JAVA GENERICS

Prof. Chris Jermaine
cmj4@cs.rice.edu

Java Is Strongly Typed

- Or is it?
- I can write:

```
Double myDouble = new Double (12.3);
ArrayList myList = new ArrayList ();
myList.add (myDouble);
Integer myInt = (Integer) myList.get (0);
// compiles OK!
```

- What does this do?
  - — Creates an ArrayList, which is a data structure holding type Object
  - — Puts a Double in (OK since a Double in an object)
  - — Takes that object out, tries to cast it into an Integer
  - — Program barfs at runtime... in general, leads to lots o' bugs!
- This was the official "Java Programming Style" (TM) in 1998

Why'd People Code This Way?

- Everyone justifiably avoids copying/pasting code
- Want to write ArrayList (or other class) just once
  - — And have it work with any type you send in
- Only way to do this way back when...
  - — Is to hard code in the class the "highest" type that could ever go in
  - — For a generic container this is "Object"
  - — That way, you can put anything in
  - — Then when you take it out, "downcast" to get original object back
- So despite evils of coding this way...
  - — Was better than the alternative of copying/pasting code

Early On, Java Designers Realized Not Good

- Thus, idea of a “generic” type was introduced
- In modern Java, can now say:

```
Double myDouble = new Double (12.3);
ArrayList <Double> myList = new ArrayList <Double> ();
myList.add (myDouble);
Integer myInt = (Integer) myList.get (0);
// won’t even compile!
```

- What’s the diff?
  - — Everything is almost exactly the same
  - — But now, can tell the compiler that ArrayList holds only objects of type Double
  - — Done via the “diamond” notation
  - — As such, the compiler can catch the error in the last line

This Might Seem Like a Small Change

- But casting was a real problem in old-school Java
- In our trivial example, clear what is in the container
- But in general case, we have no idea
- People used type “Object” all over the place
- Practiced “cast and pray” programming
  - — With generics, no need to ever cast again*
  - — Except to do numeric conversions

*well, except for a few little problems...
(more next time)

Historical Note

- Generics (aka “templates” in other languages)
  - — Have been around for a long time
- Are a key feature of Ada, for example
- Many people vehemently argue that Java generics are unique
  - — In particular, Java-heads seem really insecure wrt C++ templates
  - — Java generics **are** unique in the sense that every realization of idea has its quirks
- But the goal is always the same, regardless of language
  - — Allow people to easily re-use code (no copying and pasting) in a type-safe way

Let's Examine (Somewhat) Complex Example

- Fortunately, we can add complexity piece by piece
  - — So hopefully not overwhelming
- Key thing: my example is NOT restricted to simple containers
  - — In practice, most use of generics happens with simple containers
  - — Things like using an "ArrayList <Double>" instead of an array list of "Object"s
  - — I'll try to show you that the generic mechanism can be more powerful!

Let's Examine (Somewhat) Complex Example

- Say we want to have a generic set of objects
- And we want to associate with that set...
  - A method that computes the sum over all of the objects in the set
  - We want this code to be trivially reusable for **any** type where sum makes sense
  - How would you accomplish this?
  - Use generics!

Our First Generic Interface

```
interface ISummable <T> {
    void addYourselfTo (T addToMe);
}
```

- OK, so what's the deal here?
- We have defined an interface called "ISummable"
  - — We've seen plenty of interfaces before!
- The only difference is that **this interface is parameterized**
  - — Something can implement "ISummable of type T" only if it has the ability to add itself to an object of type T
  - — Where T is some (any) arbitrary type
  - — Ex: something can implement "ISummable of type Double" only if it has the ability to add itself to an object of type Double

Now Let's Use ISummable

- Please note I'm gonna skip the abstract class only for clarity
  - — Don't want to make generics more puzzling than they already are

```
class ChrisInt implements ISummable <ChrisInt> {
    private Integer myVal;

    public ChrisInt (int val) {myVal = val;}

    public void addYourselfTo (ChrisInt addToMe) {
        addToMe.myVal += myVal;
    }
}
```

- What's going on here?
  - — You're telling the world that ChrisInt can sum itself with ChrisInt objects
  - — And the compiler will make sure you are telling the truth!
  - — It'll check to see you have method "public void addYourselfTo (ChrisInt)"

We Can Build This Up Further

- Let's say we now want our set that can automatically sum itself
- Just have:

```
class SummableSet <T extends ISummable <T>> {

}
```

- What is this saying?
  - — Class "SummableSet" is paramaeterized on a type
  - — That type is "<T extends ISummable <T>>"
  - — This will match any type that is declared using

```
class SomeTypeHere extends ISummable <SomeTypeHere> {}
```

  - — And for this declaration to hold, the class must provide a method of the form

```
public void addYourselfTo (SomeTypeHere foo) {}
```

Implementing SummableSet

```
class SummableSet <T extends ISummable <T>> {
    ArrayList <T> myData = new ArrayList <T> ();

    void addItem (T addMe) {
        myData.add (addMe);
    }

    T getSum () {
        T sum = null;
        for (T curItem : myData) {
            if (sum != null)
                curItem.addYourselfTo (sum);
            else
                sum = curItem;
        }
        return sum;
    }
}
```

Using SummableSet

- Easy!

```
SummableSet <ChrisInt> foo = new SummableSet <ChrisInt> ();
...
ChrisInt result = foo.getSum ();
```

- So what do we get from this?
  - — We can now use “SummableSet” over any type that can be added to itself
  - — And the compiler checks everything, so there’s no possibility of type errors
  - — In the “old-school” Java way of doing things, there would have been several casts
  - — Using generics, there is little room for error!

Using SummableSet

- Can easily create a SummableSet of polynomials...
  - — First define the “Polynomial” class

```
class Polynomial implements ISummable <Polynomial> {

  private ArrayList <Double> coefs = new ArrayList<Double> ();
  ...

  public void addYourselfTo (Polynomial addToMe) {
    int i = 0;
    // this nasty code adds my coefs to his
    for (; i < coefs.size () && i < addToMe.coefs.size (); i++) {
      addToMe.coefs.set (i, addToMe.coefs.get (i) + coefs.get (i));
    }
    // and then inserts any additional coefs to the other one
    for (; i < coefs.size (); i++) {
      addToMe.coefs.add (coefs.get (i));
    }
  }
}
```

Using SummableSet

- Can easily create a SummableSet of polynomials...
  - — First define the “Polynomial” class

```
class Polynomial implements ISummable <Polynomial> {
```

- And then we’re good to go!

```
SummableSet <Polynomial> foo =
                    new SummableSet <Polynomial> ();
```

A Few Thoughts

- As intimated before...
  - — Much use of generics is for simple containers
  - — So you get class definitions like:

```
class Stack <T> { }
```

  - — This is fine and useful
- But you are using the generic mechanism at its full power when
  - — The type argument implements some interface
  - — Because then you can implement **algorithms** in a generic way
  - — You abstract out the ops you need from the data being operated on by the alg
  - — Put them in an interface
  - — Don’t worry about anything else
- This is exactly what we did with “SummableSet”

How Does This Differ From “Classic” Inher.?

- Consider what our example looks like w/o generics:

```
class SummableSet {
    ArrayList <ISummable> myData =
        new ArrayList <ISummable> ();

    void addItem (ISummable addMe) {
        myData.add (addMe);
    }
...
}
```

- What are the key diffs?

How Does This Differ From “Classic” Inher.?

- Consider what our example looks like w/o generics:

```
class SummableSet {
    ArrayList <ISummable> myData =
        new ArrayList <ISummable> ();

    void addItem (ISummable addMe) {
        myData.add (addMe);
    }
...
}
```

- What are the key diffs?
  - — Someone is free to put *anything* implementing ISummable into SummableSet
  - — Can mix all sorts of different types in there
  - — How to deal with “addYourselfTo” to make sure we don’t add mismatched types?
  - — Well, in each implementation, would need to cast to verify correctness...

How Does This Differ From “Classic” Inher.?

```
class ChrisInt implements ISummable {
    private Integer myVal;

    public ChrisInt (int val) {myVal = val;}

    public void addYourselfTo (ISummable addToMe) {
        ChrisInt temp = (ChrisInt) addToMe;
        temp.myVal += myVal;
    }
}
```

- Note the cast: could fail at runtime! Really bad!
- With generics, compiler prevents any of these errors

Questions?