Struktur Data dan Algoritma

Generic Type in Java

Suryana Setiawan, Ruli Manurung & Ade Azurat
(acknowledgments: Denny)

Fasilkom UI

Polymorphism vs Generics

- One way that object-oriented languages allow generalization is through polymorphism.
- You can write a method that takes a base class object as and then use that method with any class ***derived*** from that base class.
- Sometimes being constrained to a single hierarchy is too limiting.

Interface vs Generics

- Interface allows us to loosen the single inheritance.
- Sometimes even an interface is too restrictive. An interface still requires that your code work with that particular interface.
- Now, with generic type support, your method is more general and can be used in more places.
- And, not only method but also class!

Motivation

- Generics implement the concept of *parameterized type*.
  - We usually parameterize a method with their arguments as values. Now we can parameterized not only with values (object) but also with type (class).
- It provides *type safe* container that avoid possible runtime error on collection.
- Ability to create more general-purpose code that can be highly re-used (*reusable and expressiveness*)

Case study: Automobile Holder

- We would like to have a container of automobile.

Non-generic approach

- See: generics/Holder1.java
  - It is not very reusable, since it can't be used to hold anything else.
  - We prefer not to write a new one of these for every type we encounter.

Non-Generic Approach (2)

- See: generics/Holder2.java
  - It is quite reusable.
  - It could hold anything.
  - But in one program, you may want to have it only contains specific type.
  - Possible run-time error due to incorrect casting.

Generic Approach

- See: generic/Holder3.java
  - It is still reusable
  - It could be parameterized by any class
  - Once it is parameterized, it could only contain a specific type only.
  - No possible run-time error due to incorrect casting.
  - No casting needed.

Generic Interfaces

- Generics also work with interface.
- See: generics/Generator.java

```
public interface Generator<T> {
    T next();
}
```

Implementation of Generic Interface

- See: generics/Fibonacci.java
- Naïve algorithm.
- Let's go one step further and make an Iterable Fibonacci generator.

Iterable Fibonacci generator

- We may not always have control of the original code. (to modify it)
- We may not want to rewrite when you don't have to.
- How can we just reuse them, but we don't need to modify it and overriding some of the methods.
- Solution: We can create adapter to produce the desired interface.
- See: generics/IterableFibonacci.java

Question

- Is the IterableFibonacci.java better than Fibonacci.java?
- In what sense?
- How can we improve it?

Generic Methods

- So far we've looked at parameterizing entire classes.
- We can also parameterize methods within a class.
- The class itself may or may not be parameterized
- See: generics/GenericMethods.java

A Generic Method to use with Generators

- It is convenient to use a generator to fill a Collection.
- It make sense to “generify” this operation.
- See:
  - generics/Fibonacci.java
  - generics/CoffeGenerator.java
  - generics/Generators.java

General Purpose Generator

- Let's go further!
- We can create a class that produces a Generator for any class that has default constructor.
- See:
  - generics/BasicGenerator.java
  - generics/CountedObject.java
  - generics/BasicGeneratorDemo.java

The application of Generics

- A Set Utility
- Consider the mathematical relationships that can be expressed using Sets.
- These can be conveniently defined as generic methods, to be used with all different types.
- set of numbers, set of books and etc.
- See:
  - generics/WatercolorSets.java
  - net/mindview/util/Sets.java

Anonymous Inner Class

- Generics can also be used with anonymous inner class
- See: generics/BankTeller.java

Bound (Contrained) type

- Now suppose that we want to program a Minimum function to find the minimum value in a generically typed array?

Bound (Contrained) type

```
public class Utility {
      public static <E> E min( E[] a) {
            E smallest = a[0];
            for (int i = 1; i < a.length; i++)
            if (a[i].compareTo(smallest) < 0)
                  smallest = a[i];
            return smallest;
      }
}
```

Bound (Contrained) type variables

```
Utility.java:5: cannot find symbol
symbol  : method compareTo(E)
location: class java.lang.Object
                if (a[i].compareTo(smallest) < 0)
                       ^
1 error
```

- The previous code doesn't compile because we are assuming that type *E* implements the **`compareTo`** method.
- How to solve it, can we use interface?

Bound (Contrained) type variables

- The solution in Java 1.5 is to use a constrained type variable. The method **compareTo** is enforced by the interface **Comparable**

```
public static <E extends Comparable> E min(E[] a) {
        E smallest = a[0];
        for (int i = 1; i < a.length; i++)
        if (a[i].compareTo(smallest) < 0)
                smallest = a[i];
        return smallest;
}
```

Bound (Contrained) type variables

- We are telling the compiler that the method can be used for arrays of *E* only when *E* implements the **`Comparable`** interface.
- The method compiles but generates the warning message:

Note: Utility.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

- Why?

Bound (Contrained) type variables

- The final version is

```
public static <E extends Comparable<E>> E min(E[] a) {
        E smallest = a[0];
        for (int i = 1; i < a.length; i++)
        if (a[i].compareTo(smallest) < 0)
                smallest = a[i];
        return smallest;
}
```

- which compiles without warning messages.

Bound (Contrained) type variables

- A more complicated example is:
  `<E extends Foo & Bar>`
- where Foo and Bar are either interfaces or classes.
- Here, read the word ***extends*** as "*extends or implements*".

Wildcard types: motivation

- Consider the problem of writing a routine that prints out all the elements in a collection
- Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

```
static void printCollection(Collection c) {
        Iterator i = c.iterator();
        for (k = 0; k < c.size(); k++) {
            System.out.println(i.next());
        }
}
```

Wildcard types: motivation

- Using generics:

```
static void printCollection(Collection<Object> c) {
        for (Object o : c)
             System.out.println(o);
}
public static void main(String[] args) {
     Collection<String> cs = new Vector<String>();
     printCollection(cs); // Compile error
     List<Integer> li = new ArrayList<Integer>(10);
     printCollection(li); // Compile error
}
```

Wildcard types: motivation

- Why the previous code does not compile?

The mistery of erasure?

- See:
  - generics/ErasedTypeEquivalence.java

The mistery of erasure?

- What is erasure?
  - Any specific type information is erased when you use a generic.
- There is no information about generic parameter types available inside generic code.
- See:
  - generics/ErasedTypeEquivalence.java
  - generics/LostInformation.java

More on Erasure

- All generic type information is removed in the resulting byte-code after compilation
- So, generic type information does not exist during runtime
- After compilation, they all share same class (raw class)
- Example: The class that represents ArrayList<String>,ArrayList<Integer> is same class that represents ArrayList

Question:

- Again, Type Erasure Example Code:
- True or False?

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<String> as = new ArrayList<String>();
Boolean b1 = (ai.getClass() == as.getClass());
System.out.println("Do ArrayList<Integer> and
  ArrayList<String> share same class? " + b1);
```

Solution: use wildcard type argument <?>

```
static void printCollection(Collection<?> c) {
     for (Object o : c)
          System.out.println(o);
}
public static void main(String[] args) {
     Collection<String> cs = new Vector<String>();
     printCollection(cs); // No Compile error
     List<Integer> li = new ArrayList<Integer>(10);
     printCollection(li); // No Compile error
}
```

Solution: use wildcard type argument <?>

- **`Collection<?>`** means Collection of unknown type
- Accessing entries of Collection of unknown type with Object type is safe.
- Question:
  - Can we do:
    ```
    printCollection(Collection<? extends Object> c)
    ```
  - What is the different?

Another example: Pair

```
class Pair {
    public Object first;
    public Object second;
    public Pair (Object f, Object s) { . . }
}
```

- We can also call this type of class as the the raw type for **`Pair <T>`**

Pair: Generics Version

```
class Pair <T> {
    public T first;
    public T second;
    public Pair (T f, T s) {
        first = f; second = s;
    }
    public Pair () {
        first = null; second = null;
    }
}
```

Pair: Generics Version

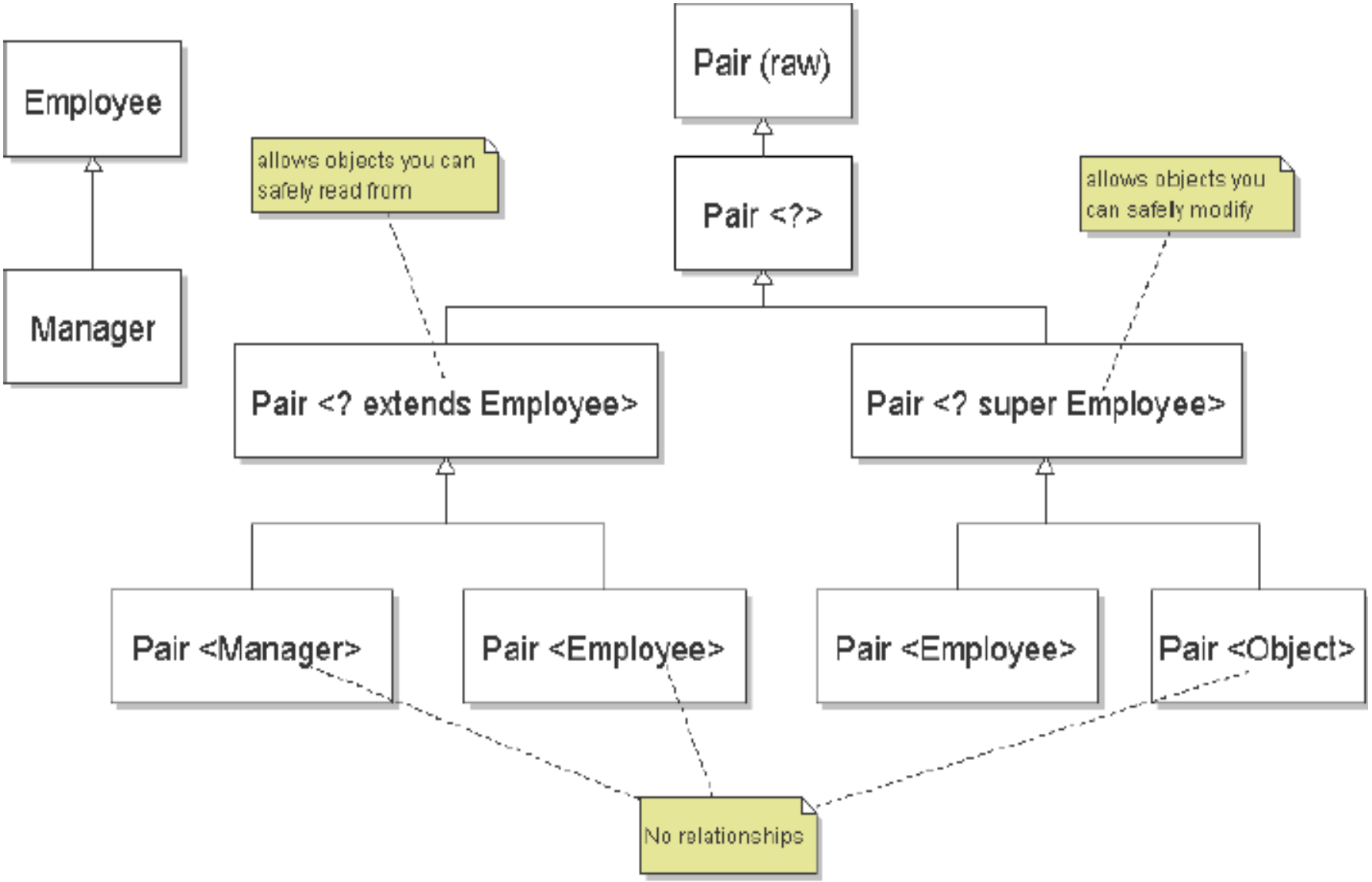
- Can we have multiple type parameters?
- Can we have different type for first and second?

Pair: Multiple type

```
class Pair <T, U> {
    public T first;
    public U second;
    public Pair (T x, U y) {
        first = x; second = y;
    }
    public Pair () {
        first = null; second = null;
    }
}
```

- to instantiate: Pair <String, Number>

Inheritances rules for generic type



Comments on inheritance relations

- Pair<Manager> matches Pair<? extends Employee>
  - subtype relation (covariant typing)
- Pair<Object> matches Pair<? super Employee>
  - supertype relation (contravariant typing)
- Pair<Employee> can contain only Employees, but Pair<Object> may be assigned anything (Numbers)
  - no subtype relation
- also: Pair<T> <= Pair<?> <= Pair (raw)

inheritance relations

- Try:

```
List <String> sl = new LinkedList <String> ();
List x = sl; // OK
x.add (new Integer (5)); // type safety warning
. .
String str = sl.get (0); // throws ClassCast
```

Generic code and the JVM

- the JVM has no instantiations of generic types
- a generic type definition is compiled once only, and a corresponding raw type is produced
  - the name of the raw type is the same name but type variables removed
- type variables are erased and replaced by their bounding types (or Object if no bounds);
- See: raw type of class Pair in the previous slide
- byte code has some generic info, but objects don't

Generic code and the JVM (cont.)

- Pair <String> and Pair <Employee> use the same bytecode generated as the raw class Pair
- when translating generic expressions, such as

```
Pair <Employee> buddies = new Pair < . .;
Employee buddy = buddies.first;
```

- the compiler uses the raw class and automatically inserts a cast from Object to Employee:

```
Employee buddy = (Employee) buddies.first;
```

Wildcard types

- some operations have no type constraints:

```
public static boolean hasNulls (Pair <?> p) {
      return p.first == null || p.second == null;
}
```

- alternatively, you could provide a generic method

```
public static <T> boolean hasNulls (Pair <T> p) {...}
```

- generally, prefer wildcard types (but use generic method with type T for multiple parameters)

```
public static <T1,T2> boolean hasNulls (Pair <T1,T2> p)
   {...}
```

Wildcard capture

- the wildcard type ? cannot be used as a declared type of any variables.

```
Pair <?> p = new Pair <String> ("one", "two"); . .
p.first = p.second; // ERROR: unknown type
```

- but, can sometimes use a generic method to capture the wildcard:

```
public static <T> void rotate (Pair <T> p) {
      T temp = p.first; p.first = p.second;
      p.second = temp;
}
```

- the compiler checks that such a capture is legal
  - e.g., the context ensures that T is unambiguous

Exercise:

- See: generics/PairTest.java
- Create generics class: Triple<A,B,C>