Lập trình hướng đối tượng

Biên soạn: Nguyễn Thị Tuyết Hải

Email: tuyethai@ptithcm.edu.vn

Chapter 8: Generic Programming

8.1 Why Generic Programming?

8.2 Defining a Simple Generic Class

8.3 Generic Methods

8.4 Bounds for Type Variables

8.5 Generic Code and the Virtual Machine,

8.6 Restrictions and Limitations

8.7 Inheritance Rules for Generic Types

8.8 Wildcard Types

8.9 Reflection and Generics

8.1 Why Generic Programming?

- Generic programming: writing code that can be reused for objects of many different types.
- The ArrayList class now has a type parameter that indicates the element type:

```
ArrayList<String> files = new ArrayList<String>();
ArrayList<String> files = new ArrayList<>();
```

8.2 Defining a Simple Generic Class

A generic class is a class with one or more type variables.

The type variables are used throughout the class definition to specify:

- method return types
- the types of fields and local variables.

- Instantiate the generic type by substituting types for the type variables

```
public class Pair<T>
{
   private T first;
   private T second;

   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first;
this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first =
newValue; }
   public void setSecond(T newValue) { second =
newValue; }
}

Pair<String> mm;
```

8.3 Generic Methods

```
class ArrayAlg {
	public static <T> T getMiddle(T... a) {
		return a[a.length / 2];
	}
}

String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

8.4 Bounds for Type Variables

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

8.4 Bounds for Type Variables

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

8.5 Generic Code and the Virtual Machine

8.5.1 Type Erasure

- Whenever you define a generic type, a corresponding raw type is automatically provided.
- The raw type: the first bounding type (or **Object** for variables without bounds)

```
public class Pair {
      private Object first;
      private Object second;

      public Pair(Object first, Object second)
      {
            this.first = first;
            this.second = second;
      }
      public Object getFirst() {
            return first;
      }

      public Object getSecond() {
            return second;
      }

      public void setFirst(Object newValue) {
            first = newValue;
      }

      public void setSecond(Object newValue) {
            second = newValue;
      }
}
```

8.5.1 Type Erasure

```
public class Interval<T extends Comparable &
Serializable> implements Serializable {
     private T lower;
     private T upper;
     ...
     public Interval(T first, T second) {
          if (first.compareTo(second) <= 0) {
               lower = first;
               upper = second;
          }
          else {
               lower = second;
               upper = first;
          }
     }
}
```

```
public class Interval implements
Serializable {
     private Comparable lower;
     private Comparable upper;
     ...
     public Interval(Comparable first,
     Comparable second) {
     . . .
     }
}
```

8.5.2 Translating Generic Expressions

- When you call to a generic method, the compiler inserts casts when the return type has been erased.

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

  - A call to the raw method Pair.getFirst
  - A cast of the returned Object to the type Employee

8.5.3 Translating Generic Methods

Before erasure:

```
public static <T extends Comparable> T
min(T[] a)

class DateInterval extends Pair<LocalDate> {
     public void setSecond(LocalDate second){
       if (second.compareTo(getFirst()) >= 0)
          super.setSecond(second);
     }
...
}
```

After erasure:

```
public static Comparable min(Comparable[] a)

class DateInterval extends Pair{
     public void setSecond(LocalDate
second){ . . . }
... }

public class Pair {
     public void setSecond(Object newValue){
          second = newValue;
     }
}
```

8.5.3 Translating Generic Methods

```
DateInterval interval = new DateInterval(. . .);
Pair<LocalDate> pair = interval; // OK--assignment to superclass
pair.setSecond(aDate);  // which setSecond()?
```

How:

The compiler generates a bridge method in the DateInterval class:

```
    public void setSecond(Object second) { setSecond((LocalDate) second);
}
```

8.6 Restrictions and Limitations

8.6.1 Type Parameters Cannot Be Instantiated with Primitive Types

- there is no Pair<double>, only Pair<Double>

8.6.2 Runtime Type Inquiry Only Works with Raw Types

```
if (a instanceof Pair<String>) // Error
if (a instanceof Pair<T>) // Error
if (a instanceof Pair<?>) // OK

Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass())
// they are equal since getClass() return Pair.class
```

8.6 Restrictions and Limitations

8.6.3 You Cannot Create Arrays of Parameterized Types

```
Pair<String>[] table = new Pair<String>[10]; // Error

ArrayList<Pair<String>> list= new ArrayList<>();
```

8.6.5 You Cannot Instantiate Type Variables

```
public Pair() { first = new T(); second = new T(); } // Error

// define makePair()
public static <T> Pair<T> makePair(Supplier<T> constr) {
        return new Pair<>(constr.get(), constr.get());
}

// call makePair()
Pair<String> p = Pair.makePair(String::new);
```

8.6 Restrictions and Limitations

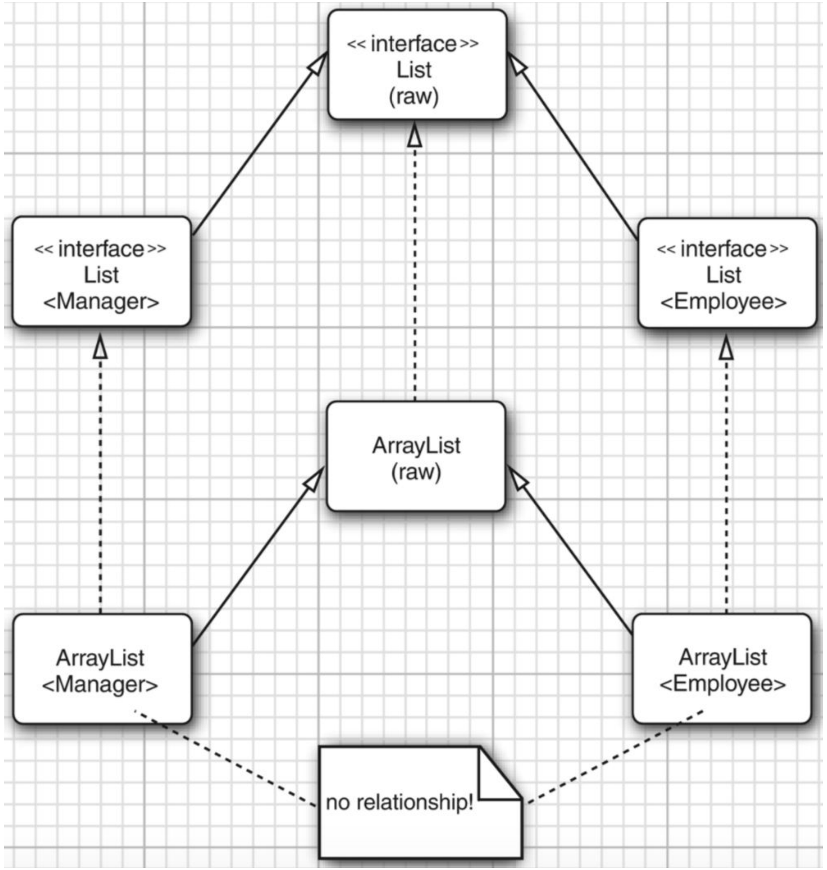
8.6.7 Type Variables Are Not Valid in Static Contexts of Generic Classes

```
public class Singleton<T> {
        private static T singleInstance; // Error
        public static T getSingleInstance() // Error { …
        }
}
```

8.6.8 You Cannot Throw or Catch Instances of a Generic Class

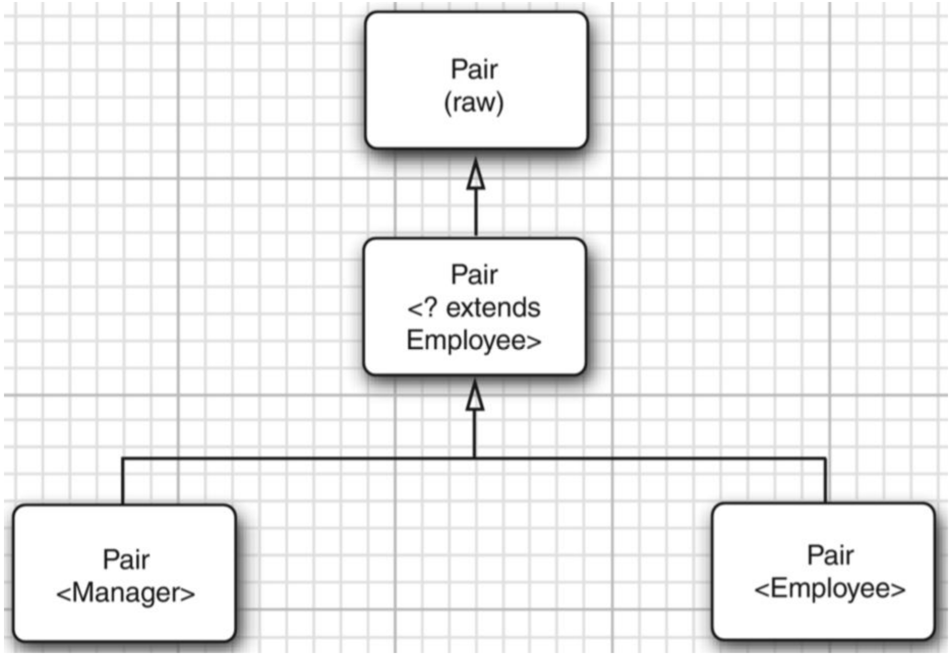
8.7 Inheritance Rules for Generic Types

- no relationship between Pair<S> and Pair<T>, no matter how S and T are related.
- convert a parameterized type to a raw type.
  - Pair<Employee> is a subtype of the raw type Pair.
- generic classes can extend or implement other generic classes.
  - ArrayList<T> implements the interface List<T>



8.8 Wildcard Types

- Pair<? extends Employee>: any generic Pair type whose type parameter is a subclass of Employee, e.g., Pair<Manager>
- Pair<Manager> is a subtype of Pair<? extends Employee>

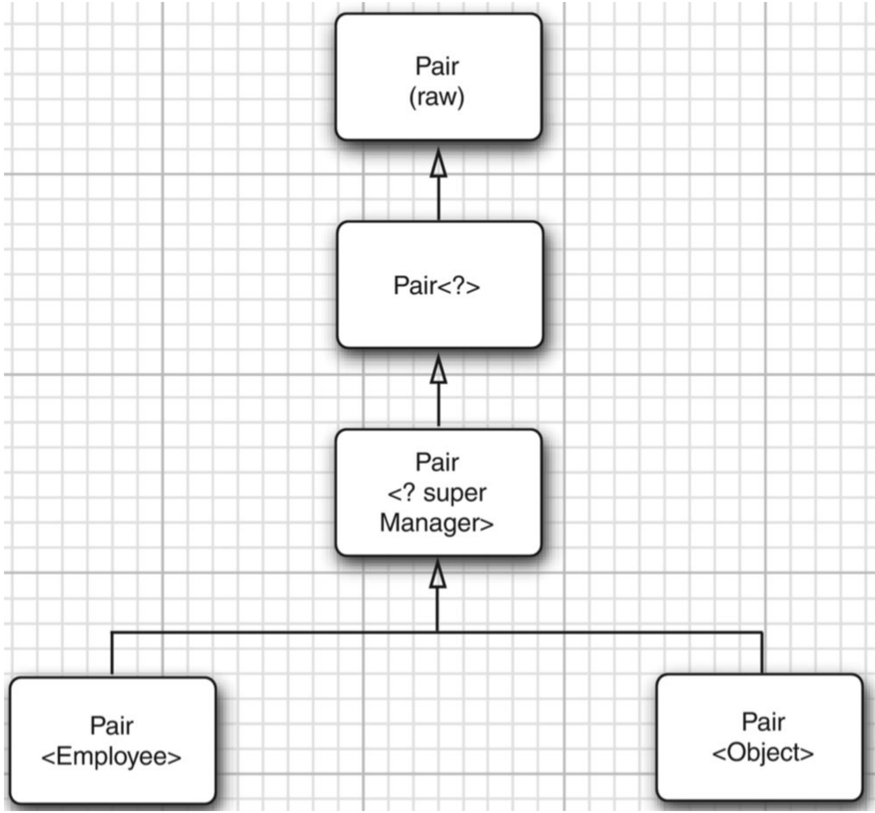


8.8 Wildcard Types

```
public static void printBuddies(Pair<? extends Employee> p) {
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " and " + second.getName() + " 
are buddies.");
}
Pair<Manager> managerBuddies = new Pair<>(new Manager("CEO", 1500, 1999, 
1, 1), new Manager("CFO", 1500, 2000, 1, 1));
printBuddies(managerBuddies);
```

8.8 Wildcard Types

```
public static void minmaxBonus(Manager[] a,
                    Pair<? super Manager> result) {
      if (a.length == 0)
            return;
      Manager min = a[0];
      Manager max = a[0];
      for (int i = 1; i < a.length; i++) {
            if (min.getBonus() > a[i].getBonus())
                  min = a[i];
            if (max.getBonus() < a[i].getBonus())
                  max = a[i];
      }
      result.setFirst(min);
      result.setSecond(max);
}
```



8.8 Wildcard Types

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

public static <T> boolean hasNulls(Pair<T> p){
        return p.getFirst() == null || p.getSecond() == null;
}
```