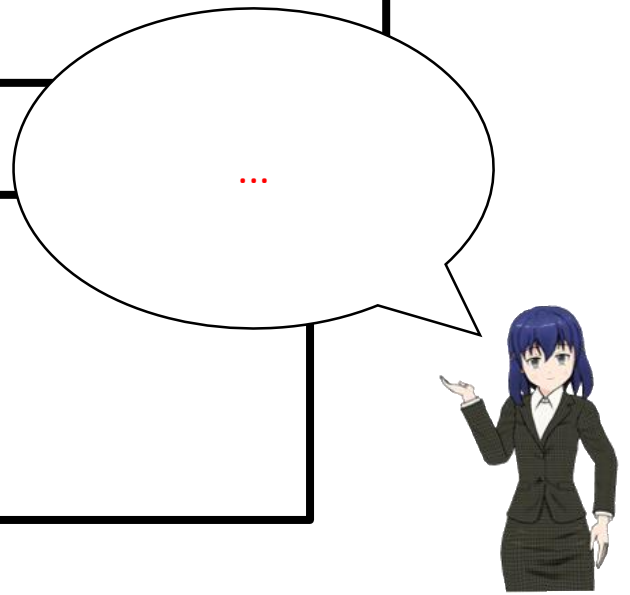
# ソフトウェア工学実習
# Software Engineering Practice
# (第07回)

## SEP07-001 コレクションフレームワーク

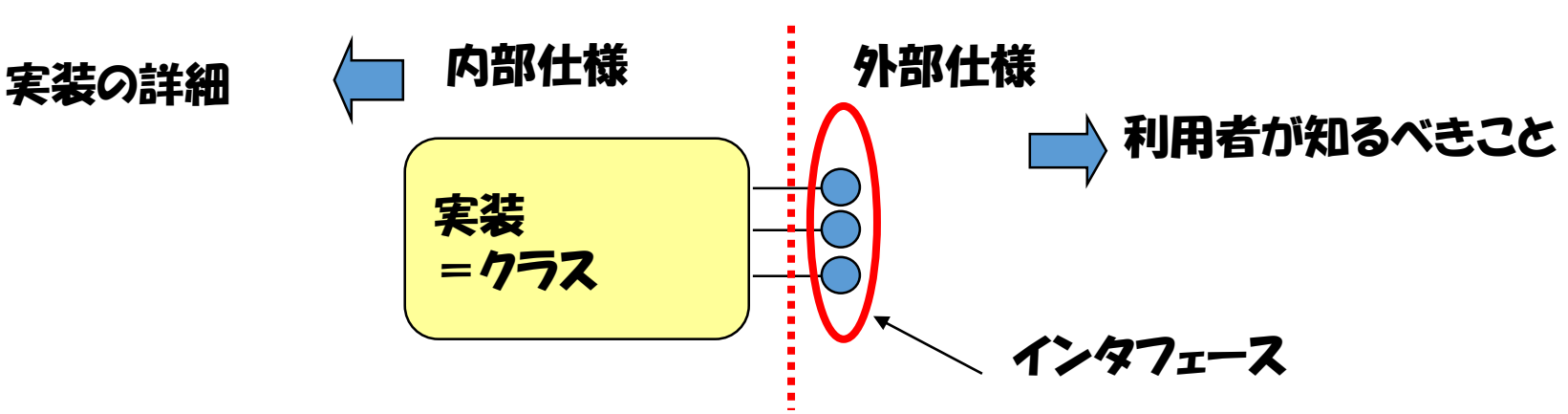
慶應義塾大学・理工学部・管理工学科
飯島 正
iijima@ae.keio.ac.jp

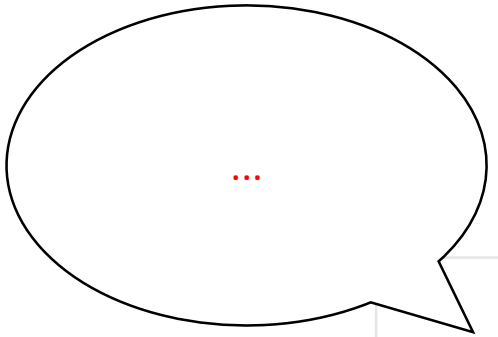
# コレクションフレームワーク



# インタフェースの概念

- インタフェースと実装の分離
  - インタフェースとクラス
    - 外部仕様と内部仕様
  - 多重インタフェース
- 分散オブジェクトの影響
  - IDL (Interface Definition Language)



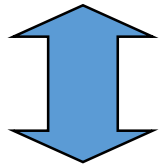

# 【参考】Javaでのインタフェースとクラス



```
interface Stack-IF {
        Object pop();
        void push( Object x );
}
```

インタフェース

外部から呼び出せる
メソッド情報だけ



```
class Stack implements Stack-IF{
        Vector  v;
        Object pop() {...};
        void push( Object x ) {...};
}
```

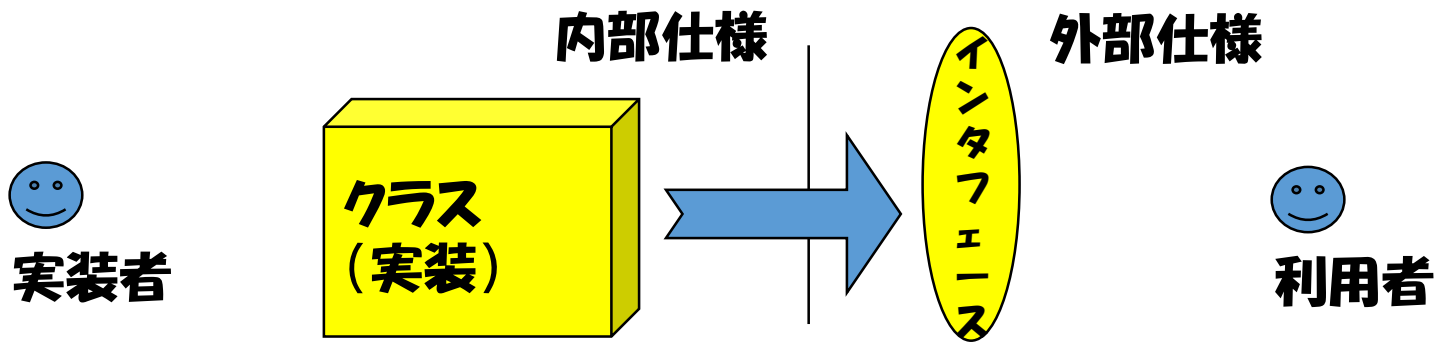
クラス

属性情報，
メソッドの定義，
内部的にしか使わない
メソッド情報を含む

# 【参考】Javaでのインタフェース指定

- インタフェースYYYは，クラスXXXの外部仕様
- クラス（コード）XXXは，インタフェースYYYの実装

```
class XXX implements YYY {

                    …

}
```




# 【参考】Javaでのインタフェースの定義



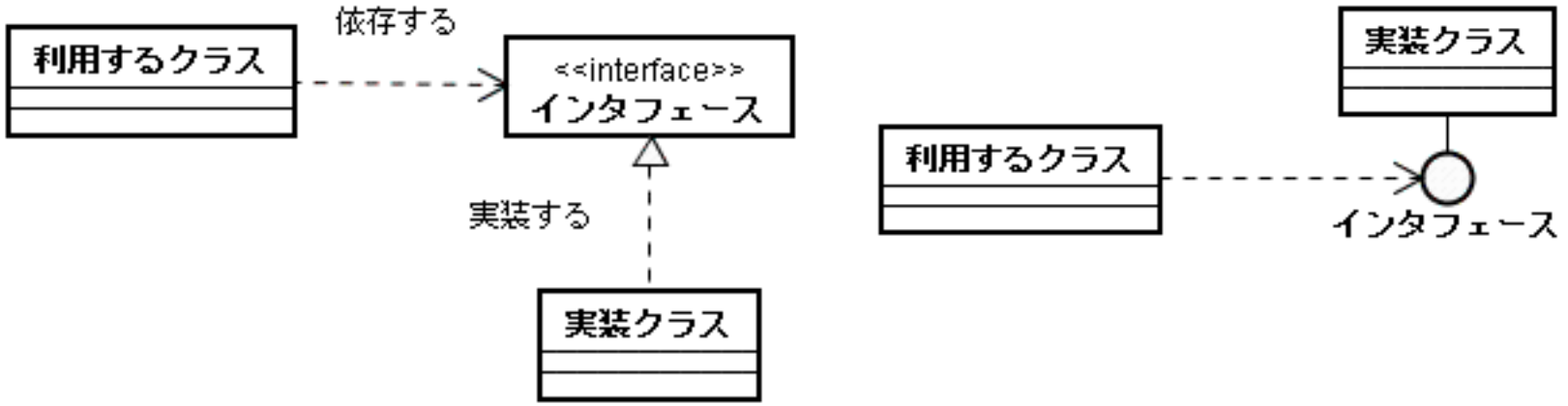
```
interface YYY extends superYYY,superYYY2 {
        メソッドシグネチャ
                ...
}
```

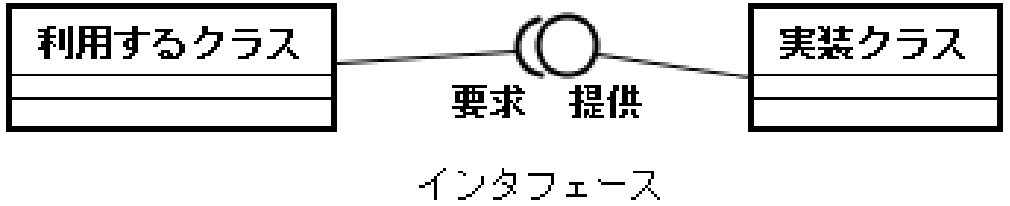
◆ インタフェースは多重継承できる

なぜ，インタフェースの多重継承は，
クラスの多重継承と違って問題がないのか？
⇒（実装が競合しないから）

# インタフェースと実装のUML表記






# 【参考】Javaでのインタフェースの利用



- クラスと同じように参照型として利用できる.
  - 受け付けられるメソッドが何なのかを規定している

# 【参考】Javaでの抽象クラスとインタフェースの使い分け



- クラスは単一継承，インタフェースは多重継承
  - インタフェースを複数指定しても，
    実装を継承しないので問題ない
- インタフェースは，複数指定できる
  （多重インタフェース）
  - 一つのオブジェクトへの複数の観点を与える
- 抽象クラスには，抽象メソッドだけでなく，
  具象メソッド(?!)も書ける
  - インタフェースには実装を書くことはできない

# 可変長配列Vectorの操作

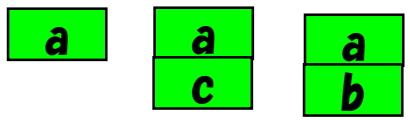


| | |
|---|---|
| 大きさ | size() |
| オブジェクトを最後に追加 | addElement(オブジェクト) |
| 指定位置(0から数える)に<br>オブジェクトを挿入 | insertElementAt(オブジェクト, 位置) |
| 指定位置の要素を置換 | setElementAt(オブジェクト,位置) |
| 指定位置の要素を除去 | removeElementAt(位置) |
| オブジェクトと等しい<br>最初の要素を除去 | removeElement(オブジェクト) |

# 例15:Vectorの使い方



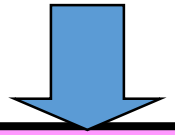
```
import java.util.*;

class VectorTest {
  public static void main( String args[] ) {
    Vector<String>  v
              = new Vector<String>();
    v.addElement("a");
    v.addElement("c");
    v.insertElementAt("b", 1);
    System.out.println(v);
    v.setElementAt("B",1);
    System.out.println(v);
    v.addElement("d");
    v.addElement("d");

    System.out.println(v);
    v.removeElement("d");
    System.out.println(v);
    System.out.println(v.size());

    for ( String e : v ) {
      System.out.println( e);
    }
  }
}
```




# × Enumeration: 列挙（とても古い）

| | |
|---|---|
| Enumerationを返す | elements() |
| Enumeration eに要素はあるか？ | e.hasMoreElements() |
| Enumeration eの次の要素を返す | e.nextElement() |

```
for ( int i = 0; i <10; i++) {
        .... x[i] ...
}
```

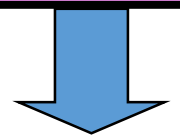


```
for ( Enumeration e = v.elements();
                    e.hasMoreElements(); ) {
    ... e.nextElement() ...
}
```

# × Iteration: 繰り返し（少し古い）

| | |
|---|---|
| Iteratorを返す | entrySet().iterator() |
| Iterator iに要素はあるか？ | i.hasNext() |
| Iterator iの次の要素を返す | i.next() |

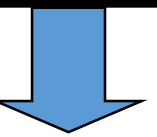
```
for ( int i = 0; i <10; i++) {
        .... x[i] ...
}
```



```
for (Iterator i = v.iterator();  i.hasNext(); ) {
        ...        i.next();      ...
}
```

# ◎　今は，foreach文（拡張for文）

```
for ( int i = 0; i <10; i++) {
        .... x[i] ...
}
```



```
for ( Object element :  v) {
        ...        element   ...
}
```


iijima@ae.keio.ac.jp

# クラス(class)と型(type)



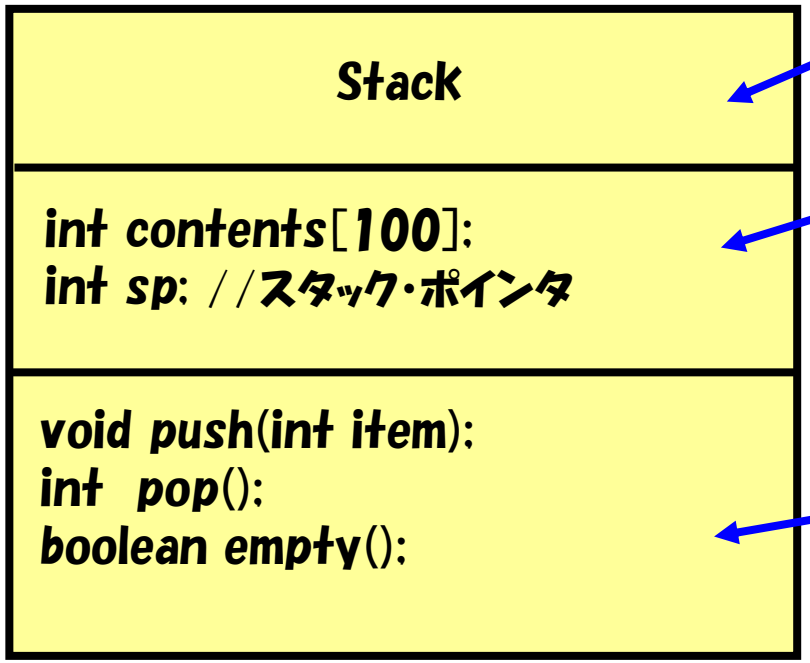
- クラスと（変数の）データ型は本来，別の概念である
- 但し，クラス参照(Class Reference)型は，
  クラス(名)を型(名)として代用しているように見える
- 型の無いオブジェクト指向言語もある

```
Stack s1=new Stack();
```

Stackクラスのインスタンスへの参照を
格納する変数の型(Stack参照型)

# 属性とメソッド

## クラス図のクラス箱



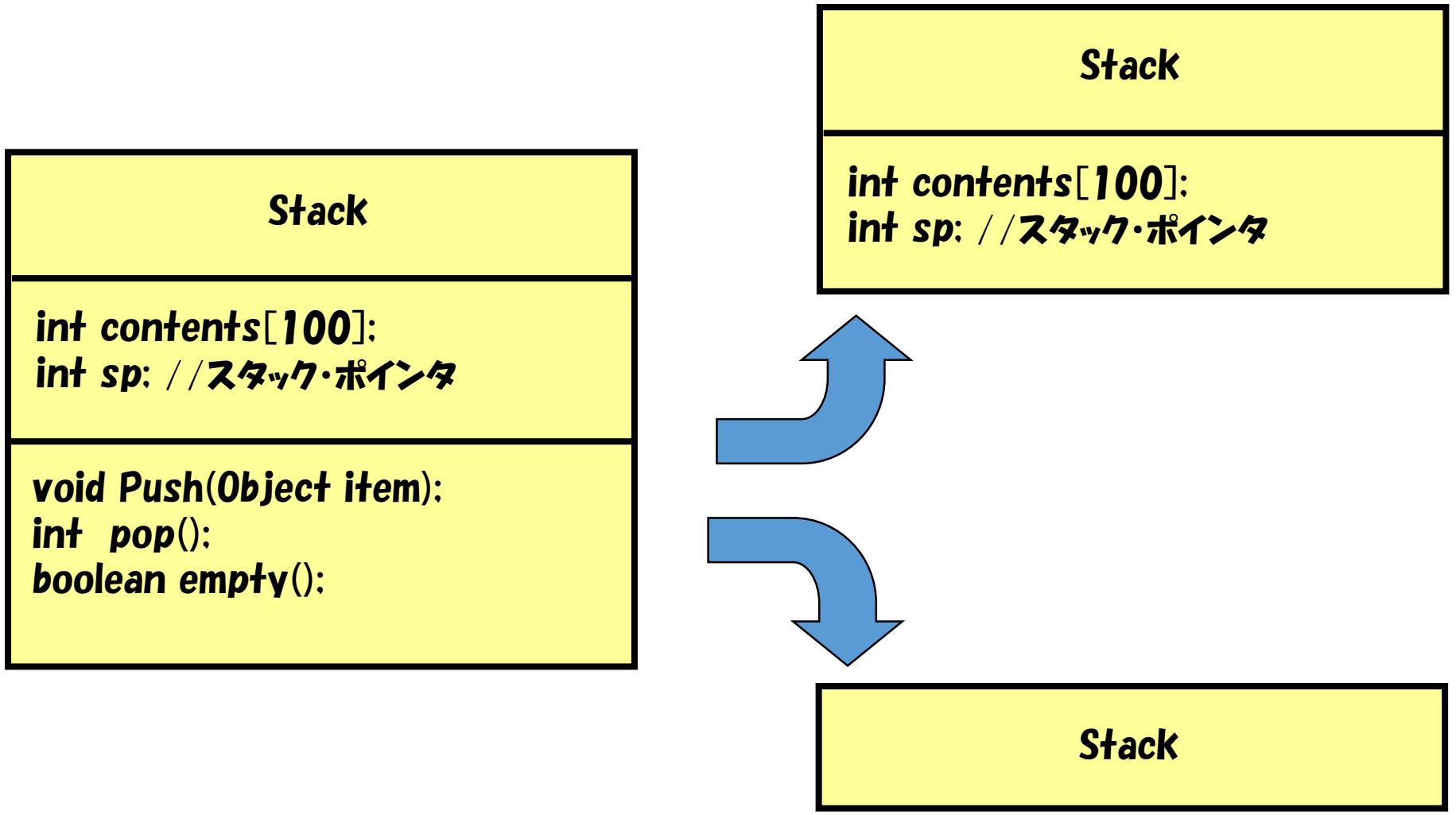

**クラス名**

**属性**
（状態を意味する
内部データの名前と型）

**メソッド**
（属性を変化させたり，
値を問い合わせる
操作の名前と型）

# 属性やメソッドに関心のないときは省略可

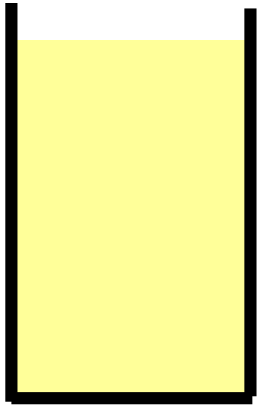





# 例：スタック (Stack) の概念 (“what”)



- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器

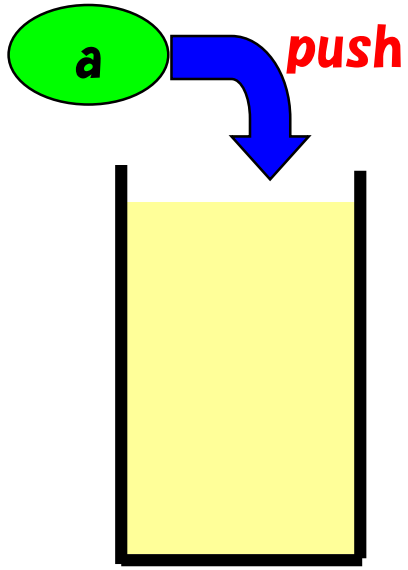


- 設計例-1：配列と整数型変数（スタックポインタ）
- 設計例-2：連結リスト

# 例：スタック (Stack) の概念 (“what”)



- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器

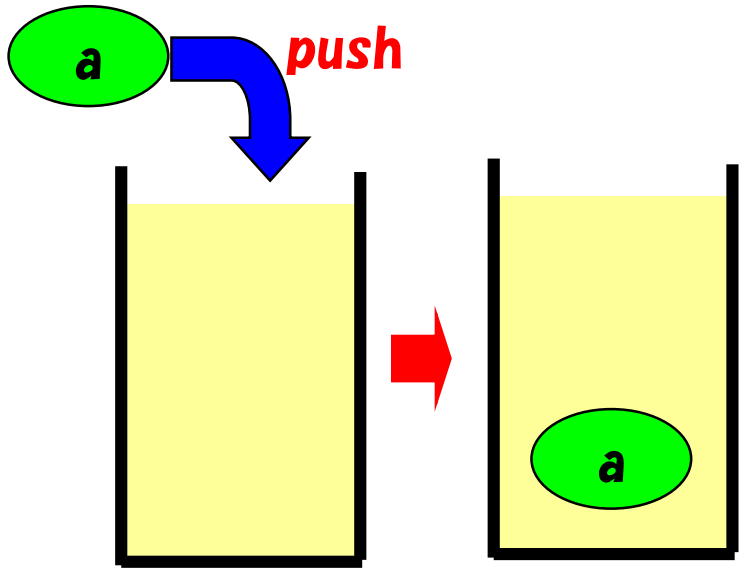



- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック (Stack) の概念 (“what”)



- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器

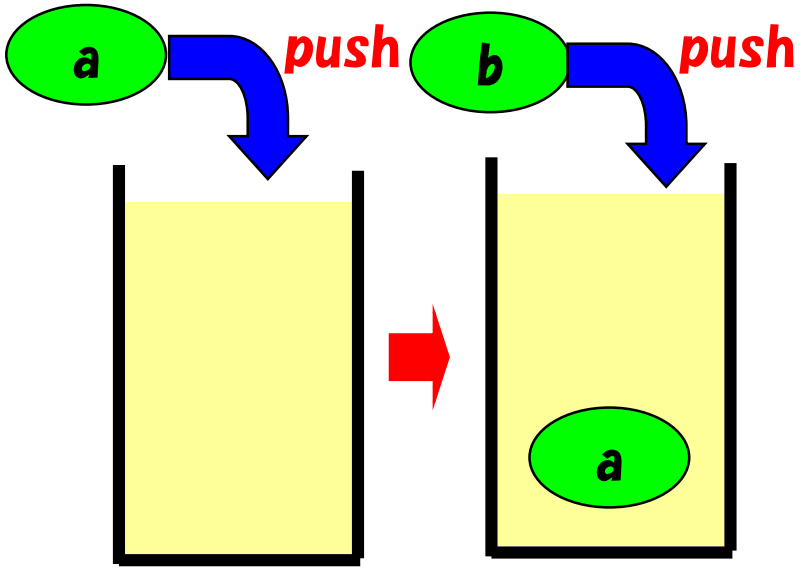



- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック (Stack) の概念 (“what”)



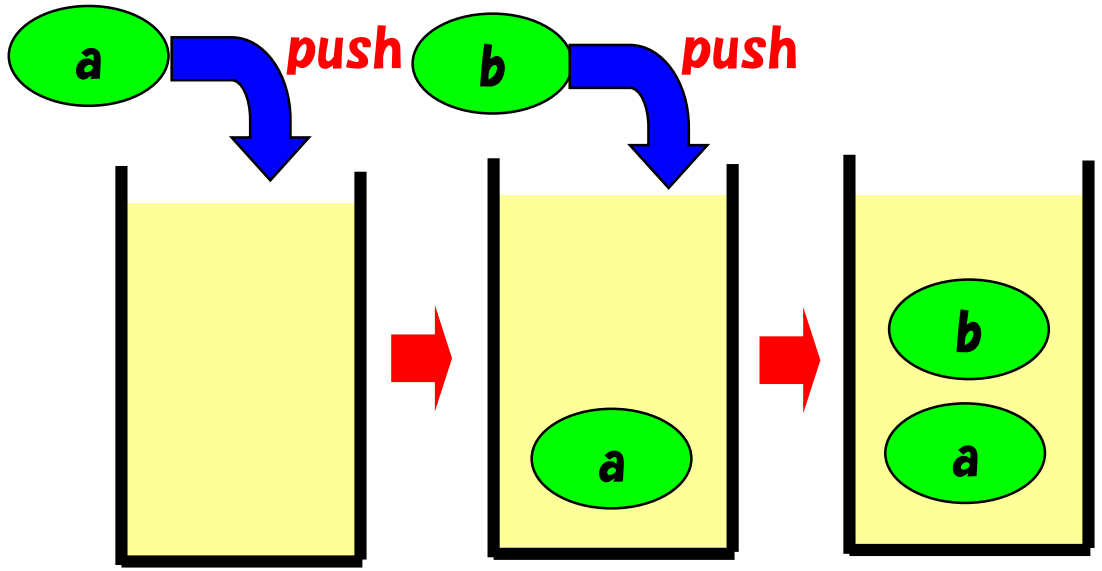
- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器




- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック (Stack) の概念 (“what”)

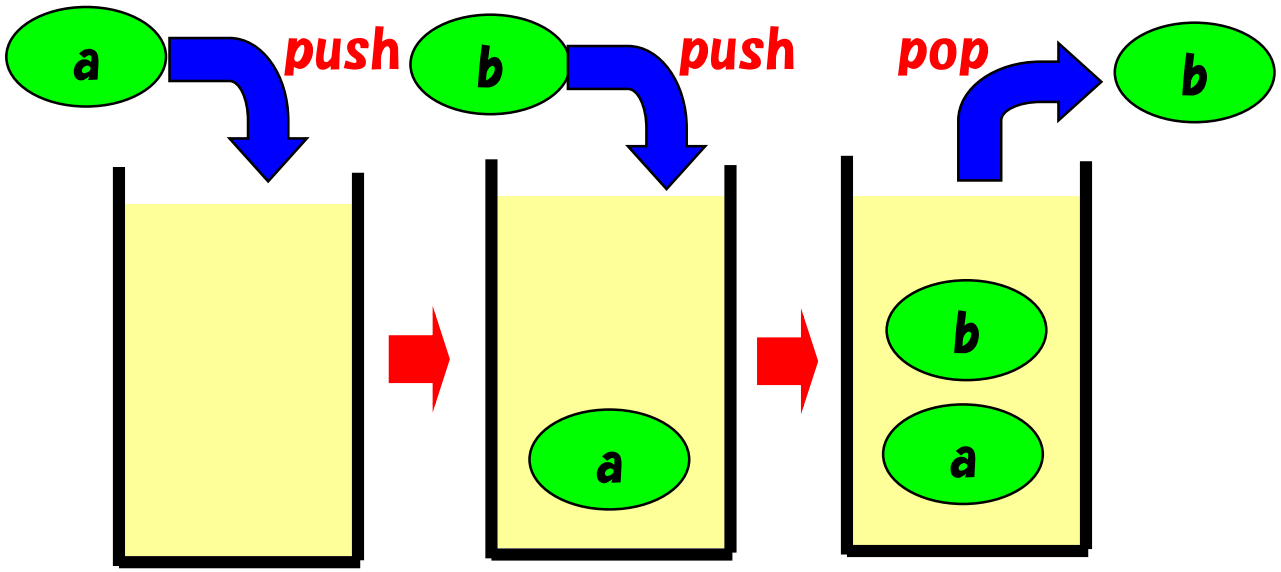
- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器




- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック (Stack) の概念 (“what”)

- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器

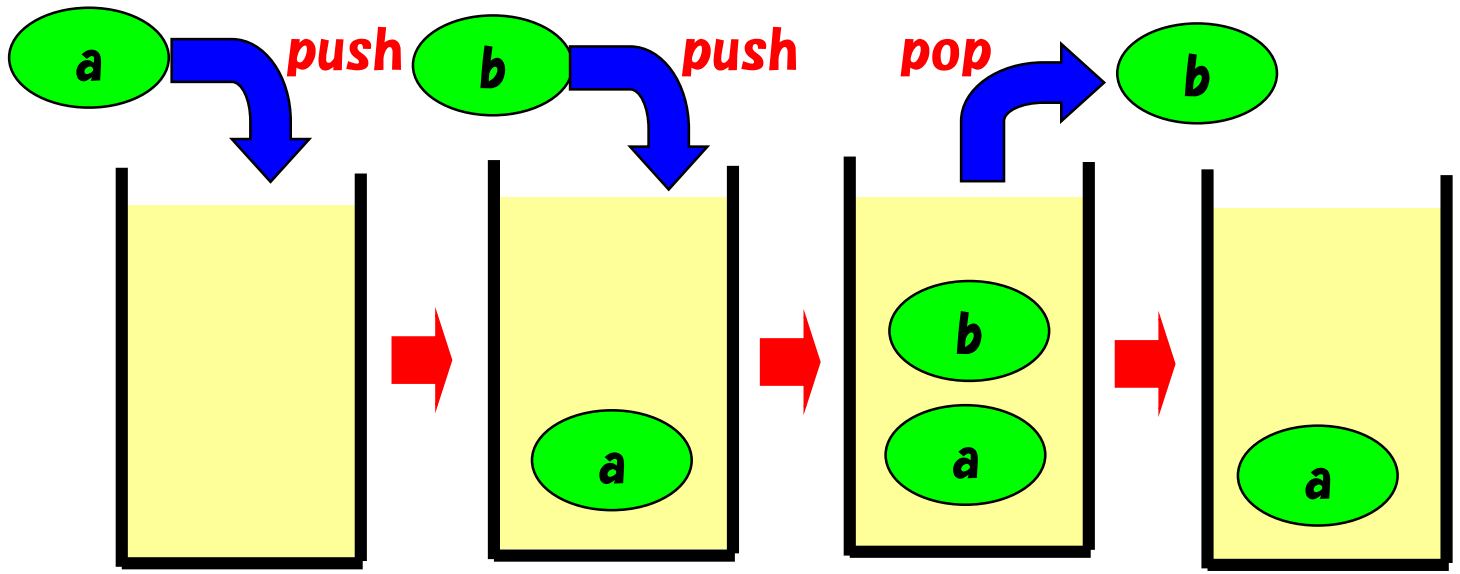



- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック(Stack)の概念(“what”)



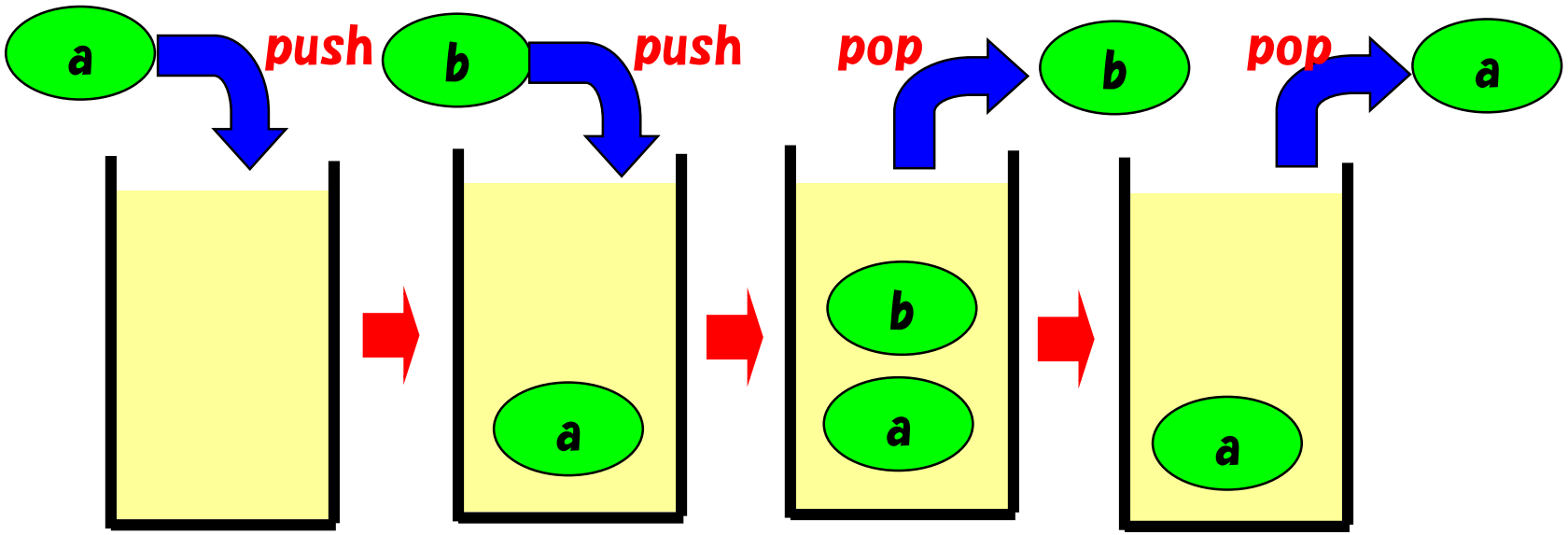
- FILO(First-in/Last-out；先入れ後出し)データ構造
- データ構造なので，そのままの形で実世界に存在しない(?)が，
  アルゴリズム(とくに再帰＝入れ子構造)の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器




- 設計例-1:配列と整数型変数(スタックポインタ)
- 設計例-2:連結リスト

# 例：スタック (Stack) の概念 (“what”)

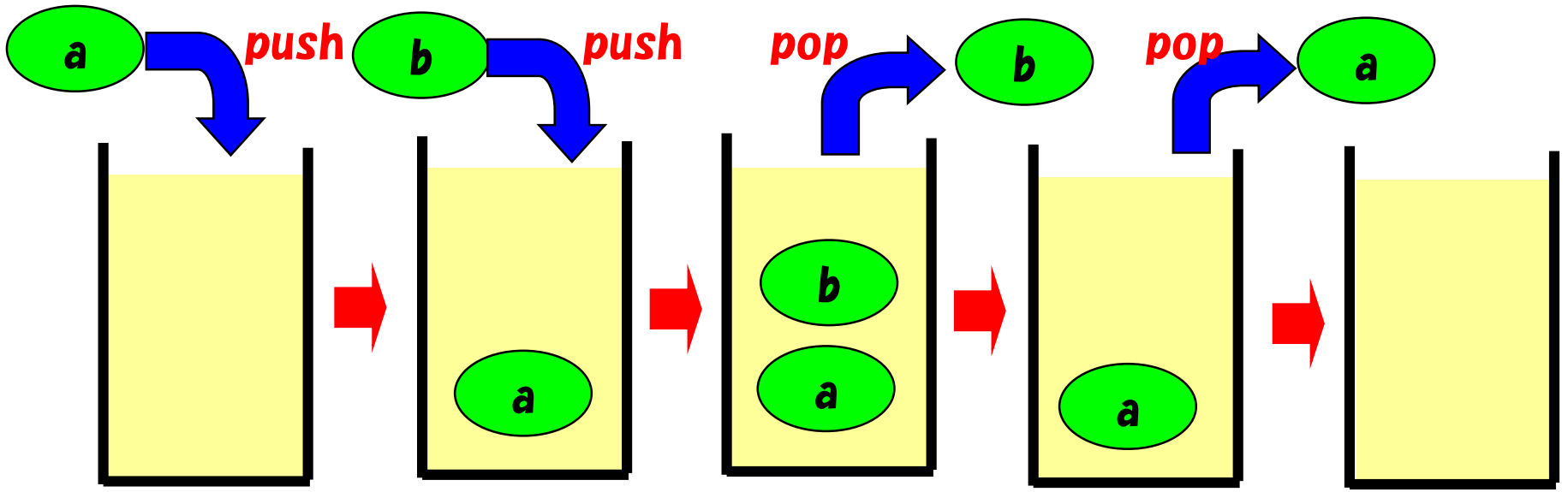
- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器




- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック (Stack) の概念 (“what”)

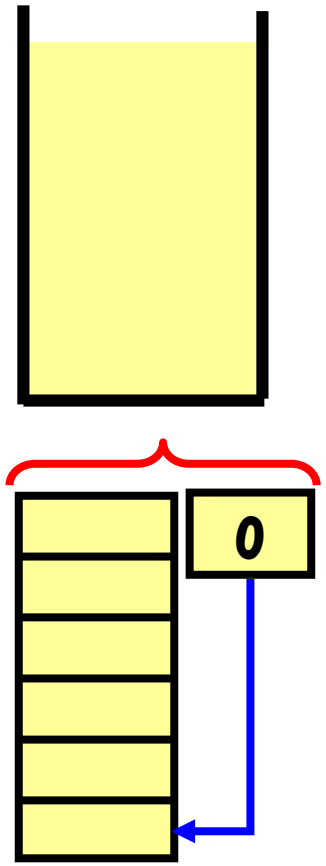
- FILO (First-in/Last-out；先入れ後出し) データ構造
- データ構造なので，そのままの形で実世界に存在しない (?) が，
  アルゴリズム (とくに再帰＝入れ子構造) の中にしばしば存在する．
- 口が一つで中に物が積み重なって入るような容器




- 設計例-1：配列と整数型変数(スタックポインタ)
- 設計例-2：連結リスト

# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～



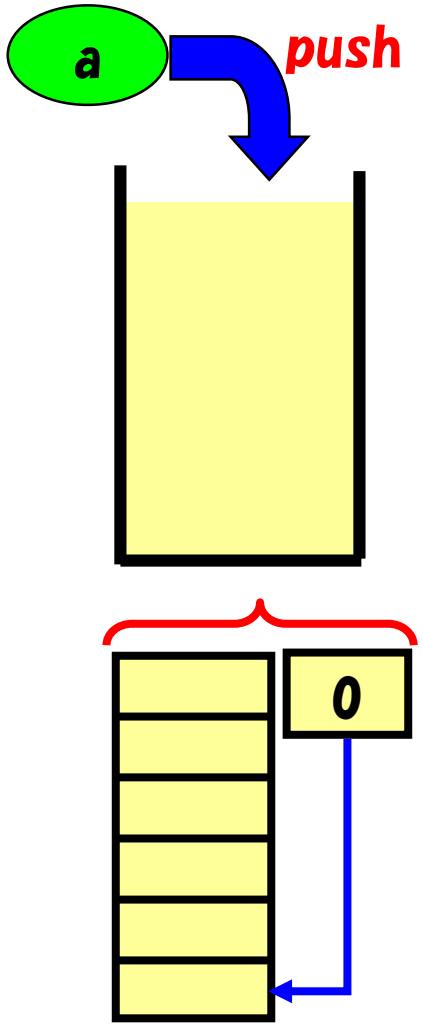




iijima@ae.keio.ac.jp

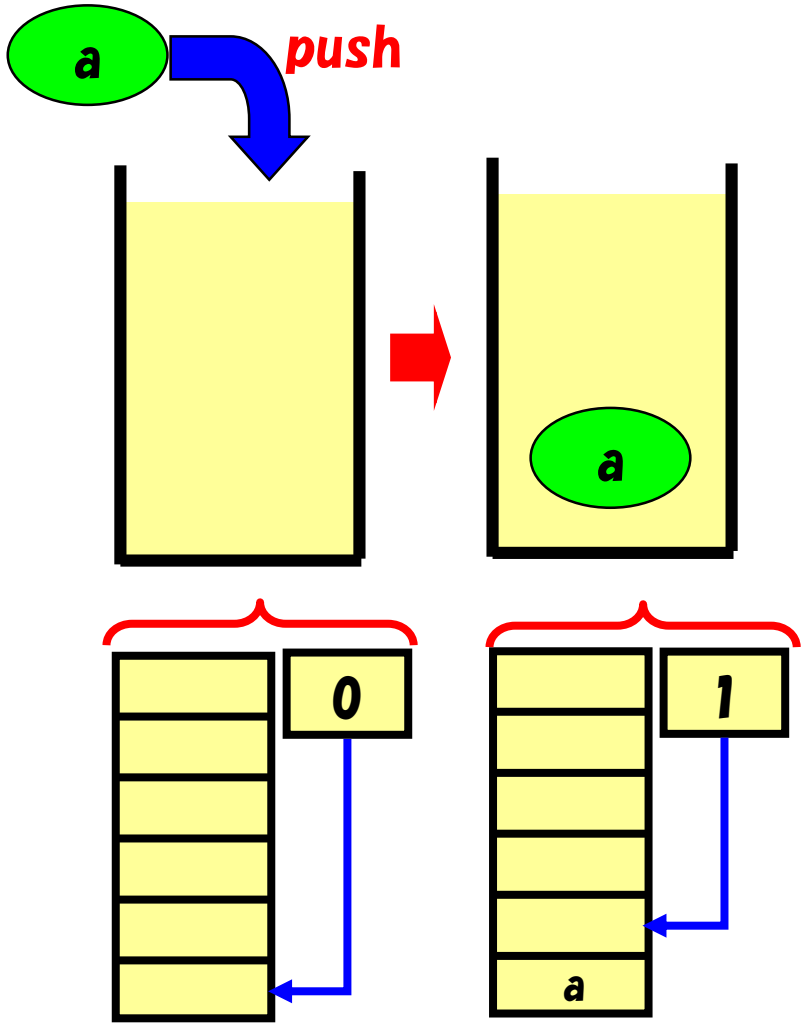
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





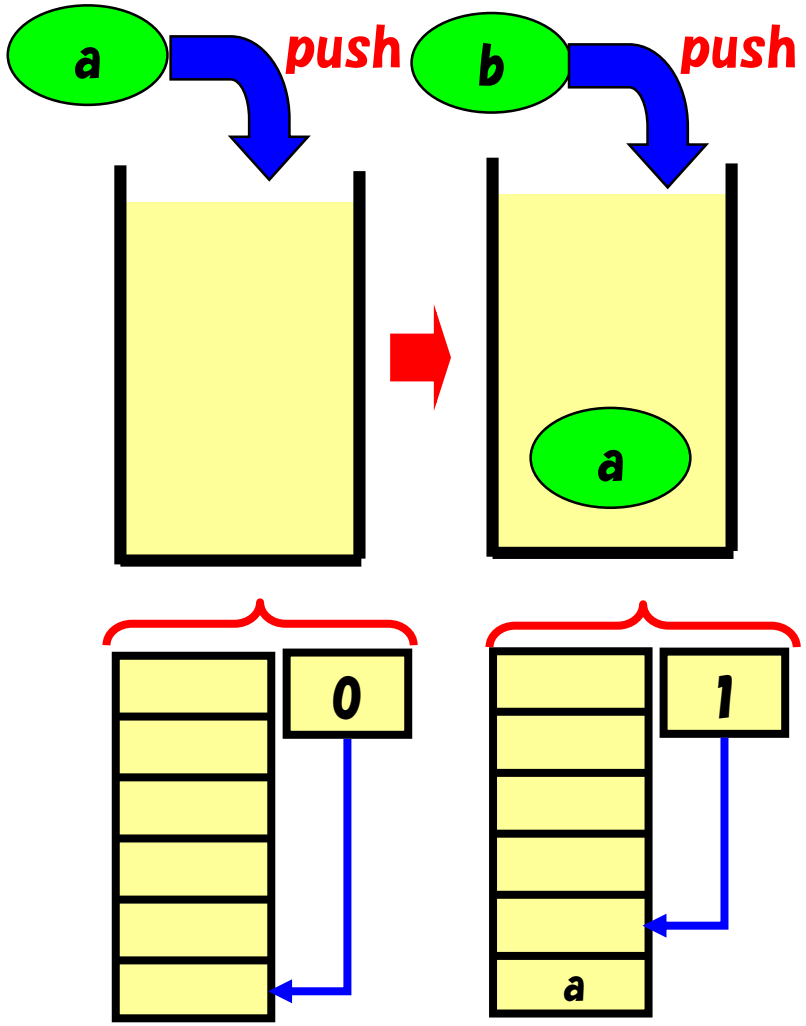
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





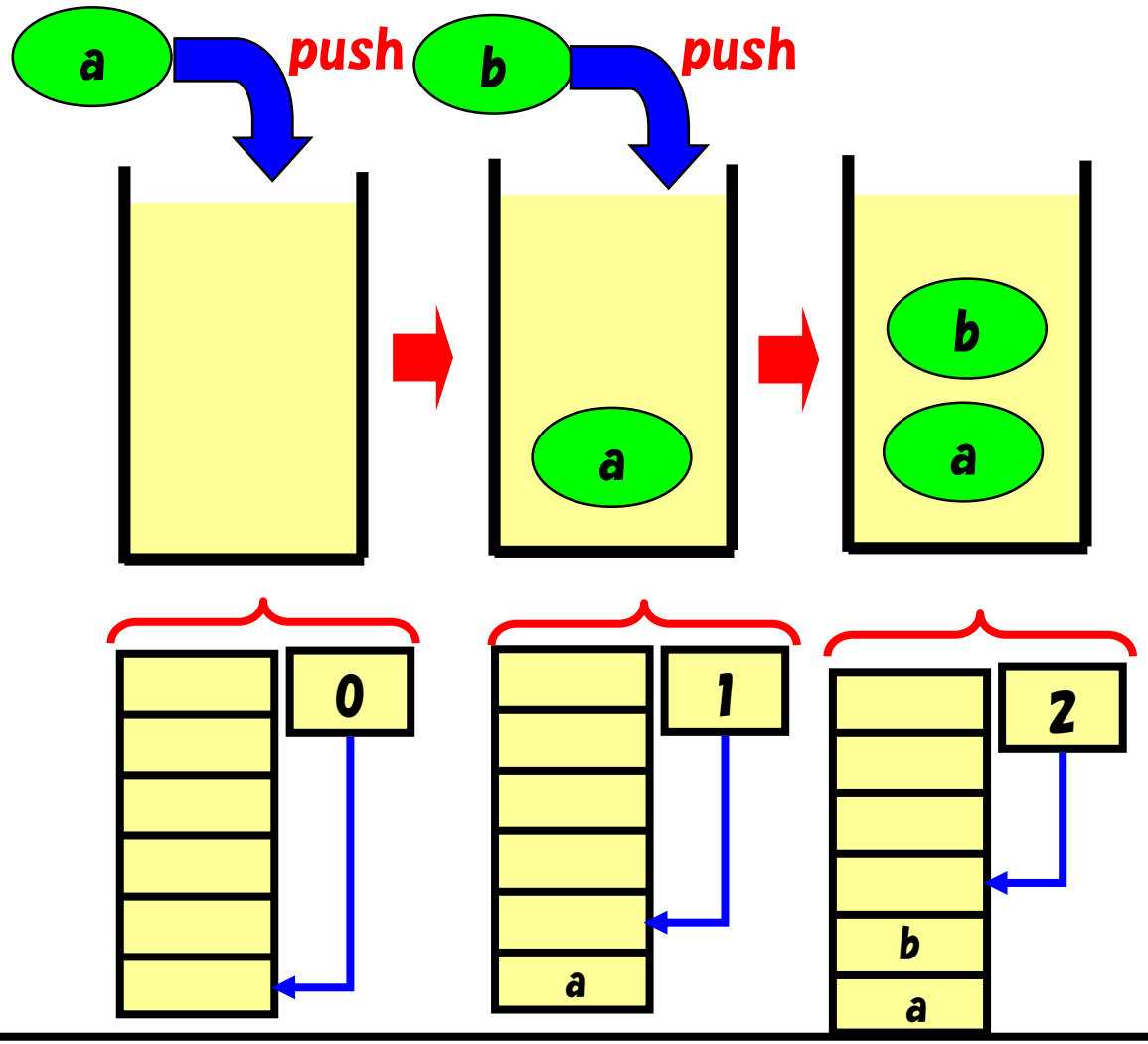
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





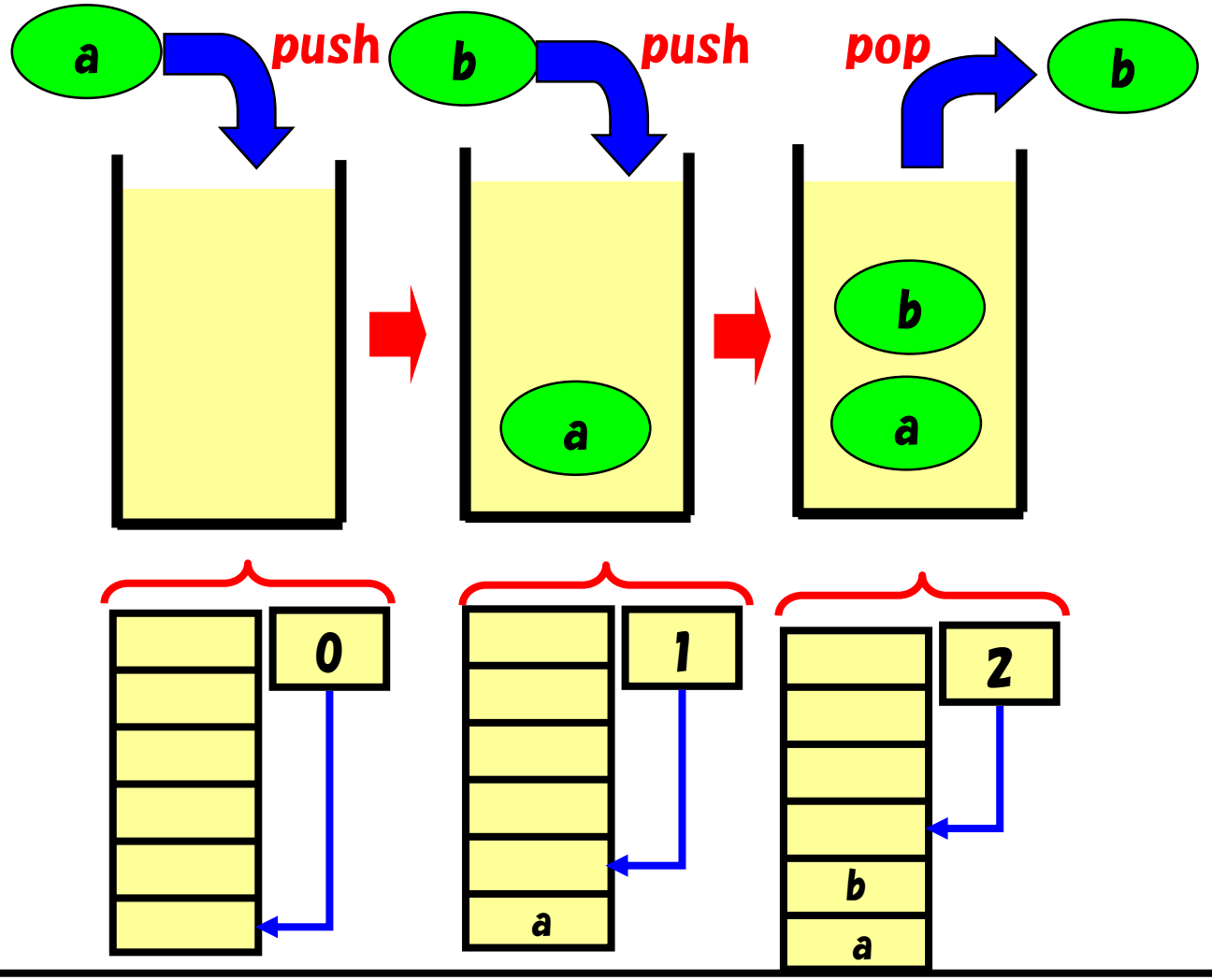
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





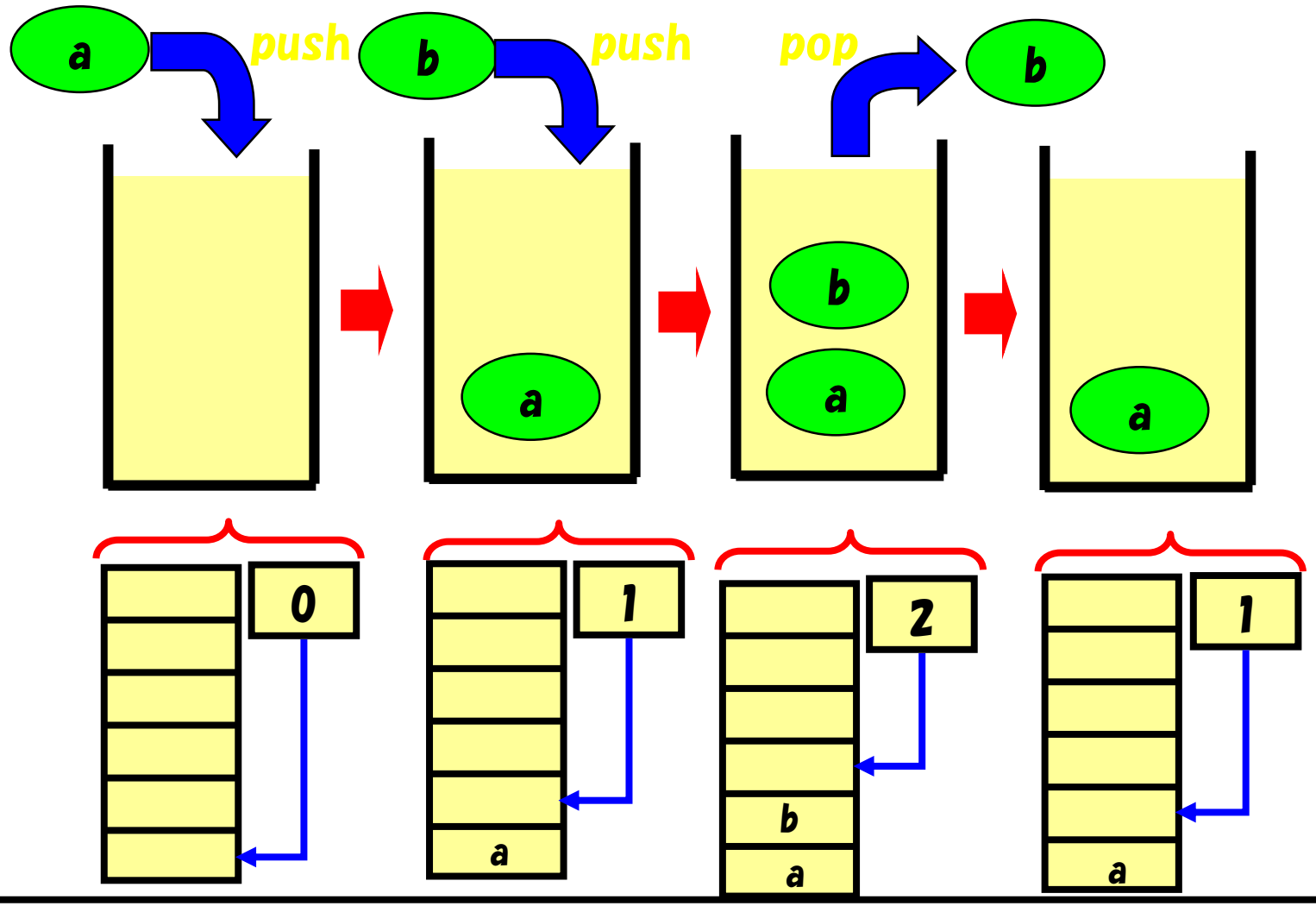
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





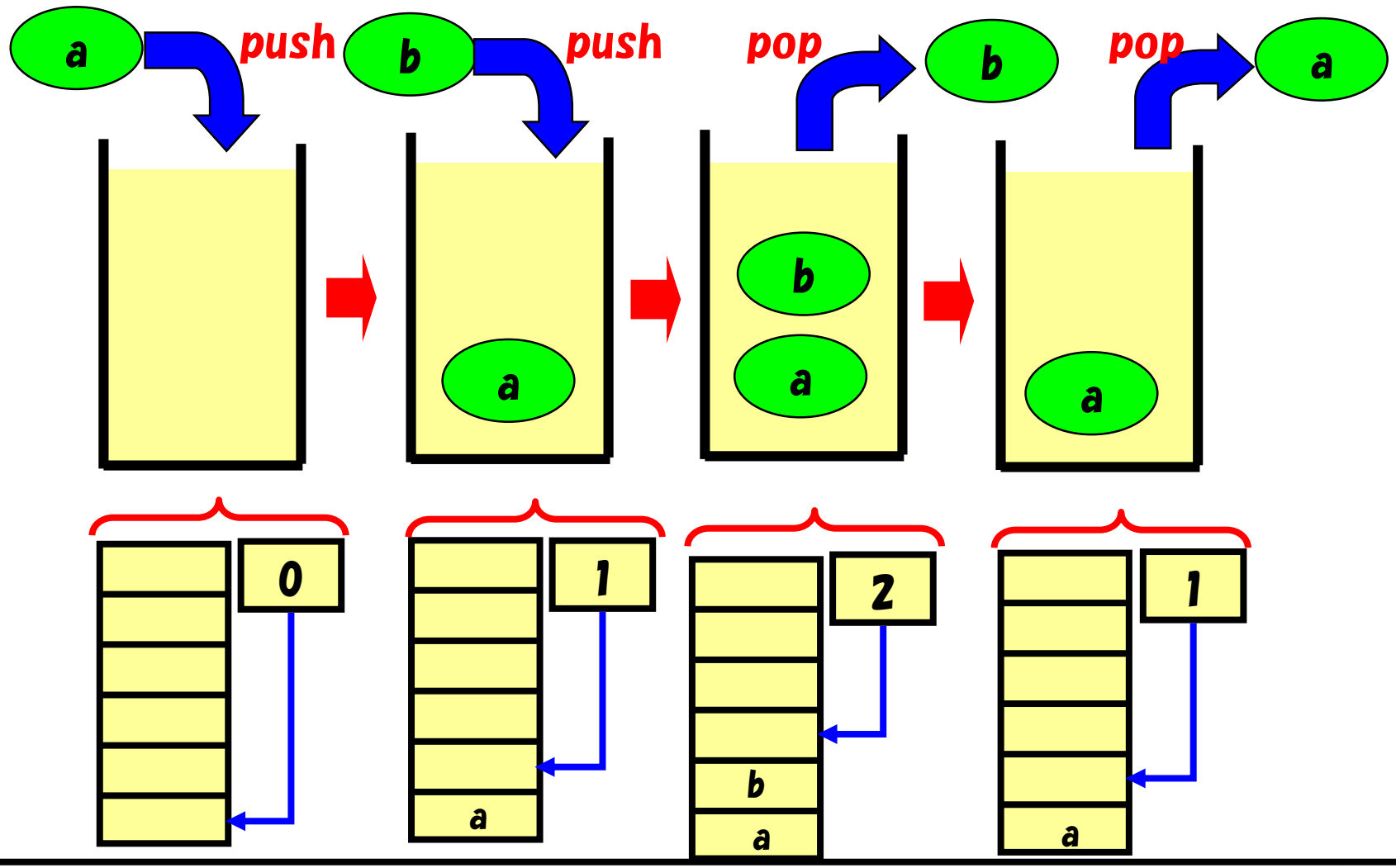
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





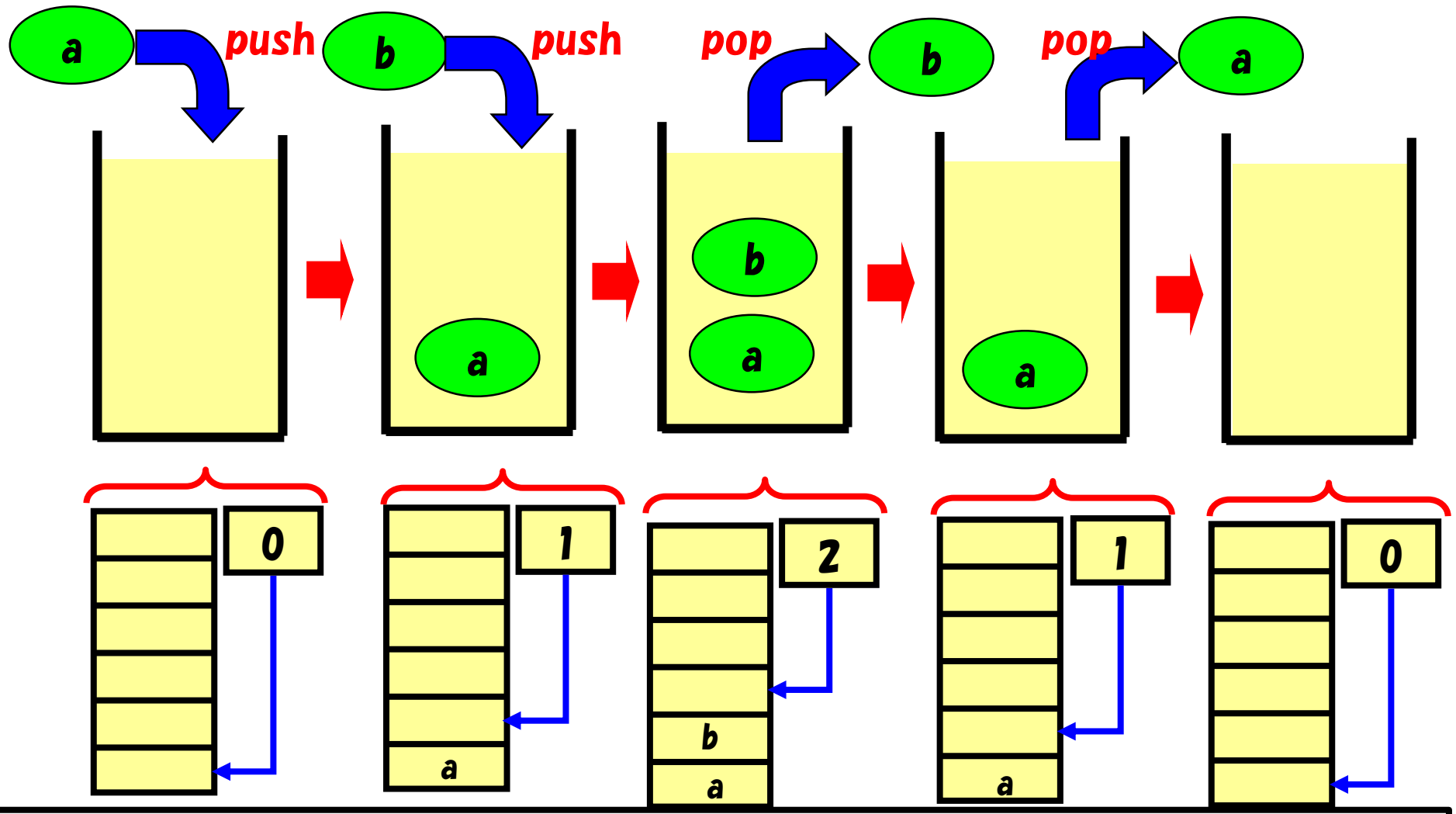
# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～





# 例：スタック(Stack)の設計(“how”)-1
## ～配列＋スタック・ポインタ～

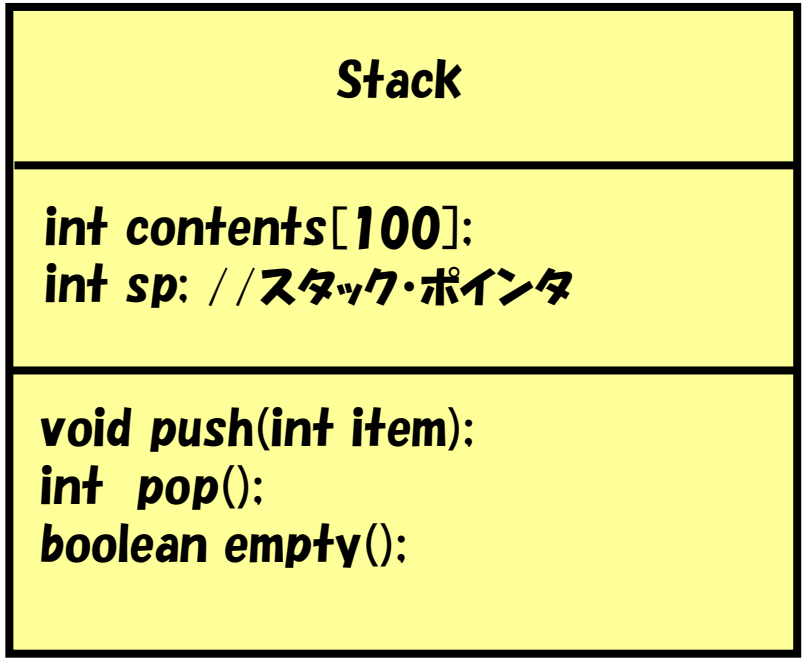




# インスタンス生成 (instantiation)

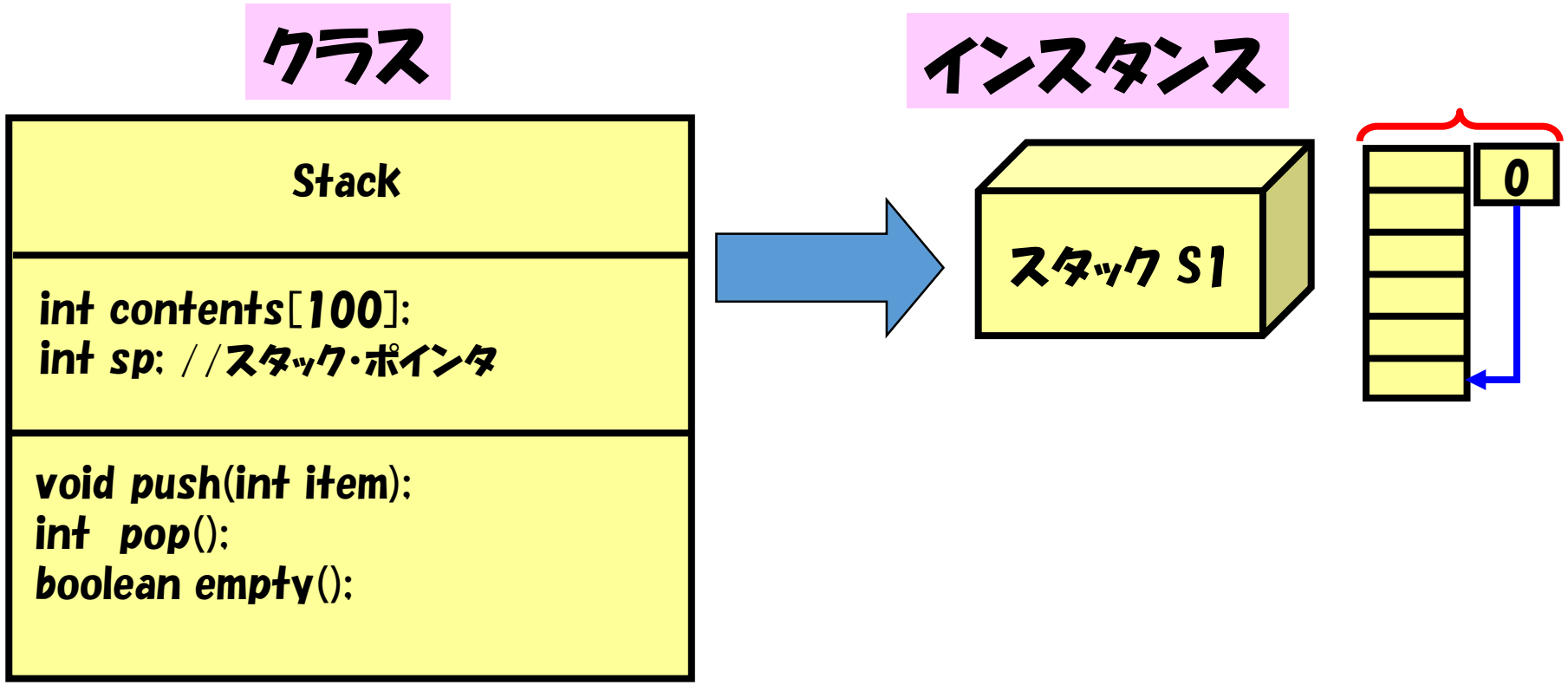


クラス




共通のクラスStackから
2つのインスタンスS1,S2を
順次，生成する
（属性保持用メモリ空間の動的確保）

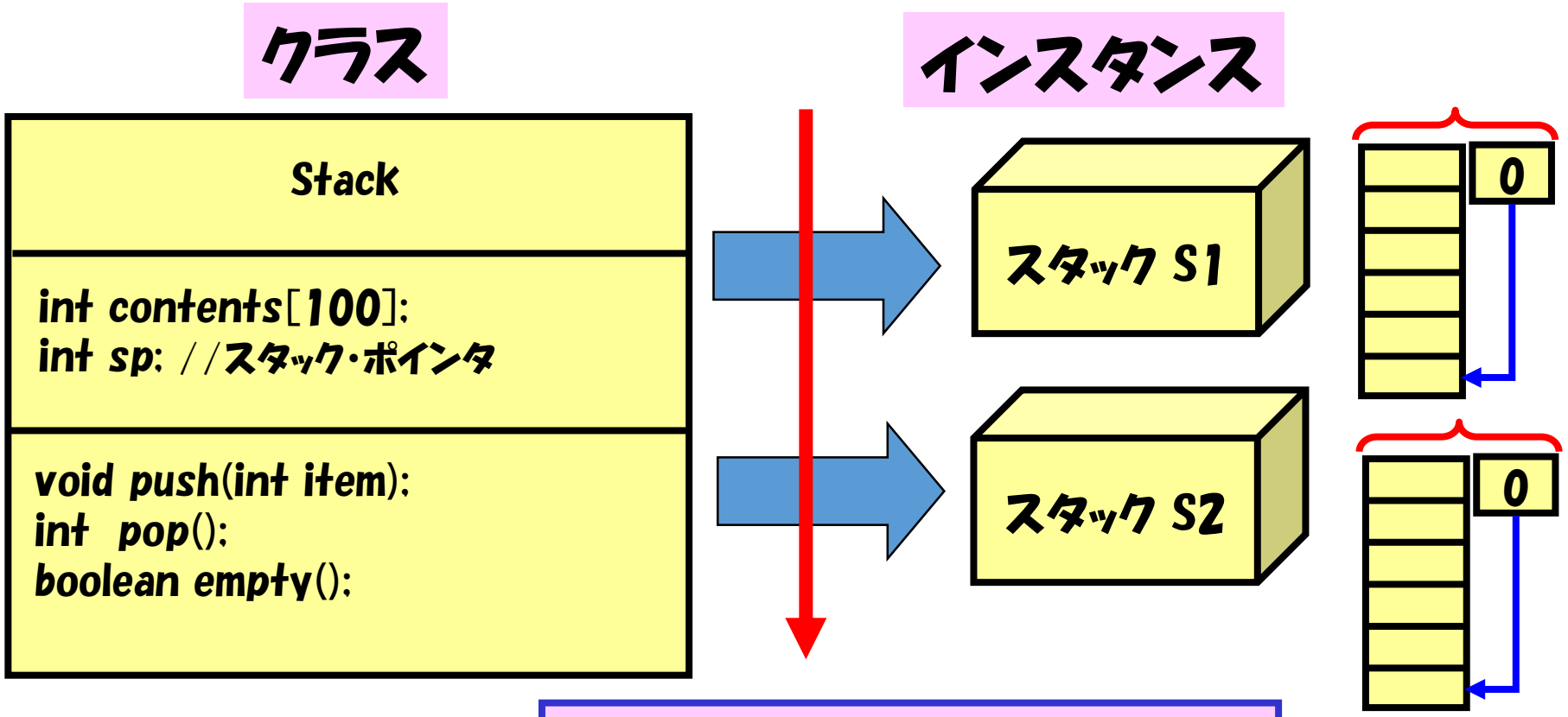
# インスタンス生成 (instantiation)




共通のクラスStackから
2つのインスタンスS1,S2を
順次，生成する
(属性保持用メモリ空間の動的確保)

# インスタンス生成 (instantiation)






共通のクラスStackから
2つのインスタンスS1,S2を
順次，生成する
（属性保持用メモリ空間の動的確保）

# 【参考】Javaの標準ライブラリに含まれている Stackクラスを使う

- 一つのクラスから二つのインスタンスを生成する

```
import java.util.*;

class StackTest {
        static public void main( String args[] ) {
                Stack s1 = new Stack();
                Stack s2 = new Stack();

                System.out.println("Stack Test");
                s1.push("s1の最初の要素");
                s2.push("s2の最初の要素");
                s2.push("s2の二番目の要素");
                s1.push("s1の二番目の要素");
                System.out.println(s1.pop());
                System.out.println(s1.pop());
                System.out.println(s2.pop());
                System.out.println(s2.pop());
        }
}
```

ライブラリのインポート

インスタンスの生成

メッセージ送信（メソッド呼び出し）

# 【参考】Javaでの簡単なスタックを作ってみる



(インスタンス)変数ないし属性

コンストラクタ

(インスタンス)メソッド

(クラス)メソッド

```
public class MyStackTest {
  static public void main(String args[]){
        MyStack s1 = new MyStack();

        System.out.println("Stack Test");
        s1.push(100);
        s1.push(200);
        System.out.println(s1.pop());
        System.out.println(s1.pop());
  }

}
```

```
public class MyStack {
  private int     v[];
  private int     p;

  MyStack() {
        v = new int[20];
        p = 0;
  }

  public int      pop() {
        if ( p > 0 ) {
              p--;
        }
        return v[p];
  }
  public void     push( int x ) {
        if ( p <= 19 ) {
              v[p] = x;
              p++;
        }
  }
}
```

# Collections Framework
# インタフェースと実装の関係

| | | 実装 | | | | |
|---|---|---|---|---|---|---|
| | | ハッシュテーブル | サイズ変更可能な配列 | バランスツリー | リンクリスト | Hash Table + Linked List |
| I/F | Set | HashSet | | TreeSet | | LinkedHashSet |
| | List | | ArrayList | | LinkedList | |
| | Map | HashMap | | TreeMap | | LinkedHashMap |

# 実装とインタフェース



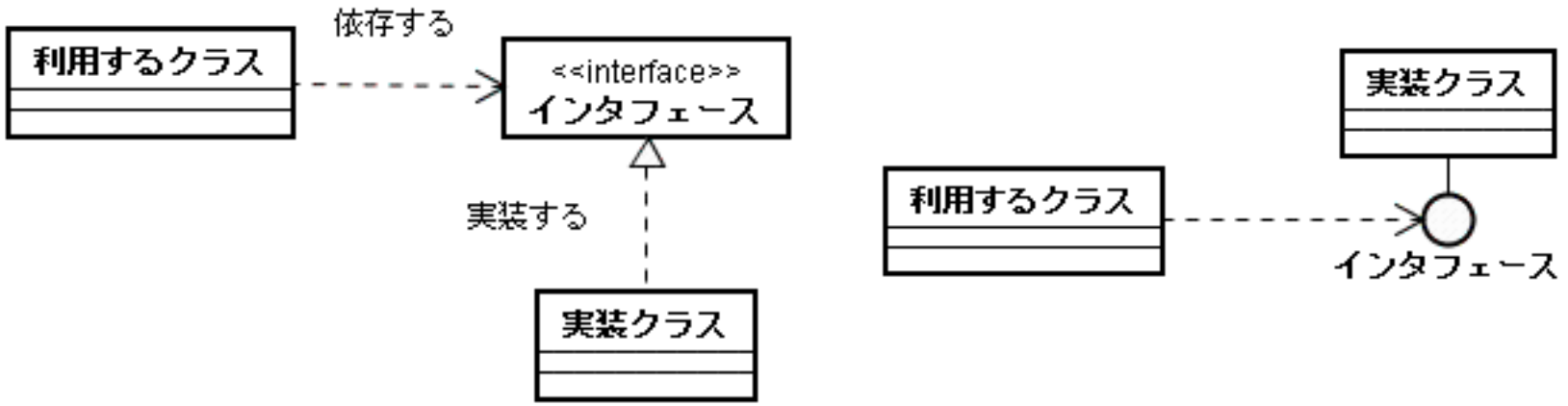
- java.lang.Object
- java.util.AbstractCollection<E>
- java.util.AbstractList<E>
- java.util.ArrayList<E>
- java.util.AbstractSequentialList<E>
- java.util.LinkedList<E>
- java.util.Vector<E>
- java.util.Stack<E>
- java.util.AbstractQueue<E>
- java.util.PriorityQueue<E>
- java.util.AbstractSet<E>
- java.util.EnumSet<E>
- java.util.HashSet<E>
- java.util.LinkedHashSet<E>
- java.util.TreeSet<E>
- java.util.AbstractMap<K,V>
- java.util.EnumMap<K,V>
- java.util.HashMap<K,V>
- java.util.LinkedHashMap<K,V>
- java.util.IdentityHashMap<K,V>
- java.util.TreeMap<K,V>
- java.util.WeakHashMap<K,V>

java.lang.Iterable<T>
  java.util.Collection<E>
    java.util.List<E>
    java.util.Queue<E>
    java.util.Set<E>
      java.util.SortedSet<E>

java.util.Map<K,V>
  java.util.SortedMap<K,V>

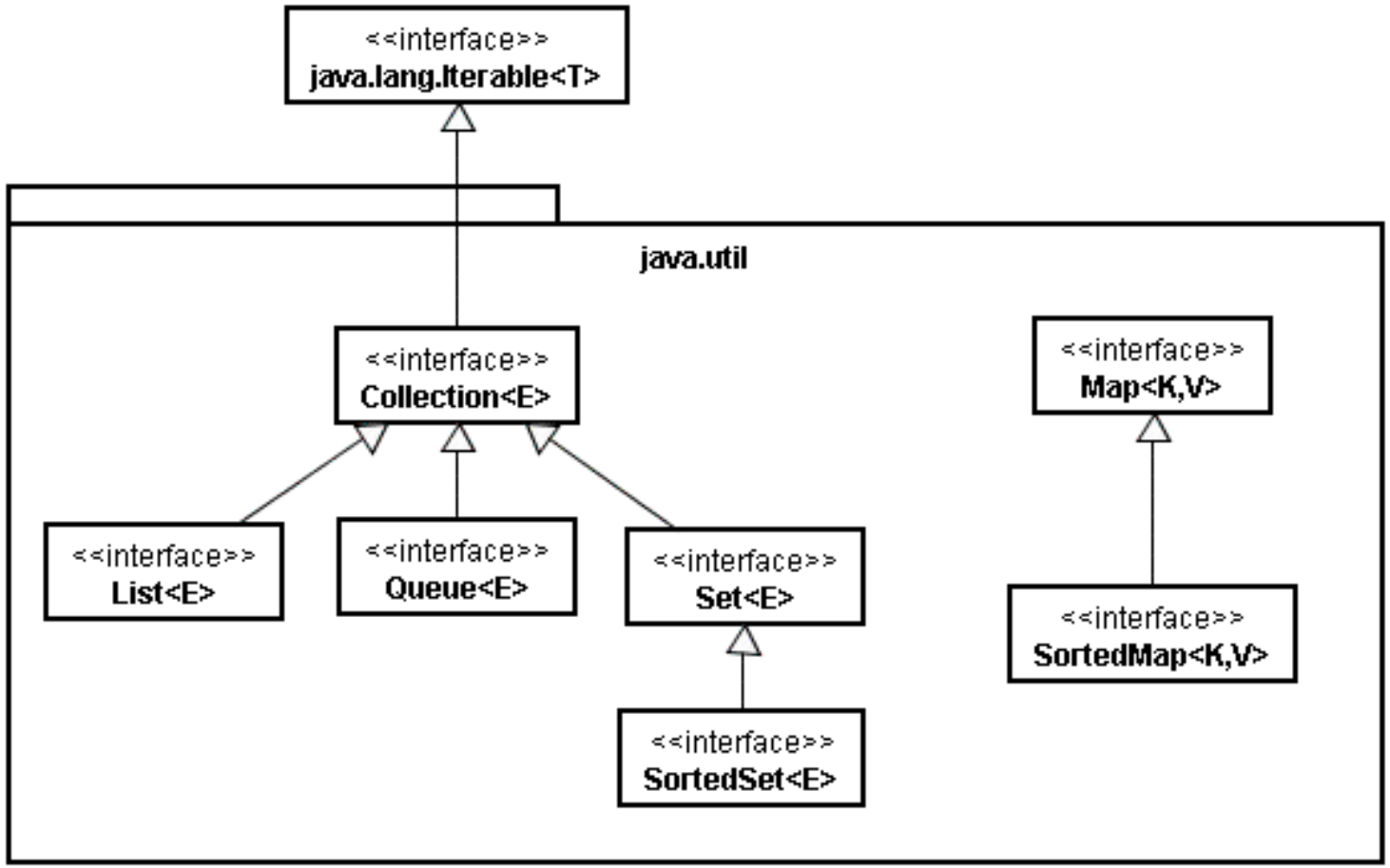
# インタフェースと実装のUML表記





# インタフェースの階層構造



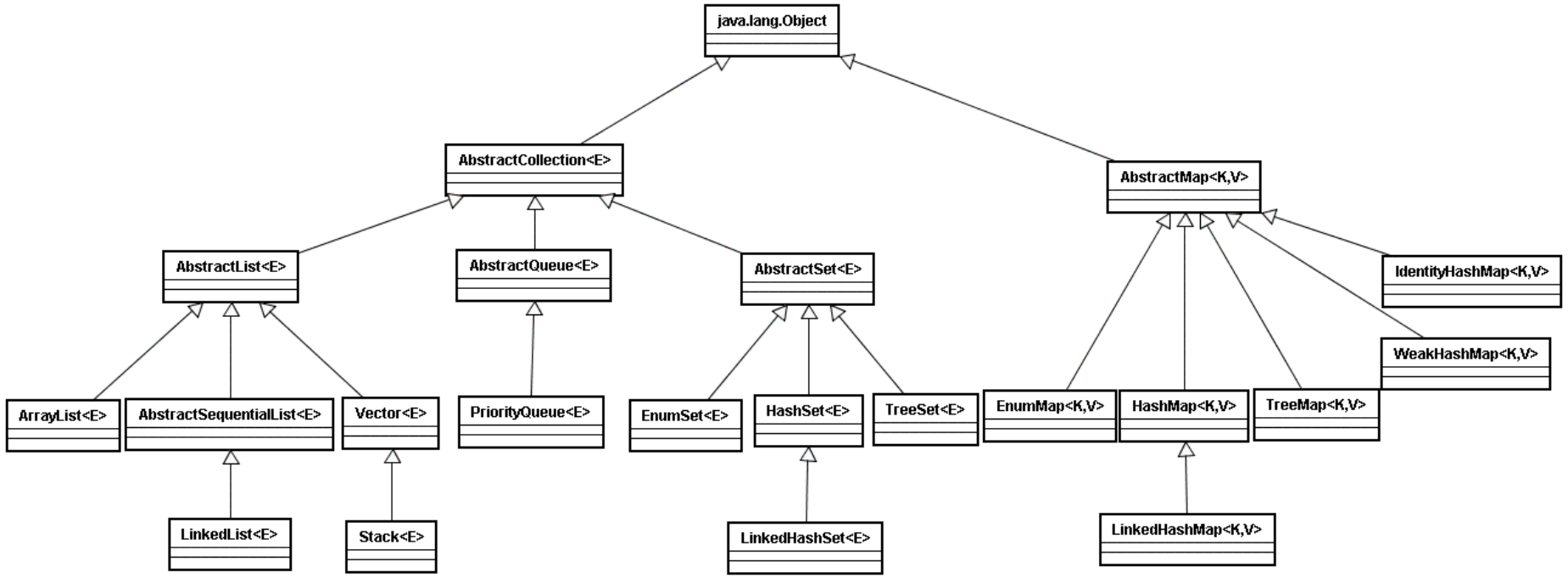




iijima@ae.keio.ac.jp

# 実装クラスの階層構造



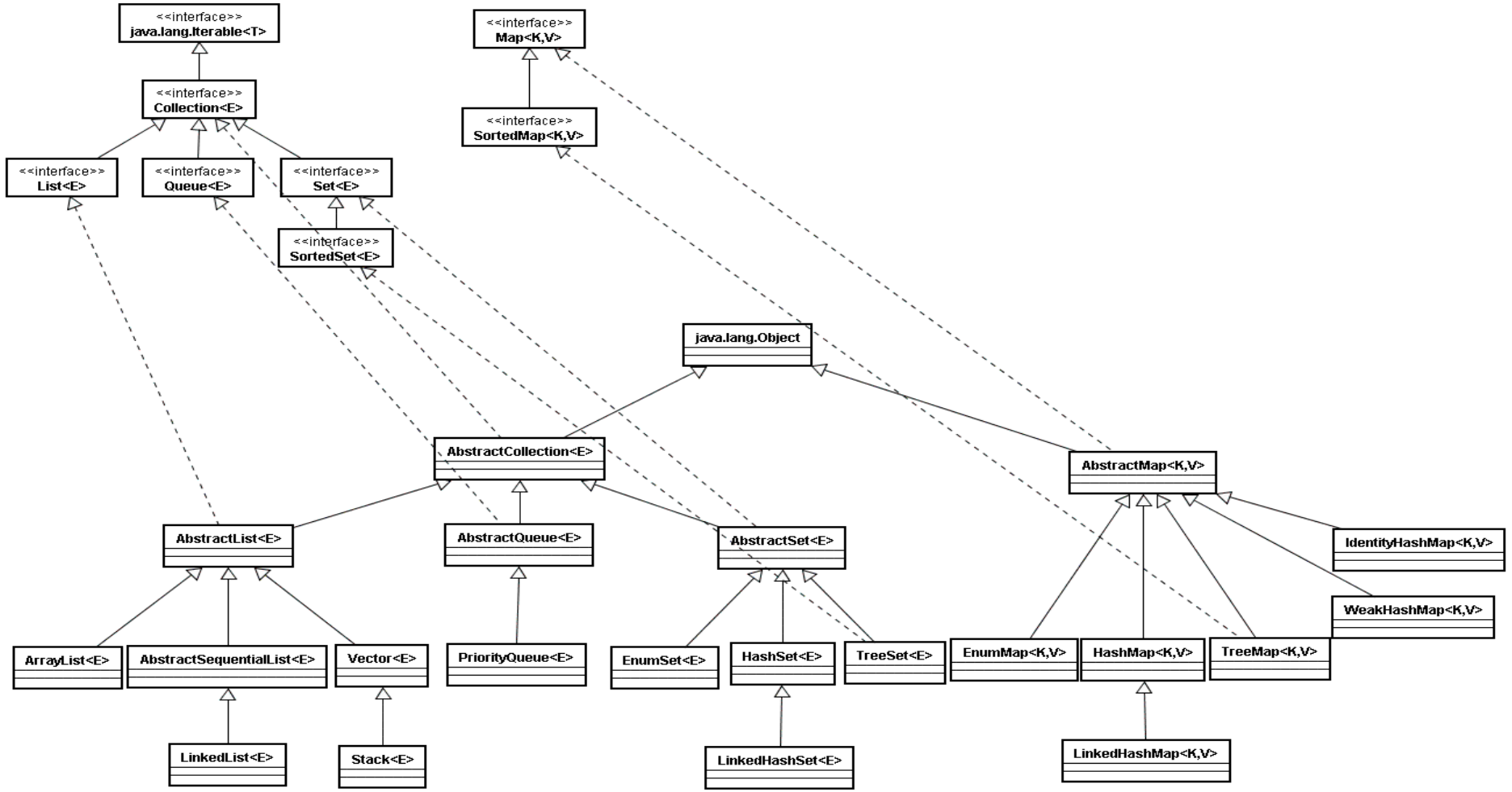




iijima@ae.keio.ac.jp

# インタフェースと実装の関係





# インタフェース



| | |
|---|---|
| Collection | オブジェクトの集まり |
| Set | 重複要素のないオブジェクトの集まり(集合) |
| List | 順序付けられたオブジェクトの集まり(リスト). 重複を許可する.<br>リスト内のどこに各要素が挿入されるかを制御することができ、<br>利用者は位置(添え字)を指定して各要素にアクセスできる. |
| Map | キーを値に対応付ける写像.<br>キーは重複して登録することはできない.<br>各キーはそれぞれ一つの値に写像する. |
| SortedSet | 要素が自然順序付けに従って昇順にソートされた集合を表す.<br>挿入される要素は, Comparable インタフェースを実装するか,<br>指定された Comparator によって受け付けられる必要があある. |
| SortedMap | キーが自然順序付けに従って昇順にソートされたマップを表す.<br>挿入されるキーは, Comparable インタフェースを実装するか,<br>指定された Comparator によって受け付けられる必要がある. |

# 実装



| | | |
|---|---|---|
| Set | HashSet | HashMap のインスタンスに基づくSet インタフェースを実装する.<br>繰り返し順序について保証しない. |
| | TreeSet | TreeMap のインスタンスに基づくSet インタフェースを実装する.<br>要素の昇順でソートされる. |
| | LinkedHashSet | 繰り返し順序を持つSet インタフェースのハッシュテーブルとリンクリストの実装である. 要素がセットに挿入された順序を保持する.<br>要素をセットに再挿入する場合、挿入順は影響を受けない. |
| List | ArrayList | Listインタフェースのサイズ変更可能な配列の実装.<br>配列のサイズを操作するメソッドを提供する. |
| | LinkedList | List インタフェースのリンクリストの実装. リストの先端および終端にある要素を取得、削除したり、先端および終端に要素を挿入したりするメソッド を提供する.<br>同期化されないことを除いてVectorとほぼ同等. |
| Map | HashMap | Map インタフェースのハッシュテーブルに基づく実装.<br>マップの順序について保証しない. 同期化されないこととnullを許容することを除いてHashtableとほぼ同等. |
| | TreeMap | SortedMap インタフェースの実装に基づく Red-Black ツリー.<br>マップがキーの昇順でソートされる. |
| | LinkedHashMap | 繰り返し順序を持つMap インタフェースのハッシュテーブルとリンクリストの実装.<br>キーがマップに挿入された順序を保持する. キーをマップに再挿入する場合、挿入順は影響を受けない. |

# Set: 集合



| | |
|---|---|
| HashSet | Setの機能だけ必要な場合 |
| TreeSet | 要素を昇順にソートする必要がある場合 |
| LinkedHashSet | 要素の挿入順を保持する必要がある場合 |

| | |
|---|---|
| Set | 重複要素のないオブジェクトの集まりを管理するクラス |

# Set：集合



| | | | |
|---|---|---|---|
| | boolean | add(Object) | 引数で指定された要素が，セットに存在しない場合追加される. |
| | boolean | addAll(Collection) | 引数で指定されたコレクションの要素すべてが，セットに存在しない場合追加される. |
| | void | clear( ) | セットからすべての要素を削除する. |
| | boolean | contains(Object) | 引数で指定された要素がセットに存在する場合，trueを返す |
| | boolean | containsAll(Collection) | 引数で指定されたコレクションの要素すべてが、セットに存在する場合，trueを返す |
| | boolean | remove(Object) | 引数で指定された要素が、セットに存在する場合その要素を削除する |
| | boolean | removeAll(Collection) | 引数で指定されたコレクションの要素の内、セットに含まれる要素を削除する |
| | Object | retainAll(Collection) | 引数で指定されたコレクションの要素の内、セットに含まれる要素を、セット内に保持する |
| | Object[] | toArray( ) | セット内のすべての要素が格納されている配列を返す. |
| | Object[] | toArray(Object[]) | セット内のすべての要素が格納されている配列を返す. 配列の実行時の型は引数で指定された型になる. |

# List: リスト

| List | 順序付けられたオブジェクトの集まりを管理するクラス. |
|---|---|
| | 重複を許可します。リスト内のどこに各要素が挿入されるかを制御することができ，ユーザーは位置(インデックス)を指定して各要素にアクセスすることができる. |

| ArrayList | List インタフェースのサイズ変更可能な配列の実装.<br>配列のサイズを操作するメソッドを提供する. |
|---|---|
| | インデックスによるランダムアクセスの性能が優れているが，<br>要素の挿入、削除には向いていない. |
| LinkedList | List インタフェースのリンクリストの実装. リストの先端および終端にある要素を取得、削除したり、先端および終端に要素を挿入したりするメソッド を提供する. 同期化されないことを除いてVectorとほぼ同等である. |
| | 要素の挿入、削除の性能が優れていますが、インデックスによるランダムアクセスには向いていない. |

| | インデックス<br>アクセス | Iterator<br>アクセス | 追加 | 挿入 | 削除 |
|---|---|---|---|---|---|
| ArrayList | ○ | ○ | ○ | 遅 | 遅 |
| LinkedList | 遅 | ○ | ○ | ○ | ○ |

# List：リスト



| | | | |
|---|---|---|---|
| 追加 | void | add(int, Object) | 引数intで指定された位置に，要素を追加する. |
| | boolean | add(Object) | リストの最後に要素を追加する. |
| | boolean | addAll(Collection) | リストの最後に，引数Collectionで指定された要素すべてを追加する |
| | boolean | addAll(int, Collection) | 引数intで指定された位置に，引数Collectionで指定された要素すべてを追加する. |
| | void | clear( ) | リストからすべての要素を削除する. |
| | Object | get(int) | リスト内のインデックス番号で指定された位置にある要素を返す. |
| | Object | remove(int) | リスト内のインデックス番号で指定された位置にある要素を削除する.<br>返り値として，削除された要素を返す. |
| | Object | set(int, Object) | リスト内のインデックス番号で指定された位置にある要素を引数Objectで指定された要素に置き換える. 返り値として，置き換えられた古い要素を返す. |
| | int | size( ) | リスト内にある要素の数を返す. |

# Map: 写像



| | |
|---|---|
| Map | キーを値にマッピングする場合に使用するクラス |

| | |
|---|---|
| HashMap | Mapの機能だけ必要な場合 |
| TreeMap | キーを昇順にソートする必要がある場合 |
| LinkedHashMap | キーの挿入順を保持する必要がある場合 |

# Map：写像



| | | | |
|---|---|---|---|
| | void | clear( ) | Mapからすべての要素を削除する. |
| | boolean | containsKey(Object) | 引数で指定されたキーがMapに存在する場合, trueを返す. |
| | boolean | containsValue(Object) | 引数で指定された値がMapに存在する場合, trueを返す. |
| | Object | get(Object) | 引数に指定されたキーに紐付けられた値を返す. |
| | Object | put(Object, Object) | 指定された引数(キー, 値)の対を, Mapに挿入する. |
| | void | putAll(Map) | 引数に指定されたMapの要素すべてを、Mapに挿入する. |
| | Object | remove(Object) | 引数に指定されたキーがMapに存在する場合、そのキーと対応付けられた値を削除する |
| | Set | entrySet( ) | Mapに格納されている要素を持つ, コレクションビューを返す. |
| | Set | keySet( ) | Mapに格納されているキーを持つ, セットビューを返す. |
| | Collection | values( ) | Mapに格納されている値を持つ, コレクションビューを返す. |

# Iterator



| | | | |
|---|---|---|---|
| 次要素の有無 | boolean | hasNext( ) | 繰り返し処理において次の要素がある場合に，<br>trueを返す． |
| 次要素 | Object | next( ) | 繰り返し処理において，次の要素を返す． |
| 削除 | void | remove( ) | 繰り返し処理において，<br>呼び出された最後の要素を削除する |

```
for (Iterator i = c.iterator(); i.hasNext();) {
        … 繰り返される処理 …
}
```

```
Iterator i = c.iterator();
while (i.hasNext() {
        … 繰り返される処理 ….
 }
```

```
for (Iterator i = hs1.iterator(); i.hasNext();) {
        i.next();
        i.remove();
 }
```

# ListIterator



| | | | |
|---|---|---|---|
| 次要素の有無 | boolean | hasNext( ) | 繰り返し処理において，<br>次の要素がある場合にtrueを返す． |
| 次要素 | Object | next( ) | 繰り返し処理において，次の要素を返す． |
| 削除 | void | remove( ) | 繰り返し処理において，<br>呼び出された最後の要素を削除する． |

| | | | |
|---|---|---|---|
| 前要素の有無 | boolean | hasPrevious( ) | 繰り返し処理において，<br>前の要素がある場合にtrueを返す． |
| 前要素 | Object | previous( ) | 繰り返し処理において，前の要素を返す． |
| 追加 | | add | |
| 変更 | | set | |
| | | nextIndex | |
| | | previousIndex | |

# 繰り返しとforeach文 (JDK1.5.0以降)

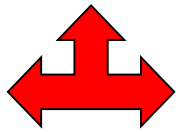


Listの繰り返し子Iteratorによるアクセス

```
List names = new ArrayList();
names.add("a");
names.add("b");
names.add("c");

for (Iterator it = names.iterator(); it.hasNext(); ) {
    String name = (String)it.next();
    System.out.println( name );
}
```

Listの添え字によるアクセス

配列の添え字によるアクセス



```
String[] names = { "a", "b", "c" };
for ( int i = 0; i < names.length; i++ ) {
    System.out.println( names[i] );
}
```

```
List names = new ArrayList();
names.add("a");
names.add("b");
names.add("c");

for ( int i = 0; i < names.length; i++ ) {
    System.out.println( names.get[i] );
}
```

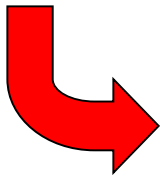
# 繰り返しとforeach文 (JDK1.5.0以降)



Listの繰り返し子Iteratorによるアクセス

```
List names = new ArrayList();
names.add("a");
names.add("b");
names.add("c");

for (Iterator it = names.iterator(); it.hasNext(); ) {
    String name = (String)it.next();
    System.out.println( name );
}
```

Listのforeach文によるアクセス(JDK-1.5以降)



```
List<String> names = new ArrayList<String>();
names.add("a");
names.add("b");
names.add("c");

for (String name: names) {
    System.out.println( name );
}
```

# Map



| | |
|---|---|
| キーからの値の検索 | get(キー) |
| 登録（既にそのキーが使われていたら，<br>対応する値を返す．さもなくば，nullを返す） | put(キー，値) |
| 削除 | remove(キー) |
| キーが登録済みかどうか？ | containsKey(キー) |
| 大きさを返す | size() |
| 空かどうか？ | isEmpty() |
| 空にする | clear() |

# 文字列カウンター (JDK-1.5.0以降)



```
import  java.util.*;

public class MapTest  {
    public static void main ( String args [] )
                CounterMap aMap =
                        new CounterMap ();
                aMap.countUp ("飯島");
                aMap.countUp ("山口");
                aMap.countUp ("飯島");
                aMap.countUp ("櫻井");
                aMap.countUp ("山口");
                aMap.countUp ("飯島");
                aMap.print ();
        }
}
```

```
class CounterMap extends
                        TreeMap<String,Integer> {
    void countUp( String w ) {
      if ( containsKey( w ) ) {
              put( w, get( w ) + 1 );
              // Java5ではInteger<-->intの変換が不要
      } else {
              put( w, 1 );
      }
    }
    void print() {
       for (String key : keySet() ) {
          System.out.println(key + " => " + get(key));
       }
    }
}
```


iijima@ae.keio.ac.jp

# 課題: 単語の出現頻度表



- ヒント：　文字列を単語で分割する
  - String line = in.readLine();
  - String[] result
    = line.split("¥¥W");//非単語構成文字

  - String word = result[x].trim();
    // 文字列のコピーを返します。
    // 先頭と最後の空白は省略されます。

# マップとソーティング(1)



- マップとは,
  {key1→Value1, Key2→Value2, … , Keyn→Valuen}
  といった写像（写像，対応関係）の意味合いです
- ここで，Keyは一意性が保証されます.
- すなわち，〈Keyi,Valuei〉という対の集合です.
- この〈Keyi,Valuei〉対をMap.Entryというクラスで表現しています.
  - （ここで，Mapはパッケージ名です．java.util.*をインポートしているので，java.util.Map.Entryクラスを意味します）

# マップとソーティング(2)



- Mapから，キーの集合を抽出するメソッドが
  keySet()です．
- 同様に，Map.Entryの集合を抽出するメソッドが
  entrySet()です．
- Map.Entryからキーを取り出すメソッドが
  getKey()で，
  バリューを取り出すメソッドが
  getValue()です．


iijima@ae.keio.ac.jp

# マップとソーティング(3)

- なので，マップの中身（一覧）を表示するには，以下のようなメソッドを書けばよいでしょう．

```
    public void print () {
        for ( Map.Entry<String,Integer> entry: entrySet () ) {
            printEntry ( entry.getKey (), entry.getValue () );
        }
    }
    private void printEntry ( String word, int freq ) {
        System.out.printf (
                "%-15s --> %3d ¥n", word, freq);
    }
```

- ついでに，棒グラフを描くには，以下を改行前に入れるといいですね．

```
for ( int i = 1; i <= freq; i++ ) { System.out.print ( '*' );}
```

# マップとソーティング(4)



- マップをソーティングするには...
  - まず，マップからEntryの集合を作り，
  - つぎに，それを元にArrayListを作ります
    - 集合という概念には本来順序がありませんから
      順序を保存するListを使うとよいでしょう
  - そして，ArrayListのソーティングには，
    Collections.sort（対象リスト，比較クラス）
    というメソッドが使えます
    - 但し，比較クラスを実装しなければなりません．
    - これには，Comparatorクラスのサブクラスとして，
      compare（x,y）という比較メソッドを定義します．

# マップとソーティング(5)



- そして，ArrayListのソーティングには，
  Collections.sort（対象リスト，比較クラス）
  というメソッドが使えます
  - 但し，比較クラスを実装しなければなりません．
  - これには，Comparatorクラスのサブクラスとして，
    compare（x,y)という比較メソッドを定義します．
  - compare(x,y)は，xとyが等しいとき0，xがyより大きいとき正の値，xがyより小さいとき負の値を返します
  - すなわち，通常の整数の比較ならx-yで表現でき，
    そのcompareメソッドを使ってソートすると，
    小さいものから大きいものへという順に（昇順に）
    ソートできます．

# マップとソーティング(6)



```
    public void sortedPrint() {
        Set<Map.Entry<String,Integer>> set = entrySet();
        List<Map.Entry<String,Integer>> list
            = new ArrayList<Map.Entry<String,Integer>>( set );
        Collections.sort( list, new EntryComparator() );
        sortedListPrint( list );
    }

    private void sortedListPrint(
                    List<Map.Entry<String,Integer>> list ) {
        for ( Map.Entry<String,Integer> entry: list ) {
            printEntry( entry.getKey(), entry.getValue() );
        }
    }
```

# マップとソーティング(7)

```
    // ソーティングのための全順序の比較関数
    // 大きいものから順に並べる降順にソートするため,
    // 本来の順序の逆順に設定する. すなわち
    // 第一引数が第二引数より小さい場合は正の整数,
    // 両方が等しい場合は0,
    // 第一引数が第二引数より大きい場合は負の整数を返す.
    private class EntryComparator
            implements Comparator<Map.Entry<String,Integer>> {
        public int compare( Map.Entry<String,Integer> p1,
                            Map.Entry<String,Integer> p2 ) {
            return( p2.getValue() - p1.getValue() );
        }
    }
```