17.3 Passing Objects of a Generic Class to a Method

Suppose we want to write a method that accepts an instance of a generic class as an argument.

```
public static void printPoint(Point<Integer> point)
{
  System.out.println("X Coordinate: " + point.getX());
  System.out.println("Y Coordinate: " + point.getY());
}
```

In the above method, the parameter `point` is a reference to `Point<Integer>` object. Then, we can call the method and pass a reference to any `Point<Integer>` object:

```
Point<Integer> iPoint = new Point<Integer>(7, 12);
printPoint(iPoint);
```

However, there is a problem if we want to pass an instance of `Point<Double>` to the method. Since the method's parameter is a reference to `Point<Integer>`, only `Point<Integer>` objects can be passed to it.

One way to solve the problem is to use the `?` type wildcard:

```
public static void printPoint(Point<?> point)
{
  System.out.println("X Coordinate: " + point.getX());
  System.out.println("Y Coordinate: " + point.getY());
}
```

The `?` character inside the angled brackets is a type wildcard, which indicates that any type argument can be used in its place. Then, we can pass any `Point` object as an argument to the method, regardless of its type argument. For example,

```
Point<Integer> iPoint = new Point<Integer>(1, 2);
Point<Double> dPoint = new Point<Double>(1.5, 2.5);
printPoint(iPoint);
printPoint(dPoint);
```

Constraining a Type Parameter

The change of the method header from

```
public static void printPoint(Point<Integer> point)
```

to

```
public static void printPoint(Point<?> point)
```

allows us to pass any **Point** object as an argument to the **printPoint** method. However, this leads to a different problem: The parameter's new type, **Point<?>**, might not be restrictive enough.

If we only want to accept **Point** object whose type argument is a subclass of **Number**, we can modify the method to

```
public static void printPoint(Point<? extends Number> point)
{
  System.out.println("X Coordinate: " + point.getX());
  System.out.println("Y Coordinate: " + point.getY());
}
```

where the parameter's type is changed to **Point<? extends Number>**. It means that the **Point** object's type argument may be **Number**, or any type that extends **Number**.

Example:

```
Point<String> sPoint = new Point<String>("1", "2");
printPoint(sPoint); // Error!
```

The code will not compile because the **String** class is not a subclass of **Number**.

The notation

```
<? extends Number>
```

means "any type that is **Number** or a subclass of **Number**."

Code Listing 17-5 (TestPoint3.java)

Defining a Type Parameter in a Method Header

As an alternative to using the wildcard in the notation

```
Point<? extends Number>
```

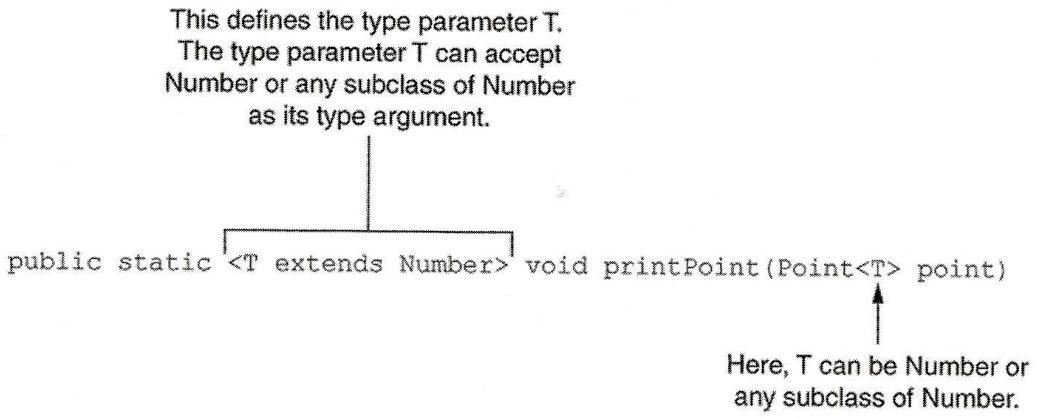
we can actually define a type parameter in a method header:

```
public static <T extends Number>
                        void printPoint(Point<T> point)
{
   System.out.println("X Coordinate: " + point.getX());
   System.out.println("Y Coordinate: " + point.getY());
}
```

The notation **`<T extends Number>`** defines a type parameter named **`T`**, and specifies that **`T`** can accept any type that is **`Number`** or a subclass of **`Number`**.

The type of the method's parameter is **`Point<T>`**. This means that the method can accept any **`Point`** object whose type parameter is **`Number`** or a subclass of **`Number`**.

Figure 17-3 Type parameter defined in a method header



Using this alternative syntax, we can simplify methods that accept multiple arguments of generic types.

For example, the method header

```
public static
     void doSomething(Point<? extends Number> arg1,
                      Point<? extends Number> arg2,
                      Point<? extends Number> arg3)
```

can be written as

```
public static <T extends Number>
     void doSomething(Point<T> arg1,
                      Point<T> arg2,
                      Point<T> arg3)
```

The `extends` Key Word Constrains a Type to an Upper Bound

When we use the **`extends`** key word with a generic type parameter, we are constraining the type parameter to an upper bound.

For example, in the notation **`<T extends Number>`**, we are constraining **`T`** to the upper bound **`Number`**. It means that **`T`** can be any type that is below **`Number`** in the class hierarchy including **`Number`** itself, but not any type that is above **`Number`**.

The `super` Key Word Constrains a Type to a Lower Bound

In addition to the **`extends`** key word, we can use the **`super`** key word to restrict a type parameter. For example,

```
public static
     void doSomething(Point<? super Integer> arg)
```

Here, the **`arg`** parameter's type is **`Point<? super Integer>`**, which means that the **`Point`** object's type argument may be **`Integer`**, or any superclass of **`Integer`**.

Because the type can be any class above **`Integer`** in the class hierarchy, it is said that we are constraining the type to a lower bound.

17.4 Writing Generic Methods

Methods themselves can also be generic. This means that they can have their own type parameters, and can use those type parameters to represent the types of arguments, the type of local variables, and their return type.

Code Listing 17-6 (GenericMethodDemo.java)

The header of the **displayArray** method

```
public static <E> void displayArray(E[] array)
```

defines a type parameter named **E**. The method has a parameter variable, **array**, which is a reference to an array of **E** objects.

The **displayArray** method is called like a regular method. Even though it is a generic method, no type argument is passed to it. When we call a generic method, the compiler determines which type to use from the context in which we are using the method.

We can constrain a type parameter in a generic method. For example, if the **displayArray** method is written as:

```
public static <E extends Number>
              void displayArray(E[] array)
{
   for (E element : array)
         System.out.println(element);
}
```

we will have constrained the type parameter **E** to any type is **Number** or a subclass of **Number**. When calling this version of the method, we can only pass any array of **Number** objects, or objects of a subclass of **Number**.

17.5 Constraining a Type Parameter in a Generic Class

Sometimes, we may want to constrain the **`Point`** class itself so that only certain type arguments can be used to create an instance of the class.

For example, we might want to allow instances of the **`Point`** class to be created using only the numeric wrapper classes as type arguments.

Code Listing 17-7 (Point.java of **`Point`** Class Version 2)

The only difference between this **`Point`** class and its earlier version is the type parameter notation:

```
public static Point<T extends Number>
```

The notation **`<T extends Number>`** defines a type parameter **`T`**, which is constrained with **`Number`** as its upper bound. It means that only **`Number`** or a subclass of **`Number`** may be passed as a type argument to this parameter.

For example, the following statement will compile without error because the type argument is a subclass of **`Number`**:

```
Point<Integer> iPoint = new Point<Integer>(1, 2);
```

However, the following statement will cause an error at compile time because the type argument is not subclass of **`Number`**:

```
Point<String> iPoint = new Point<String>("1", "2");
```

We can also use the **`super`** key word in a generic class to constrain a type parameter to a lower bound. For example,

```
public class Point<T super Double>
```

will specify that the type argument must be **`Double`**, or any superclass of **`Double`**.

17.6 Inheritance and Generic Classes

Inheritance can be used with generic classes.

Code Listing 17-8 (Point3D.java)

In the class header

```
public class Point3D<T extends Number>
                                    extends Point<T>
```

the first part of this statement

```
public class Point3D<T extends Number>
```

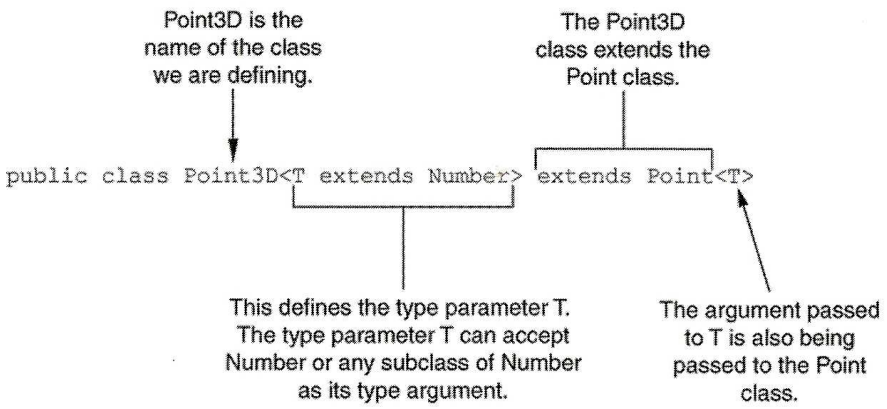
indicates that the class is **`Point3D`**, and the type parameter is **`T`**. The notation **`<T extends Number>`** is used to define the type parameter, so any type argument passed to **`T`** must be **`Number`** or a subclass of **`Number`**.

The notation

```
extends Point<T>
```

indicates that the class **`Point3D`** extends the **`Point`** class, with **`T`** passed as a type argument to the **`Point`** class.

Figure 17-4 Generic subclass header



Code Listing 17-9 (TestPoint3D.java)

The "is-a" relationship is in effect between the `Point3D` class and the `Point` class. A `Point3D` object *is a* `Point` object.

We can assign a `Point3D` object to a `Point` reference variable, or pass a `Point3D` object to a method that accepts `Point` objects.

Code Listing 17-10 (PassPoint3D.java)

The `Point3D` class is an example of a generic class that extends another generic class.

Both generic and non-generic classes may be used together in an inheritance hierarchy, in any of the following ways:

- A generic class may extend another generic class
- A generic class may extend a non-generic class
- A non-generic class may extend a generic class

17.7 Defining Multiple Type Parameters

It is possible to define multiple type parameters in a generic class or method. Different type arguments can then be passed to each type parameter. For example, the following class `MyClass` defines two type parameters:

```
public class MyClass<T, S>
{
        class code ...
}
```

We can also apply constraints to the type parameters:

```
public class MyClass<T extends Number, S extends Date>
{
    class code ...
}
```