Type Systems for Object-Oriented Languages

APLAS2005 Tutorial

Atsushi Igarashi
(Kyoto University)
igarashi@kuis.kyoto-u.ac.jp
http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/

Type Systems for an Object-Oriented Language

APLAS2005 Tutorial

Atsushi Igarashi
(Kyoto University)
igarashi@kuis.kyoto-u.ac.jp
http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/

What is This Tutorial About?

- Evolution of Java's type system
  - Simple type system before Java 5.0
  - Generics and Parametric Types
  - Wildcards
- How types contribute safety and reusability

- Not about:
  - Comparison of different languages and their type systems

Overview

- Part I: What's Java?
  - Model of (untyped) Java objects
  - Simple type system for Java (～JDK1.4)
    - Class names as types
    - Inheritance-based subtyping
- Part II: Generics for more reusable classes
  - Parametric types
- Part III: Wildcards
  - Variance-based subtyping for parametric types

} part of JDK5.0

Part I

What's Java?

Overview of Part I

- What are Java objects?
- Classes and inheritance for reusing implementation
- What is a Java type system for?
- Simple Java type system
  - Class names as types
  - Subtyping based on inheritance

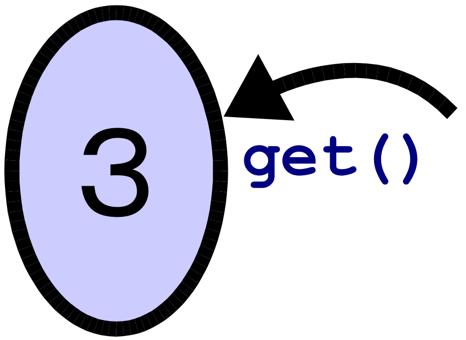
What are Objects in Java?

Just a particular kind of data structure consisting of …

- Internal state, called fields
- A set of procedures, called methods
  - Primitive operations:
    - Object creation
    - Reading field values / writing to fields
    - Invocation of a method of another object, or the object itself
- …

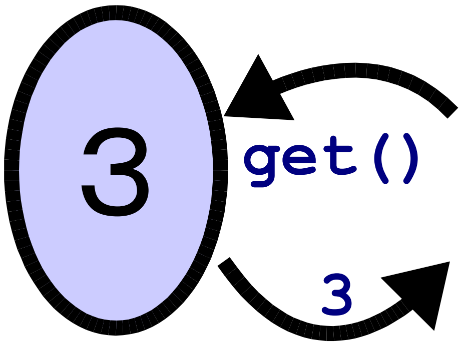
Example: （One dim.） point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



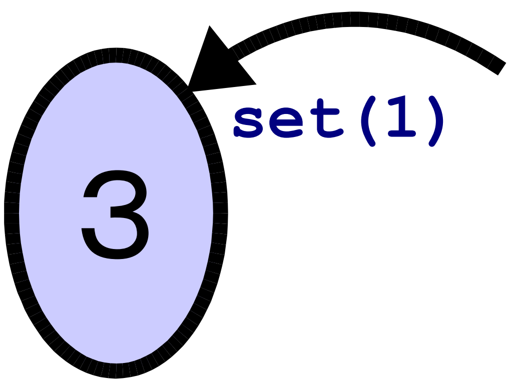
Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



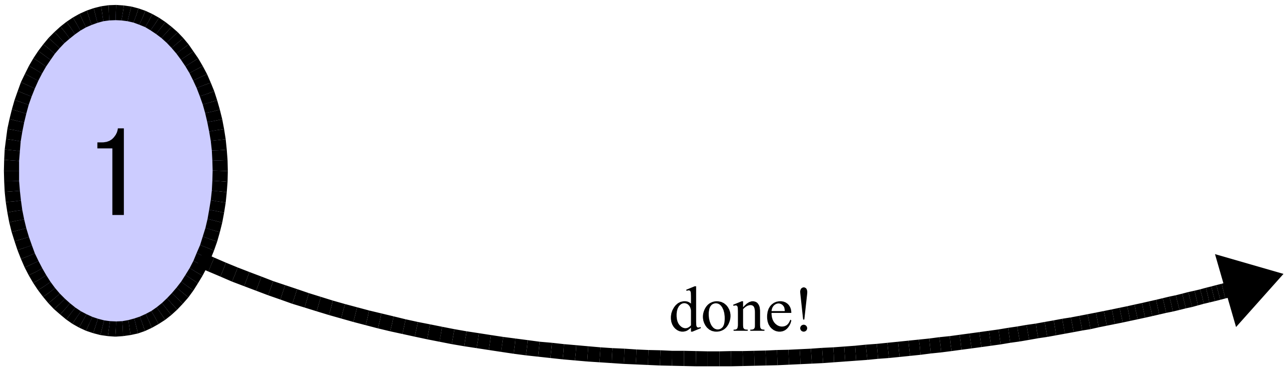
Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



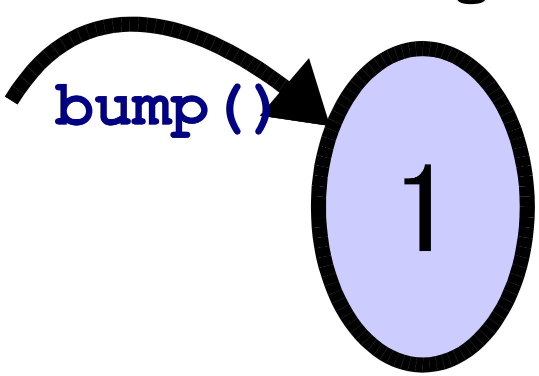
Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



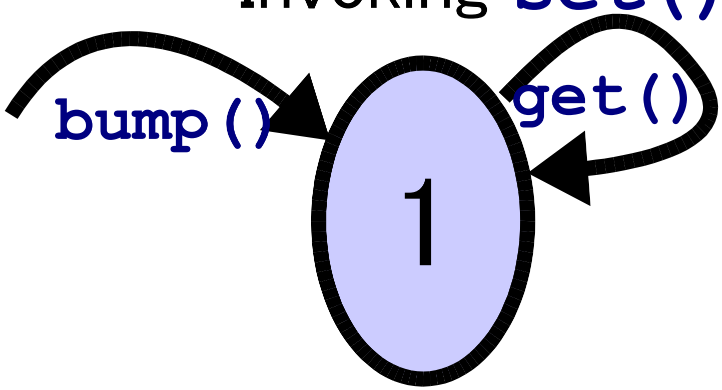
Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



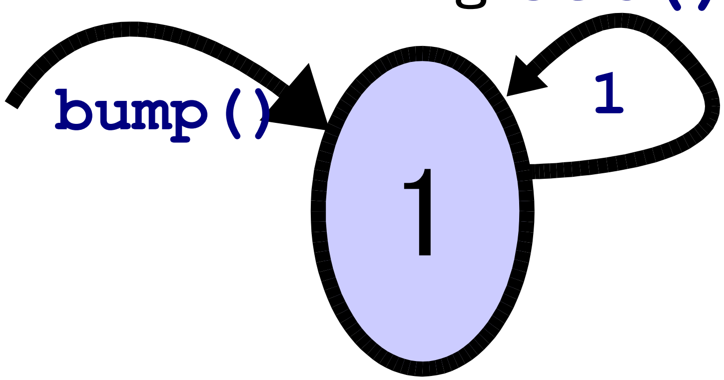
Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



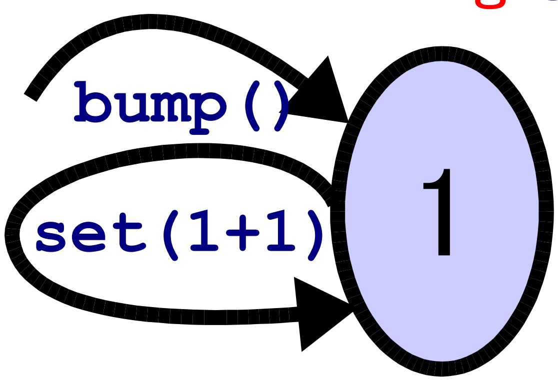
Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



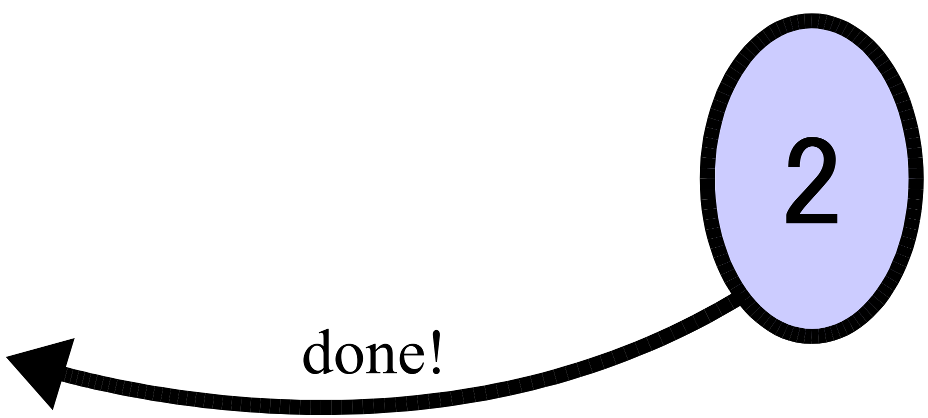
Example: （One dim.） point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



Example: (One dim.) point object

- State: coordinate value **`x`**
- Method **`get()`**: returns the value of **`x`**
- Method **`set(y)`**: sets **`x`** to **`y`**
- Method **`bump()`**: increments **`x`** by one, by
  - Invoking **`get()`** on self,
  - Adding one to the value
  - Invoking **`set()`** on self



Classes as Factories of Objects

- Description of common structure of objects
  - Field declarations
  - Method definitions
- Code to initialize objects
  - Constructor(s)
- Objects are instantiated from a class `C` by an expression `new C(...)`

Example: Class for Point Objects

```
class Point {
  field x;
  Point(initx) { x = initx; } // constructor def.
  method get() { return x; }
  method set(newx) { x = newx; return; }
  method bump() { this.set(this.get()+1); return; }
  method copy_x(p) { this.set(p.get()); return; }
}
print(new Point(5).get());  // 5
var p = new Point(3);
p.bump(); print(p.get()); // 4
var p = new Point(0);
p.copy_x(new Point(2)); print(p.get()); // 2
```

Reusing Object Implementation by Inheritance

New class definition by “extension”

- Inheriting all definitions from another class
- Adding new fields and methods, and
- Overriding (some of) inherited methods

  - Late binding of “**`this`**”
    - The meaning of **`this`** in methods is determined
      - only when an object is instantiated
      - not when a class is defined

Example: Colored Points

subclass superclass

```
class ColorPoint extends Point {
  field col;  // additional field
  ColorPoint(init_x){ x = init_x; col = Blue; }
  // method get() { return x; }
  // method bump() { this.set(this.get()+1); return; }
  // additional/overriding methods
  method get_col() { return col; }
  method set_col(new_col){ col = new_col; return; }
  method set(new_x){
    x=new_x; this.set_col(Red); return; }
}
var p = new ColorPoint(3);  p.bump(); // calls set()
print(p.get())       // 4
print(p.get_col()) // Red
```

Run-Time Test on an Object's Class

Java is equipped with constructs to check the class of an object

- **`e instanceOf C`**
  - returns **`true`** when **`e`** evaluates to an instance of **`C`** (or its subclass)
  - returns **`false`** otherwise
- **`(C)e`**
  - does nothing when **`e`** evaluates to an instance of **`C`** (or its subclass)
  - throws **`ClassCastException`** otherwise

What are Objects in Java?

Just a particular kind of data structure consisting of …

- Internal state, called fields
- A set of procedures, called methods
- Name of a class from which it is instantiated
  - Sometimes called an object's run-time type

What is a Type System?

Mechanism to detect possibility of certain kinds of errors before a program runs by analyzing its abstract syntax tree

- Types:
  - Approximation of “what a program (fragment) does” with enough information to detect the errors
- Typing rules:
  - Rules to compute such approximation from a given program fragment
- Type soundness property:
  - “Typing rules give correct approximation of the behavior of a program”

What We Are To Detect and Not To

- Errors to be detected:
  - Invocation of non-existing methods
    - **`NoSuchMethodError`**, …
- Errors not to be detected:
  - Division by zero
    - **`ArithmeticException`**
  - Failure of run-time type tests
    - **`ClassCastException`**
  - …

Type Information Required to Prevent **NoSuchMethodError**

"Interface" information of objects

- The names of methods that an object owns
- What each method takes as arguments
- What each method returns

e.g.,

- Interface of **Point** objects
  {get: ()→int, set: (int)→void, bump: ()→void, ...}

- Interface of ColorPoint objects
  {get: ()→int, set: (int)→void, bump: ()→void,
  get_col: ()→int, set_col: (col)→void, ...}

Java's Typing Principle (1) Class Names as Types

Class name as a concice representation for interface information

- Objects from the same class have the same interface
- Method names are manifest in a class definition
- Argument and return types are given by programmers

Point with Type Annotations

```
class Point {
  int x;
  Point(int initx) { x = initx; }
  int get() { return x; }
  void set(int newx) { x = newx; return; }
  void bump() { this.set(this.get()+1); return; }
  void copy_x(Point p){ this.set(p.get()); return;}
}
```

- Point is a recursively defined interface:
  Point =
  {get: ()→int, set: (int)→void,
  bump: ()→void, copy_x: Point→void}

Inheritance Requires Substitutability

- `ColorPoint` must be substitutable for `Point`, because:
  - `bump()` is typechecked under the assumption that `this` is of type `Point` (once and for all)
  - At run-time, `this` can be either `Point` or `ColorPoint`

- Subtyping relation: `C` <: `D`
  - "C is substitutable `D`"
  - Subsumption typing rule:
    - If `e` is of type `C`, then `e` is also of type `D`

Q: When is one type a subtype of another?

Java's Typing Principle (2)
Inheritance as Subtyping

`C` <: `D` iff class `C` (indirectly) **`extends`** class `D`

- The interface of C always includes that of `D`
    - `D` inherits all methods from `C`
- One subtlety: method overriding
    - Java's rule:
        - The argument/return types of an overriding method must be the same as the overridden
- Subtyping could be defined independently of inheritance
    - c.f. Objective Caml

Some Typing Rules

- Object instantiation: **`new C(e)`**
  - If **`e`**'s type is a subtype of the constructor argument type,
  - Then **`new C(e)`** is of type `C`
- Method invocation expression: **`e1.m(e2)`**
  - If **`e1`**'s type includes m:(T1) → T2 and
    `e2`'s type is a subtype of T1,
  - Then **`e1.m(e2)`** is of type T2
- Method definition in `C`: **`T m(T' x){`** body **`}`**
  - Typecheck the body under the assumption
    - **`x`** is of type **`T'`** and **`this`** is of type `C`

Type Soundness Property

"If typechecking succeeds,
`NoSuchMethodError` cannot be thrown"

- Subject Reduction Property:
  - The type of an expression is preserved by one step of execution
- Progress Property:
  - If typechecking succeeds, **`NoSuchMethodError`** cannot be immediately thrown

- Several formal proofs for various subsets of Java have been given in the literature
[DrossopoulouEisenbach97, IgarashiPierceWadler99, etc.]

Typing Rule for Typecasts **`(C)e`**

- The whole expression can be given type **`C`**, whatever the type of **`e`** is
  - In Java, actually, **`e`**'s type must be either a subtype or supertype of **`C`** (unless **`C`** is an interface type)
    - Otherwise, typecasts will always fail

Type Soundness Theorem, Revised

"If typechecking succeeds,
`NoSuchMethodError` cannot be thrown,
but **`ClassCastException`** may be thrown"

- So, the (ab)use of typecasts decreases program reliability

Summary of Part I

- Informal model of untyped Java objects
  - Object ＝ fields (internal state) + methods + class name
  - Classes and implementation reuse by inheritance
- Simple type system
  - To prevent nonexistent fields/methods from being accessed
    - Class name as a representation of type information
    - Inheritance requires substitutability (subtyping) to be taken into account
    - Inheritance as subtyping

Part II

From Java to Generic Java

Overview of Part II

- Programming generic data structure by using a Java idiom
- Problems in the Java idiom
- Generics
- Implementation of Java Generics
- Other issues in Java Generics

Programming Generic Data Structrue in Java

- Class for list structure
  - Methods: **`length()`**, **`append()`**, **`map()`**
- Various element types
  - List of strings, list of integers, ...

Definitions Specialized for Specific Elements …

```
class StrList {
  String head;  StrList tail;
  StrList(String h, StrList t) { head=h; tail=t; }
  int length() {
    if (tail==null) return 1;
    else return tail.length() + 1;
  }
  ...
}
StrList ss=new StrList("a",new StrList("b",null));
int i = ss.length();
String s = ss.head;
```

… Are Not Easy to Maintain

A number of very similar class definitions

- Code modification is cumbersome, or even error-prone

Java's "generic idiom"

Unifies specialized definitions into one class

- Use of **Object**, a top type, as an element type

```
class List {
  Object head;  List tail;
  List(Object h, List t) { head=h; tail=t; }
  int length() { ... }
  ...
}
List ss = new List("a",new List("b",null));
List is = new List(i1, new List(i2, null));
  // subsumption
int i = ss.length() + is.length();
  // So far, so good, ...
```

Oops!

```
String s = ss.head;

List.java:xx:incompatible types
found:    java.lang.Object
required:java.lang.String
    String s = ss.head;
                 ^
1 error
```

Why?

- The declared type of **`head`** is **`Object`**
- Assignment of an **`Object`** to a **`String`** variable not allowed
  - (The opposite direction is OK)
- Loss of type information in list construction

→ Workaround by typecasts

```
String s = (String)ss.head;
```

- They should succeed (if you are careful enough), but
  - The type system cannot guarantee their successes
  - The run-time system incurs some overhead

Comparisons of the Two Approaches

- Element-specific classes
  - Low reusability
    - Mostly duplicated code
  - No worry about `ClassCastException`
- Java idiom
  - High reusability
    - One definition fits all
  - Reduced safety / efficiency
    - Due to typecasts

Any way to take best of both worlds?

Introduction of Generic Classes

Classes in which some type information is abstracted by type parameters

- cf. C++ templates, ML polymorphic functions
- Viewed as a function from types to specialized classes
  - **`new List<String>(...)`**
- Type parameters are used as types in their scopes

```
class List<X> {
  X head; ...
}

...   new List<String>("a",
        new List<String>("b",null)) ...
```

Parametric Types

Generic class name + actual type arguments, such as **List<String>**

- Representing the interface of the class in which **X** is instantiated with **String**
  - The field **head** of **List<String>** is of **String**
- Class names by themselves are not types

```
class List<X> {
  X head;  List<X> tail;
  List(X h, List<X> t) { head=h; tail=t; }
  int length() { ... }
  ...
}
List<String> ss= new List<String>("a",...);
String s = ss.head;  // OK!
```

More Generally, ...

- Generic classes with multiple type parameters

```
class Pair<X,Y> {
  X fst;  Y snd;   ...
}
Pair<String,Integer> p = ...;
Integer i = p.snd;
```

- Nested parametric types

```
List<List<String>> ss=...;
int i = ss.length()+ss.head.length()
          +ss.head.head.length();
List<Pair<String,Integer>> ps=...;
```

Other Features of Java Generics (1): Parameterized Methods

- Implementing the map function for lists

```
class Fun<X,Y> { /* functions from X to Y */}
class List<X> { ...
  <Y> List<Y> map(Fun<X,Y> f) {
     ...
} }
List<String> l = ...;
Fun<String,Integer> f1 = ...;
Fun<String,String> f2 = ...;
List<Integer> l1 = l.<Integer>map(f1);
List<String> l2 = l.<String>map(f2);
```

Other Features of Java Generics (2): Method Type Argument Inference

- Automatic synthesis of type arguments from types of value arguments

```
class C {
  <Y> Y choose(Y y1, Y y2) {
    if ... return y1; else return y2;
  }
}

C c = …;  Integer i = …; Float f = …;
Number n = c.<Number>choose(i,f);
 // Y is implicitly instantiated to Number
```

Other Features of Java Generics (3): Bounded quantification

- The upperbound of the range of a type variable
  - **`Object`** when omitted

```
class NumList<X extends Number> {
  X head;  NumList<X> tail;
  Byte byteHead() {
    return this.head.byteValue();
    //     ^^^^^^^^^
    //     subsumption using X <: Number
} }
NumList<Integer> il = …;
NumList<String> sl = …;   // typing error!
```

- Recursive bounds (F-bounded quantification)

```
interface Comparable<X> { boolean cmp(X that);}
class CmpList<X extends Comparable<X>> {
  X hd;  CmpList<X> tl;
   void sort() { … this.hd.cmp(this.tl.hd) … }
}
class A implements Comparable<A> {
  boolean cmp(A that) { … }}
CmpList<A> al = …;  al.sort();
```

Implementation of Java Generics

By so-called "erasure" translation

- One generic class to one class file
  - `class C<X> {...}` ⇒ `class C { ... }`
- Type parameter `X` ⇒ `Object`
- Typecasts are inserted where type mismatch occurs

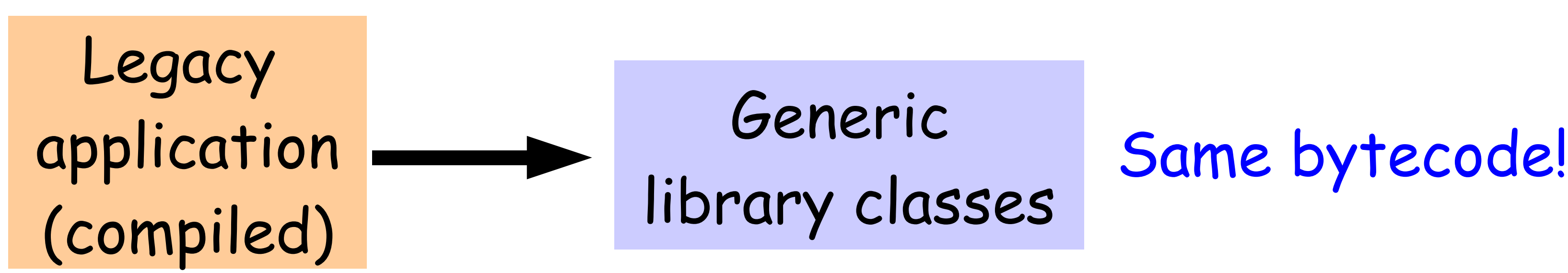
```
class List<X> {
  X head;
  List<X> tail;
  ...
}
List<String> ss =
 new List<String>(...);
String s = ss.head;
```

⇒

```
class List {
  Object head;
  List tail;
  ...
}
List ls = new List(...);
String s = (String)ls.head;
```

What's the Point?
Or, didn't you say typecasts are unsafe?

- Safety by automating the generic idiom
  - Typechecking with parametric types
  - Mechanical translation by erasure, which inserts typecasts
    - proven to succeed
      - [Igarashi, Pierce, Wadler; OOPSLA99]
- Compatibility with the idiom
  - (Library) classes written with the generic idiom and ones with generics result in the same bytecode
    - Old applications run without recompiling



Restriction due to Erasure Translation(1) : Type Abstraction only for Object Types

```
class List<X> {
  X car;  List cdr;
}
List<Integer> il = …;
List<int> sl = …;   // typing error!
```

- In Java 5.0, **`int`** and **`Integer`** are automatically converted to each other, though

Restriction (2): Typecasts

```
class List<X> { … }
class MyList<X> extends List<X> { … }

Object o;  List<String> ss;
(List<String>)o       // compile-time error!
(MyList<String>)ss    // OK!
```

- Both **`new List<String>()`** and **`new List<Integer>()`** are tagged only with **`List`** (w/o type argument information)
  - **`o`** may be **`new List<Integer>()`**
  - False positive must be excluded

Summary of Part II

- Generic classes for generic data structure
  - Reusability by parameterization
  - Safety by refined type information
- Implementation by the erasure translation
  - Automated idiomatic programming
  - Typecasts that eventually succeed
  - Somewhat unnatural restrictions
    - Could be avoided by "type-passing" implementation [NextGen, LM]

Part III

Even More Reusability by Wildcards

Overview of Part III

- Interaction between parametric types and subtyping
  - Subtyping schemes for parametric types
    - Subtyping based on inheritance
    - Subtyping based on variance
  - Safety issues
- Introduction of wildcards

Inheritance-based Subtyping

Instantiating the inheritance relation ("**extends**" clause) by type arguments

```
class MyList<X> extends List<X> { … }

List<String> ss = new MyList<String>(…);

// MyList<T> <: List<T> for any T
```

Variance-based Subtyping

Subtyping between parametric types from the same class

- Invariant subtyping rule
  - **C<S> <: C<T>** if **S** = **T**
- Covariant subtyping rule
  - **C<S> <: C<T>** if **S <: T**
  - e.g., **List<String> <: List<Object>**
- Contravariant subtyping rule
  - **C<S> <: C<T>** if **T <: S**
  - e.g., **List<Object> <: List<String>**

type safe?

Java Array Types **T[]**

- A kind of parametric types (～**Array<T>**)
- Covariant subtyping permitted

```
String[] ss = ...;
Object[] os = ss; // covariant subtyping
os[0] = new Integer(10);
int i = ss[0].length(); // NoSuchMethodError!?
```

- Run-time check for safety
  - Exception for illegal assignments
  - Again, to prevent **NoSuchMethodError**

```
os[0]=new Integer(10);  // ArrayStoreException!
```

Variance vs Safety

- More subtypes for more reusability
  - **`String[]`** can be passed to a method that takes **`Object[]`**
- Run-time checks to prevent **`NoSuchMethoError`**

Java Arrays Can Be Made Safe!

- Covariant subtyping for array types is always safe if you never assign anything
- Trade-off between covariance and assignments

→ Let programmers choose!

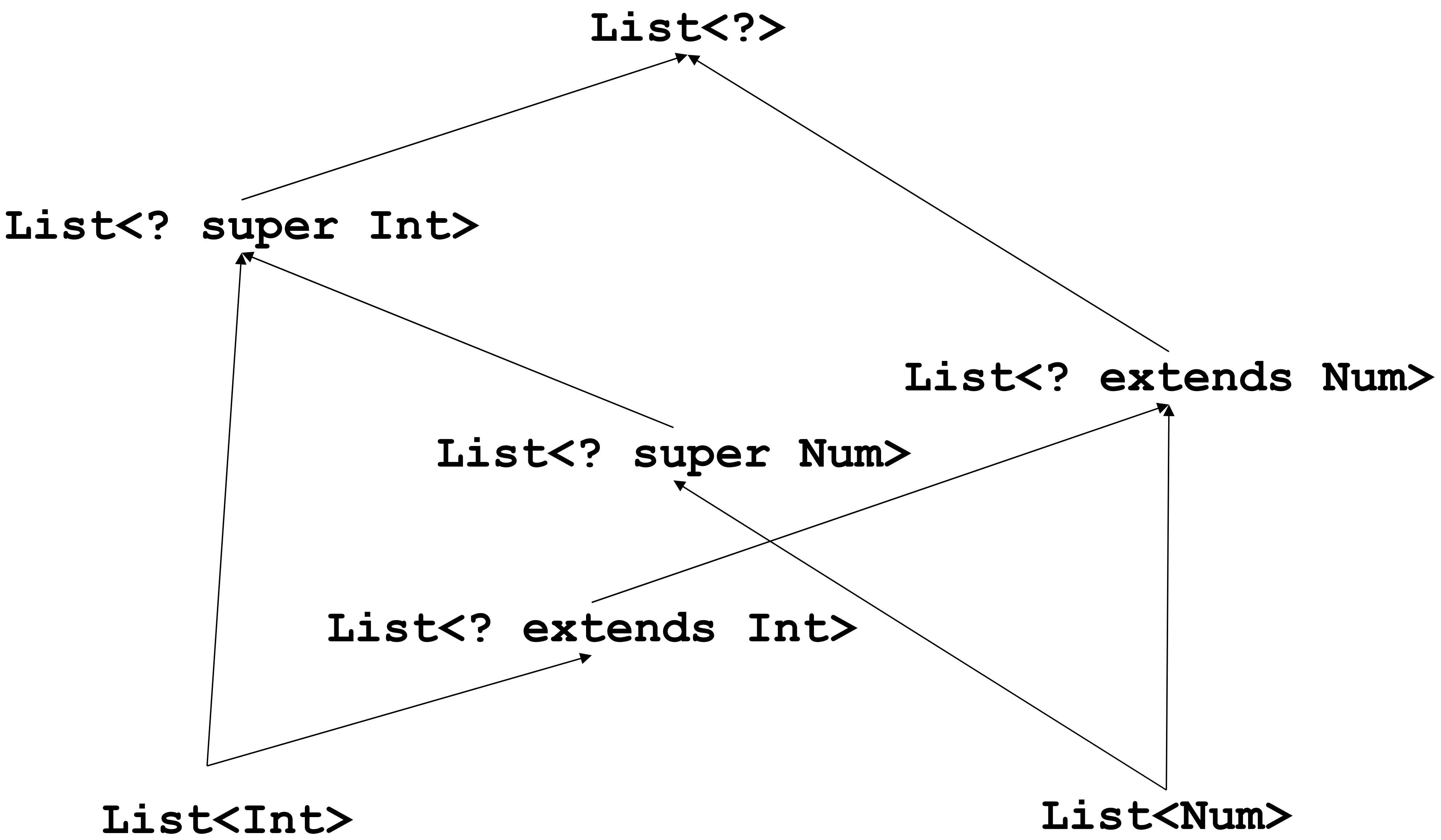
  - **T[]**: invariant but both reading and assigments permitted
  - **T[+]**: covariant but assignments prohibited

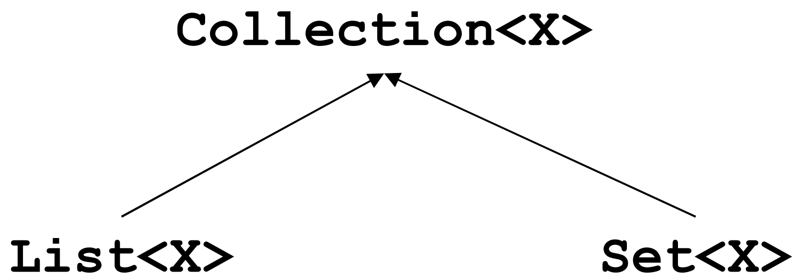
```
String[] ss = ...;
Object[+] os = ss; // covariant subtyping
os[0] = new Integer(10); // typing error!
```

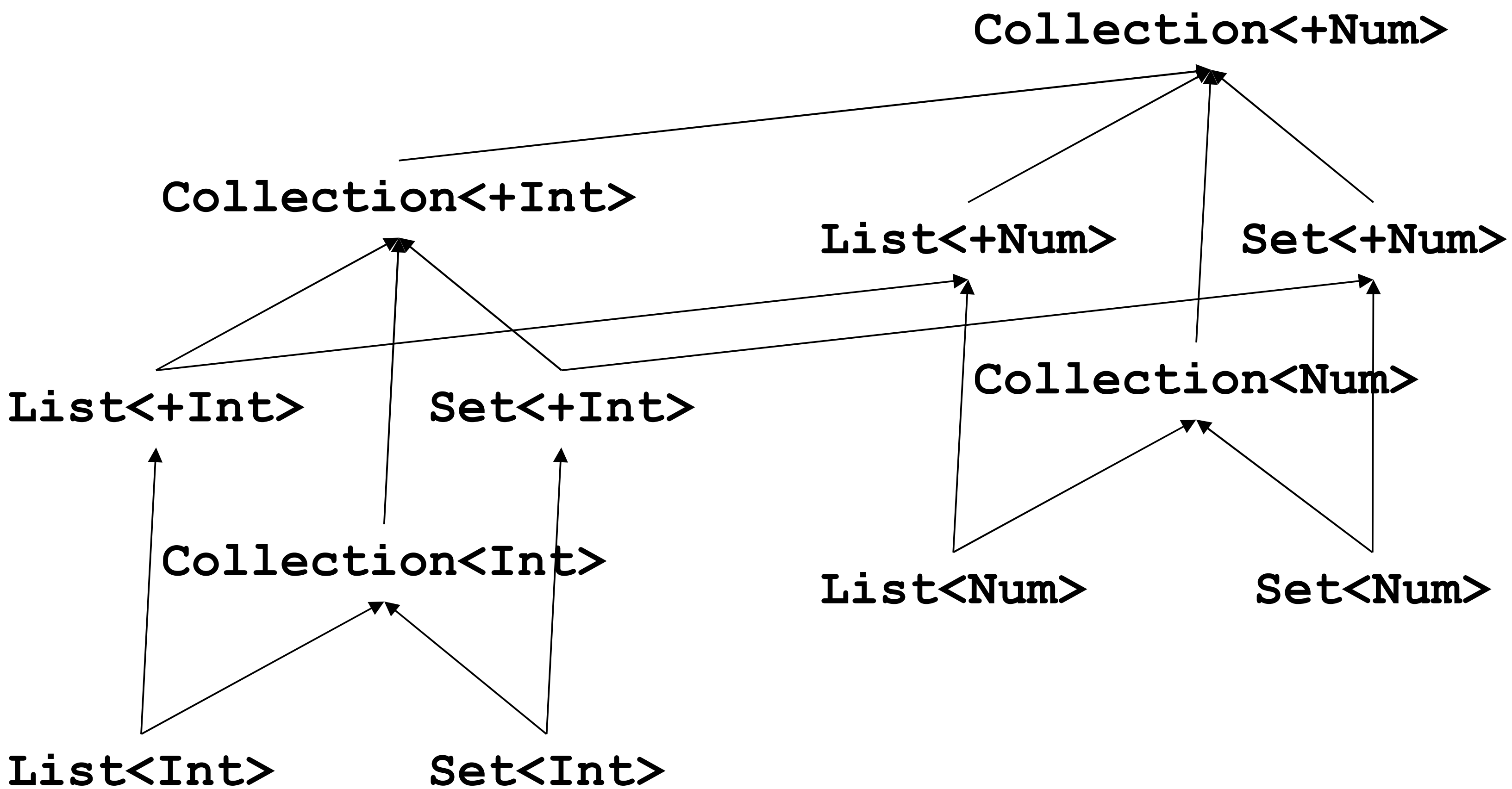
```
String[] ss = ...;
Object[] os = ss; // typing error!
os[0] = new Integer(10);
```

Introduction to Wildcards

- Invariant types: **`List<T>`**
  - Object instantiation, any method invocation permitted
- Covariant types: **`List<? extends T>`**
  - e.g., **`List<? extends String>`** `<:` **`List<? extends Object>`**
  - Invocation of methods to, e.g., assign new elements prohibited
- Contravariant types: **`List<? super T>`**
  - e.g., **`List<? super Object>`** `<:` **`List<? super String>`**
  - The types of read elements are **`Object`**
- **`List<?>`**
  - No assignments allowed, elements are read as **`Object`**
    - **`length()`** can be still invoked
    - All kinds of types above are subtypes


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


Collection<X>
List<X>
Set<X>


Collection<+Num>
Collection<+Int>
List<+Num>
Set<+Num>
Collection<Num>
List<+Int>
Set<+Int>
Collection<Int>
List<Num>
Set<Num>
List<Int>
Set<Int>

Intuition behind Wildcards

- **`List<?>`**
  - List of something you don't know
- **`List<? extends Number>`**
  - List of some **`Number`**s (maybe **`Integer`**s or **`Float`**s)
  - The element is not exactly known but reading elements yields **`Number`**s (by subsumption)
  - Assignment is prohibited since its element type is unknown
    - Only **`null`** can be assigned
- c.f. Existential types
  - **`∃X.List<X>`**
  - **`∃X<:Number.List<X>`**

Applications of Wildcards

- Parameter of a covariant type
  - Declaration of read-only use
- More applicability of the method

```
class List<X> { ...
  List<X> append(List<? extends X> l) {
    if (tail == null) return this;
    else return
      new List<X>(l.head, this.append(l.tail));
} }
List<Number> ns = ...;
List<Integer> is = ...;
List<Number> ns2 = ns.append(is);
// argument type: List<? extends Number>
```

```
interface Collection<X> {
  <Y> Y choose(Y y1, Y y2) {…}
}
class Set<X> implements Collection<X> {…}
class List<X> implements Collection<X>{…}

// without wildcards
Object x = choose(intSet, stringList);
// with wildcards
Collection<? extends Object> x =
    choose(intSet, stringList);
```

```
<Y> Set<Y> unmodifiableSet(Set<Y> s) {…}

Set<Integer> s1;
Set<Integer> s2 = unmodifiableSet(s1);
// here, Y is instantiated with Integer

Set<? extends Integer> s3;
Set<? extends Integer> s4 = unmodifiableSet(s3);
// Q: What is Y instantiated with?
//    A: The unknown type "?"!
```

Summary of Part III

- Wildcards and subtyping for parametric types
- More reusability for methods using parameters in a limited way
- Yet safe: Tradeoff between subtyping and access restriction

Conclusion: Safety and Reusability by Improving Type Systems

- Simple Type System
  - Towards no **`NoSuchMethodError`**
  - Typecasts and covariant array types
    - Loopholes to allow "useful" programs
    - Their abuse may reduce both safety and efficiency
- Generic Classes
  - Reusability by type parameterization
  - Refined type information by parametric types
- Wildcards
  - Flexible subtyping for parametric types

Departure from the “Class Names as Types” Principle

- Parametric types
  - Type = class name + type arguments
  - Run-time types = class name (+ type arguments)
- Wildcards
  - Type = class name + type arguments (possibly with “**`? super T`**” etc.)
  - Run-time types ⊂ types
    - Only invariant types can be a target of “**`new`**”

Types = Interface Information

References

- 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. To appear.
- 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
- S. Drossopoulou and S. Eisenbach. Java is Type Safe – Probably. In Proc. ECOOP'97