More on Generics

Hsuan-Tien Lin

Department of CSIE, NTU

OOP Class, May 30, 2013

This is Where It Starts.....

- array “direct inheritance” introduces type unsafety

```
1  Fruit[] frArr = new Fruit[3];
2  Food[] foArr = frArr; //Compiler: Yes!
3  foArr[0] = new Meat(); //Compiler: Yes!
4  Fruit fr = frArr[0];  //Compiler: Yes!
```

- generic: **not using direct inheritance**

```
1  ArrayList<Fruit> frArr = new ArrayList<Fruit>(3);
2  ArrayList<Food> foArr = frArr; //Compiler: No!
3  foArr.add(0, new Meat()); //Compiler: Yes!
4  Fruit fr = frArr.get(0);  //Compiler: Yes!
```

What If We Want to "Convert From" An Existing List?

```
1  MyList<Fruit> frArr = new MyList<Fruit>();
2  MyList<Food> foArr = new MyList<Food>(frArr); //convert
3
4  class MyList<T>{
5    public MyList<T>(){ }
6    public <S> MyList<T>(MyList<S> input){ /* code */ }
7  }
```

- shouldn't work because we can convert Fruit (S) List to Meat (T) List arbitrarily
- enforce type compatibility: `S extends T`

```
1    public <S extends T> MyList<T>(MyList<S> input){ }
2    //or, if S is not needed
3    public MyList<T>(MyList<? extends T> input){ }
```

- now this converting is like "upper cast"

What If We Want to "Convert To" A New List?

```
1  MyList<Fruit> frArr = new MyList<Fruit>();
2  MyList<Food> foArr = new MyList<Food>();
3  frArr.dumpInto(foArr);
4
5  class MyList<T>{
6    public MyList<T>(){ }
7    public <S> dumpInto(MyList<S> output){ /* code */ }
8  }
```

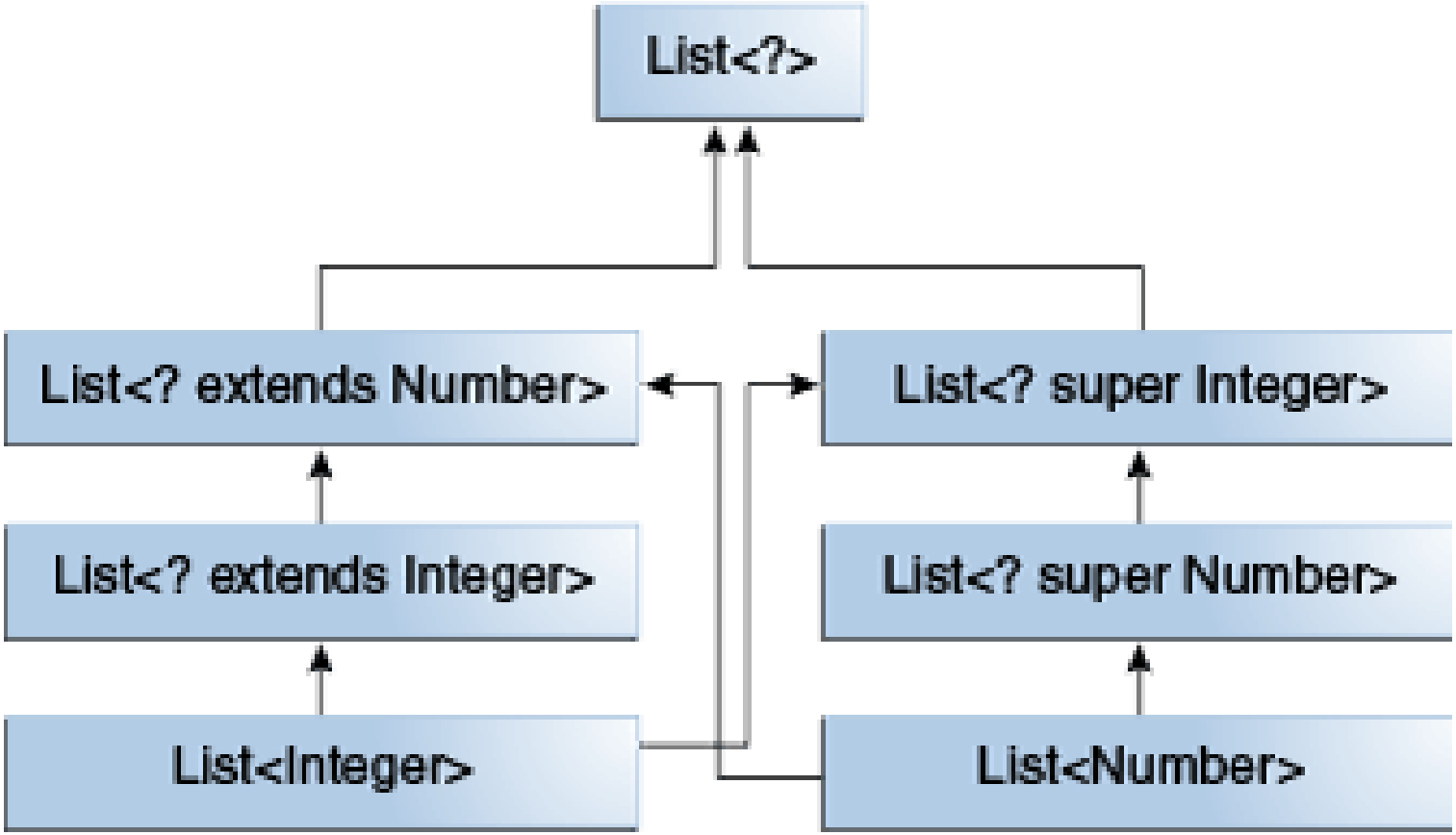
- shouldn't work because we can dump Fruit (S) List to Meat (T) List arbitrarily
- enforce type compatibility: `S super T`

```
1    public <S super T> dumpInto(MyList<S> output){ }
2    //or, if S is not needed
3    public dumpInto(MyList<? super T> output){ }
```

- again, "upper cast", but in another direction

More on “?” (Wildcard)

(figure from `http://docs.oracle.com/javase/tutorial/java/generics`)



Guidelines on Wildcards

(figure from `http://docs.oracle.com/javase/tutorial/java/generics`)

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

Make Things More Sophisticated

```
1  static <T extends Object & Comparable<? super T>>T max(Collection
       <? extends T> coll);
```

More about Type Erasure

- one main reason: JVM doesn't need changing between 1.4 and 1.5
- replace types by upper bounds
- cast automatically inserted with safe check
- allow using "legacy code" **unsafely**
  - List
  - List<Object>
  - List<?>
  - List<Fruit>

  Calling legacy code from generic code is inherently dangerous; once you mix generic code with non-generic legacy code, all the safety guarantees that the generic type system usually provides are void.
- cons: lose run-time ability of generic types (`new T()`)

More on Polymorphism

- ad-hoc polymorphism: one method name, many different uses (via signature): `print(int)` and `print(Object)`
- subtype polynorphism (often seen in OOP): one entry point (parent method), many different future uses (via overriding the method): `String.valueOf(Object)` which calls `Object.toString()`
- parametric polymorphism (generics, often seen in Functional Programming): one class (type-erased), many different uses: `printList(List<T> lst)`