Advanced Programming

Generic Programming

Introduction

Prior to the JDK 5.0 release, when you created a Collection, you could put any object in it.

```
List myList = new ArrayList(10);
myList.add(new Integer(10));
myList.add("Hello, World");
```

Introduction

Getting items out of the collection required you to use a casting operation:

```
Integer myInt =
(Integer)myList.iterator().next();
```

If you cast the wrong type, the program would successfully compile, but an exception would be thrown at runtime.

Introduction

Solution: Use `instanceof` to avoid a blind cast

```
Iterator listItor = myList.iterator();
Object myObject = listItor.next();
Integer myInt = null;
if (myObject instanceof Integer)
     myInt = (Integer)myObject;
```

Objective

- Writing code that can be reused for objects of many different types
- For example, you don’t want to program separate classes to collect String and File objects
- E.g.: The single class ArrayList collects objects of any class

Classical ArrayList

- ArrayList class can simply maintain an array of Object references

```
public class ArrayList // before generic classes
{
   private Object[] elementData;
   . . .
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
}
```

Problems?

This approach has two problems.

1. A cast is necessary whenever you retrieve a value:

```
ArrayList files = new ArrayList();
. . .
String filename = (String) files.get(0);
```

2. There is no error checking. You can add values of any class:

```
files.add(new File(". . ."));
```

This call compiles and runs without error. Elsewhere, casting the result of get to a String will cause an error.

Defining Simple Generic Class

```
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; }
}
```

Instantiate as `Pair<String>` or with any other data type

Using Generic Pair

```
Pair<String> my_pair = new Pair<String>();

my_pair.setFirst(“Jade”);

my_pair.setSecond(“Carl”);

String first = my_pair.getFirst();
```

Defining Generic Method

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

Calling a generic method:

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.",
"Kim");
```

Translating Generic Methods: Erasure

- There are no generics in the virtual machines, only ordinary classes and methods.
- All type parameters are replaced by their bounds.
- Casts are inserted as necessary to preserve type safety
- Bridge methods are synthesized to preserve polymorphism.

After Erasure

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

Translating Expressions

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

Translates to

```
Object temp = buddies.getFirst();
Employee buddy = (Employee)temp;
```

Bounds

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

Bounds

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

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

Erasure with bounds

Before Erasure

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

After Erasure

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

Erasure with bounds

Before Erasure

```
public class Interval<T extends Comparable & Serializable>
implements Serializable
{
public Interval(T first, T second)
```

After Erasure

```
public class Interval implements Serializable
{
public Interval(Comparable first, Comparable second)
```

Restrictions

- Type Parameters Cannot Be Instantiated with Primitive Types – No double, only Double

Think of

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

Will translate to

```
Object temp = buddies.getFirst();
int buddy = (int)temp;
```

Restrictions

- Type Parameters Cannot Be Instantiated with Primitive Types – No double, only Double

Think of

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

Will translate to

```
Object temp = buddies.getFirst();
Integer buddy = (Integer)temp;
```

Restrictions

- You Cannot Instantiate Type Variables

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

Think of translation with T = `Integer`

```
public Pair(Object a)
{
	first = (Integer)new Object();
	second = (Integer)new Object();
}
```

Restrictions

- Arrays of Parameterized Types Are Not Legal

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

After erasure

```
Object[] objarray = table;
```

- An array remembers its component type and throws an ArrayStoreException if you try to store an element of the wrong type:

objarray[0] = "Hello"; // ERROR--component type is Pair

But erasure renders this mechanism ineffective for generic types.

Restrictions

- You Cannot Throw or Catch Instances of a Generic Class - not legal for a generic class to extend Throwable

```
public static <T extends Throwable>
void doWork(Class<T> t)
{
	try{
		do work
	}
	catch (T e) { // ERROR--can't catch type variable
		Logger.global.info(...)
	}
}
```

Restrictions

- Runtime Type Inquiry Only Works with Raw Types – all type inquiries yield only the raw type
- Type Variables Are Not Valid in Static Contexts of Generic Classes

Restrictions

- Beware of Clashes After Erasure

```
public class Pair<T>
{
  public boolean equals(T value) {
    return first.equals(value) && second.equals(value);
  }
}
```

Pair<String> has two equals methods:

```
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object
```

The erasure of the method `boolean equals(T)` is `boolean equals(Object)` which clashes with the `Object.equals` method

Inheritance Rules for Generic Types

- Consider a class and a subclass, such as Employee and Manager.
- Is Pair<Manager> a subclass of Pair<Employee>?

Inheritance Rules for Generic Types

- Consider a class and a subclass, such as Employee and Manager.
- Is Pair<Manager> a subclass of Pair<Employee>?
- Answer: No

```
Manager[] topmen = . . .;
Pair<Employee> result = ArrayAlg.minmax(topmen); // ERROR
```

The minmax method returns a Pair<Manager>, not a Pair<Employee>, and it is illegal to assign one to the other.

Inheritance Rules for Generic Types

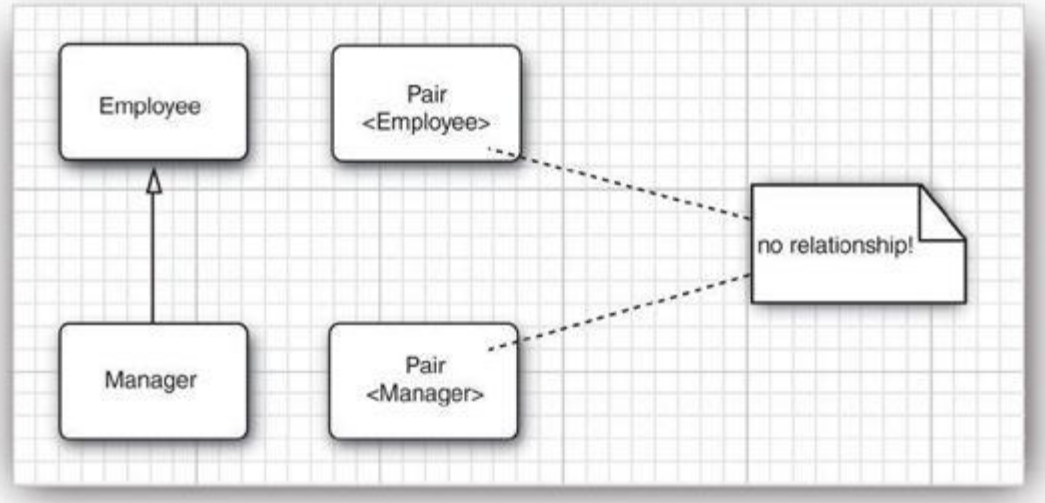


Image Source: Core Java, Volume-1

- However, generic classes can extend or implement other generic classes

Another Example

```
public class A { }
public class B extends A { }
public class C extends A { }


List<A> listA = new ArrayList<A>();
List<B> listB = new ArrayList<B>();


listA = listB;
listB = listA;
```

Another Example

Suppose, you want to write a method that prints out pairs of employees

```
public static void printBuddies(Pair<Employee> p)
{
   Employee first = p.getFirst();
   Employee second = p.getSecond();
   System.out.println(first.getName() + " and " +
second.getName() + " are buddies.";
}
```

You cannot pass a Pair<Manager> to the method, which is rather limiting

Wildcard Types

- `Pair<? extends Employee>`
  denotes any generic Pair type whose type parameter is a subclass of Employee, such as Pair<Manager>, but not Pair<String>.

- Using wildcard type
  `public static void printBuddies(Pair<? extends Employee> p)`

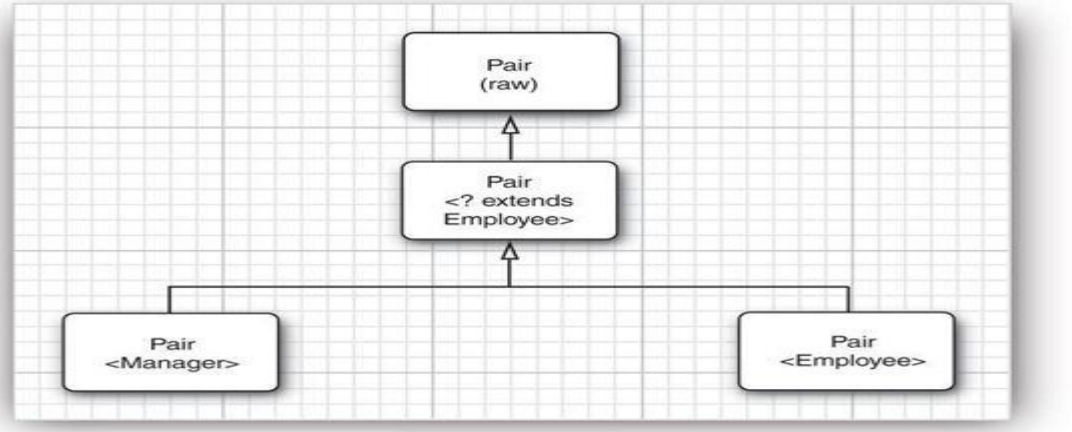


Image Source: Core Java, Volume-1

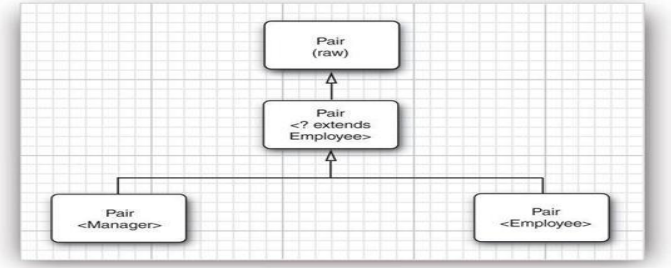
Wildcard Types

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);

Pair<? extends Employee> wildcardBuddies =
managerBuddies;// OK

wildcardBuddies.setFirst(lowlyEmployee); // compile-
time error
```

Wildcard Types




```
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error, Why??
```

Methods of Pair<? extends Employee> look like this:

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

- Call the setFirst method: The compiler only knows that it needs some subtype of Employee, but it doesn’t know which type. It refuses to pass any specific type—after all, ? might not match it.

- Call getFirst: It is perfectly legal to assign the return value of getFirst to an Employee reference.

Supertype Bounds for Wildcards

- `? super Manager`
  This wildcard is restricted to all supertypes of Manager

- A wildcard with a supertype bound gives you a behavior that is opposite to that of the wildcards described before
  - You can supply parameters to methods, but you can’t use the return values

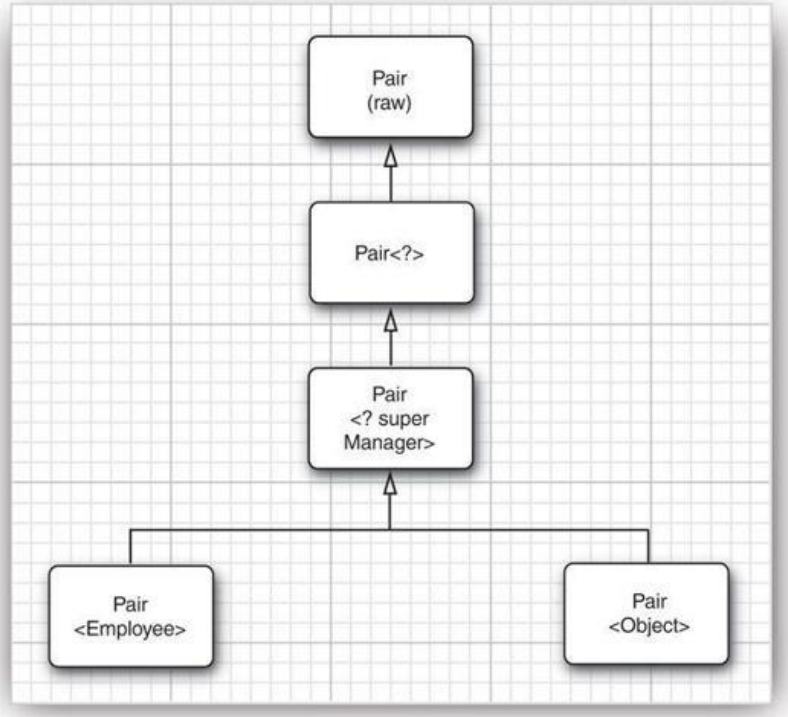


Image Source: Core Java, Volume-1

Supertype Bounds for Wildcards

Pair<? super Manager> has methods

```
void setFirst(? super Manager)
? super Manager getFirst()
```

- The compiler doesn’t know the exact type of the setFirst method and therefore can’t call it with an object of type Employee or Object, but only with type Manager or a subtype such as Executive.

- Moreover, if you call getFirst, there is no guarantee about the type of the returned object. You can only assign it to an Object.

Supertype Bounds for Wildcards

Given array of managers, put the manager with the lowest and highest bonus into a Pair object.
Pair object?? Consider Pair<Employee>

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
   if (a == null || 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);
}
```

Wildcards(Continued..)

- Wildcards with
  - supertype bounds let you write to a generic object,
  - subtype bounds let you read from a generic object

Bounded wildcards are useful in situations where only partial knowledge about the type argument of a parameterized type is needed.

```
public class Collections {
  public static <T> void copy(List<? super T> dest, List<?
extends T> src) {
          for (int i=0; i<src.size(); i++)
            dest.set(i,src.get(i));
  }
}
```

Unbounded Wildcards

- Pair<?> equivalent to Pair<? Extends Object>
- Pair<?> : Not same as Pair.
- Pair<?> has methods such as

```
? getFirst()
void setFirst(?)
```

- The return value of getFirst can only be assigned to an Object. The setFirst method can never be called, not even with an Object