

IKI10400 • Struktur Data & Algoritma:

Abstract Data Types dan Java Collections API

Fakultas Ilmu Komputer • Universitas Indonesia

Slide acknowledgments:

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung



Abstract Data Type (ADT) adalah ...

- **Spesifikasi dari sekumpulan data termasuk operasi** yang dapat dilakukan pada data tersebut. (Wikipedia)
- **Sekumpulan data dan operasi** terhadap data tersebut yang **definisi-nya tidak bergantung pada implementasi tertentu.** (www.nist.gov/dads/)



Interface

- Spesifikasi *Abstract Data Type* biasa disebut sebagai *interface*.
- *Interface* menyatakan apa yang dapat dilihat dan digunakan oleh *programmer*.
- Dalam Java, hal tersebut dinyatakan sebagai *public method*.
- Operasi-operasi yang dapat dilakukan pada *abstract data type* dituliskan dalam *interface* dan dinyatakan *public*.



Pemisahan *interface* dengan implementasi

- Pengguna dari sebuah *abstract data type* hanya perlu memikirkan dan mempelajari *interface* yang diberikan tanpa perlu mengetahui banyak bagaimana implementasi dilakukan. (prinsip: **enkapsulasi**)
- Implementasi dapat saja berubah namun *interface* tetap.
- Dengan kata lain, implementasi dari sebuah *abstract data type* dapat saja berbeda-beda namun selama masih mengikuti *interface* yang diberikan maka program yang menggunakan *abstract data type* tersebut tidak akan terpengaruh.



Struktur data = *container*



Container



Data

- Sebuah struktur data dapat dipandang sebagai tempat penyimpanan benda (**container**).
- Beberapa hal yang dapat dilakukan:
 - **Menaruh** benda
 - **Mengambil** benda
 - **Mencari** benda tertentu
 - **Mengosongkannya** (atau **periksa** apakah kosong)

Contoh *Interface* struktur data :

```
void add(Benda x) ;  
void remove(Benda x) ;  
Benda access(Benda x) ;  
void makeEmpty() ;  
boolean isEmpty() ;
```



ADT: List



Indeks → 1

2

3

4

- Sebuah **List** adalah kumpulan benda di mana setiap benda memiliki **posisi**.
- Setiap benda dalam List dapat diakses melalui **indeks**-nya.
- Contoh paling gampang: array!

Contoh Interface **list** :

```
void add(int indeks, Benda x); //insert
void add(Benda x); //append
Benda remove(int indeks);
boolean remove(Benda x);
Benda get(int indeks);
```



ADT: List

- *Concrete implementation* List di Java:
 - `java.util.ArrayList`
 - menggunakan array
 - `java.util.Vector`
 - Jangan gunakan `java.util.Vector`. Higher overhead because it is synchronised for concurrent modification.
 - Gunakan
 - `List list = Collections.synchronizedList(new ArrayList(...));`
 - `java.util.LinkedList`
 - implementasi menggunakan Doubly-linked list



ADT: Stack



- Sebuah **Stack** adalah kumpulan benda di mana hanya benda yang **most recently inserted** dapat diakses.
- Bayangkan setumpuk koran.
- Benda yang paling terakhir ditambahkan ditaruh di atas tumpukan (**top**).
- Operasi pada Stack membutuhkan waktu konstan (**$O(1)$**).

Contoh Interface **stack** :

```
void push(Benda x) ;  
Benda pop() ;  
Benda top() ;
```



ADT: Queue



- Sebuah **Queue** adalah kumpulan benda di mana hanya benda yang ***least recently inserted*** dapat diakses.
- Bayangkan antrian *printer job* pada jaringan.
- Benda yang paling awal ditambahkan berada di depan antrian (**front**).
- Operasi pada Queue membutuhkan waktu konstan (**$O(1)$**).

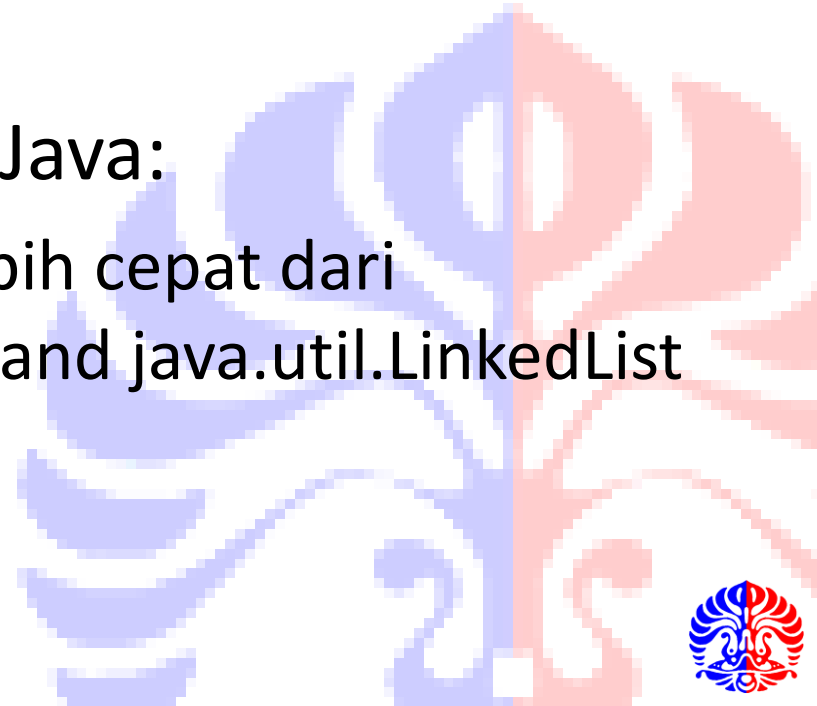
Contoh Interface **queue** :

```
void add(Benda x); //enqueue x  
Benda poll(); //dequeue or getFront
```



ADT: Deque

- Double-ended queue (dibaca 'deck')
 - Kumpulan yang bisa diakses dari depan maupun belakang dengan efisien
 - bisa dipakai sebagai Stack dan Queue
- *Concrete implementation* di Java:
 - Gunakan ArrayDeque yang lebih cepat dari `java.util.Stack` (sebagai stack) and `java.util.LinkedList` (sebagai queue)



ADT: Priority Queue



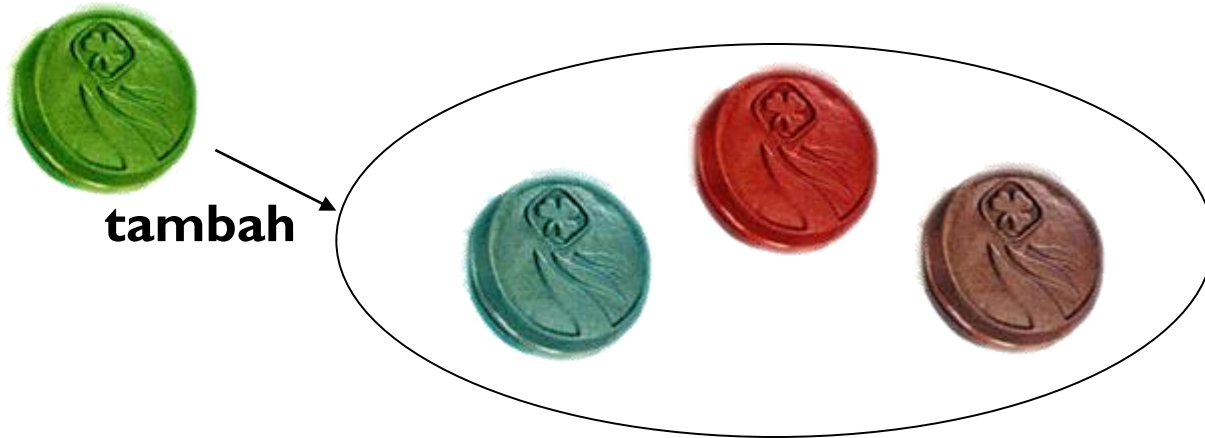
- **Priority Queue** adalah struktur data *queue* yang tiap elemen data dapat **miliki nilai prioritas**. Data dengan **nilai prioritas tertinggi**lah yang **dapat diakses** terlebih dulu.
- Bayangkan sebuah antrian pada printer jaringan. Misalkan ada sebuah permintaan cetak untuk **100 halaman** hanya beberapa detik lebih awal dari permintaan cetak **selembar halaman**.

Contoh Interface sebuah *Priority Queue* :

```
void add(Benda x); (Menambahkan)
Benda poll(); (menghapus dan mengembalikan Min)
Benda peek(); (mengembalikan Min)
```



ADT: Set



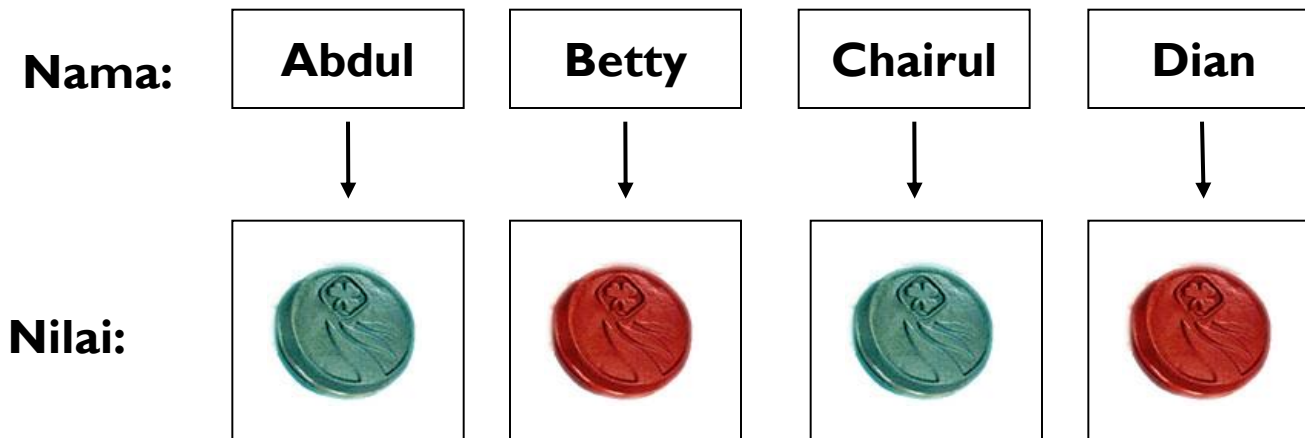
- **Set** adalah struktur data yang **tidak mengizinkan duplikasi data**.
- Bandingkan dengan struktur data lain yang mengizinkan kita menyimpan dua data yang sama.
- Bayangkan peserta kuliah ini: Setiap peserta unik, **tidak ada yang terdaftar dua kali!**

Contoh Interface **set**:

```
void add(Benda x);  
void remove(Benda x);  
boolean contains(Benda x); //isMember
```



ADT: Map



- **Map** adalah struktur data yang berisi sekumpulan pasangan **nama (keys)** dan **nilai (values)** dari nama tersebut.
- **Nama (Keys)** harus unik, tapi **nilai (values)** tidak.
- Bayangkan basis-data yang berisi informasi peserta kuliah. Apa yang menjadi “**nama**” (**keys**)?

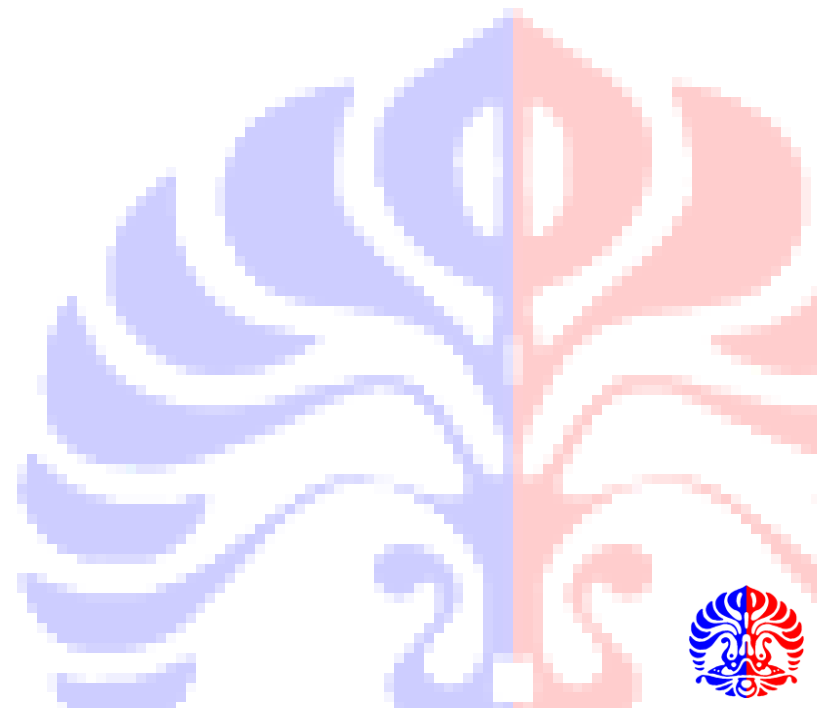
Contoh Interface sebuah **Map** :

```
Nilai put(Kunci id, Nilai x);  
Nilai remove(Kunci id);  
Nilai get(Kunci id);
```



ADT: Set dan Map

- Concrete implementation di Java:
 - Set:
 - `java.util.TreeSet`
 - sorted
 - `java.util.HashSet`
 - Map:
 - `java.util.TreeMap`
 - sorted by keys
 - `java.util.HashSet`



JAVA COLLECTIONS API



Apa itu *Collections*?

- *Collection* — di sebagian literatur disebut sebagai: *container*
- Dalam bahasa pemrograman Java, *collection* adalah sebuah *object* yang mengelompokkan beberapa element dalam satu unit.
- *Collections* digunakan untuk menyimpan, mengambil, memanipulasi dan untuk menghubungkan/menggabungkan data.



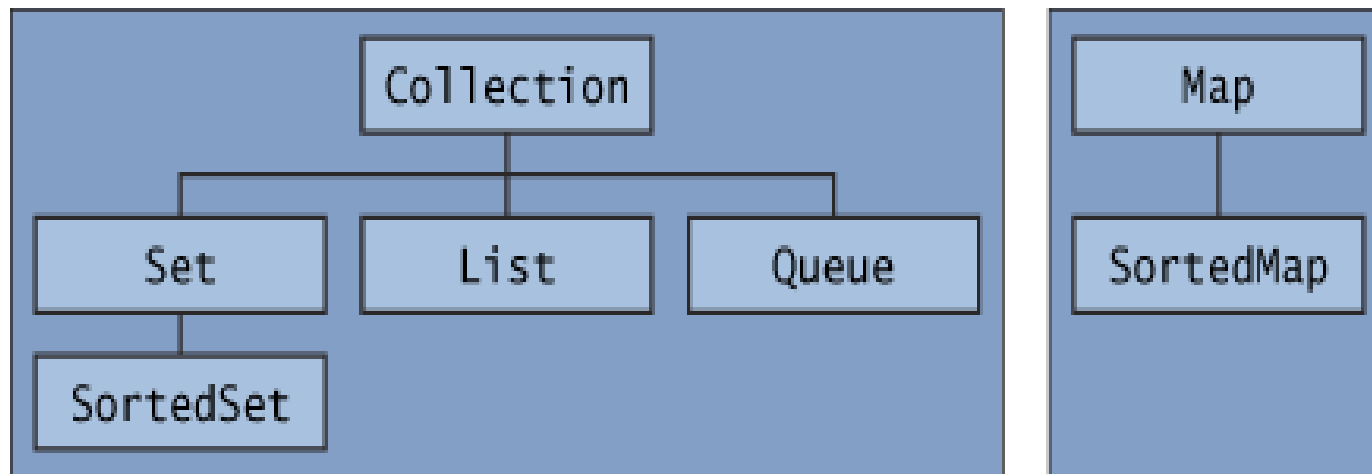
API vs doing it yourself

- Jika anda menggunakan *Java collections framework*, *programmer* lain dapat lebih mudah mengadaptasi program anda.
- Jika anda membuat implementasi sendiri, maka *programmer* lain belum tentu dapat dengan mudah mempelajari program anda.
- Namun demikian, sebagai mahasiswa/i ilmu komputer, perlu memahami bagaimana data tersusun dalam *memory* dan perlu memahami konsep-konsep apa yang mendasarinya.
- Tidak menutup kemungkinan untuk *meng-extend/meng-implement Collections*



Interfaces

- Java *collections framework* didasari pada sekumpulan *interface* yang mendikte metode-metode apa saja yang harus diimplementasikan dan membantu standarisasi penggunaan.
- Hubungan antara beberapa *interface*:



The Collection interface

- Interface Collection adalah interface utama yang menetapkan operasi-operasi dasar, antara lain:

```
int size();
```

```
boolean isEmpty();
```

```
boolean contains(Object element);
```

```
boolean add(E element);
```

```
boolean remove(Object element);
```

```
Iterator<E> iterator();
```

```
.....
```

- E** menyatakan tipe parameter.



Primitive types

- *Collections* adalah kumpulan dari referensi terhadap *object* dan tidak bisa berisi tipe primitif.
- Jika membutuhkan kumpulan data bertipe primitif misalnya *characters*, kita tidak bisa memparameterisasi *collection* dengan tipe **char**. kita harus menggunakan kelas: **Character**
- Namun Java menyediakan fasilitas: **boxing** and **unboxing**
- Jika hendak meletakkan data bertipe **char** dalam sebuah *collection* bertipe **Character**, maka Java akan secara otomatis melakukan “**boxing**” (membungkus) **char** dalam kelas **Character**
- Jika hendak mendapatkan data **char** tersebut, Java akan secara otomatis melakukan “**unbox**” dan memberikan data dalam tipe **char**.



Collections in use

```
import java.util.*;

public class Example{
    private List<String> list;
    public List<String> getList(){
        return list;
    }

    Example() {
        list = new ArrayList<String>();
        list.add(new String("Hello world!"));
        list.add(new String("Good bye!"));
    }
}
```



```
public void printList() {  
    for (Object s:list) {  
        System.out.println(s);  
    }  
}
```

```
public static void main(String argv[]) {  
    Example e = new Example();  
    e.printList();  
    Collections.sort(e.getList());  
    e.printList();  
}
```



GENERIC COLLECTIONS



Kebutuhan akan generic programming

Apa yang salah dengan program berikut:

```
void addStuffToCollection(Collection c) {  
    c.add(new String("Hello world!"));  
    c.add(new String("Good bye!"));  
    c.add(new Integer(95));  
    printCollection(c);  
}  
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    while(i.hasNext()) {  
        String item = (String) i.next();  
        System.out.println("Item: "+item);  
    }  
}
```

- Terjadi **run-time** error!



Pembatasan tipe *collection*

- Sebelum Java 5.0, tidak ada cara untuk menetapkan tipe elemen dari sebuah *collection*.
- Sehingga, kita harus melakukan variabel *casting* di berbagai tempat.
 - *Cumbersome*: melakukan *casting* saat tidak diperlukan.
 - *Error-prone*: dapat mengakibatkan *run-time exception*.
- *Generics* adalah sebuah konsep yang dapat digunakan untuk membatasi tipe elemen sebuah *collection*.
- Pada Java versi > 5.0, *Collections* adalah *generic classes* yang dapat menerima parameter tipe yang menentukan tipe dari elemennya.



Instantiasi kelas generic

```
List<MyType> myList = new ArrayList<MyType>();
```

- Sebuah kelas *collection generic* dapat diinstantiasi melalui parameter tipe.
- Kita dapat saja membuat kelas dan method *generic* sendiri dan menginstantiasi dengan berbagai tipe.



Kebutuhan akan generic programming

Apa yang salah dengan program berikut:

```
void addStuffToCollection(Collection<String> c) {  
    c.add(new String("Hello world!"));  
    c.add(new String("Good bye!"));  
    c.add(new Integer(95));  
    printCollection(c);  
}  
void printCollection(Collection<String> c) {  
    Iterator<String> i = c.iterator();  
    while(i.hasNext()) {  
        String item = i.next();  
        System.out.println("Item: "+item);  
    }  
}
```

- Dapat terdeteksi ketika **compile-time** !



Iterator

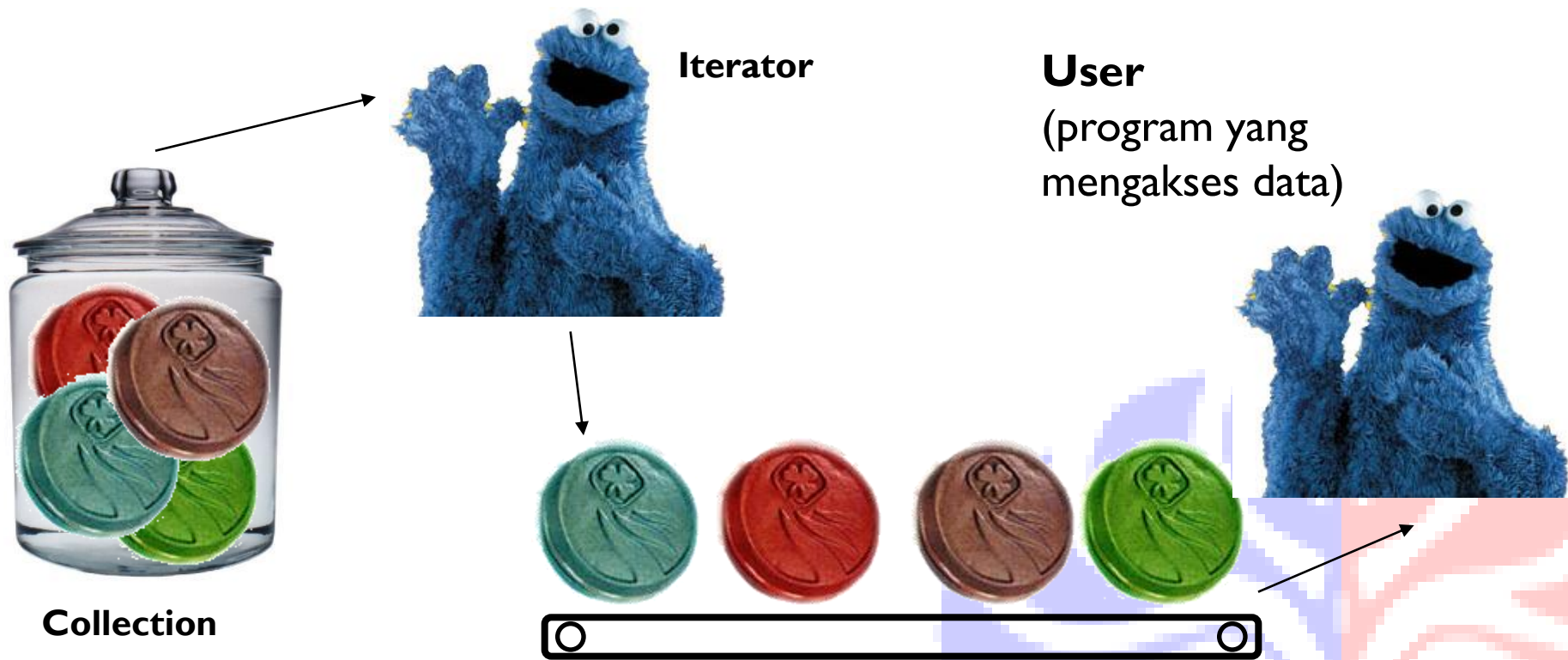
- Perhatikan kedua contoh sebelumnya menggunakan kelas *Iterator*:

```
Iterator i = c.iterator();
```

- Hal yang umum dilakukan pada *collection* adalah membaca seluruh elemen.
- *i* adalah objek *iterator* yang mengendalikan iterasi pembacaan data pada *collection c*.
- Secara umum *Iterator* bekerja sebagai berikut:
 - Mulai dengan mengatur *iterator* pada elemen pertama pada *collection*.
 - Satu-persatu berlanjut pada elemen selanjutnya
 - Berakhir ketika tidak ada lagi elemen pada *collection* yang belum dibaca.



Ilustrasi: Iterator



Contoh lain penggunaan iterator:

```
void printCollection(Collection<String> c) {  
    Iterator itr = c.iterator();  
    for(itr = v.first(); itr.isValid(); itr.advance())  
        System.out.println(itr.getData());  
}
```

