



Java 1.5

# QuiDome

Peter van Rijn

DGA

XP-DSDM methodiek

Java architect/trainer/coach

generatoren: xslt

# Agenda

- enumerated types
- generics
- verbeterde for-lus
- autoboxing
- static import
- varargs
- metadata

# Enumeraties oud 1/2

```
class Weekdag
{
    public static final int MAANDAG = 1;
    public static final int DINSDAG = 2;
    public static final int WOENSDAG = 3;
    public static final int DONDERDAG = 4;
    public static final int VRIJDAG = 5;
    public static final int ZATERDAG = 6;
    public static final int ZONDAG = 7;
}
```

# Enumeraties oud 2/2

```
Class EnumeratiesOud {  
    public static void main(String[] args) {  
        int dag = Weekdag.ZATERDAG;  
        switch ( dag )    {  
            case Weekdag.ZATERDAG:  
            case Weekdag.ZONDAG: System.out.println( "Weekend!" );  
        }  
    }  
}
```

# Enumeraties nieuw

```
class Enumeraties {  
    enum Weekdag {maandag, dinsdag, woensdag, donderdag, vrijdag,  
        zaterdag, zondag}  
  
    public static void main(String[] args) {  
        Weekdag dag = Weekdag.zaterdag;  
  
        switch ( dag ) {  
            case Weekdag.zaterdag:  
            case Weekdag.zondag:  
                System.out.println( "Weekend!" );  
        }  
    }  
}
```

# Enumerated types

- Eenvoudiger en helder
- Alle datatypen zijn mogelijk
- Typesafe
- De enumeratie elementen zijn klassen waaraan je methoden en variabelen kunt toevoegen.

# Zonder generics

```
class StringListTest {  
    public static void main(String[] args) {  
        ArrayList list = new ArrayList();  
        list.add("een");  
        list.add("twee");  
        for (int i=0; i < list.size(); i++) {  
            Object o = list.get(i); <--  
            if (o instanceof String) { <--  
                String str = (String) o; <--  
                System.out.println(str);  
            }  
        }  
    }  
}
```

# Generics

```
class StringListTest {  
    public static void main(String[] args) {  
        ArrayList<String> list = new ArrayList<String>();           <--  
        list.add("een");  
        list.add("twee");  
        for (int i=0; i < list.size(); i++)    {  
            String str = list.get(i);  
            <--  
            System.out.println(str);  
        }  
    }  
}
```

# Hoe werkt generics

- type erasure
  - De compiler vertaalt alle typen in de broncode naar het bounding type. Als er geen type is opgegeven wordt Object genomen.
  - `class NotBoundClass<T>`  
Object als bounding type
  - `class BoundClass<T extends javax.swing.JComponent>`  
Jcomponent als bounding type
- De compiler voegt de benodigde typecasts toe.

# Voordelen generics

.Eenvoudige en heldere code.

.Backwards compatible met bestaande programma's

.Geen wijzigen aan de JVM. De code wordt toegevoegd in compile-time.

.Veiliger code omdat de type check nu compile-time wordt uitgevoerd.

.Geen performance verlies. Er wordt min of meer dezelfde code toegevoegd als je zelf zou schrijven. Dus is er ook geen performance winst

# Zonder autoboxing

```
class IntListOud {  
    public static void main(String [] args) {  
        List intList = new ArrayList();  
        for (int i=0; i < 10; i++)  
            intList.add(new Integer(i));  
        int sum = 0;  
        for (int i=0; i < intList.size(); i++) {  
            int num = ((Integer) intList.get(i)).intValue();  
            sum += num;  
        } System.out.println(sum);  
    }  
}
```

<--

<--

# Met autoboxing

```
class IntListNieuw {  
    public static void main(String [] args) {  
        List<Integer> intList = new ArrayList<Integer>();           <--  
        for (int i=0; i < 10; i++)  
            intList.add(i);                                         <--  
  
        int sum = 0;  
        for (int i=0; i < intList.size(); i++)  
            sum+= i;                                               <--  
  
        System.out.println(sum);  
    }  
}
```

# Voordelen autoboxing

- Weg met de Wrapper classes
- Weg met de irritante typecasts.

# For iterator

```
Class StringListTest {  
    public static void main(String[] args) {  
        ArrayList<String> list = new ArrayList<String>();  
        list.add("een");  
        list.add("twee");  
        for (String str : list) {  
            System.out.println(str);  
        }  
    }  
}
```

<--

# For array

```
class ArrayFor {  
    public static void main(String[] args) {  
        int [] rij = { 1, 2, 4, 6 } ;  
  
        for (int i : rij) {  
            <--  
            System.out.println(i);  
        }  
    }  
}
```

# Voordelen for

- Korter en helder
- Het doet precies wat je verwacht
- Geen index of iterator variabele meer nodig
- Dus ook geen variabele verwarring meer.

# Static import

```
import static java.lang.Math.*;
import static java.lang.Integer.*;
class Static {
    public static void main(String[] args) {
        String getal = "3";
        int x = parseInt(getal);
        double y = sin(PI/x);
        System.out.println("getal: "+ x + " sin: " + y );
    }
}
```

# Voordelen static import

- Leesbaarheid

- Bij het gebruik moet je nu wel beter dan voorheen de naam van de methode kiezen, omdat deze nu zonder het klasse voorvoegsel gebruikt wordt.

# Varargs

Class Varargs {

```
public static void printf(String fmt, Object... args) { <--
```

```
    int i=0;
```

```
    for (char c : fmt.toCharArray()) {
```

```
        if (c == '%')                System.out.print(args[i++]);
```

```
        else if (c == '\n')          System.out.println();
```

```
        else                          System.out.print(c);
```

```
    }
```

```
}
```

```
public static void main(String[] args) {
```

```
    printf("Optelling: % plus % is gelijk aan %\n", 1, 1, 2); <--
```

```
}
```

# Voordelen varargs

- *Iets dergelijks kon voorheen met een array als argument. Bijvoorbeeld de main-methode. Dit is soepeler.*

# Zonder metadata

```
public interface CoffeeOrderIF extends java.rmi.Remote {  
    public Coffee [] getPriceList()    throws java.rmi.RemoteException;  
    public String orderCoffee(int quantity)  throws java.rmi.RemoteException;  
}
```

```
public class CoffeeOrderImpl implements CoffeeOrderIF {  
    public Coffee [] getPriceList() {  
        ...  
    }  
    public String orderCoffee(int quantity) {  
        ...  
    }  
}
```

# Met metadata

```
public class CoffeeOrder {  
    @Remote public Coffee [] getPriceList() {  
        ...  
    }  
    @Remote public String orderCoffee(String name, int quantity) {  
        ...  
    }  
}
```

# Voordelen metadata

- Ondersteuning door tools
- Je geeft aan in de code (`@Remote`) waar de tools code moet genereren.
- Lijkt veel op attributes in C#
  - Een hook in de compiler
  - Die naar een klasse springt

```
[WebMethod(CacheDuration=90)]
```

```
public DataSet FindSenatorsByState(string strState)
```

# Zelf spelen

*[http://developer.java.sun.com/developer/earlyAccess/adding\\_generics/](http://developer.java.sun.com/developer/earlyAccess/adding_generics/)*

J2SE14=c:\j2sdk1.4.0

JSR14DISTR=C:\j2sdk1.4.0\adding\_generics-2\_2-ea

C:\j2sdk1.4.0\adding\_generics-2\_2-ea\examples>..\scripts\javac  
Test.java

C:\j2sdk1.4.0\adding\_generics-2\_2-ea\examples>..\scripts\java  
Test

# Conclusie

- Java heeft adequaat gereageerd op de uitdagingen van C#.
- Java is nu weer de krachtigste programmeertaal.
- Het is wachten op de reactie van C#. Van deze competitie worden alle programmeurs alleen maar beter.