COMP 1231 COMPUTER PROGRAMMING II

GENERICS

Topics

- Introduction to Generics
- Writing a Generic Class
- Passing Objects of a Generic Class to a Method
- Writing Generic Methods
- Constraining a Type Parameter in a Generic Class
- Defining Multiple Parameter Types
- Generics and Interfaces

Introduction to Generics

- A generic class or method is one whose definition uses a placeholder for one or more of the types it works with.
- The placeholder is really a type parameter.
- For a generic class, the actual type argument is specified when an object of the generic class is being instantiated.
- For a generic method, the compiler deduces the actual type argument from the type of data being passed to the method.

The ArrayList Class

The ArrayList class is generic: the definition of the class uses a type parameter for the type of the elements that will be stored in that collection

ArrayList<String> specifies a version of the generic ArrayList class that can hold String elements only

ArrayList<Integer> specifies a version of the generic ArrayList class that can hold Integer elements only

Instantiation and Use of a Generic Class

ArrayList<String> is used as if it was the name of any non-generic class:

```
ArrayList<String> myList = new
  ArrayList<String>();
myList.add("Java is fun");
String str = myList.get(0);
```

Writing a Generic Class

- Consider a “point” as a pair of coordinates *x* and *y*, where *x* and *y* may be of any one type (i.e. both are the same).

A Generic Point Class

```
class Point<T>      // T represents a type parameter
{
  private T x, y;
  public Point(T x, T y) //  Constructor
  {
     set(x, y);
  }
  public void set(T x, T y)
  {
      this.x = x;   this.y = y;
  }
  T getX(){ return x;}
  T getY(){ return y;}
  public String toString()
  {
      return "(" + x.toString() + ","
             + y.toString() + ")";
  }
}
```

Using a Generic Class

```
public class Test
{
   public static void main(String [] s)
   {
      Point<String> str = new Point<String>
         ("Anna", "Banana");
      System.out.println(str);
      Point<Number> pie = new Point<Number>
         (3.14, 2.71);
      System.out.println(pie);
    }
}
```

Program Output:

(Anna,Banana)

(3.14,2.71)

Reference Types and Generic Class Instantiation

- Only reference types can be used to declare or instantiate a generic class.

ArrayList<Integer> myIntList = new ArrayList<Integer>; // OK

ArrayList<int> myIntList = new ArrayList<int>; // Error

- Since int is not a reference type, it cannot be used to declare or instantiate a generic class. You must use the corresponding wrapper class to instantiate a generic class with a primitive type argument.

Autoboxing

- Automatic conversion of a primitive type to the corresponding wrapper type when it is used in a context where a reference type is required.

```
// Autoboxing converts int to Integer
Integer intObj = 35;
// Autoboxing converts double to Number
Point<Number> pie = new Point<Number>(3.14, 2.71);
```

Unboxing

- Automatic unwrapping of a wrapper type to give the corresponding primitive type when the wrapper type is used in a context that requires a primitive type.

```
// Unboxing converts Integer to int
int i = new Integer(34);
// AutoBoxing converts doubles 3.14, 2.71 to Double
Point<Double> pied = new Point<Double>(3.14, 2.71);
// p.getX() returns Double which is unboxed to double
double pied = p.getX();
```

Autoboxing, Unboxing, and Generics

Autoboxing and unboxing are useful with generics:

- Use wrapper types to instantiate generic classes that will work with primitive types

```
Point<Double> pied = new Point<Double>(3.14, 2.71);
```

- Take advantage of autoboxing to pass primitive types to generic methods

```
pied.set(3.141593, 2.71828);
```

- Take advantage of unboxing to receive values returned from generic methods

```
double pi = pied.getX();
```

Raw Types

- You can create an instance of a generic class without specifying the actual type argument.
- An object created in this manner is said to be of a raw type.

```
Point rawPoint = new Point("Anna", new Integer(26));
System.out.println(rawPoint);
```

Output:

(Anna, 26)

Raw Types and Casting

- The Object type is used for unspecified types in raw types.
- When using raw types, it is necessary for the programmer to keep track of types used and use casting

```
Point rawPoint = new Point(“Arthur Dent", new Integer(42));
System.out.println(rawPoint);
String name = (String)rawPoint.getX();      // Cast is needed
int age = (Integer)rawPoint.getY();          // Cast is needed
System.out.println(name);
System.out.println(age);
```

Commonly Used Type Parameters

Name	Usual Meaning
T	Used for a generic type.
S	Used for a generic type (with T for multiples)
E	Used to represent generic type of an element in a collection.

Generic Objects as Parameters

- Consider a method that returns the square length of a Point object with numeric coordinates.
  - Square length of Point(3, 4) is 3*3 + 4*4 = 25

```
static int sqLength(Point<Integer> p)
{
    int x = p.getX();
    int y = p.getY();
    return x*x + y*y;
}
```

- The method is called as in

```
int i = sqLength(new Point<Integer>(3, 4));
```

Generics as Parameters

sqLength(Point<Integer> p) will not work for other numeric types and associated wrappers: for example, it will not work with Double.

We want a generic version of sqLength that works for all subclasses of the Number class.

- Declaring the method parameter as Point<Number> works for Point<Number>, but not for any Point<T> where T is a subclass of Number

```
static double sqLength(Point<Number> p)
{
  double x = p.getX().doubleValue();
  double y = p.getY().doubleValue();
  return x*x + y*y;
}
```

Works for:

```
Point<Number> p = new Point<Number>(3,4);
System.out.println(sqLength(p));
```

Does not work for:

```
Point<Integer> p = new Point<Integer>(3,4);
System.out.println(sqLength(p));            // Error
```

Wildcard Parameters

- Generic type checking is very strict:
  Point<Number> references cannot accept Point<T> objects unless T is Number.
- A Point<Number> reference will not accept a Point<Integer> object, even though Integer is a subclass of Number.

- The wildcard type symbol ? stands for any generic type:
  Point<?> references will accept a Point<T> object for any type T.

Use of Wildcards

- A version using wildcards works for all subclasses of Number, but loses benefits of type checking, and requires casts!

```
static double sqLength(Point<?> p)
{
    Number n1 = (Number)p.getX();    // Needs cast to Number
    Number n2 = (Number)p.getY();
    double x = n1.doubleValue();
    double y = n2.doubleValue();
    return x*x + y*y;
}
```

Call as in

```
Point<Integer> p = new Point<Integer>(3,4);
System.out.println(sqLength(p));
```

Constraining Type Parameters

- Benefits of type checking can be regained by constraining the wildcard type to be a subclass of a specified class:

```
Point <?> p1;                          // Unconstrained wildcard
Point <? extends Number> p2;  // Constrained wildcard
```

p2 can accept a Point<T> object, where T is any type that extends Number.

Constraining Type Parameters

```
static double sqLength(Point<? extends Number> p)
{
    Number n1 = p.getX();
    Number n2 = p.getY();
    double x = n1.doubleValue(); // cast no longer needed
    double y = n2.doubleValue();
    return x*x + y*y;
}
```

Call as in:

```
Point<Integer> p = new Point<Integer>(3,4);
System.out.println(sqLength(p));
```

Defining Type Parameters

- The type parameter denoted by a wild card has no name.
- If a name for a type parameter is needed or desired, it can be defined in a clause included in the method header.
- The type definition clause goes just before the return type of the method.
- Defining a type parameter is useful if you want to use the same type for more than one method parameter, or for a local variable, or for the return type.

Defining Type Parameters

Using the same type for several method parameters:

```
static <T extends Number>
void doSomething(Point<T> arg1, Point<T> arg2)
{ … }
```

Using the name of the generic type for the return type of the method:

```
static <T extends Number>
Point<T> someFun(Point<T> arg1, Point<T> arg2)
{ … }
```

Constraining Type Parameters

```
class Point<T extends Number>      //  T constrained to a subclass of
   Number
{
  private T x, y;
  public Point(T x, T y)  {  this.x = x;  this.y = y;  }
  double sqLength()
  {   double x1 = x.doubleValue();
      double y1 = y.doubleValue();
      return x1*x1 + y1*y1;
  }
  T getX(){ return x;}
  T getY(){ return y;}
  public String toString()
  { return "(" +  x.toString() + ","  + y.toString() + ")";
  }
}
```

Type parameters can be constrained in Generic classes:

```
Point<Integer> p = new Point<Integer>(3,4);      //  Ok
System.out.println(p.sqLength());                //  Ok

Point<String> q = new Point<String>("Anna", "Banana");
//  Error, String is not a subclass of Number
```

Upper and Lower Bounds

- The constraint <T extends Number > establishes Number as an upper bound for T. The constraint says T may be any subclass of Number.

- A similar constraint <T super Integer> establishes Integer as a lower bound for T. The constraint says T may be any superclass of Integer.

Defining Multiple Type Parameters

- A generic class or method can have multiple type parameters:

  class MyClass<S, T>

  { … }

- Multiple type parameters can be constrained:

  class MyClass<S extends Number, T extends Date)

  { … }

A Class with Multiple Type Parameters

```
class Pair<T, S>
{
    private T first;
    private S second;
    public Pair(T x, S y)
    {
        first = x; second = y;
    }
    public T getFirst() { return first; }
    public S getSecond() { return second; }
}
```

Use of Multiple Type Parameters

```
import java.util.Date;
public class Test
{
   public  static void main(String [ ] args)
   {
      Pair<String, Date> p =
         new Pair<String, Date>("Joe", new Date());
      System.out.println(p.getFirst());
      System.out.println(p.getSecond());
   }
}
```

Generics and Interfaces

- Interfaces, like classes, can be generic.
- An example of a generic interface in the class libraries is

```
public interface Comparable<T>
{
    int compareTo(T obj);
}
```

- This interface is implemented by classes that need to compare their objects according to some natural order.

The Comparable Interface

```
public interface Comparable<T>
{
    int compareTo(T obj);
}
```

- The `compareTo` method:
  - returns a negative integer if the calling object is “less than” the other object
  - returns 0 if the calling object is “equal” to the other object
  - returns a positive integer if the calling object is “greater than” the other object

Implementing the Comparable Interface

```
class Employee implements Comparable<Employee>
{
  private int rank;
  private String name;
  public int compareTo(Employee e)
  {
    return this.rank - e.rank;
  }
  public Employee(String n, int r)
  { rank = r; name = n; }
  public String toString()
  { return name + " : " + rank; }
}
```

Comparing Employee Objects

```
Employee bigShot = new Employee("Maxwell Manager",
  10);
Employee littleShot = new Employee("Walter Worker", 1);
// want to show them in rank order
if (bigShot.compareTo(littleShot) > 0)
{
    System.out.println(bigShot);
    System.out.println(littleShot);
}
else
{
    System.out.println(littleShot);
    System.out.println(bigShot);
}
```

Type Parameters Implementing Interfaces

A type parameter can be constrained to a type implementing an interface:

```
public static <T extends Comparable<T>>
T greatest(T arg1, T arg2)
{
    if (arg1.compareTo(arg2) > 0)
        return arg1;
    else
        return arg2;
}
```

Type Parameters Implementing Interfaces

```
public  static void main(String [ ] args)
{
    Employee bigShot = new Employee(“Maxwell Manager", 10);
    Employee littleShot = new Employee(“Walter Worker", 1);
    Employee great = greatest(bigShot, littleShot);
    System.out.println(great);
}
```

This avoids the need to pass objects as interfaces and then cast the return type from the interface back to the type of the object