COMP 401

*Prasun Dewan*[1]

# 23. Generics and Adapters

The term generic implies unspecialized/common behavior. In an object oriented language, it seems to apply to the class Object, which describes the behavior of all objects. In fact, a generic type in a programming language is more complicated. It allows us to have the benefits of the Object type, which is the ability to assign to a variable an object of any type, without the drawback of having to cast the variable to access specialized behavior of the assigned object. Thus, it allows us to have our cake and eat it too!

Generics are particularly useful for increasing the reusability of the variable-sized collections we implement. In Java, the array type is a fixed-size collection type provided by the language that takes the type of the element as a parameter. As a result, it can be used to create arrays of arbitrary element types such as int, double, and Point.  Generics allow us to similarly create a variable-size collection that takes the type of the element as a parameter.

Generics will also make us look more in depth at the idea of an IS-A relationship, by answering the following question: If a type T1 IS-A T1, what is the relationship between array, histories, databases and other collections of elements of type T1 and type T2.

Java provides several useful generic types – in particular, those that define variable-sized collections. However, a proper use of them requires the application of adapter objects, which, like real-life adapters, are converter objects sitting between two objects that need to interact with each other. This chapter will briefly look at some the Java collection generic types, explain the nature of adapter objects, and illustrate their use in defining variable-sized collections from the Java collection types. The correct use of adapters will require the use of delegation, which is briefly introduced here, and discussed in more depth in a separate section.

## The Need for Code Reuse in Definition of Collections

By now we have seen several ways to reuse code. The method abstraction allows us to share code within a class – all callers of a method share the code in the called method. Inheritance allows us to share code among different classes – all subclasses of a class share the code of the inherited method. As we have seen, delegation is a more flexible but less automated approach, relying on the HAS-A rather than IS-A relationship between classes. We have also seen several patterns such as MVC and Iterators that provide structured ways to use these code-sharing mechanisms in certain situations.

[1] © Copyright Prasun Dewan, 2010.

None of these concepts, however, helps us reduce the code duplication in the definition of variable-sized collections. Consider the following two history definitions.

```
public interface  StringHistory  {
            public void addElement(String element);
            public int size();
            public String elementAt(int index);
}
```

```
public interface PointHistory {
            public void addElement (Point newVal);
            public Point elementAt (int index);
            public int size();
}
```

These two definitions are identical except for the type of the argument of addElement() and the return type of elementAt() – in the first case it is String, and the second one, is Point. We might be tempted to create a single history definition in which this type is Object, the common super-type of both String and Object.

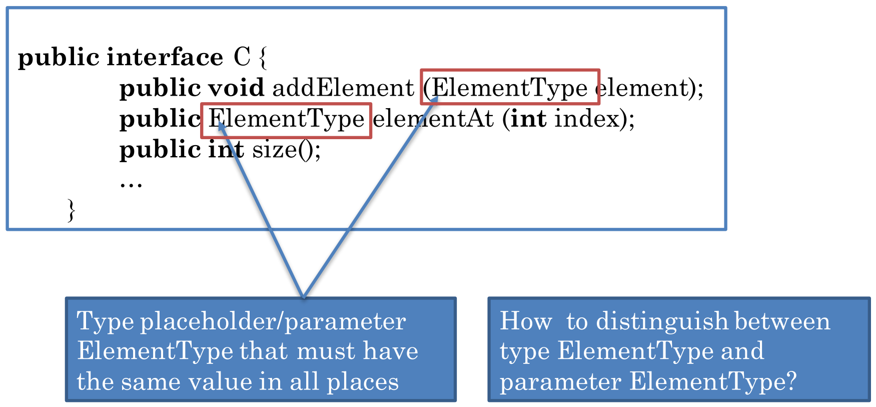
```
public interface  ObjectHistory  {
            public void addElement(Object element);
            public int size();
            public Object elementAt(int index);
}
```

However, unlike the point and string histories, this history would allow a mix of Strings, Points, and any other object. Thus, this interface is not really equivalent to the two history interfaces given above. The code duplication problem remains.

## Type Parameters, Generics, and Elaborations

Thus, what we need is the ability to define histories in which all elements are of some type designated by us, without duplicating code for different element types? In other words, we need a single definition in which the type of the element is itself a “parameter” or placeholder that can be “assigned” by us to different types to create different kinds of histories.

The notion of a type placeholder/variable/parameter was actually used in the arrays chapter to describe the ObjectEditor conventions for variable-sized collections, reproduced below.

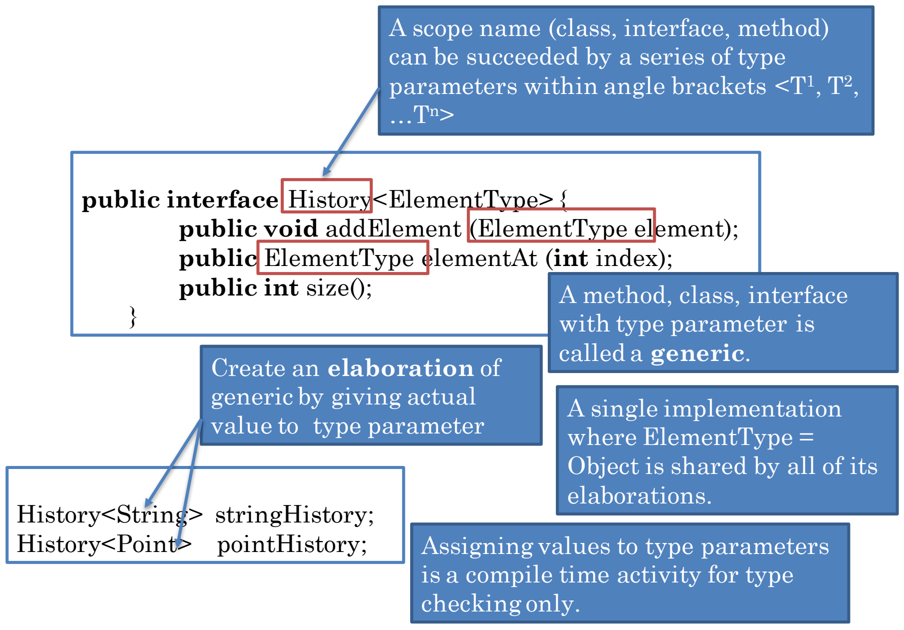


```
public interface C {
        public void addElement (ElementType element);
        public ElementType elementAt (int index);
        public int size();
        …
    }
```

Type placeholder/parameter ElementType that must have the same value in all places

How to distinguish between type ElementType and parameter ElementType?

This was just an informal way to document the conventions; it was not something a compiler could accept to create a type, partly because the compiler could not distinguish type literal, T, from type parameter, T. We have seen earlier the use of angle brackets, <T>, to denote type placeholders/variables/parameters, which remove the ambiguity but are more cumbersome to use. Java provides a mix of these schemes, illustrated below.



A scope name (class, interface, method) can be succeeded by a series of type parameters within angle brackets $<T^1, T^2, \ldots T^n>$

```
public interface History<ElementType> {
        public void addElement (ElementType element);
        public ElementType elementAt (int index);
        public int size();
    }
```

A method, class, interface with type parameter is called a **generic**.

Create an **elaboration** of generic by giving actual value to type parameter

A single implementation where ElementType = Object is shared by all of its elaborations.

```
History<String>  stringHistory;
History<Point>   pointHistory;
```

Assigning values to type parameters is a compile time activity for type checking only.

The name of a scope such as a method, interface and class can be followed by a comma separated sequence of type parameter names enclosed in angle brackets such as:

$<T^1, T^2, .. T^n>$

This sequence tells Java that these names are placeholders for types rather than actual type literals. Within that scope now, these names can be used, without the angle brackets, wherever type (literals) are accepted. For example, in the History definition, we use a single type parameter, ElementType:

History<Element Type>

In the body of the scope, the type placeholder name can be used without the angle brackets. All occurrences of the type parameter denote or, *unify to*, the same value. Thus, the two uses of the type placeholder below ensure that the type of the element added by, addElement(), is the same as the type of the value returned by elementAt():

> **public** addElement (ElementType element);
> **public** ElementType elementAt(**int** index);

A scope such as method, class, and in this example, interface, that has one or more type parameters is called a *generic*. Before using a generic, we should assign values to its type parameters, as shown below:

History<Point> pointHistory;

The type values are specified within a sequence of type literals or parameters enclosed within angle brackets that follows the name of the generic.  Assigning values to the type parameters of a generic is called *elaboration* of the generic. It is possible to use a generic without explicitly elaborating it, as shown below:

History objectHistory;

In this case, Java assigns Object to each type parameters, and some programming environments, such as Eclipse, give a warning message saying that the generic should be explicitly elaborated.

## Generic Classes and Erasure

It is possible for an elaboration of a generic to itself contain type parameters declared in the scope elaborating the generic, as illustrated in the definition of a generic history class, AHistory, shown below.

```
public class AHistory<ElementType> implements
History<ElementType> {
    public final int MAX_SIZE = 50;
    Object[] contents = new Object[MAX_SIZE];
    int size = 0;
    public int size() {return size;}
    public ElementType elementAt (int i
        return (ElementType) contents[i
    }
    boolean isFull() {return size == MAX_SIZE;}
    public void addElement(ElementType element) {
        if (isFull())
          System.out.println("Adding item to a full history");
        else {
          contents[size] = element;
          size++;
    }
}
```

A single implementation where ElementType = Object is shared by all of its elaborations.

The header of this class declares the type parameter, ElementType, and then uses it in the elaboration of the interface, History. This means that type parameters of the interface and the class will have the same value when the class is elaborated. The class uses this parameter in the declaration of all methods that access elements of the history.

One counter-intuitive aspect of the class implementation is the declaration of the array. We have declared it as an object array:

```
Object[] contents = new Object[MAX_SIZE];
```

This, in turn, requires us to use a cast when we wish to use an array element as an instance of <ElementType>:

```
return (ElementType) contents[index];
```

Why not use the type parameter in the definition of the array, in the following manner?

```
ElementType[] contents = new ElementType [MAX_SIZE];
```

Such code makes perfect sense if Java elaborated a generic by creating a copy of it in which the type parameters are replaced with the values assigned to them. Some languages such as C++ do take this approach, creating a different implementation of a generic for each elaboration. Java, on the other hand, creates a single implementation that is shared by all elaborations. Elaboration of a generic is purely a compile time activity to do type checking. All information about type parameters is erased at the end of compilation; this approach to implementing generic types is therefore called *erasure*. As a single implementation of a generic exists, the complier does not know what code to generate if we use a type parameter in the instantiation of the array. In fact, a type parameter can never be used in an instantiation. Thus, the following is illegal:

ElementType t = **new** ElementType();

If Element is an abstract class or interface, instantiation of it makes no sense. Even it is a class, Java does not know what constructors it has.

It is, of course, possible to cast an instantiated object to a variable typed using a type parameter.

```
ElementType[] contents = (ElementType[]) new Object [MAX_SIZE];
```

Here we have cast the instantiated Object array to an array variable declared using a type parameter: This code will not generate a runtime exception in Java because of erasure – the type variable ElementType has no meaning at runtime. Now we do not have to cast individual elements of the array variable:

```
return contents[index];
```

Thus, the approach of casting the entire array at variable declaration time is better than the approach of casting individual array elements when they are accessed. It would be even better if Java automatically did such a cast, that is, processed:

```
        ElementType[] contents = new ElementType [MAX_SIZE];
```

as:

```
        ElementType[] contents = (ElementType[]) new Object [MAX_SIZE];
```

This would be consistent with, for instance, automatically converting between wrapper and primitive types.

Because of the erasure approach, we do not have to explicitly elaborate a generic class when it is instantiated:

```
        History<String> stringHistory = new AHistory();
        History<Point>  pointHistory =   new AHistory();
```

The fact that we do not have to give the element type of a generic collection seems counterintuitive given that we have to specify the element type in an array instantiation:

```
        new String[50];
        new double[50];
```

The reason is that the representation depends on the type of the element. For example, the instantiation, **new** String[50], allocates a block of space consisting of Object pointers, while the instantiation, **new** double[50], allocates a block of space consisting of double values. As we saw before, a double value takes a different amount of space than a pointer. Thus, the two blocks are of different sizes even though they contain the same number of elements. On the other hand, in the case of a generic class, a single implementation exists in which each variable that is declared using a type parameter is a pointer to an Object.

It is possible to specify a type parameter when a generic collection is instantiated:

```
        new AHistory<Point>();
```

In fact, some programming environments will give a warning if you do not do so. We will see look at the reason in depth later.

## Renaming Type Parameters

As in the case of method parameters, it is the position rather than the names of type parameters that matter to the compiler. This is illustrated below.

```
public interface I1 <T>{
        …
}
```

```
public interface I2 <T>{
        …
}
```

```
public class C<T1, T2> implements  I1 <T1>, I2<T2>{
        …
}
```

Here, two generic interfaces, I1 and I2, use the same name, T, for their type parameter. The elaboration of these interfaces in class C refers to these type parameters using the names, T1 and T2, respectively.

## Named Elaborations

Using angle brackets in elaborations can be tedious, especially because they involve the use of the Shift key.  It is possible to name a particular elaboration of a class or interface by creating an extension of an elaboration of the class or interface, as shown below.

```
public interface StringHistory extends History<String> {

}
```

Now we can use the name of the extension in declarations of multiple variables:

```
StringHistory stringHistory1 = new AHistory();
StringHistory stringHistort2 = new AHistory();
```

rather than repeat the elaboration in the declaration of each of these variables:

```
History<String> stringHistory1 = new AHistory();
History<String> stringHistort2 = new AHistory();
```

## IS-A Relationship among Elaborations

Consider the following code.

```
History<String> stringHistory = new AHistory();
History<Point>  pointHistory   = new AHistory();
pointHistory = stringHistory;         ✖
```

The third second assignment is illegal. We now that we cannot assign a String object to a Point variable, because a String is not a Point. It stands to reason, then, that we cannot assign a collection of String objects to a variable that expects a collection of Point objects. In other words, the IS-A relationship does not hold between types T1 and T2, then it does not hold between collections of type T1 and T2.

What about the converse? If type T1 IS-A T2, then is a collection of T1 a subtype of a collection of type T2? The answer, perhaps counter-intuitively, is no. This is illustrated in the code below.

```
History<String>  stringHistory   = new AHistory();
History<Object> objectHistory = stringHistory;
```

It is not possible to assign a history of String instances to a variable that expects a history of plain Object instances. The following code shows what could go wrong Java allowed the assignment.

```
stringHistory.addElement(“hello”);
objectHistory.addelement(new ACartesianPoint(5,10));
String secondElement = stringHistory.elementAt(1);
```

We would now have two references to the same collection that are typed differently. The reference, stringHistory, would allow us to add a String element to the history. The reference, objectHistory, would allow us to a Point to the history. Thus, now the variable, stringHistory, which expects to be refer a collection in which each element is a String (or a subtype of String), would now refers to collection in which an element that does not have these properties! As a result, the third statement above, which would pass compile time checking, would give a run time exception. Java prevents such an error from occurring. In general, T1 IS-A T2 does not mean that two elaborations of a generic that assign parameters T1 and T2, respectively, to a type parameter of the generic, and assign the same value to all other type parameters of the generic, do not have an IS-A relationship between them.

## WildCard Expressions

The above rule is constraining in several situations. Consider the following code.

```
void printX (History<Point> pointHistory) {
        for (int index = 0; index < pointHistory.size(); index++)
                System.out.println(pointHistory.elementAt(index).getX());
}
```

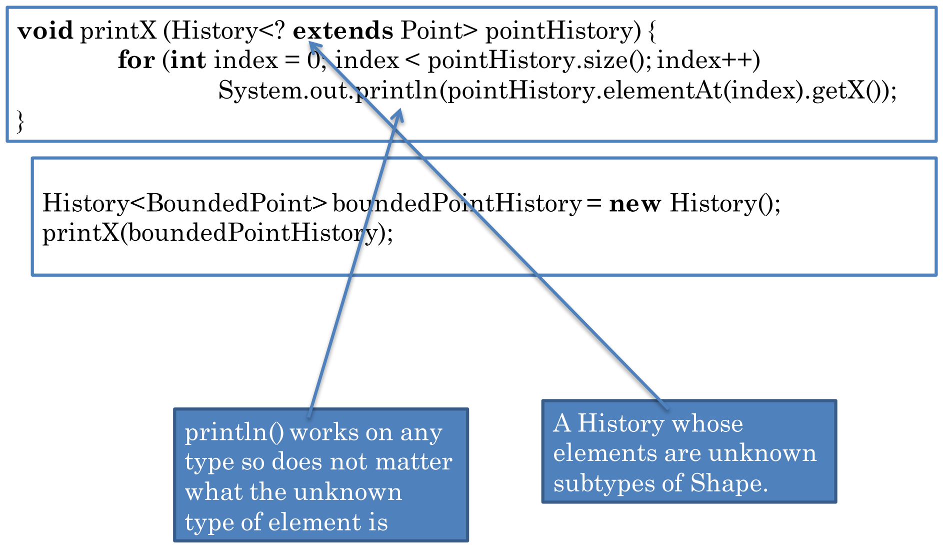
```
History<BoundedPoint> boundedPointHistory = new History();
printX(boundedPointHistory);
```

Disallowed even though no type violations occur.

Can we type in such a way that safe operations on generics are allowed and unsafe are not?

The printX() method expects a history of Point objects and prints the X coordinate of each of the elements. Unfortunately, because of the above, we cannot pass the method a history of BoundedPoint objects, even though BoundedPoint IS-A Point, and thus, defines the getX() method invoked by printX. It would be nice if we can type generics in such a way that safe operations such as getX() are allowed and unsafe operations such as add() are not.

Java provides type wildcards as a form of such typing. We can assign a type parameter a type expression of the form ? extends T, where T is a type literal or parameter, as shown in the modified version of printX() given below.



```
void printX (History<? extends Point> pointHistory) {
        for (int index = 0; index < pointHistory.size(); index++)
                System.out.println(pointHistory.elementAt(index).getX());
}
```

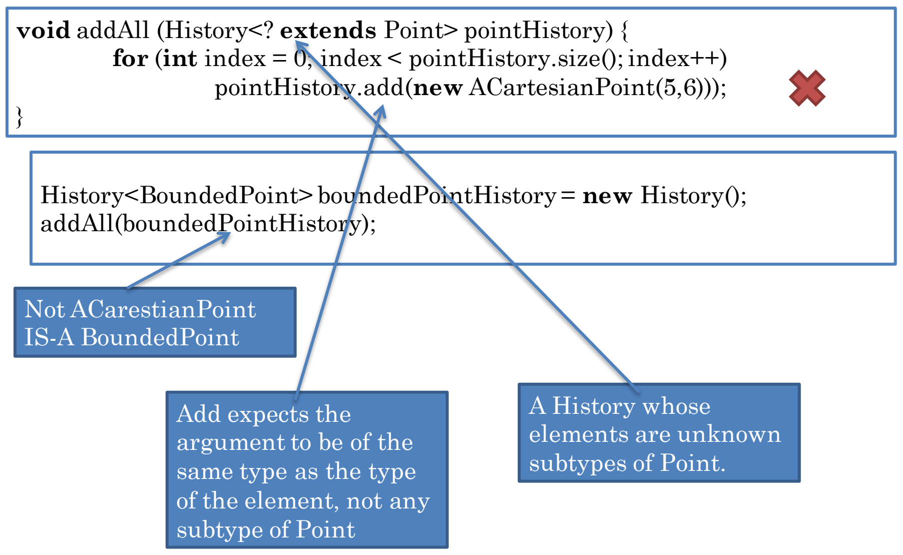
```
History<BoundedPoint> boundedPointHistory = new History();
printX(boundedPointHistory);
```

println() works on any type so does not matter what the unknown type of element is
A History whose elements are unknown subtypes of Shape.

Here, instead of assigning ElementType the type Point, we have assigned it the expression

? **extends** Point.

This expression tells Java that the ElementType of History is not bound to a specific type but to any type that extends Point. Thus, the exact type is unknown to Java. Java uses this information to allow operations that need to only assume that element type is any subtype of Point. In the above example, the getX() method can be invoked on any element whose type is a subtype of Point. Thus, it is legal. On the other hand, the code below is erroneous.



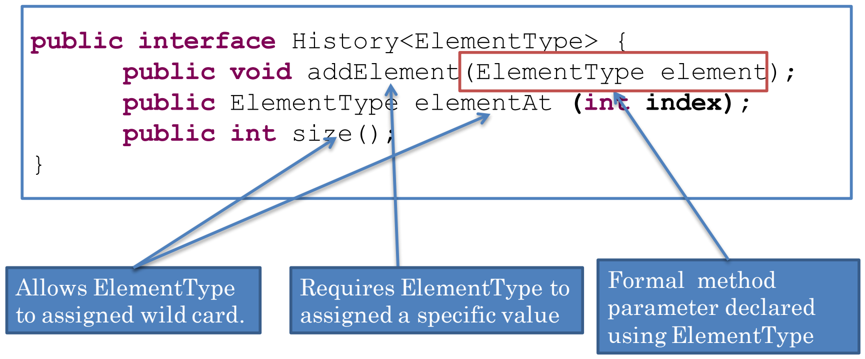
```
void addAll (History<? extends Point> pointHistory) {
        for (int index = 0; index < pointHistory.size(); index++)
                pointHistory.add(new ACartesianPoint(5,6)));
}
```

```
History<BoundedPoint> boundedPointHistory = new History();
addAll(boundedPointHistory);
```

Not ACarestianPoint IS-A BoundedPoint
Add expects the argument to be of the same type as the type of the element, not any subtype of Point
A History whose elements are unknown subtypes of Point.

The reason is that the add() method must know the specific value of ElementType to ensure that an illegal element cannot be added to the collection. By making the invocation of the add method above illegal, Java prevents the addAll() from adding an instance of ACartesianPoint(), which is not a BoundedPoint, to be added to the history boundedPointHistory, which expects each element to be a subtype of BoundedPoint.

In general, if a type parameter is assigned a wildcard expression, then a method that declares a formal argument typed using the parameter cannot be invoked, while any other method can be invoked. This is the reason why, when a History is elaborated using a wildcard expression, it is possible to invoke the elementAt() and size() methods on it but not the  addElement() method. This is illustrated below.



```
public interface History<ElementType> {
      public void addElement(ElementType element);
      public ElementType elementAt (int index);
      public int size();
}
```

Allows ElementType to assigned wild card.
Requires ElementType to assigned a specific value
Formal method parameter declared using ElementType

## IS-A Relationship between Arrays and Covariance

Based on the discussion about the lack of an IS-A relationship between generic collections elaborated using different type values, it is not surprising that the third statement below is illegal.

```
String[] strings = new String[50];
Point[] points   = new Point[50];
points = strings;
```

If

Not T1 IS-A T1

then

Not T1[] Is-A T2[]

We also learned that T1 IS-A T2 does not imply that a collection of T1 IS-A collection of T2. Surprisingly, then, the following code results in no errors.

```
String[] strings  = new String[50];
Object[] objects  = strings;
```

As a result, the following code will result in no compile time error.

```
strings[0] = "hello";
objects[1] = new ACartesianPoint(5,10);        ✖
String s =  strings[1];
```

However, when the second statement is executed, Java will throw an ArrayStoreException. The reason is that because the array was instantiated as a String array, it expects each element to be a String.

This behavior is surprising because, in the case of a generic collection, Java assignment rules prevent illegal elements to be included the collection. Why not do the same for arrays? Thus, why not prevent the following assignment:

```
Object[] objects  = strings;
```

The probably reason is that this rule would prevent the following array version of the printX() code we saw before.

```
void printX (Point[] points) {
        for (int index = 0; index < points.size(); index++)
                System.out.println(points.elementAt(index).getX());
}
```

We were able to create the generic version of this code by using wildcard type expressions. As an array is not a generic, we cannot use this approach for the array version. Java makes the optimistic assumption at compile time that illegal elements will not be assigned to an array, and raises an exception at run time when this assumption does not hold. It can throw such an exception because with each array, information about its element type is kept at runtime. This is not the case with generics, as a single implementation is created for all possible element types, and information about the element type is erased at the end of compilation. This is the reason why two different sets of assignment rules are used for arrays and generic collections.

If T1 IS-A T2 implies that a collection of T1 IS-A collection of T2, then the collection is called *covariant*. In Java, arrays are covariant but generic collections are not.

## Adapters

Consider the following version of  PointHistory.

```
public interface PointHistory {
        public void addElement (int x, int y);
        public Point elementAt (int index);
        public int size();
}
```

It is different from the version we saw earlier in that the addElement() method takes as arguments the Cartesian coordinates of a point rather than a Point object. Naturally, we would like to reuse the code in AHistory in an implementation of this interface. One way to do so is to create the following extension of AHistory.

```
public class APointHistory extends AHistory<Point>
implements PointHistory{
       public void addElement(int x, int y) {
              addElement(new ACartesianPoint(x,y));
       }
}
```

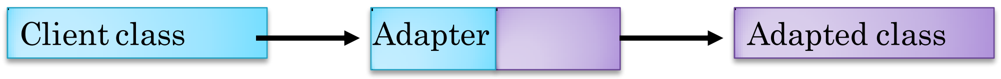
Has both addElement methods!

However, this class implements more than the (second version of the) PointHistory interface in that it implements both forms of addElement(). There is no harm in doing so, but let us, for argument sake, assume we want only the addElement(int, int) defined in PointHistory. The answer is to *delegate* to rather than inherit from AHistory, as shown below.

```
public class APointHistory implements PointHistory {
       History<Point> contents = new AHistory();
       public void addElement(int x, int y) {
              contents.addElement(new ACartesianPoint(x,y));
       }
       public Point elementAt(int index) {
              return contents.elementAt(index);
              }
       public int size() {
              return contents.size();
       }
}
```

In delegation, a class A reuses an existing class C, not by inheriting from it, but by having a reference to C, which is stored in an instance variable. Thus, in this example, APointHistory reused AHistory, not by extending AHistory, but having a pointer to AHistory, stored in the instance variable *contents*.

In both the inheritance and delegation cases, we have put an intermediary class, A, between a class, C and its user or client, U, that adapts the functions of C without implementing new functionality. Such an intermediary class is called an *adapter*. Like a real-world adapter, an adapter class performs only a translation task, transforming the interface of the adapted class to one expected by the clients.



The adapter class can inherit from or delegate to the adapter class. As we have seen above, the delegation approach is more flexible because it allows the adapter to hide methods of the adapter class from the client class.

## Non Collection Generics

It is possible to create generics for objects other than stored collections. As we see below, we can replace separate definitions of StringIterator and PointIterator with a generic Iterator interface that takes the type of the iterated element as a parameter.

```
public interface StringIterator {
        public String next ();
        public boolean hasNext();
}
```

```
public interface PointIterator {
        public Point next ();
        public boolean hasNext();
}
```

```
public interface Iterator<T> {
        public T next ();
        public boolean hasNext();
}
```

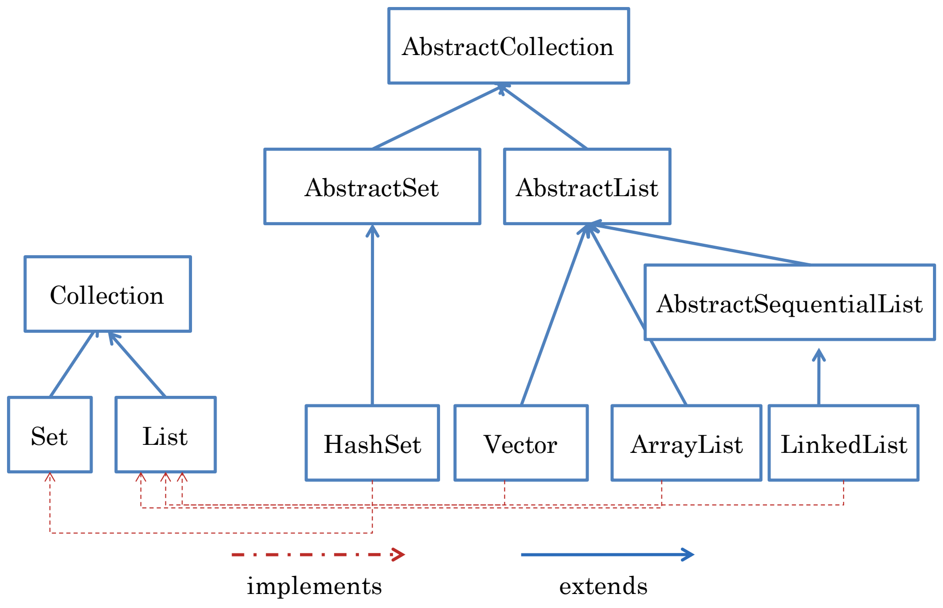
Both iterators and collections are sequences of elements, the difference is that the latter stores all of the elements in its instance variables. It is possible to define generics that are not sequences, as shown below.

```
public interface BackedUpObject<T> {
        public T getObject ();
        public T getBackup();
        public void setObject (T newVal);
        public void setBackup(T newVal);
}
```

The type, BackedUpObject, stores an object and a backup copy of it, and defines a type parameter to ensure that these two values have the same type.

## Java Generic Collections

Type parameters are used extensively in collection types provided by Java in the package java.util. The figure below shows some of these types. All of them are generic types that, like History and AHistory, define type parameters for their elements.  The Set types, like the set types implemented in the arrays chapter, defines collections without duplicates. The List, Vector, and ArrayList types support extensions of the database types of the arrays chapter. List is an interface, and ArrayList and Vector are implementations of this interface that inherit common code from the abstract class, AbstractList.



The class Vector implements a superset of the methods provided by ArrayList.  Both implement methods defined in the interface List, some of which are given below. Here, ElementType is a type parameter.

```
public int size();
public ElementType get(int index);
public void add(ElementType obj) ;
public void set(int index, ElementType obj);
public void insert(int index,  ElementType obj);
public boolean remove(ElementType obj);
public void remove(int index);
public int indexOf(ElementType obj);
public Iterator elements();
```

In addition, for backward compatibility with earlier versions of Vector, the Vector class defines several additional methods, some which are given below:

```
public ElementType elementAt(int index);
public void addElement(ElementType obj) ;
public void setElementAt(ElementType obj, int index);
public void insertElementAt(ElementType obj,  int index);
public boolean removeElement(ElementType obj);
public void removeElementAt(int index);
public Enumeration elements();
```

As you can see, the conventions defined by ObjectEditor for collections are inspired by these types.

When implementing our own collection types, it is much more convenient to use these collection types rather than arrays, as we do not have to worry about allocating enough storage and keeping track of which part of it is occupied and which is unoccupied. It is possible to inherit from or delegate to these classes, as shown below in the following implementations of AStringHistory.

```
import java.util.ArrayList;
public class AStringHistory implements StringHistory {
    List<String> contents = new ArrayList();
    public void addElement (String s) {contents.add (s);}
    public String elementAt (int index) {return  contents.get(index); }
    public int size() {return contents.size();}
}
```

```
import java.util.ArrayList;
public class AStringHistory extends ArrayList<String> {  }
```

Exposing only methods in a history.

remove() not visible.

Delegating adapter often used with ArrayList and Vector

The example again shows the flexibility of delegation. AStringHistory should not expose methods that remove elements from the collection or add elements at arbitrary positions. However, the inheritance approach exposes all of these elements of ArrayList. The delegation approach creates an adapter that hides all unwanted methods from the user of AStringHistory. ArrayList and Vector classes are often accessed through delegating adapters.

## Elaboration in Instantiation of Generic Classes

Consider the following statement in the delegating implementation:

```
List<String> contents = new ArrayList();
```

Here we have (explicitly) elaborated the interface, List, but not the class, ArrayList. As elaboration has meaning only at compile time, it seems redundant to elaborate the class instantiation. Yet Java will allow us to elaborate an instantiation, as shown below.

```
List<String> contents = new ArrayList<String>();
```

In fact, Eclipse will give a warning if we do not elaborate an instantiated generic class. If all accesses to the instantiated class occur through the variable to which it is assigned, then typing the variable correctly is sufficient to ensure that these accesses are safe. In the delegating implementation given above, this is indeed the case, as the variable, contents, has been typed correctly. Elaborating the instantiation makes no difference to compile time checking.

Elaborating an instantiated class makes sense only if we access the instantiated object directly, without assigning it to a variable. The following code shows when elaborating an instantiation can make a difference.

```
String  nonExistingElement = (new ArrayList()).get(0);
```

✖

```
String  nonExistingElement = (new ArrayList<String>()).get(0);
```

In both statements, the instantiated generic class is not assigned to a variable. Instead, the get() method is called on the new instance, and assigned to a String variable. The first statement gives a compile time error because ArrayList has not been explicitly elaborated. As a result, Java assigns Object to its ElementType parameter, which in turn implies that get() returns Object. An Object  value cannot be assigned to a String variable; hence the error. The elaboration of the second instantiation ensures that the return type of get() can be assigned to the String variable.

This example shows that elaborating an instantiation can make a difference but is not very convincing as in both cases we will given an index out of bounds exception – a newly created ArrayList has no elements that can be retrieved using get(). To create a more convincing example using ArrayList, we must instantiated a non-empty ArrayList. It is, in fact, possible to do so, as the class provides a constructor that takes a List as an argument and returns a copy of the list.

Suppose we create an ArrayList with one element:

```
List<String> contents = new ArrayList();
contents.add(“test”);
```

We can now create a copy of this object using by passing it to the ArrayList constructor and assign it in a type safe manner to variable.

```
List<String> correctCopy = new ArrayList(contents);
```

However, we can also assign the copy to a variable in a type unsafe manner.

```
List<Point> incorrectCopy= new ArrayList(contents);
```

The reason is that we did not explicitly elaborate the instantiation. This assignment allows us to add a Point object to the copy.

```
incorrectCopy.add(new ACartesianPoint(5,5));
```

In addition, it will also allow the following statement to be compiled without any errors.

```
Point firstElement = incorrectCopy.get(0);
```

However, as the first element is the String "test", this statement will give a run time ClassCastException when we assign it to the Point variable. Had we elaborated the second instantiation, we would not have been able to assign it to a Point collection. If we assign the type parameter of the instantiation to the type String:

```
List<Point> checkedCopy1 = new ArrayList<String>(contents);
```

Then Java would complain that the correctly created instance cannot be assigned to the variable checkedCop1. We could assign the type parameter to the type Point.

```
List<Point> checkedCopy2 = new ArrayList<Point>(contents);
```

then the assignment is correct but the instantiation is incorrect as the constructor of the instantiated class expects its argument type to be List<Point>. However, the variable, contents, was elaborated as List<String>, hence we get a compile error.

Thus, the moral is that the class should be elaborated at instantiation time if the constructor used for instantiation takes an argument whose type is described by a type parameter.

## Java Iterators, Iterables, and Associated For Loops

The Java util package illustrates that it is possible and useful to iterate over the elements of a collection. The package defines the following paramaterized Iterable interface.

```
package java.util;
public interface Iterable<ElementType> {
        public Iterator<ElementType>  iterator();
}
```

Any object that implements this interface must implement the iterator() method, which returns an object that iterates elements whose type is given by the type parameter, <ElementType>. Any class that implements this interface is called an iterable.

The Java interface List extends this interface.

```
package java.util;
public interface List <ElementType> extends Iterable<ElementType> {
            …
}
```

This means that class ArrayList and Vector are iterables. The following code shows how we can iterate over the Iterable object, arrayList.

```
java.util.Iterator <Point> elements =  pointList.iterator();
while ( elements.hasNext()) {
    Point element = elements.next();
    System.out.println(element.getX()  + " " + element.getY());
}
```

Thus, we do not have to create and increment an index variable to access the elements of an iterable. All we have to do is invoke the iterator() method on it to create an iterator, and then repeatedly call next() and hasNext() on the iterator.

Even these steps are not necessary, as shown below in the following loop that is equivalent to the code given above.

```
for  (Point element: pointList) {
      System.out.println(element.getX()  + " " + element.getY());
}
```

In general, if iterable is an instance of Iterable<ElementType>, then:

```
            for (ElementType element: iterable) {
                        …
            }
```

Is equivalent to:

```
            Iterator<ElementType> elements;
            while (elements.hasNext()) {
                        ElementType element;
                        …
            }
```

As we have deduced from the code above, the Iterator interface implemented in the java.util package is a generic providing the next() and hasNext() methods. Interestingly, it provides an additional method.

```
package java.util;
public interface Iterator<ElementType> {
          public ElementType next ();
          public boolean hasNext();
          public void remove(ElementType element);
}
```

Given that an iterator is not required to store its elements, the remove method seems to make no sense. It is meaningful only when the iterator is created from an iterable such as a collection that stores its elements. It is an optional method that is called on such an iterable to remove the element from the iterable. It is the responsibility of each class implementing the Iterator interface to appropriately implement this operation. Java designers must have decided that it is common to remove (but not add or replace) elements from a collection that is iterated.

## Summary

Java arrays are predefined, fixed-sized collections that take the type of the element as a parameter. Whenever, we instantiate an array or declare an array variable, we must provide the value of this parameter.

```
        Point[] points = new Point[50];
        String[] strings = new String[50];
```

Java generics allow us to create programmer-defined, variable-sixed collections that also take the type of the element as a parameter. As in the case of an array, when we instantiate a generic or declare a variable that stores an instance of it, we can supply the value of this parameter to elaborate the generic.

```
        History<Point>  points  = new AHistory<Point>();
        History<String> strings = new AHistory<String>();
```

Generics are implemented by *erasure*, that is, all information about type parameters is erased at the end of compilation. As a result, a single representation of a generic exists for all possible values of its type parameters. In the case of an array, on the other hand, its representation depends on the element type, and information about this type is kept at runtime.

This difference, in turn, causes several other differences between generics and arrays. The value of a type parameter need not be explicitly supplied by a user of a generic. If it is not provided, it is assumed to be Object. Often this value is not required at instantiation time, even if the instance is assigned to a collection whose elements are subtypes of Object. Arrays are covariant collections, while generic collections are not. A collection is covariant if, for type checking purposes, a collection of elements of type T1 IS-A collection of elements of type T2, if T1 IS-A. A covariant collection can cause run time errors when elements of the wrong type are added to the collection. Such runtime errors cannot be thrown for non-covariant collections. In fact, they cannot be thrown for collections implemented by erasure.

Wild card type expressions make it possible to write a single piece of code that works for collections elaborated using different types that are subtypes of a common super type that is guaranteed to not cause such errors.

Generics are useful for defining not only variable-sized collections but also, iterators, types that are not sequences, and methods.

An iterable collection provides a method to iterate its elements. Java provides a special loop to access elements of this collection that does not require explicit indexing or iteration.

An adapter object is an intermediary that transforms or omits methods of another class without adding functionality. A predefined Java collection is often used through a delegating adapter that deletes/transforms the signature of one or methods of the collection.