Programmiertechnik II

Datentypen in Java

Primitive Typen in Java

- Wertesemantik (keine Referenzsemantik)
  - Vordefinierte Hüllklassen (etwa java.lang.Float)
- Integrale Typen: char, byte, short, int, long
- Gleitkommatypen: float, double
- bool
- Symbolische Konstanten in Hüllklassen
  - java.lang.{Byte,Short,Integer,Long,Float,Double}.{MIN_VALUE,MAX_VALUE}
  - java.lang.Boolean.{FALSE,TRUE}
  - java.lang.{Float,Double}.{NaN,NEGATIVE_INFINITY,POSITIVE_INFINITY}

Strings

- Strings: java.lang.String
  - Unterstützt durch Zeichenkettenliterale
  - Repräsentiert durch char[]
    - aber: unveränderbar (immutable)
    - java.lang.StringBuffer für änderbare Zeichenketten
- String-Addition: s1 + s2
  - String + String -> String
  - String s + Object o => s + o.toString()
  - Object o + String s => o.toString + s
  - String s + <primitiver Typ T> v => s + new <Hülltyp für T>(v).toString()
  - <primitiver Typ T> v + String s => new <Hülltyp für T>(v).toString() + s
- Strings sind Objekte, == testet auf Identität
  - .equals zur Ermittlung der Wertegleichheit
  - aber: String-Interning
- Viele Standardoperationen (startsWith, split, substring, charAt, …)

Objekttypen

- java.math.BigInteger: beliebig große ganze Zahlen
  - new BigInteger(“123456789”).pow(20)
- java.math.BigDecimal: beliebig große Gleitkommazahlen
  - Dezimal, nicht binär
  - variable Zahl von Nachkommatstellen

Arraytypen

- Jeder Datentyp impliziert einen Arraytyp
  - int[]
  - String[]
  - …
  - Arraytypen haben als Basistyp java.lang.Object (implements Cloneable, java.io.Serializable)
- Typen sind zuweisungskompatibel falls Elementtypen zuweisungskompatibel sind
- Sprachkonstrukte:
  - Deklaration von Variablen, Klassenattributen, Parametern
    - int[] a; int b[];
  - Erzeugung von null-initialisierten Feldern: new String[100];
  - Erzeugung von initialisierten Feldern: int x[] = {1, 2, 3};
- Mehrdimensionale Felder

Datentypen

- Ein *Datentyp* ist eine Mengen von Daten zusammen mit einer Familie von Operationen
  - abstrakter Datentyp: beschrieben wird lediglich die Menge und die Semantik der Operationen, nicht aber die interne Repräsentation der Daten oder die Implementierung der Operationen

Mengenkonstruktionen

- Konstruktion neuer Mengen (evtl. auf Basis existierender Mengen M, M_i)
- Aufzählung: Menge $\{e_1, e_2, e_3, ..., e_n\}$
  - C, C++, Java 5: enum (*enumerations*)
- Teilmengen: Gegeben Prädikat P(x), bilde $\{x \in M \mid P(x)\}$
- k-te Potenz: Gegeben $k \in \mathbb{N}$: Menge M^k aller k-Tupel (a_i, a_i, ..., a_i) mit $a_i \in M$
  - C, C++: Array (Felder) “von M” der Länge k
  - Python: Liste (Beschränkung auf Länge per Konvention)
- Kartesisches Produkt: $M_1 \times M_2 \times ... \times M_k$: Menge der k-Tupel ($a_i$, a_i, ..., a_i) mit $a_i \in M_i$
  - C: struct

Mengenkonstruktionen

- Konstruktion neuer Mengen (evtl. auf Basis existierender Mengen M, M_i)
- Aufzählung: Menge $\{e_1, e_2, e_3, ..., e_n\}$
  - C, C++, Java 5: enum (*enumerations*)
- Teilmengen: Gegeben Prädikat P(x), bilde $\{x \in M \mid P(x)\}$
- k-te Potenz: Gegeben $k \in \mathbb{N}$: Menge M^k aller k-Tupel (a_i, a_i, ..., a_i) mit $a_i \in M$
  - C, C++: Array (Felder) “von M” der Länge k
  - Python: Liste (Beschränkung auf Länge per Konvention)
- Kartesisches Produkt: $M_1 \times M_2 \times ... \times M_k$: Menge der k-Tupel ($a_i$, a_i, ..., a_i) mit $a_i \in M_i$
  - C: struct

Abstrakte Datentypen

- Collections: Containertypen für andere Objekte
  - Java: enthaltene Objekte sind vom Typ java.lang.Object
    - GNU Trove: typ-spezifische Container
    - JDK 5: Generic Types
- java.util.Collection: Sammlungen von Objekten
  - Set, List, SortedSet
  - unmodifiable/modifiable
  - immutable/mutable
  - fixed-size/variable-size
  - random access/sequential access
- java.util.Map: Abbildung von Schlüssel auf Wert
  - SortedMap

java.util.Collection

```
interface Collection{
    boolean add(Object o);
    boolean addAll(Collection c);
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection c);
    boolean equals(Object o);
    int hashCode();
    boolean isEmpty();
    Iterator iterator();
    boolean remove(Object o);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    int size();
    Object[] toArray();
    Object[] toArray(Object[] a);
}
```

java.util.Set

- Collection ohne doppelte Elemente
  - Wann sind zwei Elemente e1 und e2 gleich? e1.equals(e2), oder e1 == null und e2 == null
- Implementiert durch java.util.AbstractSet
  - abstrakt: konkrete Spezialisierungen sind HashSet und TreeSet
  - LinkedHashSet extends HashSet (bewahrt “insertion order”)
- java.util.SortedSet extends Set
  - Elemente der Menge in Bezug auf Ordnungsrelation sortiert
    - java.util.Comparator definiert Ordnungsrelation
    - alternativ: Elemente implementieren java.lang.Comparable

Ordnungsrelationen

```
interface Comparable{
    int compareTo(Object o);
}
```

- “natural comparison method”
- ergibt Zahl <0, =0, >0
  - muss Ordnungsrelation definieren (reflexiv, antisymmetrisch, transitiv)
- sollte konsistent mit .equals() sein
  - (x.compareTo(y) == 0) == x.equals(y)
  - nicht zwingend gefordert
    - “Note: this class has a natural ordering that is inconsistent with equals”

Ordnungsrelationen (2)

```
interface Comparator{
   int compare(Object o1, Object o2);
   boolean equals(Object o1, Object o2);
}
```

- erlaubt Definition von “besonderen” Ordnungsrelationen
  - Sortieren entsprechend den Regeln des Telefonbuchs
  - Ignorieren von Groß- und Kleinschreibung
  - …
- Comparator-Implementierung sollten Serializable implementieren

java.util.List

- geordnete Sammlung von Elementen
  - doppelte Elemente i.d.R. erlaubt
- Indizierter Zugriff möglich, Index startet bei 0
  - Komplexität u.U. linear mit Index
    - I.d.R. ist es besser, über die Liste zu iterieren, als der Reihe nach mit aufsteigenden Indizes zuzugreifen
- Implementiert durch AbstractList
  - LinkedList extends AbstractSequentialList extends AbstractList
  - ArrayList: Speicherung der Objekte in Array
    - capacity: aktuelle Größe des Arrays
    - size, isEmpty, get, set, iterator, listIterator: konstante Zeit
    - add: amortisiert-konstante Zeit
    - alle anderen Operationen: lineare Komplexität
      - insbesondere Löschen von Elementen
  - Vector: wie ArrayList, aber zusätzlich synchronisiert
    - ältere Implementierung, zusätzliche Operationen
    - anderer Algorithmus zur Vergrößerung (Vector: 100%, ArrayList: 50%)

java.util.iterator

```
interface Iterator{
    boolean hasNext();
    Object next(); // throws NoSuchElementException
    void remove(); // throws UnsupportOperationException, IllegalStateException
}
```

- remove entfernt letztes Element, das von .next zurückgegeben wurde
  - darf nur einmal pro next-Aufruf gerufen werden

java.util.Map

- Assoziatives Feld: Abbildung von Schlüsseln auf Werte
  - Funktion, Relation, “Dictionary”, (“Hash”, “Hashmap”, “Hashtable”)
- 3 “collection views”:
  - Menge von Schlüsseln
  - Sammlung von Werten
  - Menge von Schlüssel-Wert-Paaren
- Ordnung der Map: Reihenfolge, in der Iterator die “collection views” bereitstellt
  - Garantierte Reihenfolge z.B. in TreeMap
  - SortedMap extends Map: Schlüssel sind nach Kriterium sortiert
- Basisklasse der Implementierungen ist AbstractMap
  - Ausnahme: Hashtable extends Dictionary

java.util.Map

```
interface Map{
    void clear();
    boolean isEmpty();
    int size();
    Object get(Object key);
    Object put(Object key, Object value);
    void putAll(Map t);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Set entrySet();
    Set keySet();
    Collection values();
    int hashCode();
}
```

Map-Implementierung: HashMap

- Implementiert durch “hash table”
  - basiert auf java.lang.Object.hashCode
  - java.lang.Object.equals zur Unterscheidung unterschiedlicher Objekte mit gleichem Hashwert
  - ein Schlüssel darf null sein
- Operationen .get und .put im Mittel in konstanter Zeit
  - vorausgesetzt, die Hashwerte sind gleichverteilt
- LinkedHashMap extends HashMap
  - Schlüssel werden in “insertion order” gespeichert
- java.util.Hashtable extends Dictionary
  - “historische” Implementierung
  - im Wesentlich gleich der HashMap
    - aber: synchronisiert
    - aber: null ist als Schlüssel nicht erlaubt

Map-Implementierung: IdentityHashMap

- Implementiert durch “hash table”
  - ebenfalls auf Basis von java.lang.Object.hashCode
  - aber: zum Vergleich wird == verwendet, nicht .equals
    - Streng genommen Verletzung der Schnittstelle java.util.Map
- Verwendet zur Erkennung “doppelter” Objekte etwa in Graph-Algorithmen
  - Serialisierung: jedes gefundene Objekt wird in IdentityHashMap eingetragen, mit “Objekt-Nummer” als Wert

Map-Implementierung: TreeMap

- Implementiert durch Rot-Schwarz-Bäume
  - Schlüssel müssen einer Ordnungsrelation unterliegen

Collections im Überblick

		Implementierungen				
		Hash-tabelle	Vergößer-bares Feld	Baum	Verkettete Liste	Hashtabelle + verkettete Liste
Schnitt-stellen	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList, Vector		LinkedList	
	Map	HashMap, Hashtable		TreeMap		LinkedHashMap

Java 5: Generic Types

- bisher: Collections enthalten java.lang.Object

```
List zahlen = new LinkedList();
zahlen.add(new Integer(57));
Integer x = (Integer)zahlen.iterator().next();
```

- Java 5: Klassen und Schnittstellen können *Typparameter* haben

```
List<Integer> zahlen = new LinkedList<Integer>();
zahlen.add(new Integer(57)); // oder: zahlen.add(57);
Integer x = zahlen.iterator().next();
```

Generic Types (2)

- Definition der Klassen: formale Typparameter

```
public interface List<E>{
    void add(E x);
    Iterator<E> iterator();
    // ...
}
public interface Iterator<E>{
    E next();
    boolean hasNext();
}
```

-