Using Java generics

Lecture 17

Methods for writing general-purpose code

- **Polymorphism**: a method that takes a base class object as an argument, and then uses that method with any class derived from that base class. Now your method is more *general* and can be used in more places. Anything but a **final** class can be extended, so this flexibility is automatic.
- Sometimes, being constrained to a single hierarchy is too limiting. If a method argument is an **interface** instead of a class, the limitations are loosened to include anything that implements the interface.
- Sometimes even an interface is too restrictive. An interface still requires that your code work with that particular interface. You could write even more general code if you could say that your code works with "**some unspecified type**".

Concept of generics

- Generics implement the concept *of parameterized types*, which allow multiple types.
- The term "generic" means "pertaining or appropriate to large groups of classes."
- Generics allow the programmer the greatest amount of expressiveness possible when writing classes or methods

Generic **classes** Example 1: Tuple library

- One of the things you often want to do is return multiple objects from a method call.
- The **return** statement only allows you to specify a single object, so the answer is to create an object that holds the multiple objects that you want to return.
- You can write a special class every time you encounter the situation, but with generics it’s possible to solve the problem once and save yourself the effort in the future. At the same time, you are ensuring compile-time type safety.

Generic classes example 1: TwoTuple

Read-only object

```
public class TwoTuple<A,B> {
      public final A first;
      public final B second;

      public TwoTuple(A a, B b) {
            first = a; second = b;
      }

      public String toString() {
            return "(" + first + ", " + second + ")";
}
```

Generic classes example 1: ThreeTuple

```
public class ThreeTuple<A,B,C> extends TwoTuple<A,B>
{
      public final C third;
      public ThreeTuple(A a, B b, C c) {
            super(a, b);
            third = c;
      }

      public String toString() {
            return "(" + first + ", " + second + ", " +
                                          third +")";
      }
}
```

Generic classes example 1: FourTuple

```
public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C>
{
	public final D fourth;
	public FourTuple(A a, B b, C c, D d) {
		super(a, b, c);
		fourth = d;
	}

	public String toString() {
		return "(" + first + ", " + second + ", " +
					third + ", " + fourth + ")";
	}
}
```

Generic classes example 1: TupleTest

```java
class Amphibian {}
class Vehicle {}

public class TupleTest {
	static TwoTuple<String,Integer> f(String s, Integer i) {
		return new TwoTuple<String,Integer>(s, i);
	}

	static ThreeTuple<Amphibian,String,Integer> g(Amphibian a,
				String s, Integer i) {
		return new ThreeTuple<Amphibian, String, Integer>(new
						Amphibian(),s, i);
	}

	public static void main(String[] args) {
		TwoTuple<String,Integer> ttsi = f("hi",47);
		System.out.println(ttsi);
		// ttsi.first = "there"; // Compile error: final
		System.out.println(g(new Amphibian(), "hi",47));
	}
}
```

```
(hi, 47)
(generics.Amphibian@93dcd, hi, 47)
```

Generic classes example 2: RandomList

```java
import java.util.*;
public class RandomList<T> {
	private ArrayList<T> storage = new ArrayList<T>();
	private Random rand = new Random(47);
	public void add(T item) { storage.add(item); }
	public T select() {
		return storage.get(rand.nextInt(storage.size()));
	}

	public static void main(String[] args) {
		RandomList<String> rs = new RandomList<String>();

		for(String s: ("The quick brown fox jumped over " +
			          "the lazy brown dog").split(" "))
			rs.add(s);

		for(int i = 0; i < 11; i++)
			System.out.print(rs.select() + " ");
		}
}
```

Generic **methods**

- A generic method allows the method to vary independently of the class.
- As a guideline, you should use generic methods rather than the generic classes.
- In addition, if a method is **static**, it has no access to the generic type parameters of the class, so if it needs to use genericity it must be a generic method.

Generic method syntax

To define a generic method, you simply place a generic parameter list before the return value:

```
public <T> void f(T x)
```

Generic methods example 1: print type

```
public class GenericMethods {
	public <T> void f(T x) {
		System.out.println(x.getClass().getName());
	}

	public static void main(String[] args) {
		GenericMethods gm = new GenericMethods();
		gm.f("");
		gm.f(1);
		gm.f(1.0);
		gm.f(1.0F);
		gm.f('c');
		gm.f(gm);
	}
}
```

```
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
generics.GenericMethods
```

We expect that generic class knows the actual type of its parameters

```java
import java.util.*;

public class LostTypeInfo {
	public static void main (String [] args)
	{
		TwoTuple <String,Integer> t1=new TwoTuple
					<String,Integer>("name",1);
		System.out.println(Arrays.toString(
			t1.getClass().getTypeParameters()));

		TwoTuple <Integer,Integer> t2=new TwoTuple
				<Integer,Integer>(1,2);
		System.out.println(Arrays.toString(
			t2.getClass().getTypeParameters()));
	}
}
```

```
[A, B]
[A, B]
```

The cold truth is:

There’s no information about generic parameter types available inside generic code during run time.

Erasure at run time

- Java generics are implemented using ***erasure***.
- This means that any specific type information is erased when you use a generic.
- Inside the generic, the only thing that you know is that you're using an Object. So **List<String>** and **List< Integer**> *are*, in fact, the same type at run time. Both forms are "erased" to their *raw type*, **List**

C++ templates: only look similar

```
//: generics/Templates.cpp
#include <iostream>
using namespace std;
template<class T> class Manipulator {
        T obj;
        public:
        Manipulator(T x) { obj = x; }
        void manipulate() { obj.f(); }
};

class HasF {
        public:
        void f() { cout << "HasF::f()" << endl; }
};
int main() {
        HasF hf;
        Manipulator<HasF> manipulator(hf);
        manipulator.manipulate();
}
```

```
HasF::f()
```

C++ approach to generics

- **manipulate( )** method calls a method f**( )** on **obj**. How can it know that the f**( )** method exists for the type parameter T?
- The C++ compiler checks when you instantiate the template, so at the point of instantiation of **Manipulator <HasF>**, it sees that **HasF** has a method f**( )**.
- If it were not the case, you'd get a compile-time error, and thus type safety is preserved.

Trying it with Java (1/2)

```
public class HasF {
     public void f() {
          System.out.println("HasF.f()");
     }
}
```

Trying it with Java (2/2)

```
// {CompileTimeError} (Won’t compile)
       class Manipulator <T> {
              private T obj;
              public Manipulator(T x) { obj = x; }
                              // Error: cannot find symbol: method f():
              public void manipulate() { obj.f(); }

              public static void main(String[] args) {
                     HasF hf = new HasF();
                     Manipulator<HasF> manipulator =
                            new Manipulator<HasF>(hf);
                     manipulator.manipulate();
              }
}
```

Java bounds

- Because of erasure, the Java compiler can’t map the requirement that **manipulate( )** must be able to call f**( )** on **obj** to the fact that **HasF** has a method f**( )**.
- In order to call f**( )**, we must assist the generic class by giving it a *bound* that tells the compiler to only accept types that conform to that bound.

Using type bounds

Extends means *extends* or *implements*

```
class Manipulator2<T extends HasF> {
      private T obj;
      public Manipulator2(T x) { obj = x; }
      public void manipulate() { obj.f(); }
}
```

The same can be done without generics

Programming to an interface: parameter of type HasF

```
class Manipulator3 {
      private HasF obj;
      public Manipulator3(HasF x) {
            obj = x;
      }
      public void manipulate() {
            obj.f();
      }
}
```

Bottom line

- Erasure reduces the "genericity" of generics. Generics are still useful in Java, just not as useful as they could be, and the reason is erasure.
- In an erasure-based implementation, generic types are treated as second-class types that cannot be used in some important contexts.
- The generic types are present only during static type checking (compilation), after which every generic type in the program is erased by replacing it with a non-generic upper bound.
- The reason: migration compatibility with the previous non-generic java code