13 An Introduction to Generics

Topics

- Why Generics?
- Declaring a Generic Class
  - "Primitive" Limitation
- Constrained Generics
- Declaring a Generic Method

Generics

- Included in Java's latest release
- Problem with typecasting:
  - Downcasting is a potential hotspot for *ClassCastException*
  - Makes our codes wordier
  - Less readable
  - Destroys benefits of a strongly typed language
  - Example: *ArrayList* object

```
String myString = (String) myArrayList.get(0);
```

Generics

- Why generics?
  - Solve problem with typecasting
- Benefits:
  - Allow a single class to work with a wide variety of types
  - Natural way of eliminating the need for casting
  - Preserves benefits of type checking
  - Example: *ArrayList* object

```
//myArrayList is a generic object
String myString = myArrayList.get(0);
```

Generics

- Caution:

```
Integer data = myArrayList.get(0);
```

  - Removal of downcasting doesn't mean that you could assign anything to the return value of the *get* method and do away with typecasting altogether
  - Assigning anything else besides a *String* to the output of the get method will cause a compile time type mismatch

```
found: java.lang.String
required: java.lang.Integer
```

Generics

```
1 //Code fragment
2 ArrayList<String> genArrList =
3                                 new ArrayList<String>();
4 genArrList.add("A generic string");
5 String myString = genArrList.get(0);
6 //int myInt = genArrList.get();
7 JoptionPane.showMessageDialog(this, myString);
```

Declaring a Generic Class

- For the previous code fragment to work, we should have defined a generic version of the *ArrayList* class

- Java's newest version already provides users with generic versions of all Java *Collection* classes

Declaring a Generic Class

```
1  class BasicGeneric<A> {
2      private A data;
3      public BasicGeneric(A data) {
4          this.data = data;
5      }
6      public A getData() {
7          return data;
8      }
9  }
10 //continued...
```

Declaring a Generic Class

```
11 public class GenSample {
12    public String method(String input) {
13       String data1 = input;
14       BasicGeneric<String> basicGeneric = new
15                              BasicGeneric<String>(data1);
16       String data2 = basicGeneric.getData();
17       return data2;
18    }
19 //continued...
```

Declaring a Generic Class

```
20    public Integer method(int input) {
21        Integer data1 = new Integer(input);
22        BasicGeneric <Integer> basicGeneric = new
23                              BasicGeneric <Integer>(data1);
24        Integer data2 = basicGeneric.getData();
25        return data2;
26    }
27 //continued...
```

Declaring a Generic Class

```
20    public static void main(String args[]) {
21        GenSample sample = new GenSample();
22        System.out.println(sample.method(
23                                "Some generic data"));
24        System.out.println(sample.method(1234));
25    }
26 }
```

Declaring a Generic Class

- Declaration of the BasicGeneric class:

```
class BasicGeneric<A>
```

  - Contains type parameter: `<A>`
  - Indicates that the class declared is a generic class
  - Class does not work with any specific reference type

- Declaration of field:

```
private A data;
```

  - The field *data* is of generic type, depending on the data type that the *BasicGeneric* object was designed to work with

Declaring a Generic Class

- Declaring an instance of the class
  - Must specify the reference type to work with
  - Examples:

```
BasicGeneric<String> basicGeneric = new
                    BasicGeneric<String>(data1);
```

    - Class works with variables of type *String*

```
BasicGeneric<Integer> basicGeneric = new
                    BasicGeneric<Integer>(data1);
```

    - Class works with variables of type *Integer*

Declaring a Generic Class

- Declaration of the *getData* method:

```
public A getData() {
    return data;
}
```

  - Returns a value of type A, a generic type
  - The method will have a runtime data type
  - After you declare an object of type *BasicGeneric*, A is bound to a specific data type

Declaring a Generic Class

- Instances of the *BasicGeneric* class

```
BasicGeneric<String> basicGeneric = new
                        BasicGeneric<String>(data1);
String data2 = basicGeneric.getData();
```

  - *basicGeneric* is bound to *String* type
  - No need to typecast

```
BasicGeneric<Integer> basicGeneric = new
                        BasicGeneric<Integer>(data1);
Integer data2 = basicGeneric.getData();
```

  - *basicGeneric* is bound to *Integer* type
  - No need to typecast

Generics: "Primitive" Limitation

- Java generic types are restricted to reference types and won't work with primitive data types
  - Example:

```
BasicGeneric<int> basicGeneric = new
                        BasicGeneric<int>(data1);
```

- Solution:
  - Wrap primitive types first
  - Can use wrapper types as arguments to a generic type

Compiling Generics

- To compile using JDK (v. 1.5.0):

```
javac -version -source "1.5" -sourcepath src -d
  classes src/SwapClass.java
```

  where

  - *src* refers to the location of the java source code
  - *class* refers to the location where the class file will be stored
  - Example:

```
javac -version -source "1.5" -sourcepath c:\temp
 -d c:\temp c:/temp/MyFile.java
```

Constrained Generics

- Preceding example:
  - Type parameters of class *BasicGeneric* can be of any reference data type
- May want to restrict the potential type instantiations of a generic class
  - Can limit the set of possible type arguments to subtypes of a given type bound

Constrained Generics

- Limiting type instantiations of a class
  - Use the *extends* keyword in type parameter

```
class ClassName <ParameterName extends ParentClass>
```

  - Example: generic ScrollPane class
    - Template for an ordinary *Container* decorated with scrolling functionality
    - Runtime type of an instance of this class will often be a subclass of *Container*
    - The static or general type is *Container*

Constrained Generics

```
1  class ScrollPane<MyPane extends Container> {
2      ...
3  }
4  class TestScrollPane {
5      public static void main(String args[]) {
6          ScrollPane<Panel> scrollPane1 =
7                                  new ScrollPane<Panel>();
8          // The next statement is illegal
9          ScrollPane<Button> scrollPane2 =
10                                 new ScrollPane<Button>();
11     }
12 }
```

Constrained Generics

- Gives added static type checking
  - Guarantee that every instantiation of the generic type adheres to assigned bounds
  - Can safely call any methods found in the object's static type
- No explicit bound on the parameter
  - Default bound is *Object*
  - An instance can't invoke methods that don't appear in the *Object* class

Declaring a Generic Method

- Java also allows us to declare a generic method
- Generic Method
  - Polymorphic methods
  - Methods parameterized by type
- Why generic method?
  - Type dependencies between the arguments and return value are naturally generic
  - But the generic nature change from method call to method call rather than class-level type information

Declaring a Generic Method

```
1 class Utilities {
2     /* T implicitly extends Object */
3     public static <T> ArrayList<T> make(T first) {
4         return new ArrayList<T>(first);
5     }
6 }
```

Declaring a Generic Method

- Java also uses a type-inference mechanism
  - Automatically infers the types of polymorphic methods based on the types of arguments
  - Lessens wordiness and complexity of a method invocation

- To construct a new instance of *ArrayList<Integer>*, we would simply have the following statement:

```
Utilities.make(Integer(0));
```

Summary

- Why Generics?
- Declaring a Generic Class

```
class ClassName<TypeParameter> {
    ...
}
```

  - "Primitive" Limitation

Summary

- Constrained Generics

```
class ClassName<ParameterName extends ParentClass>
```

- Declaring a Generic Method
  - Example:

```
public static <T> ArrayList<T> make(T first) {
    return new ArrayList<T>(first);
}
```