzungseinheit pro parametrisierbaren Typ.
- LinkedList<Integer> , LinkedList<String> => LinkedList.class

Implementierung von Generics(2)

- Es gibt nur ein Klassen-Objekt für alle Instanziierungen eines generischen Typs

- Source

```
public class Stack<T>{

    public void push(T t){
        data.add(t);
    }
    public T pop(){
        return data.remove(data.size()-1);
    }
    private List<T> data;
}
```

- Bytecode(.class)

```
public class Stack{
    public void push(Object o){
        data.add(o);
    }
    public Object pop(){
        return (Object)data.remove
        (data.size()-1);
    }
    private List data;
}
```

Implementierung von Generics(3)

```
LinkedList<Integer> l = new
    LinkedList<Integer>();
l.add(new Interger(5));
Integer i = l.getAt(0);
```

```
LinkedList l = new LinkedList();
l.add(new Integer(5));
Integer i = (Integer)l.getAt(0);
```

- Ersetzung des Parametertyps durch den „Upper bound“ bei der Klassendefinition.

- Hinzufügen geeigneter Typkonvertierungen beim Aufruf.

Folgerung

- Keine Typüberprüfung zur Laufzeit möglich.
- Kein Unterschied mehr zwischen LinkedList<Integer> und LinkedList<String> zur Laufzeit.
- Übersetzter Code entspricht dem Code einer LinkedList ohne Generics.
- Java 5 Bytecode von VM 1.4 ausführbar (theoretisch).
- Primitive Typen sind nicht zulässig.

Auswirkung von Type Erasure(1)

- Erwartete Ausnahme nicht geworfen.

```
LinkedList<Integer> i_list;
Object o = new LinkedList<String>();
i_list = (LinkedList<Integer>)o;
```

- Gefährlich und verwirrend bei der Fehlersuche.

- Gefährdet die Typsicherheit von Java.

Auswirkung von Type Erasure(2)

- Keine Laufzeitoperationen auf parametrisierte Typen.

```
LinkedList<Integer>.class;    //unzulässig.
i_list instanceof LinkedList<Integer> //unzulässig.
parametrisierte Exception sind nicht zulässig.
T t = new T(); // unzulässig.
```

Auswirkung von Type Erasure(3)

- Keine parametrisierten Typen als Array-Elemente.

```
LinkedList<Integer>[] lists = new LinkedList<Integer>[5];
```

- Heftige Einschränkung wenn man ernsthaft mit Generics arbeitet.
- Wäre das erlaubt, wäre Java keine typsichere Programmiersprache.
- Warum?

Arrays mit parametrisierten Typen

```
public class Flasche<T>{…} //T Inhalt
Object[] kasten = new Flasche<Wasser>[6];
kasten[0] = new Flasche<Milch>();
Flasche<Wasser> f = (Flasche<Wasser>)kasten[0];
Wasser w = f.entleeren();
```

- Angenommen Array-Konstrukt zulässig.
- Übersetzung fehler- und warnungsfrei.
- Milchflasche in dem Wasserkasten.
- Array Store Check kann es wegen Type erasure nicht verhindern.
- ClassCastException beim Entleeren der Milchflasche als Wasserflasche.

Statische Attribute

- Ein statisches Attribut gehört allen instanziierten Typen.

- Folgen:

```
class Container<T>{
        private static T t;  //nicht zulässig.
}
```

  Zählen der Exemplare generischer Typen unmöglich.

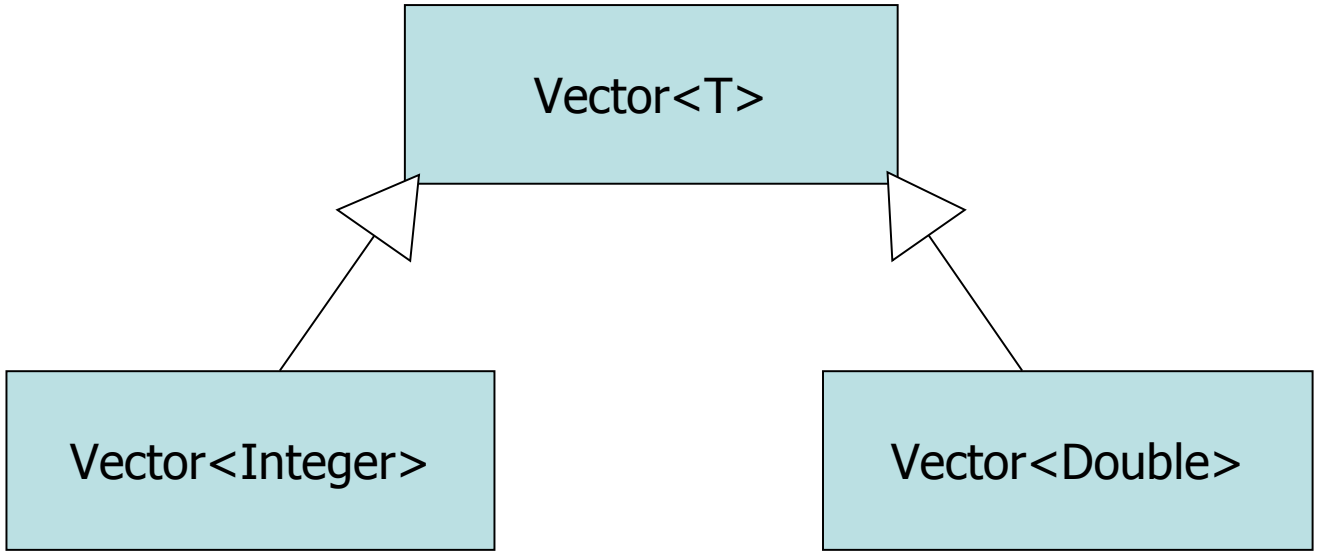
NextGen

- Weiterentwicklung von Generic Java.
- Kein Type erasure => Keine Einschränkungen.
- Informationen über Parametertyp zur Laufzeit vorhanden.
- Erweiterung des Übersetzers.
- Class Loader.

NextGen Implementierung(1)

- Bilde die abstrakte „type erased“ Klasse C zu der Klasse C<T>.
- Erzeuge zu jeder Instanzklasse zur Laufzeit (by demand) eine Unterklasse von C.
- Jede erzeugte Klasse verfügt über notwendige weiterleitende Konstruktoren.
- Erzeuge zu jeder Operation, die vom dynamischen Typ abhängt, eine abstrakte Methode („snippet“) in der Klasse C.
- „Snippet“ Methoden werden in den Instanzklassen implementiert.

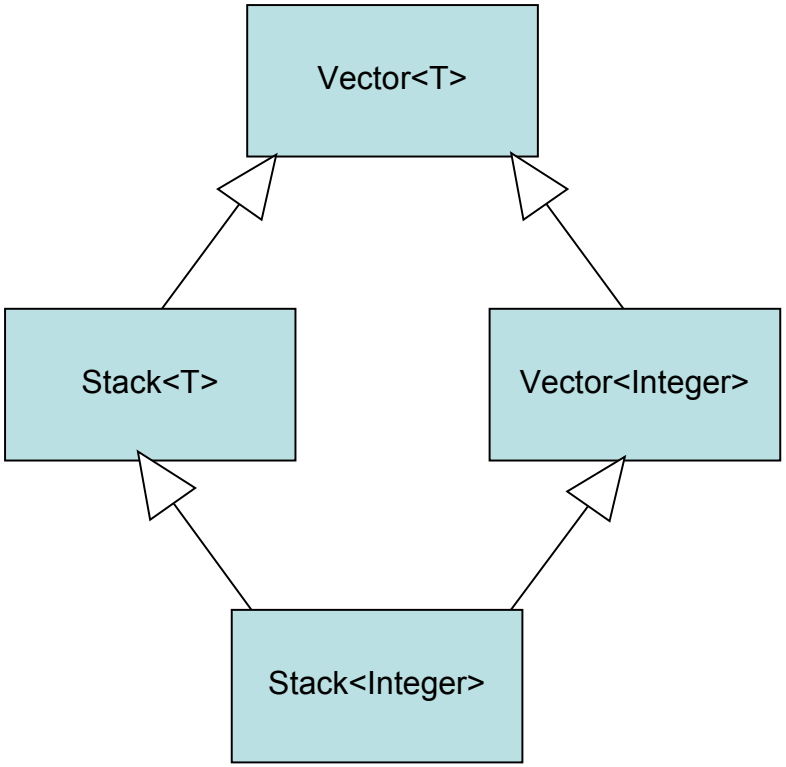
NextGen Implementierung(2)



Naive Implementierung

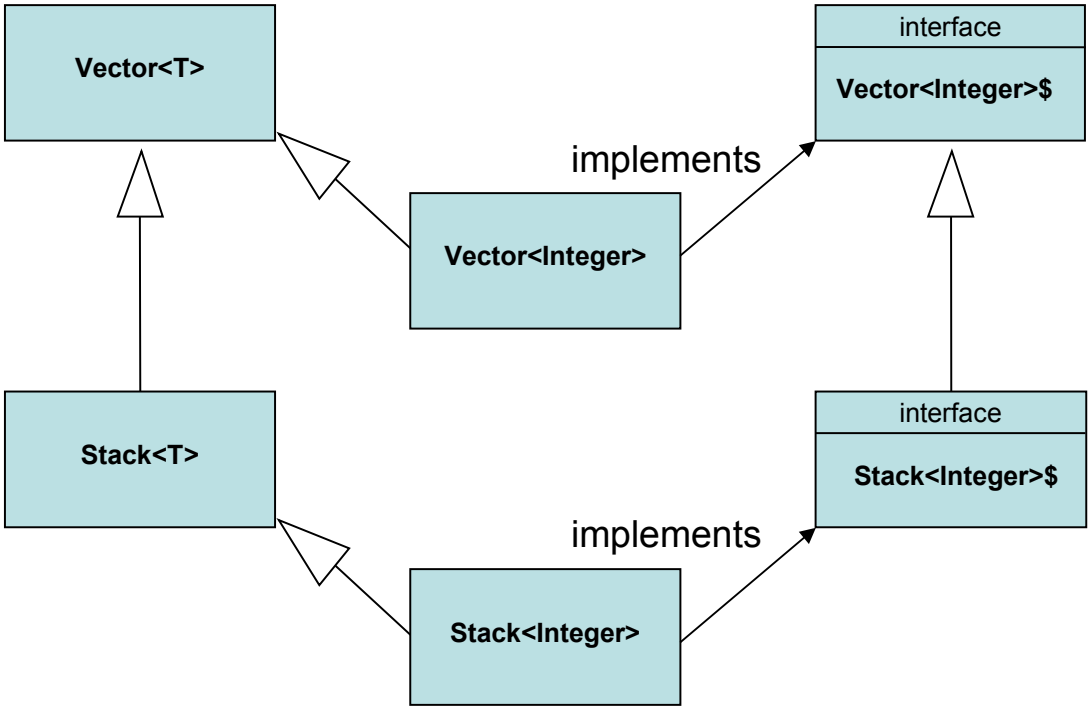
- NextGen Implementierung ist komplexer als die oben dargestellte Implementierung, denn...

NextGen Implementierung(3)



- Mehrfachvererbung ist verboten in Java.

NextGen Implementierung(4)



- Zulässig in Java.

NextGen Implementierung(5)

- Bilde die abstrakte „type erased“ Klasse C zu der Klasse C<T>.
- Erzeuge zu jeder Instanzklasse zur Laufzeit (by demand) eine Unterklasse von C.
- Bilde zu jeder Instanzklasse eine leere Superschnittstelle.

- Jede erzeugte Klasse verfügt über notwendige weiterleitende Konstruktoren.
- Erzeuge zu jeder Operation, die vom dynamischen Typ abhängt, eine abstrakte Methode („snippet“) in der Klasse C.
- „Snippet“ Methoden werden in den Instanzklassen implementiert.

Name Mangling

- VM verbietet Bezeichner mit spitzen Klammern.
- Name Mangling notwendig.
- Vector<Integer> => $$Vector$_Integer_$ und Vector<Integer[]> => $$Vector$_array$$_Integer_$_$.
- Superschnittstellen werden nur mit einem führenden $ bezeichnet $Vector$_Integer_$.
- NextGen Übersetzer verbietet das Zeichen $ in den Bezeichnern, damit eine eventuelle Namenskollision vermieden wird.

Snippet Methoden

- Sind Methoden für die Implementierung der Operationen, die vom Typ der Argumente abhängen, wie new, instanceof und cast.

- Werden in der oberen Klasse als abstract definiert und in den Instanzklassen implementiert.

- Aus Performance-Gründen werden Snippets inline deklariert (final).

Beispiel(1)

```
public class Vector<T>
{
    private T[] elements;
    public Vector(int capacity){
        elements = new T[capacity];
    }
    public T elementAt(int idx){
        return elements[idx];
    }
    public void setElementAt(T e, int idx)
    {
        elements[idx] = e;
    }
}
```

```
public abstract class Vector
{
    private Object[] elements;
    public Vector(int capacity){
        elements = $snip$1(capacity);
    }
    public Object elementAt(int idx){
        return elements[idx];
    }
    public void setElementAt(Object e, int idx)
    {
        elements[idx] = e;
    }
    abstract protected Object[] $snip$1(int
    capacity);
}
```

Beispiel(1)

```
public interface $Vector$_Integer_$ {}

public class $$Vector$_Integer_$ extends Vector
                                        implements $Vector$_Integer_$
{
    public $$Vector$_Integer_$ (int capacity){
        super(capacity);
    }
    protected final Object[] $snip$1(int capacity){
        return new Integer[capacity];
    }
}
```

- final aus Performance-Gründen.

Was ist, wenn die Klasse nicht sichtbar ist?

Snippet Modellierung(1)

- Man könnte die Sichtbarkeit der Klasse manipulieren.
- Einfache Lösung, verletzt aber die Sicherheit der Sichtbarkeit.
- Gibt es eine andere Lösung ohne Verletzung der Sichtbarkeit?
- Snippets können als Environment-Klassen modelliert werden.

Snippet Modellierung(2)

```
public abstract class Vector
{
    private Object[] elements;
    public Vector(int capacity){
        elements = $snip$1(capacity);
    }
    public Object elementAt(int idx){
        return elements[idx];
    }
    public void setElementAt(Object e, int
    idx)
    {
        elements[idx] = e;
    }
    abstract protected Object[] $snip$1(int
    capacity);
}
```

```
public abstract class Vector
{
    private Object[] elements;
    public Vector(int capacity){
        elements = $snip$1(capacity);
    }
    public Object elementAt(int idx){
        return elements[idx];
    }
    public void setElementAt(Object e, int idx)
    {
        elements[idx] = e;
    }
    public abstract static class $IsolatedSnippets{
        Object[] $1(int capacity);
    }
}
```

Snippet Modellierung(3)

```
public class $$Vector$_Integer_$ extends
    Vector
    implements $Vector$_Integer_$
{
    public $$Vector$_Integer_$ (int
    capacity){
        super(capacity);
    }
    protected final Object[] $snip$1(int
    capacity){
        return new Integer[capacity];
    }
}
```

```
public class $$Vector$_Integer_$ extends Vector
    implements $Vector$_Integer_$
{
    public $$Vector$_Integer_$ (int capacity){
        super(capacity);
    }
    protected final Object[] $snip$1(int capacity){
        return $si.$1();
    }

    static Vector.$IsolatedSnippets $si = null;

    public static void
    $init(Vector.$IsolatedSnippets $si){
        this.$si = $si;
    }
}
```

Snippet Modellierung(4)

```
package client;
import java.util.*;
class Name {...} //kein publicTyp
class Main{
    static Vector<Name> vn = new
    Vector<Name>(100);
}
```

```
package client;
import java.util.*;
class Name {...} //kein publicTyp
class Main{
    static{
    $$Vector$_Name_$.$init(
            new Vector.$IsolatedSnippets(){
            Object[] $1(int capacity){
                    return new Name[capacity];
            }
    });
    }
    static $$Vector$_Name_$ vn = new
    $$Vector$_Name_$(100);
}
```

Snippet Modellierung(5)

- Wie werden new , new[], instanceof und cast abgebildet?
- Input: class C<T>{…} und Instance C<E>
- Output:

  C : abstrakte Basisklasse.

  $$C$_E_$: konkrete Instanzklasse.

  C_E_$: Instanzschnittstelle.

Snippet Modellierung(6)

- new C<E>()
- new C<E>[]
- (C<E>)x
- instanceof C<E>

- new $$C$_E_$()
- new C_E_$[]
- (C)((C_E_$)x)
- instanceof C_E_$

Wozu ein Class Loader

- Kann man nicht bei der Übersetzung die Menge der generischen Klassen und deren Instanziierungen feststellen und so die benötigten Klassen und Schnittstellen zur Übersetzungszeit generieren?
- Generischen Klassen – ja
- Instanziierungen dieser Klassen – nein

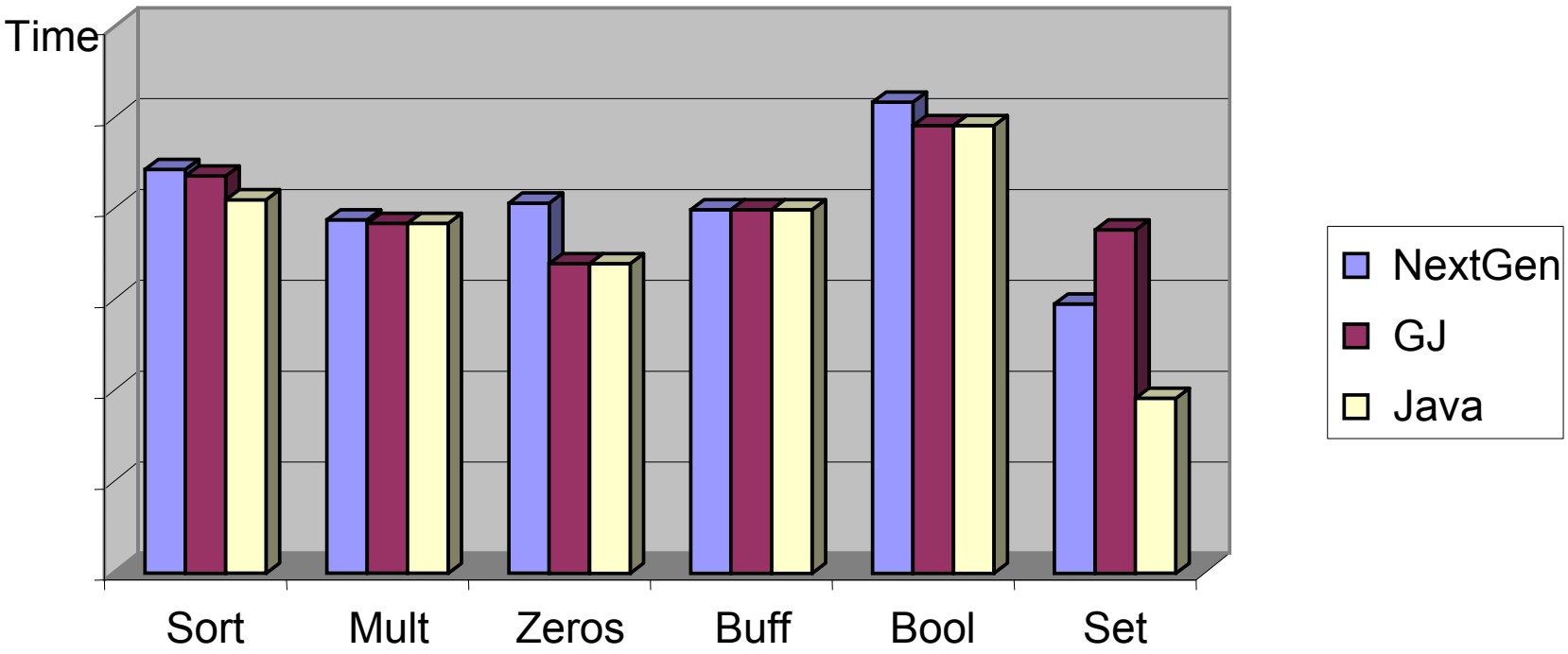
```
class C<T>{
    public Object nest(int n){
        if(n == 0) return this;
        else return new C<C<T>>().nest(n-1);
    }
}
```

Class Loader

- Generiert zur Laufzeit die benötigten Instanzklassen und Schnittstellen.

- Behält aus Performance-Gründen die generierten Klassen im Cache.

Performance



Fazit

- Generics in Java sind mittels „type erasue“ implementiert.
- Dieser Ansatz hat schlimme Folgen auf das Typesystem von Java.
- Andere Ansätze, wie NextGen bieten eine bessere und flexiblere Nutzung von generischen Typen in Java.