The Craftsman: 44
Brown Bag I
Java Generics 2

Robert C. Martin
22 November 2005

...Continued from last month.

June, 1944.

Intelligence analyst Jennifer Kohnke scanned the satellite photos again and again. Since the birds had begun flying again last month, her workload had become nearly unbearable. Her supervisors kept on handing her photos of some kind of complex in the eastern part of the Reich. They wanted to know what the Germans were doing there. The heavy train traffic implied that the complex was industrial. So did the large number of barracks for the workers. The problem was that she didn't see any kind of significant factory. There was just one small building with big chimneys. It was too small for any real industrial purpose. Yet huge volumes of smoke continuously poured from those chimneys.

Avery and I learned a lot from experimenting with Java generics last Friday. So we decided to make such explorations a daily ritual from now on. We planned to meet at 11:00 every morning for a few minutes to discuss and show off, some new thing we had learned. Today it was my turn.

Monday, 25 Feb 2002, 1100

Japer and I had been working all morning on the Suspend User story, making good progress on it. At 11:00, Avery walked up and said: "Ready?"

I looked at Jasper and said: "Would you mind if I took a break for 15 min?"

He gave me that bright-eyed toothy grin of his and said: "Sure, Alphonse, have a blast!"

"Thanks. I'll see you shortly."

Avery and I found a conference room. We shut the door, and I set up my laptop. The two of us huddled around it. And I began.

"OK, now suppose I have a function that looks like `getSuitsOverdueForInspection`:

```
public class SuitInventory {
  private List<MensSuit> mensSuits = new ArrayList<MensSuit>();
  private List<WomensSuit> womensSuits = new ArrayList<WomensSuit>();

  public void getSuitsOverdueForInspection(List<Suit> overdueSuits) {
    for (Suit s : mensSuits)
      if (s.overdueForInspection())
        overdueSuits.add(s);

    for (Suit s : womensSuits)
      if (s.overdueForInspection())
```

```
        overdueSuits.add(s);
  }
}
```

"Ack!" said Avery. "Duplicate code! You have to refactor that!" So Avery grabbed the keyboard and reduced the two loops into a single loop:

```
  public void getSuitsOverdueForInspection(List<Suit> overdueSuits) {
    getSuitsOverdueForInspectionFromList(mensSuits, overdueSuits);
    getSuitsOverdueForInspectionFromList(womensSuits, overdueSuits);
  }

  private void getSuitsOverdueForInspectionFromList(
    List<Suit> suitList, List<Suit> overdueSuits) {
    for (Suit s : suitList)
      if (s.overdueForInspection())
        overdueSuits.add(s);
  }
```

I stopped him right there. "Yes, Avery, Right. But notice that this doesn't compile!"

Avery looked carefully at the screen and noticed that the two calls to `getSuitsOverdueFor-InspectionFromList` were complaining about the type of the first argument.

"Right!" He said. "This is just what we talked about on Friday. A list of derivatives, like `List<MensSuit>` cannot be passed as a List of base classes like `List<Suit>`. And we know what to do about this."

So Avery continued typing, fixing the compile errors as follows:

```
  private void getSuitsOverdueForInspectionFromList(
    List<? extends Suit> suitList, List<Suit> overdueSuits) {
    for (Suit s : suitList)
      if (s.overdueForInspection())
        overdueSuits.add(s);
  }
```

"Precisely!" I said. "This makes the `getSuitsOverdueForInspectionFromList` method more generic than either of the two earlier loops. So in that sense `? extends Suit` is more generic than `Suit`. However, I have a problem. I would like to call the first function, `getSuitsOverdueForInspection`, from a number of different places. Let me show you. Let's say that I have a `SuitInspector` class like this:"

```
public class SuitInspector {
  private SuitInventory inventory = new SuitInventory();

  public void inspectSuits() {
    List<Suit> overdueSuits = new ArrayList<Suit>();
    inventory.getSuitsOverdueForInspection(overdueSuits);
    for (Suit overdueSuit : overdueSuits) {
      overdueSuit.inspect();
    }
  }
}
```

Avery looked as this for a second or two and said: "OK, This class just finds all the suits that are overdue for inspection, and inspects them."

"Right!" I said. "But I also have a `GeneralInspector` class that looks like this:"

```
public class GeneralInspector {
  private SuitInventory suits = new SuitInventory();
  private ValveInventory valves = new ValveInventory();

  public List<Object> getOverdueItems() {
    List<Object> overdueItems = new ArrayList<Object>();
    suits.getSuitsOverdueForInspection(overdueItems);
    valves.getValvesOverdueForInspection(overdueItems);
    return overdueItems;
  }

  public void PrintOverdueInspectionReport() {
    List<Object> overdueItems = getOverdueItems();
    for (Object overdueItem : overdueItems) {
      System.out.printf("%s is overdue\n", overdueItem);
    }
  }
}
```

Again, Avery stared and the screen for a few seconds and then said: "OK, sure. you are just gathering up the overdue Suit and Valve objects and then printing them. Hay! What's that `printf` statement?"

"That?" I said with a smirk. "That's actually pretty retro. It's sort of like the old C `printf` statement. Maybe you can talk about that one tomorrow. Or maybe I will. For now, the more interesting thing is that the two `getXXXOverdueForInspection` lines don't compile."

"Well of course they don't" Avery Sneered. "You are passing a `List<Object>` to them, when they respectively take `List<Suit>` and `List<Valve>`".

He was falling into my trap. Good. "So how should we fix this?"

"You need to change the arguments of the getXXXOverdueForInspection functions to take `List<? extends Object>`." And Avery grabbed the keyboard and began to type.

Before he was finished I said: "I'd like to point out that widening the type of the argument to `getSuitsOverdueforInspection` is a bit strange. It should be taking a `List<? extends Suit>`!" But Avery pretended not to hear as he pounded out the following code:

```
  public void getSuitsOverdueForInspection(
    List<? extends Object> overdueSuits) {
    getSuitsOverdueForInspectionFromList(mensSuits, overdueSuits); // broken
    getSuitsOverdueForInspectionFromList(womensSuits, overdueSuits); // broken
  }
```

"Oh no!" Avery shrieked. "This made GeneralInventory compile, but broke the calls to `getSuits-OverdueForInspectionFromList`.

"Right!" I said, because you can't pass a `List<? extends Object>` to an argument that expects a `List<Suit>`."

"OK." Avery said. "So we just have to widen the argument of `getSuitsOverdueForInspection-FromList` like this:"

```
  private void getSuitsOverdueForInspectionFromList(
    List<? extends Suit> suitList, List<? extends Object> overdueSuits) {
    for (Suit s : suitList)
      if (s.overdueForInspection())
        overdueSuits.add(s); // broken
  }
```

"Aarrrgghh!" Avery cried. Now the 'add' statement is broken."

"Right again!" I said. "That's because you can't add something to a list of unknown type. `?`

`extends Object` is a truly unknown type!"

Avery looked near panic. I could see the fear of casts and `instanceof`s in his face. But then he seemed to suddenly realize that I had sprung a trap on him. He visibly relaxed; put a sheepish smile on his face, and said: "OK, so I'm sure you have a solution to this conundrum."

"Indeed I do!"

"Play on, Maestro! Play on!"

"It turns out, my dear Avery, that you can widen the type of a list by using `? extends X`, only if you plan on *reading* from that list. If, however, you plan on writing to that list; as our `getSuitsOverdue-ForInspection` functions do, then you have to use a different form. You have to use `<? super X>`.

Avery drew breath in an audible hiss.

I continued: "Allow me to show you how this works:" And I made the following changes to the `SuitInventory` class.

```
public void getSuitsOverdueForInspection(List<? super Suit> overdueSuits) {
  getSuitsOverdueForInspectionFromList(mensSuits, overdueSuits);
  getSuitsOverdueForInspectionFromList(womensSuits, overdueSuits);
}

private void getSuitsOverdueForInspectionFromList(
  List<? extends Suit> suitList,
  List<? super Suit> overdueSuits) {
  for (Suit s : suitList)
    if (s.overdueForInspection())
      overdueSuits.add(s);
}
```

"This settles the issue." I said. "Now the functions take a `List<? super Suit>`, which is a list of unknown types that are known to be lists of something *no more derived* than `Suit`."

"No more derived..." echoed Avery. "That makes my head hurt."

"Think of it this way, Avery." When you put an object into a list, all you care about is that the object is of a type that is compatible with the type held by the list. So you want the list to take the type of that object, or one of the superclasses of that object."

"Ow!, Ow! Ow!. My head is hurting"

"On the other hand, when you *read* from a list, you want the type you are reading to be the type contained by the list, or a derivative of that type."

"Ow!"

"So when you read from a list, you use `<? extends X>`, but when you write to a list you use `<? super X>`. See?

"Ow!"

"Don't you think that's cool?"

"Ow!"

This was fun! Avery is usually the superior one, and he just couldn't keep up with me on this one. I'm sure it would be my turn next. But that would be fun too.

To be continued...
