**.Net Generics**

**Folie 3:**

Problem1: ~16byte overhead für boxing am heap bei einfügen von int als object

Problem2: Einfügen von double wert und holen als string….. typprüfung erst zur laufzeit

**Generics (#4)**

Gewinnung von Typsicherheit

Type erasure, als Elementtyp können hier auch wertetypen verwendet werden ( gegensatz java)

**Derivation Constraint: (#5)**

Von K&V grundsätzlich mal nur Methoden die Object auch kann verwenden (ToString, HashCode,..)

Mit Angabe von where setzt man bestimmte Funktionalität voraus, Anwender muss diese für angegebenen typ implementieren

LinkedList<string, int> strList //richtig, string implementiert icomparable

LinkedList<object, int> objList //object implementiert icomparable nicht -> compareto(o1) funktioniert nicht

Unterschiede zu C++ und Java werden vorausgesetzt zu wissen (by Heinzi)

In c++ gibt es keine solche angabe, wird alles compilieren aber dann bei nicht implementierung nicht funktionierten, in java macht man dies angabe mit: type extends interface

**Constructor Constraing (#6)**

Man kann oben auch angeben: where V : struct dann muss typ ein wertedatentyp sein

Mit where V : new() kann man dann den typen V instanzieren

**Implementierung von Generics in .Net (#8)**

Laufzeitsystem von c# kennt generische typen, java laufzeitsystem weiß nur compiler von

Generics bescheid und generiert dann code der ohne generics auskommt. Hat auswirkung auf

Reflection. Für jeden wertedatentyp wird eine instanz erzeugt, und für alle referenzdatentypen

gemeinsam wird eine instanz erstellt, bsp: person und student laufzeitoverhead vermeiden

**Vorteile von Generics (#9)**

z.B c++ mehrere übersetzungseinheiten erzeugen alle stacks von int (code bloat)

kein boxing/unboxing notwendig da sowieso nur der typ zurückkommt

Zugriff auf generische typparameter zur laufzeit: in java könnte man die typparameter nicht

herausfinden, der schmeißt die angabe von int weg, kann man sich so vorstellen dass er daraus

object macht

**Unterschiede zu Generics in Java #10**

ArrayList<Integer> list = … //zur laufzeit ist diese information nicht mehr abgreifbar dass es typ int ist -> type erasure (gibt es in .net nicht)

**#11**

Da bei bytecode object benutzt wird braucht man mehr speicher und muss boxing/unboxing

durchführen, und type erasure da typinfos verloren gehen -> viel overhead in java

**#13**

Bei c++ werden generics (templates) vom compiler instanziert, dafür hat man wieder code bloat da

es nicht vom laufzeitsystem gehandled wird

**Objektinitialisierer (#4)**

Problem: für alle möglichen variationen konstruktoroverloads machen -> schlecht

Abhilfe: Objektinitiliasierer, angabe aller nötigen member mit zuweisung

**Behälterinitialisierer (#5)**

Selbe vorgehensweise wir bei objektinitialisierer

**Automatische Typableitung (#6)**

Bei angabe von var leitet compiler automatisch den typ her, dieser wird auch nicht geändert weil

bereit feststeht

**Anonyme Typen (#7)**

Neuer typ mit angegebener schnittstelle wird erstellt (für temporäre bzw einmalige anwendung)

**Lambda Ausdrücke (#8)**

Wenn nur 1 ausdruck kann man return weglassen und geschwungene klammern, typen werden

hergeleitet also kann man weglassen

**Erweiterungsmethoden (#9)**

Mit angabe von using xy; wird erweiterungsmethode aktiviert

**Kovarianz bei Generics (#19)**

Ist invariant, compiler lässt das nicht zu

**Kovarianz bei Generics (#20)**

Bei speziell gekennzeichneten Interfaces wo der Typparameter nur als Ausgangstyp verwendet wird, also keine add-methoden oder methoden wo der Typ verändert wird unterstützt, dann ist kovarianz gültig -> z.B enumerable

Kennzeichnung bei Generic mit out : public interface IEnumerable<out T>

**Kontravarianz bei Generics (#21)**

Anders als bei kovarianz hier mit in kennzeichnen : IEnumerable<in T> wenn Typ nur bei Eingangsseite verwendet wird und nicht als Ausgabe