Generics

Generics

- Added to Java v5.0
- Generics = class and method definitions that include parameters for types
- Generic class – type parameter allows one to write code that applies to any class
- Ex. list of items of type T
    - when T=Double, it's a list of Doubles, etc.

Generics

- Classes and methods can have a type parameter (actually, more than one).
- The type parameter can be any reference (class) type.
- (started with Java version 5.0; in C++ as well but much restricted in Java)
- Generic class = parameterized class = class with a parameter for a type

Generics

- Once a parameterized class is compiled, it can be used like any other class.
  - However, the class type plugged in for the type parameter must be specified before it can be used in a program.
  - Doing this is said to *instantiate* the generic class.

```
Sample<String> stringSample = new Sample<String>();
```

Display 14.4 A Class Definition with a Type Parameter

```
public class Sample<T>
{
    private T data;

    public void setData(T newData)
    {
        data = newData;
    }

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

T is a parameter for a type.

Generics

- A class that is defined with a parameter for a type is called a generic class or a parameterized class
  - The type parameter is included in angular brackets after the class name in the class definition heading.
  - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter.
  - The type parameter can be used like other types used in the definition of a class.

What is Generics ?

- Collections can store Objects of any Type
- Generics restricts the Objects to be put in a collection
- Generics ease identification of runtime errors at compile time

How is Generics useful ?

Consider this code snippet

```
List v = new ArrayList();
v.add(new String("test"));
Integer i = (Integer) v.get(0);
    // Runtime error. Cannot cast from String to Integer
```

This error comes up only when we are executing the program and not during compile time.

How does Generics help ?

The previous snippet with Generics is

```
List<String> v = new ArrayList<String>();
v.add(new String("test"));
Integer i = v.get(0);
  // Compile time error. Converting String to Integer
```

- The compile time error occurs as we are trying to put a String and convert it to Integer on retrieval.
- Observe we don’t have to do an explicit cast when we invoke the get method.
- We can also use interfaces in Generics.

Generic methods

```
public class Utility {
  …
  public static <T> T getMidpoint ( T[] a ) {
        return a[ a.length/2 ];
  }
  public static <T> T getFirst ( T[] a ) {
        return a[0];
  }
  …
}
```

Generic methods

```
public class Utility {
  public static <T> T getMidpoint ( T[] a ) {
        return a[ a.length/2 ];
  }
  public static <T> T getFirst ( T[] a ) {
        return a[0];
  }
}

String  midString = Utility.<String>getMidPoint( b );
double  firstNumber = Utility.<Double>getFirst( c );
```

Generic (parameterized) classes

```
public class Sample<T> {
  private T  data;
  public void setData ( T newData ) {
        data = newData;
  }
  public T getData ( ) {
        return data;
  }
}
```

Generic (parameterized) classes

```
public class Sample<T> {
   private T  data;
   public void setData ( T newData ) {
          data = newData;
   }
   public T getData ( ) {
          return data;
   }
}

Sample<String>  sample= new Sample<String>();
sample.setData( “Hello” );
```

Generic class for ordered pairs

```
Pair<String>  secretPair
        = new Pair<String>( “Happy”, “Day” );

Pair<Integer>  pairOfDice
        = new Pair<Integer>(new Integer(2),
                            new Integer(3) );

Pet  aPet = new Pet();
Pet  bPet = new Pet();
Pair<Pet>  walkingPair = new Pair<Pet>( aPet, bPet );
```

Defining the ordered pair class

```
public class Pair<T> {
   private T  first;
   private T  second;
   public Pair ( ) {
            first = null;
            second = null;
   }
   public Pair ( T f, T s ) {
            first = f;
            second = s;
   }
   …
}
```

Defining the ordered pair class

```
public class Pair<T> {
   private T  first;
   private T  second;
   …
   public boolean equals ( Object other ) {
            if (other==null) return false;
            if (getClass() != other.getClass())  return false;
            Pair<T>  o = (Pair<T>)other;
            return first.equals(o.first) && second.equals(o.second);
   }
   …
}
```

Defining the ordered pair class

```
public class Pair<T> {
   private T  first;
   private T  second;
   …
   public boolean equals ( Object other ) {
            if (other==null) return false;
            if (getClass() != other.getClass())  return false;
            Pair<T>  o = (Pair<T>)other;
            return (first==o.first) && (second==o.second);
   }
   …
}
if walkingPair.equals(runningPair);   // usage examle
```

More then one type parameter can be specified

```
Pair<String,Integer>
  p = new Pair<String,Integer>( “Kyle Jones”,
                                new Integer(123456789) );
```

Defining the ordered pair class

```
public class Pair<T1,T2> {
   private T1  first;
   private T2  second;
   public Pair ( ) {
            first = null;
            second = null;
   }
   public Pair ( T1 f, T2 s ) {
            first = f;
            second = s;
   }
   …
}
```

Defining the ordered pair class

```
public class Pair<T1,T2> {
  private T1  first;
  private T2  second;
  …
  public T1 getFirst ( ) {  return first;  }
  public T2 getSecond ( ) {  return second;  }
  …
}
```

Defining the ordered pair class

```
public class Pair<T1,T2> {
   private T1  first;
   private T2  second;
   …
   public boolean equals ( Object other ) {
            if (other==null) return false;
            if (getClass() != other.getClass())           return false;
            Pair<T1,T2>  o = (Pair<T1,T2>)other;
            return first.equals(o.first) && second.equals(o.second);
   }
   …
}
```

A Primitive Type Cannot be Plugged in for a Type Parameter!!!

- The type plugged in for a type parameter must always be a reference type:
  - It cannot be a primitive type such as **`int`**, **`double`**, or **`char`**
  - However, now that Java has automatic boxing, this is not a big restriction.

Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

```java
import java.util.Scanner;

public class GenericPairDemo2
{
   public static void main(String[] args)
   {
         Pair<Integer> secretPair =
              new Pair<Integer>(42, 24);

         Scanner keyboard = new Scanner(System.in);
         System.out.println("Enter two numbers:");
         int n1 = keyboard.nextInt();
         int n2 = keyboard.nextInt();
         Pair<Integer> inputPair =
             new Pair<Integer>(n1, n2);

         if (inputPair.equals(secretPair))
         {
             System.out.println("You guessed the secret numbers");
             System.out.println("in the correct order!");
         }
         else
         {
             System.out.println("You guessed incorrectly.");
             System.out.println("You guessed");
             System.out.println(inputPair);
             System.out.println("The secret numbers are");
             System.out.println(secretPair);
         }
```

Automatic boxing allows you to use an **int** *argument for an* **Integer** *parameter.*

Limitations on Type Parameter Usage

- The type parameter cannot be used in simple expressions using new to create a new object
  - For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T();
T[] a = new T[10];
```

Limitations on Generic Class Instantiation

- Arrays such as the following are illegal:

```
Pair<String>[] a = new Pair<String>[10];
```

- Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes.