日本ソフトウェア科学会チュートリアル

「Java言語の最新事情」

Java 5.0 の新機能

五十嵐　淳

京都大学　大学院情報学研究科

igarashi@kuis.kyoto-u.ac.jp

http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/

チュートリアルの内容

J2SE 5.0 (2004秋リリース)において導入された新言語機能紹介

- ジェネリクス
- ボクシングとアンボクシング
- イテレータ・配列のための for 文拡張
- 列挙(enum)型
- 可変長パラメータ

裏テーマ

- 「言語拡張　＝　イディオムのコンパイラサポート」
  - イディオムへの自動展開
  - さらなる安全性チェック
- Java言語拡張の限界を暴く（？）
  - 実装方式が及ぼす言語機能への影響

1.4から5.0へ

1.4

```
public static void main(String[] args) {
  List lst = new ArrayList();
  lst.add(new Integer(1));
  lst.add(new Integer(2));

  int sum = 0;
  for(Iterator i = lst.iterator(); i.hasNext(); ) {
    Integer elm = (Integer)i.next();
    sum += elm.intValue();
    System.out.println(
      "value = " + elm + ", sum = " + sum);
  }
}
```

5.0

新しい型の表記
(ジェネリクス)

```
public static void main(String[] args) {
  List<Integer> lst = new ArrayList<Integer>();
  lst.add(1);
  lst.add(2);

  int sum = 0;
  for(int elm : lst) {
    sum += elm;
    System.out.printf(
      "value = %d, sum = %d%n", elm, sum);
  }
}
```

消えたラッパークラス
（ボクシング）

消えたイテレータ
(拡張for構文)

Java に printf !?
（可変長引数）

目次

- Part I
  - 単純なジェネリクスの使い方
  - ボクシングとアンボクシング
  - 可変長パラメータ
  - イテレータのための for 文拡張
  - 列挙(enum)型
- Part II
  - ジェネリクス詳説

目次

- Part I
  - 単純なジェネリクスの使い方
  - ボクシングとアンボクシング
  - 可変長パラメータ
  - イテレータのための for 文拡張
  - 列挙(enum)型
- Part II
  - ジェネリクス詳説

コレクションを使ったプログラム

- 格納される要素の種類によらず単一の実装
  - **add(Object e)**: オブジェクトはなんでも格納できる
  - **Object get(int i)**: 取り出し後にキャストが必要
    - キャストが成功するかはプログラマの責任
    - 実行時オーバーヘッド

```
List lst = new ArrayList();
lst.add(new Integer(1));
lst.add(new Integer(2));

Integer i = (Integer)lst.get(1);
```

ジェネリックコレクションを使う

5.0

```
List<Integer> lst = new ArrayList<Integer>();
lst.add(new Integer(1));
lst.add(new Integer(2));

Integer i = lst.get(1); // キャスト不要！
```

- パラメトリック型： **List<参照型>**
- **List<Integer>** の持つメソッドのシグネチャ:
  - **void add(Integer o)**
  - **Integer get(int index)**

パラメータ化された クラス・インターフェース

- クラス **`ArrayList<E>`**
- インターフェース **`List<E>`**
  - 型パラメータ **`E`**
  - **`E`** を含むメソッド
    - **`void add(E o)`**
    - **`E get(int index)`**
    - など
- **`List<T>`**：**`E`**を参照型**`T`**に具体化したインターフェース
- c.f. **`Integer[]`**：要素型を**`Integer`**に具体化した配列型

パラメータ化されたメソッド

5.0

```
List<Integer> lst = …;

Integer[] iarray
  = lst.<Integer>toArray(new Integer[](10));
Number[] narray
  = lst.<Number>toArray(new Number[](10));
```

同じメソッドなのに
呼出し毎に
型が違う！

- **public <T> T[] toArray(T[] a)**
  - **<T>** ： メソッドの型パラメータ宣言
  - **lst.<Integer>toArray(...)** で **T** を具体化
    - **<...>** は省略できる（ことが多い） ⇒ メソッド型引数推論

単純なジェネリクスの使い方

- インスタンス生成
  - **new List<型>()**
- パラメトリック型
  - **List<型> x = ...;**
- メソッド呼び出し
  - **x.<型>m()**

後半で詳しく扱うトピック

- ジェネリックスの機能詳細
- 実装方法とそれに由来する不自然な機能制限
- 再利用性促進のためのワイルドカード型

目次

- Part I
  - 単純なジェネリクスの使い方
  - ボクシングとアンボクシング
  - 可変長パラメータ
  - イテレータのための for 文拡張
  - 列挙(enum)型
- Part II
  - ジェネリクス詳説

基本型とラッパクラス

- 基本型: **int**, **short**, **char**, ...
  - オブジェクトではない
- ラッパクラス: **Integer**, **Short**, **Character**, ...
  - 基本型とオブジェクトの世界の橋渡し
  - 直接足し算などはできない

```
List lst = new ArrayList();
// lst.add(1); はできない
lst.add(new Integer(1));

Integer i = ...; Integer j = ...;
// int k = i + j; もできない
int k = i.intValue() + j.intValue();
```

ボクシング(boxing)と アンボクシング(unboxing)

5.0

- boxing: 基本型⇒ラッパクラスへの自動変換

```
List lst = new ArrayList();
lst.add(1);  // boxing 1
```

- unboxing: ラッパクラス⇒基本型への自動変換

```
Integer i = ...; Integer j = ...;
int k = i + j;  // unboxing i and j
```

  - **if** や **switch** にも使用可

注意

- 対応する基本型・ラッパクラス間の変換に愚直に **new** や **intValue()** などを挿入するだけの実装
  - キャストの作法

```
Object o = ...;
int i = (Integer)o;   //   (int)o はエラー
Short s = (short)i;   //   (Short)i はエラー
```

  - 演算ごとのインスタンス生成

```
Integer i = new Integer(100);
i = i * 3 + 2;   //   オブジェクトふたつがゴミ
```

比較演算 ==

- 片方が基本型なら unboxing

```
Integer x = 256;
Integer y = x * 1;
int z = x;
//   (Q1) x==y ?    (Q2) y==z ?   (Q3) x==z ?
//        false          true          true
```

- 「小さい」数の boxing は **new** しない

```
Integer x = 4;
Integer y = x * 1;
int z = x;
//   (Q1) x==y ?    (Q2) y==z ?   (Q3) x==z ?
//        true           true          true
```

目次

- Part I
  - 単純なジェネリクスの使い方
  - ボクシングとアンボクシング
  - 可変長パラメータ
  - イテレータのための for 文拡張
  - 列挙(enum)型
- Part II
  - ジェネリクス詳説

使用例：PrintStream クラス

- **public PrintStream**
  printf(String format,
  Object ... args)

```
System.out.printf("%d + %d = %d", x, y, x+y);
System.out.printf("Error: %s", msg);
```

可変長パラメータ

- メソッド・コンストラクタのパラメータ宣言 “`T ... x`”
  - 最後の引数のみ可能
  - 本体内では `x` は配列 `T[]` として使える
- 可変長パラメータには配列を渡せる

```
static void m(int ... nums) {...}
m(new int[] {1,2,3}); // OK
```

- 配列型パラメータは可変長の引数をとれない

```
static void m(int[] nums) {...}
m(1,2,3); // エラー
```

実装

- コンパイラ内部・仮想機械レベルでは単なる配列型の引数
- よって、
  - オーバーロードできません

```
class Foo {
  void m(String... strs) {...}
  void m(String[] strs) {...}
}
```

  - オーバーライドしちゃいます（警告は出ます）

```
class Foo { void m(String... strs) {...} }
class Bar extends Foo { void m(String[] strs) {...} }
```

微妙すぎる気が…

- メソッド定義

```
static void m(Integer ... nums) {...}
```

- メソッド呼び出し三態
  - 長さ0の配列が渡されます
  - 警告が出ます

```
m();
```

```
m(null);
```

    - null を含んだ長さ1の配列なのか ((Integer)null)
    - 配列として null を渡したいのか ((Integer[])null)
      - メソッド定義側では null を受け取る可能性に留意

目次

- Part I
  - 単純なジェネリクスの使い方
  - ボクシングとアンボクシング
  - 可変長パラメータ
  - イテレータのための for 文拡張
  - 列挙(enum)型
- Part II
  - ジェネリクス詳説

コレクションに対するよくある処理

- イテレータを使って各要素に対して処理を行う

```
List lst = ...;
for(Iterator i = lst.iterator(); i.hasNext(); ) {
  String elem = (String)i.next();
   /*
    * elem に対する処理 （iやlstは使わない）
    */
}
```

拡張 for 構文

5.0

List<E> implements Iterable<E>

```
List<String> lst = ...;
for(String elem : lst) {  /*
for(Iterator i = lst.iterator(); i.hasNext(); ) {
  String elem = i.next(); */
   // elem に対する処理 （iやlstには触れない）
}
```

- 配列にも使用可
  - 添え字0から順にループ

```
String[] strs = ...;
for(String elem : strs) {
  ...
}
```

注意

- ループ中でイテレータは見えない
  - 要素の削除はできない
- 配列の添字はわからない
  - カウンタを自前で設ければもちろんその限りではない

目次

- Part I
  - 単純なジェネリクスの使い方
  - ボクシングとアンボクシング
  - 可変長パラメータ
  - イテレータのための for 文拡張
  - 列挙(enum)型
- Part II
  - ジェネリクス詳説

列挙型とは？

- それに属する定数の列挙による型の定義
  - 要素は宣言された定数のみ
  - 条件分岐

C言語

```
enum color { BLACK, BLUE, RED, PURPLE,
             GREEN, CYAN, MAGENTA, WHITE };
  enum color c = BLACK;
  switch (c) {
    case BLACK: ...
    case BLUE: ...
    ...  }
```

  - Cでは単なる整数型の別名

C言語

```
int i = CYAN * WHITE;
enum color c = 100;
```

Java 列挙型の基本(1)

- **enum** キーワードによる宣言

```
public enum Color {
  BLACK, BLUE, RED, PURPLE,
  GREEN, CYAN, MAGENTA, WHITE };
```

- **switch**文による条件分岐

```
static Color reverse(Color c) {
   switch (c) {
     case BLACK: return WHITE;
     case BLUE: return MAGENTA;
     ...  }
}
```

Java列挙型の基本(2)

- 全定数の入った配列を返す **values** メソッド

```
for (Color c : Color.values()) {
   System.out.println(c);
}
```

- **int ≠ Color**

```
int i = Color.BLACK  // コンパイルエラー
Color c = 2;         // コンパイルエラー
```

コンストラクタと定数共通のメソッド

```
public enum Color {
   BLACK("black"), BLUE("blue"), RED("red"),
   PURPLE("purple"), GREEN("green"), CYAN("cyan"),
   MAGENTA("magenta"), WHITE("white");

   private final String name;
   Color(String s) { name=s; }  // コンストラクタ
   public String toString() { return name; } // メソッド
}

System.out.println(Color.BLACK); // "black"
```

- コンストラクタ呼び出しを伴った定数宣言
- **switch**文以外では"型名.定数名"で参照

定数毎の異なる振舞い

- メソッド定義を伴った定数宣言
  - 同名の (abstract) メソッド宣言が必要

```
public enum Color {
  BLACK(“black”) {
    Color reverse() { return Color.WHITE;}
  } ,
  BLUE(“blue”) {
    Color reverse() { return Color.MAGENTA; }
  } ,
  ...
  abstract Color reverse();
}

System.out.println(Color.BLACK.reverse());
        // “white”
```

実装の種明かし

Joshua Bloch著
“Effective Java”

- いわゆる「タイプセーフenumパターン」に変換
  - 定数はインスタンスで static フィールドに格納
  - 宣言された定数以外のインスタンスの生成は禁止
    - コンストラクタは private

```
public class Color extends java.lang.Enum<Color> {
  public static final Color BLACK =
    new Color(“black”) {  // 匿名クラス
      Color reverse() { return Color.WHITE;}
    };
  public static final Color BLUE = ...;
}
```

`java.lang.Enum` クラス

- 列挙型一般に関するメソッド群を提供
  - **`equals`**, **`hashCode`**, **`clone`**
  - **`final String name()`**: 定数名を文字列として返す
    - デフォルトの `toString()` の振る舞いと同じ
  - **`final int ordinal()`**: 定数と一対一対応する整数値を返す

java.util.EnumMapクラス

- 列挙型の値をキーとするテーブル
  - 配列添字に列挙型定数が使えないことへの補填
    - a[BLACK] みたいなコードって結構便利
  - 列挙型のサイズが既知なので効率的に実装できる

```
EnumMap<Color,String> map =
  new EnumMap<Color,String>(Color.class);

map.put(Color.BLACK, “Foo”);

for (Color c : Color.values()) {
  String s = map.get(c); ...
}
```

java.util.EnumSetクラス

- 列挙型の値の集合
  - 固定長ビットマップによる効率的な表現
  - インスタンス生成はファクトリーメソッドのみ
  - 普通の Set, イテレータとして使用できる

```
EnumSet<Color> mono =
  EnumSet.of(Color.BLACK, Color.WHITE);
EnumSet<Color> notMono = Enumset.complementOf(mono);
EnumSet<Color> notMono =
  Enumset.range(Color.BLUE, Color.MAGENTA);

for(Color c: notMono) { ... }
```

Part I まとめ

- ボクシング、可変長パラメータ、for、列挙型
  - よく使うイディオムを簡潔に記述
  - コンパイラによる展開
    - 凡ミスの防止
  - 展開後を知らないとわからない落とし穴も

Part II 目次

- ジェネリックスの機能詳説
- 実装方式
- ワイルドカード

Part II 目次

- ジェネリックス機能詳説
  - ジェネリッククラスを定義する
  - 型パラメータと上限
- 実装方式
- ワイルドカード

復習：ジェネリック(総称)クラスとは

- 総称クラス ＝　型についてパラメータ化されたクラス
  - c.f.　手続き ＝ 値・変数についてパラメータ化された文
- パラメトリック型 **C<T>**
  - 総称クラス・インターフェースに型の実引数を与える

簡単な総称クラス定義

- 要素型をパラメータとする linked list クラス

```
class List<E> {
  private E head;
  private List<E> next;

  public List<E>(E h, List<E> n) {
    head=h; next=n;
  }
}
class SortedList<E> extends List<E> { ... }
```

  - クラス名直後の型パラメータ **E** の宣言
  - **E** はあたかも普通の型であるかの如く使える
    - 制限あり(後述)

パラメトリック型

- （概念的には）型引数に関して特化したクラス
  - 例： **List<String>** は以下のクラスと大体同じ

```
class List_String {
  private String head;
  private List_String next;

  public List_String(String h, List_String n) {
    head=h; next=n;
  }
}
```

- 多引数総称クラス・インターフェース

```
interface Map<K,V> {
  public void clear();
  public Set<K> keySet();
  public V put(K key, V value);
  ...
}
```

- 入れ子パラメトリック型

```
List<String> lst = new List<String>(“Foo”, null);
List<List<String>> llst =
  new List<List<String>>(lst,
    new List<List<String>>(lst, null));
```

map メソッドを定義してみる

- リストの各要素に関数(**Fun<X,Y>**)を作用
  - 解答例1

```
interface Fun<X,Y> { pubilc Y apply(X x); }
class List<E> { ...
  public List<E> map(Fun<E,E> f) {
    if (next==null) {
      return new List<E>(f.apply(head),null));
    } else {
      return new List<E>(f.apply(head),map(f));
    }
  }
}
```

あれ、作用される関数が要素と同じ
型を返さなきゃいけないのか。。。

- 解答例2「返り値の型もパラメータ化してみよう」

```
class List<E,A> { ...
  public List<A,?> map(Fun<E,A> f) { ... }
}
```

  - 返り値のリスト型の第二引数は何にすべき？
  - インスタンス生成時に返り値の型が固定されてしまう

総称メソッド

- (型について)パラメータ化されたメソッド
  - 呼び出し毎に別の型でパラメータを具体化できる

メソッドに対する
型パラメータ宣言

```
class List<E> { ...
  public <A> List<A> map(Fun<E,A> f) { ... }
}

Fun<Integer,String> f1 = ...;
Fun<Integer,Number> f2 = ...;
lst.<String>map(f1);
lst.<Number>map(f2);
```

型パラメータと上限

- 型パラメータの動く範囲の指定
  - 具体化に使う型が制限される
  - 型パラメータの値は上限型として使用できる
  - 逆は不可
  - 省略時は **extends Object**

総称メソッドにも使えます！

型パラメータは基本型では具体化できない

```
class NumList<X extends Number> {
   X head;  NumList<X> tail;
   Byte byteHead() { returnhead.byteValue(); }
   // void illegal(Number n) { head=n; }
}
NumList<Integer> il = …;
NumList<String> sl = …;  // エラー！
```

再帰的な上限

- 自分自身と比較可能な型を動くパラメータ
  - java.util.Collectionsより

```
interface Comparable<T> {
 public int compareTo(T o); // Tと比較するメソッド
}
class Collections {
   public static <T extends Comparable<T>>
    void sort(List<T> list) { ...
     int j = list.get(i).compareTo(list.get(i+1));
    }
}
class Byte implements Comparable<Byte> {...}

 List<Byte> blist = ...;
 Collections.<Byte>sort(blist);
```

複数の上限指定

- 上限は(二つ目以降がインターフェースなら)複数あってもよい

```
class MyList<X extends Comparable<X> & Cloneable>
          extends List<X> {
   ...
}
```

Part II 目次

- ジェネリックス機能詳説
- 実装方式
- ワイルドカード

Listクラスをコンパイルしてみる

- コンパイル前

```
class List<E> {
  private E head;
  private List<E> next;

  public List<E>(E h, List<E> n) {head=h; next=n;}
  public E get_head() { return head; }
  public void set_head(E h) { head=h; }
}

List<String> lst = new List<String>(“Foo”, null);
String s = lst.get_head();
```

Listクラスをコンパイルしてみる

- コンパイル後

```
class List {
  private Object head;
  private List next;

  public List(Object h, List n) {head=h; next=n;}
  public Object get_head() { return head; }
  public void set_head(Object h) { head=h; }
}

List lst = new List("Foo", null);
String s = (String)lst.get_head();
```

```
class List<E> {
  private /* E */ Object head;
  private List<E> next;

  public List(/* E */ Object h, List<X> n)
      {head=h; next=n;}
  public /* E */ Object get_head() { return head; }
  public void set_head(/* E */ Object h) { head=h; }
}

List<X> lst = new List<String>("Foo", null);
String s = (String)lst.get_head();
```

- コンパイルの方法（通称 erasure 変換）
  - 1総称クラス⇒1クラスファイル
  - <>の類は消去
  - 単独の型変数はその（一番目に書かれた）上限に置換
  - 型が合わなくなるところには適当なキャストを挿入

特化したサブクラスとブリッジメソッド

- 型パラメータを固定して継承&メソッド上書き

```
class SList extends List<String> {
  // private String head;
  // private List<String> next;

  public void set_head(String h) { head = h+h; }
}
```

- オーバーライドするためだけの「ブリッジ」メソッド挿入

```
class SList extends List<String> {
  public void set_head(String h) { head = h+h; }
  public void set_head(Object h) { // ブリッジメソッド
    set_head((String)h);
  }
}
```

結局何がご利益なのか

- 安全性
  - 詳細な型情報(パラメトリック型)による型検査
  - コンパイラが挿入したキャストは「必ず」成功する
    [Igarashi et al. TOPLAS2001]

✗ 実行効率面には特にない

実装方式が及ぼす妙な影響

- 型引数情報が実行時にはないので…
  - **`new X()`** とかはダメ
    - そもそもコンストラクタは継承されないから、という噂も
  - **`<X> void m(X x)`** と **`void m(Object x)`** はオーバーロードできない
  - 型引数情報がないと判定できないキャストは不可

```
Object o = new List<Integer>(1,null);
List<String> lst = (List<String>)o; // 失敗すべき
```

```
Object o = new List(new Integer(1),null);
List lst = (List)o;  // 成功！？
```

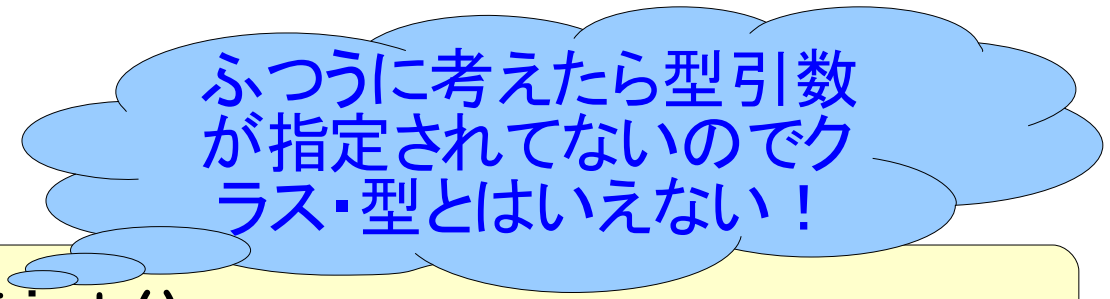
考えられる実装方式比較

- Java 5.0方式： erasure変換
- C++方式： １総称クラス(テンプレート)⇒nクラス
  - コード量の増加
- C#方式： Java 5.0とC++方式のハイブリッド
  - 基本型が引数の場合だけ具体化した定義を生成
- その他： 実行時まで型引数情報をとっておく
  - reflection を使う[LM]
  - 仮想機械を改造する
  - 型情報に相当するインターフェースなどを生成するコンパイル技法[NextGen]

Java 5.0方式の利点

旧バージョンとの互換性

- ライブラリのバイトコードはほとんど変わっていない
  - パラメトリック型の情報が注釈として付加
- 昔のクライアントコードも（一応）そのままコンパイルできる
  - 原型(raw type)機能




```
List lst = new ArrayList();
...
String s = (String)lst.get(1);
```

Part II 目次

- ジェネリックス機能詳説
- 実装方式
- ワイルドカード
  - 動機付け
  - ワイルドカード型の概略
  - 応用例

パラメトリック型と部分型関係

- 継承関係にもとづく部分型関係
  - 任意の型 **T** に対し **ArrayList<T>** は **List<T>** の部分型

```
List<Integer> lst = new ArrayList<Integer>();
List<String> lst2 = new ArrayList<String>();
```

- 部分型を与える他の可能性？
  - **List<Integer>** は **List<Number>** の部分型？
    - いわゆる 共変(covariant)部分型規則
  - 逆は？

共変部分型が許されるなら…

- メソッド再利用性の向上

```
static double sum(List<Number> list) {
  double sum = 0.0;
  for (Number n : list) sum += n.doubleValue();
  return sum;
}

List<Integer> l = ...;
double sum = sum(l);
```

  - ただし、総称メソッドを使うという手もある

共変部分型は安全ではない

- 安全ではない例

```
List<Integer> lst = new ArrayList<Integer>(10);
List<Object> badList = lst;
badList.add("Foo");
int i = lst.get(0)  // !?
```

- Java の配列では…

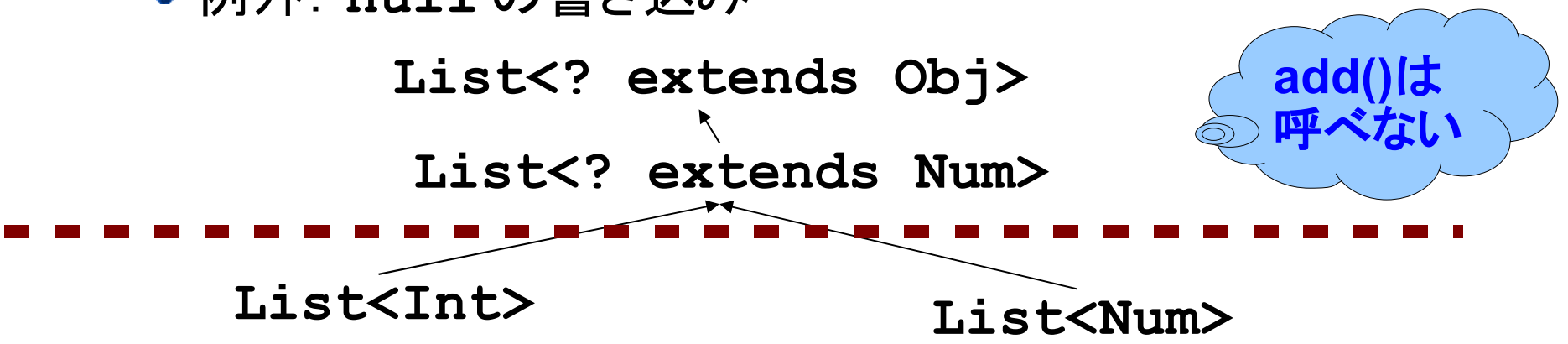
```
Integer[] iarray = new Integer[]();
Object[] badArray = iarray;
badArray[0] = "Foo";  // throws an Exception!
int i = lst[0]
```

教訓

- 共変部分型と「書き込み」をするメソッドは組み合わせるとまずい
  - 一般的には、共変部分型で変化する型パラメータが引数型に現れるようなメソッド

ワイルドカード型へ

- **List<Number>**: 共変ではない
  - **add()** メソッドなどを呼んでもよい
- **List<? extends Number>**: 共変ワイルドカード型
  - **List<Number>**, **List<Integer>**, **List<Float>** などを部分型とする
  - **add()** などの「危険」なメソッドは呼べない
    - 例外: **null** の書き込み



使用例

```
static double sum(List<? extends Number> list) {
  double sum = 0.0;
  for (Number n : list) sum += n.doubleValue();
  return sum;
}

List<Integer> l = ...;
double sum = sum(l);
```

- 内部では読み出し操作である **hasNext()**, **next()** しか呼んでいない

安全性の確保

- 共変ワイルドカードを使わないと

```
List<Integer> lst = new ArrayList<Integer>(10);
List<Object> badList = lst; // コンパイルエラー！
```

- 使ったとしても

```
List<Integer> lst = new ArrayList<Integer>(10);
List<? extends Object> badList = lst;
badList.add("Foo"); // コンパイルエラー！
```

直感的な意味

- **`List<? extends Number>`**
  - **`Number`** の部分型の「何か」を要素とするリスト
  - 要素型が何かはっきりしない
    - ので、書き込んでいいものがわからない　⇒　書き込み禁止
    - が、読み出したものが **`Number`** であることはわかる

総称メソッド vs ワイルドカード

- **sum** メソッドは総称メソッドを使っても書ける

```
static <X extends Number> double sum(List<X> list) {
   ...
}

List<Integer> l = ...;
double sum = <Integer>sum(l);
```

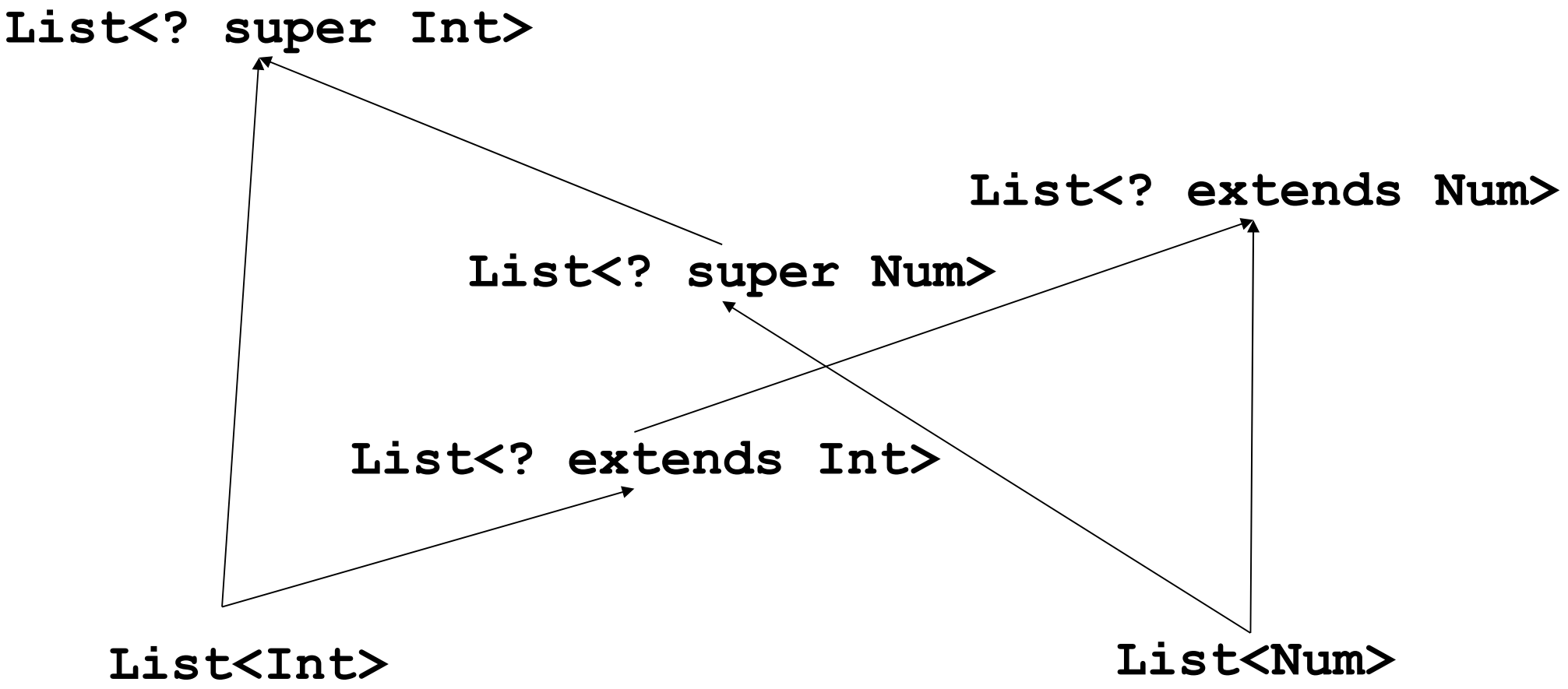
  - 型パラメータ **X** が引数で一箇所出現するだけ
    - ワイルドカードで表現可能

反変ワイルドカード

- **List<? super Number>**
  - **Number** のスーパータイプの**List**を部分型としてもつ
    - 例：**List<Number>**, **List<Object>**など
  - 書き込みOK、(**Object** として以外の)読み出し不可

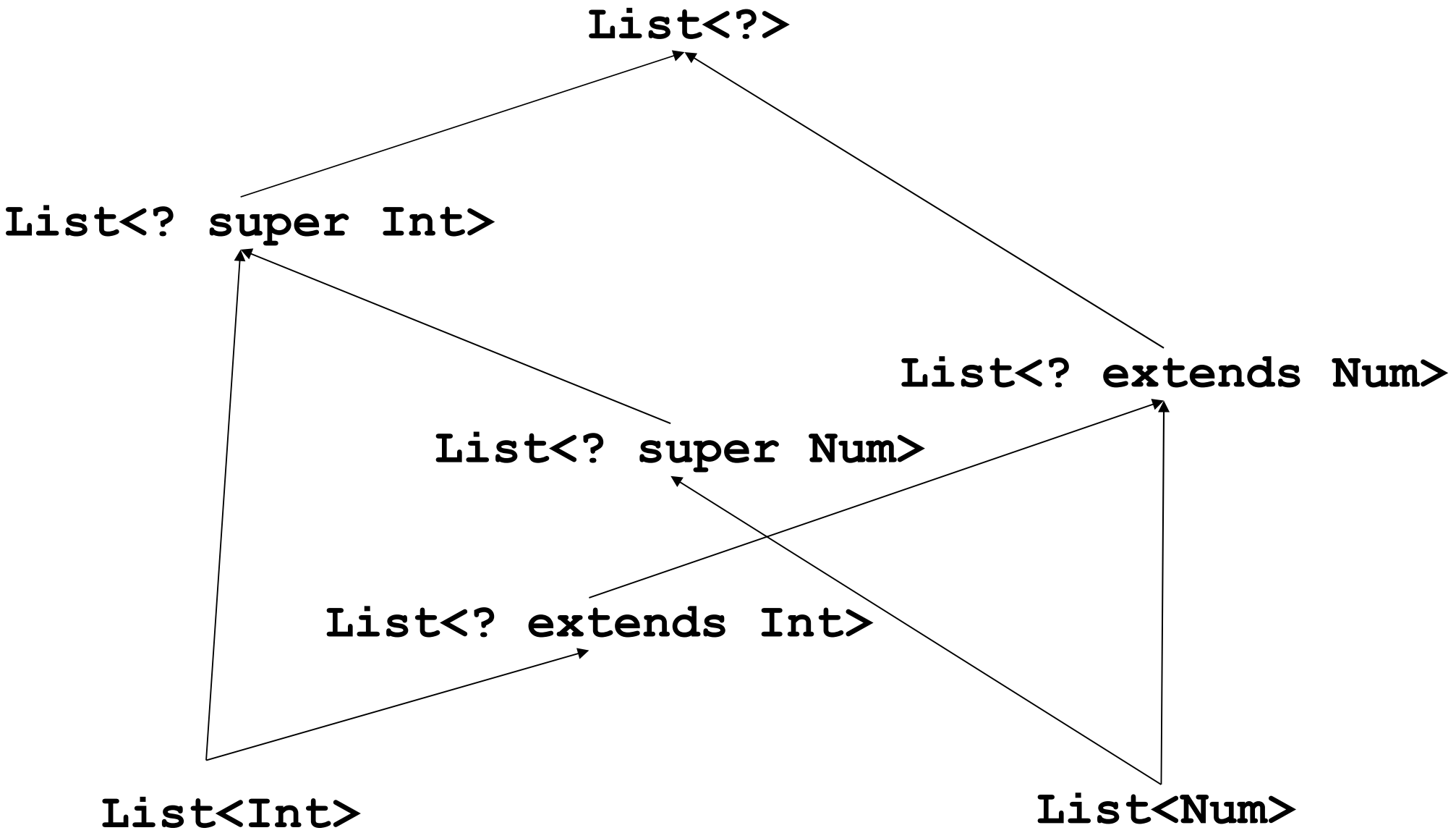
```
static double addTenTo(List<? super Number> list) {
  list.add(0,new Integer(10));
  return;
}

List<Object> l = ...;
addTenTo(l);
```


List<? super Int>
List<? extends Num>
List<? super Num>
List<? extends Int>
List<Int>
List<Num>

制限なしワイルドカード

- `List<?>`
  - `List<? extends Object>` の略記
    - 型パラメータの上限が `Object` であることを反映
  - 全ての `List<...>` のスーパータイプ


List<?>
List<? super Int>
List<? extends Num>
List<? super Num>
List<? extends Int>
List<Int>
List<Num>

やや高度な例

- 自分自身(のスーパータイプ)と比較可能な要素のリストのソート
  - 再帰的上限の不便さを解消

```
class Collections { ...
  static <T extends Comparable<? super T>>
                  void sort(List<T> list) { ...
} }
class MyDate extends java.util.Date {
  int compareTo(Date x) {...} }

List<Date> lst1 = ...;
Collections.sort<Date>(lst1);
ArrayList<Date> lst2 = ...;
Collections.sort<Date>(lst2);
List<MyDate> lst3 = ...;
Collections.sort<MyDate>(lst3);
```

```
MyDate
 <: Date
 <: Comparable<Date>
 <: Comparable<? super Date>

MyDate
 </: Comparable<MyDate>
```

ワイルドカードの「キャプチャ」

? に隠れた「未知の型引数」もれっきとした(無名)型

- コンパイラ内部では “capture of ?” と呼ばれる
  - プログラマは書けない

```
List<? extends Object> badList = ...;
badList.add("Foo");
// List<capture of ? extends java.lang.Object>
// に add がないとかなんとか
```

- 総称メソッドの型引数になることもある！

```
static <T> List<T> synchronizedList(List<T> list){...}
    // java.util.Collections より
List<?> lst = ...;
List<?> slist = synchronizedList(lst);
          // <capture of ?>synchronizedList(lst) と思える
```

ジェネリクスは使いこなせるのか？

- ライブラリを使うだけならなんとかなると思う
- 自分で総称クラスを設計するのはたしかに難しい
  - クライアントがどう使うのか充分な考察が必要
- ジェネリック化する refactoring 技法・ツールなどの研究に期待

ワイルドカードの典型的使用パターン

- 引数型が共変型(例えば **`List<? extends E>`**)
  - **`E`** の部分型のリストなら何でも渡せる
  - メソッド内部では(おそらく)その引数は読み出し専用で使われている
- 引数型が反変型(例えば **`List<? super E>`**)
  - **`E`** のスーパータイプのリストなら何でも渡せる
  - メソッド内部では(おそらく)その引数に要素が書き込まれる
- 返値型が共変型
  - 書き込みはできない

まとめ

まとめ

- Java 5.0 新機能の概観
  - ジェネリクス
  - boxing, 可変長引数、拡張 for, enum
  - このチュートリアルでは扱わなかったトピック
    - static インポート
    - アノテーション

Java の新機能の意義と問題点

- イディオムの自動化
  - よくあるパターンを簡潔に書ける
- コンパイラによるイディオム誤使用の検査
  - 今まで以上の安全性保証
    - コンパイラが挿入するダウンキャストは失敗しない
- 互換性にこだわりすぎ？
  - Java 1.4≦への変換による実装
  - 変換手法を知らずに深く理解することが不可能
    - 「適切な抽象化の手段を提供する」という言語の役割？

参考文献

- G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ: Extending the Java Programming Language with type parameters. http://homepages.inf.ed.ac.uk/wadler/gj/Documents/
- A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Core Calculus for Java and GJ. ACM TOPLAS, 2001.
- A. Igarashi and M. Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. ACM TOPLAS, 2006.
- M. Torgersen et al. Adding Wildcards to the Java Programming Language. In Proc. Of ACM SAC2004. http://bracha.org/selected-pubs.html

- [NextGen] Robert Cartwright and Guy L. Steele Jr. Compatibe Genericity with Run-Time Types for Java. In Proc. OOPSLA'98
- [LM] Mirko Viroli and Antonio Natali. Parametric Polymorphism in Java: An Approach to Translation based on Reflective Features. In Proc. OOPSLA2000
- 柴田 芳樹. Java 2 Standard Edition 5.0 Tiger―拡張された言語仕様について. ピアソンエデュケーション.
- Java 5.0 Tiger (単行本)
- Brett McLaughlin, David Flanagan, 菅野 良二 (訳). Java 5.0 Tiger. オライリージャパン