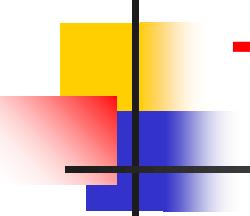


Simple Generics

Corky Cartwright

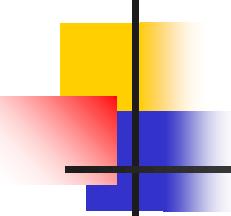
Stephen Wong

Department of Computer Science
Rice University



Today's goals

- .Develop the notion and syntax to represent generic typing in Java, also called “parametric polymorphism”.
- .Only cover the basic notions—advanced generics will be left for Comp310.
 - .Comp310 Java generics Resources page covers basic and advanced notions:
<http://www.clear.rice.edu/comp310/JavaResources/generics/>



Example: Boxes of things

```
public class BoxOfInt {  
    private int data;  
  
    public BoxOfInt(int data) {  
        this.data = data;  
    }  
  
    public int getData() {  
        return data;  
    }  
}
```

```
public class BoxOfStr {  
    private String data;  
  
    public BoxOfStr(String data) {  
        this.data = data;  
    }  
  
    public String getData() {  
        return data;  
    }  
}
```

```
public class BoxOfMyStuff {  
    private MyStuff data;  
  
    public MyStuff(MyStuff data) {  
        this.data = data;  
    }  
  
    public MyStuff getData() {  
        return data;  
    }  
}
```

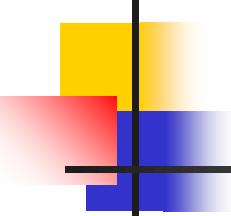
What's the difference? → *Not much!*

Key Feature: *The code does not depend on the type of the data!*

Re-write the Box code for a “generic” data type

```
public class Box<T> {  
    private T data;  
  
    public Box(T data) {  
        this.data = data;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

- **T** is a specific but as-yet, unspecified type. **T** will be specified when an instance of this class is created.
- **T** tells the Java compiler to make sure that whatever **T** is, that it be consistent throughout its usage. → “type-safe”
- *There is nothing special about using the letter “T” here!* Convention is to use single letters, but you can use whole words if desired.



Using a generic class

```
// Specify T when instantiating a Box object
Box<Integer> intBox = new Box<Integer>(42);
Box<String> strBox = new Box<String>("Yahoo!");
Box<MyStuff> myStuffBox = new Box<MyStuff>(new MyStuff());

int x = intBox.getData();
String s = strBox.getData();
MyStuff ms = myStuffBox.getData();

// The following will cause a compiler error!
String s = intBox.getData();
Box<Integer> b = new Box<Integer>("Oh dear!");
```

- Notice how **T** is specified when the `Box<T>` object is instantiated. This sets the type of **T** for that *object*.
- The *compiler* will flag an error if **T** is used inconsistently for that object.

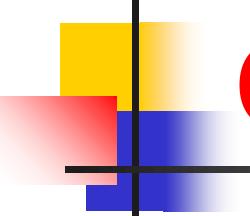
Specifying more than one generic parameter

- Simply separate the different generic type parameters by a comma:

```
public class Dyad<F, S> {  
    private F first;  
    private S second;  
  
    public Dyad(F first, S second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public F getFirst() { return first; }  
    public void setFirst(F first) {  
        this.first = first;  
    }  
  
    public S getSecond() { return second; }  
    public void setSecond(S second) {  
        this.second = second;  
    }  
}
```

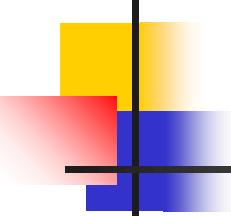
```
// Usage:  
  
Dyad<Integer, String> intStrPair =  
    new Dyad<Integer, String>(42,  
                             "Hello world!");  
  
Integer i = intStrPair.getFirst();  
  
String s = intStrPair.getSecond();  
  
intStrPair.setFirst(-99);  
  
intStrPair.setSecond("Bye bye!");
```

Generics in the Java Collections Framework



The Java Collections Framework is a set of classes and interfaces that represent groups of objects. Because of this, most use generic type specifications for the data that they hold.

- Java Collections Framework references:
 - <http://download.oracle.com/javase/tutorial/collections/index.html>
 - <http://java.sun.com/developer/onlineTraining/collections/Collection.html>
- Useful types
 - **Collection<E>** - a collection of objects of type `E`, the superclass of all single-element type Collection types.
 - **Set<E>** - an unordered set of objects of type `E`.
 - **List<E>** - an ordered set of objects of type `E`.
 - **Vector<E>** - an auto-resizing array of objects of type `E`.
 - **Map<K,V>** - a dictionary that maps a key of type `K` to a value of type `V`.



Parameterized methods

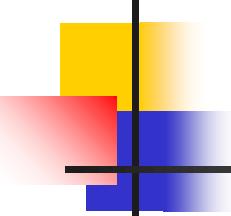
Sometimes, one needs extra generic parameters just for a specific method. In that case, it is possible to specify extra generic parameters for an individual method.

Suppose we had a algorithm to process the contents of a `Box<T>`, where we specify both the type data the algorithm works on, `T`, and the return type of its processing, `R`:

```
public abstract class BoxAlgo<T,R> {  
    abstract public R process(T data);  
}
```

We would add the following method to `Box<T>`, specifying the extra return type parameter the `BoxAlgo<T,R>` needs:

```
public <R> R runAlgo(BoxAlgo<T,R> algo) {  
    return algo.process(data);  
}
```



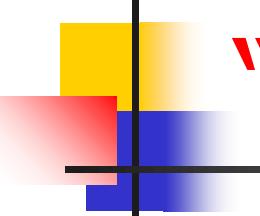
Immutable Generic List

Please download and open the DrJava project in [lec_29.zip](#).

Things to note:

- The type of the data held by the list is now determined by the generic parameter `E`.
- The visitor to the list requires 3 generic type parameters, one for the type of data, one for the return type and one for the input parameter type.
- The `SumIntListVisitor` contains no casts of the recursive result. This is because the visitor defines a specific return type, not the generalized `Object` type.
- Javadocs allows you to add the generic type parameters as a documented “parameter” with the following syntax:

```
@param <X> description of type X
```



“Gotcha’s” in generics

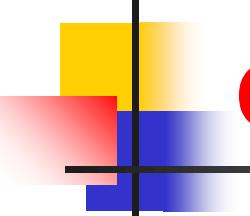
- If type `A` is a superclass of type `B`, `Box<A>` is NOT a superclass of `Box`. That is, the following will NOT compile:

```
Box<Number> boxOfNumber = new Box<Integer>(42);
```

- The empty list of `List<E>`, i.e. `EmptyList<E>`, cannot be defined as a singleton because `E` is a variant and differs from one list to another.
- In our simplistic use of generics, an algorithm on a `List<E>`, i.e. a visitor to it, `IVisitor<E, R, P>`, must be defined on the exact type of its host list, not a sub-type. That is, the following will NOT work:

```
List<Number> myNumList = ...; // some instantiation  
myNumList.accept(new IVisitor<Integer, AReturnType, AParamType>() {...});
```

This issue gets fixed with more advanced generics techniques.



“Gotcha’s” in generics, continued...

- JUnit tests cannot tell what a generic returned type is—the compiler will think that it is of type `Object`. Using a local variable of the correct type to hold the actual and expected values, will get around this problem.
- Javadoc comments are essentially HTML code, which means that the Javadoc processor thinks that “`<`” and “`>`” are part of HTML tags. When writing “`<`” or “`>`” in the body of a comment, use the HTML code for those characters: `<` and `>` respectively.