WORKED EXAMPLE 18.1 Making a Generic Binary Search Tree Class

In Section 17.3, we developed a binary search tree class that held `Comparable` objects. That is not typesafe—someone might add `Integer` objects to the tree and then try to find a `String` object, causing a `ClassCastException` in the `compareTo` method of either the `String` or `Integer` classes.

Problem Statement Turn the binary search tree class into a generic `BinarySearchTree<E>` that stores elements of type `E`.

Adding the Type Parameter

The types that we use in a binary search tree must be comparable, so we declare the class as

```
public class BinarySearchTree<E extends Comparable>
```

We replace the parameter variables of type `Comparable` in the following methods with the type parameter `E`:

```
public void add(E obj)
public boolean find(E obj)
public void remove(E obj)
```

As it happens, there are no other local variables of type `Comparable` to replace. But the `data` instance variable of the inner `Node` class needs to be changed from `Comparable` to `E`.

```
public class BinarySearchTree<E extends Comparable>
{
   . . .
   class Node
   {
      public E data;
      public Node left;
      public Node right;
      . . .
   }
}
```

Note that the `Node` class is *not* a generic class. It is a regular class that is nested inside the generic `BinarySearchTree<E>` class. For example, if `E` is `String`, we have an inner class `BinarySearchTree<String>.Node` with a `data` instance variable of type `String`.

In contrast, let us supply an `inorder` method that accepts a visitor, and let's make `Visitor` a top-level interface (unlike the implementation in Section 17.4 where it was declared inside the tree class.) We need a type parameter for the parameter variable of the `visit` method. Because `Visitor` is not nested inside a generic class, we must make it generic.

```
public interface Visitor<E>
{
   void visit(E data)
}
```

We can then implement the `inorder` method in the usual way:

```
public void inorder(Visitor<E> v)
{
   inorder(root, v);
}

private void inorder(Node parent, Visitor<E> v)
{
   if (parent == null) { return; }
   inorder(parent.left, v);
```

```
      v.visit(parent.data);
      inorder(parent.right, v);
   }
```

Note that the parent parameter variable doesn't need a type parameter.

With these modifications, we have a fully functioning BinarySearchTree class. You can try out the TreeTester program, and it will work correctly.

worked_example_1/TreeTester.java

```
public class TreeTester
{
   public static void main(String[] args)
   {
      BinarySearchTree<String> names = new BinarySearchTree<>();
      names.add("Romeo");
      names.add("Juliet");
      names.add("Tom");
      names.add("Dick");
      names.add("Harry");

      class PrintVisitor implements Visitor<String>
      {
         public void visit(String data)
         {
            System.out.print(data + " ");
         }
      }

      names.inorder(new PrintVisitor());
      System.out.println();

      System.out.println("Expected: Dick Harry Juliet Romeo Tom");
   }
}
```

worked_example_1/BinarySearchTree.java

```
/**
   This class implements a binary search tree whose
   nodes hold objects that implement the Comparable
   interface.
*/
public class BinarySearchTree<E extends Comparable>
{
   private Node root;

   /**
      Constructs an empty tree.
   */
   public BinarySearchTree()
   {
      root = null;
   }

   /**
      Inserts a new node into the tree.
      @param obj the object to insert
   */
```

```
   public void add(E obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) { root = newNode; }
      else { root.addNode(newNode); }
   }

   /**
      Tries to find an object in the tree.
      @param obj the object to find
      @return true if the object is contained in the tree
   */
   public boolean find(E obj)
   {
      Node current = root;
      while (current != null)
      {
         int d = current.data.compareTo(obj);
         if (d == 0) { return true; }
         else if (d > 0) { current = current.left; }
         else { current = current.right; }
      }
      return false;
   }

   /**
      Tries to remove an object from the tree. Does nothing
      if the object is not contained in the tree.
      @param obj the object to remove
   */
   public void remove(E obj)
   {
      // Find node to be removed

      Node toBeRemoved = root;
      Node parent = null;
      boolean found = false;
      while (!found && toBeRemoved != null)
      {
         int d = toBeRemoved.data.compareTo(obj);
         if (d == 0) { found = true; }
         else
         {
            parent = toBeRemoved;
            if (d > 0) { toBeRemoved = toBeRemoved.left; }
            else { toBeRemoved = toBeRemoved.right; }
         }
      }

      if (!found) { return; }

      // toBeRemoved contains obj

      // If one of the children is empty, use the other

      if (toBeRemoved.left == null || toBeRemoved.right == null)
      {
```

```
         Node newChild;
         if (toBeRemoved.left == null)
         {
            newChild = toBeRemoved.right;
         }
         else
         {
            newChild = toBeRemoved.left;
         }

         if (parent == null) // Found in root
         {
            root = newChild;
         }
         else if (parent.left == toBeRemoved)
         {
            parent.left = newChild;
         }
         else
         {
            parent.right = newChild;
         }
         return;
      }

      // Neither subtree is empty

      // Find smallest element of the right subtree

      Node smallestParent = toBeRemoved;
      Node smallest = toBeRemoved.right;
      while (smallest.left != null)
      {
         smallestParent = smallest;
         smallest = smallest.left;
      }

      // smallest contains smallest child in right subtree

      // Move contents, unlink child

      toBeRemoved.data = smallest.data;
      if (smallestParent == toBeRemoved)
      {
         smallestParent.right = smallest.right;
      }
      else
      {
         smallestParent.left = smallest.right;
      }
   }

   /**
      Prints the contents of the tree in sorted order.
   */
   public void inorder(Visitor<E> v)
   {
      inorder(root, v);
   }
```

```
   /**
      Prints a node and all of its descendants in sorted order.
      @param parent the root of the subtree to print
   */
   private void inorder(Node parent, Visitor<E> v)
   {
      if (parent == null) { return; }
      inorder(parent.left, v);
      v.visit(parent.data);
      inorder(parent.right, v);
   }

   /**
      A node of a tree stores a data item and references
      of the child nodes to the left and to the right.
   */
   class Node
   {
      public E data;
      public Node left;
      public Node right;

      /**
         Inserts a new node as a descendant of this node.
         @param newNode the node to insert
      */
      public void addNode(Node newNode)
      {
         int comp = newNode.data.compareTo(data);
         if (comp < 0)
         {
            if (left == null) { left = newNode; }
            else { left.addNode(newNode); }
         }
         else if (comp > 0)
         {
            if (right == null) { right = newNode; }
            else { right.addNode(newNode); }
         }
      }
   }
}
```

worked_example_1/Visitor.java

```
public interface Visitor<E>
{
   /**
      This method is called for each visited node.
      @param data the data of the node
   */
   void visit(E data);
}
```

worked_example_1/Person.java

```
/**
   A person with a name.
*/
public class Person implements Comparable<Person>
{
   private String name;

   /**
      Constructs a Person object.
      @param aName the name of the person
   */
   public Person(String aName)
   {
      name = aName;
   }

   public String toString()
   {
      return getClass().getName() + "[name=" + name + "]";
   }

   public int compareTo(Person other)
   {
      return name.compareTo(other.name);
   }
}
```

worked_example_1/Student.java

```
/**
   A student with a name and a major.
*/
public class Student extends Person
{
   private String major;

   /**
      Constructs a Student object.
      @param aName the name of the student
      @param aMajor the major of the student
   */
   public Student(String aName, String aMajor)
   {
      super(aName);
      major = aMajor;
   }

   public String toString()
   {
      return super.toString() + "[major=" + major + "]";
   }
}
```

In the following sections, we will discuss additional refinements that are described in Special Topic 18.1 and Common Error 18.2. You can skip this discussion if you are not interested in the finer points of Java generics.

Wildcards

Consider the following simple change to the `PrintVisitor` class in the `TreeTester` program. We don't really need to require that `data` is a string. The printing code will work for any object:

```
class PrintVisitor implements Visitor<Object>
{
   public void visit(Object data)
   {
      System.out.print(data + " ");
   }
}
```

Unfortunately, now the `inorder` method of a `BinarySearchTree<String>` will no longer accept a new `PrintVisitor()`. It wants a `Visitor<String>`, not a `Visitor<Object>`. That's a shame. Wildcards were invented to overcome this problem.

There is no harm in passing a `String` value to a `visit` method with an `Object` parameter. In general, the data value of type `E` can be passed to a `visit` method that receives a supertype of `E`. You use a wildcard to spell this out:

```
public void inorder(Visitor<? super E> v)
```

The `inorder` method works with a visitor for any supertype of `E`.

The Generic Comparable Type

The `Comparable` type is a generic type. A `Comparable<T>` has a `compareTo` method with a parameter of type `T`:

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

For example, `String` implements `Comparable<String>`.

We should make use of the type parameter in the declaration of the `BinarySearchTree` class. Instead of

```
public class BinarySearchTree<E extends Comparable>
```

we can write

```
public class BinarySearchTree<E extends Comparable<E>>
```

With this change, the unsightly warnings at the calls to `compareTo` go away.

But that's not quite good enough. Consider the following class:

```
public class Person implements Comparable<Person>
{
   . . .
   public int compareTo(Person other)
   {
      return name.compareTo(other.name);
   }
}
```

People are just compared by name.

We have a subclass

```
public class Student extends Person { . . . }
```

Students are people. How are they compared? Also by name. Note that `Student` implements `Comparable<Person>`, not `Comparable<Student>`.

That means we can't have a `BinarySearchTree<Student>`! Again, wildcards come to the rescue. The proper type bound is

```
public class BinarySearchTree<E extends Comparable<? super E>>
```

Static Contexts

Look again into the first section, where we implemented the `inorder` method in the "usual way":

```
private void inorder(Node parent, Visitor<E> v)
{
   if (parent == null) { return; }
   inorder(parent.left, v);
   v.visit(parent.data);
   inorder(parent.right, v);
}
```

Actually, that wasn't quite the usual way. In the usual way, the recursive helper method is static. But if you try that, you get an error. In a "static context", the generic parameters don't work as expected. There is only a single static method for *all* parameters `E`, so you can't use `E` in a static method. (This is a consequence of type erasure. There is only a single `BinarySearchTree` class in which `E` is erased.)

The workaround is to make the method generic, like this:

```
private static <T> void inorder(Node parent, Visitor<T> v)
```

That is better, but it isn't quite right because `Node` is defined inside a generic class, so we need to specify what kind of node we need.

```
private static <T> void inorder(BinarySearchTree<T>.Node parent, Visitor<T> v)
```

We are getting closer, but the compiler now complains that a `BinarySearchTree` is only defined for types `T` that implement `Comparable`. Fair enough:

```
private static <T extends Comparable<? super T>> void inorder(
      BinarySearchTree<T>.Node parent, Visitor<T> v)
```

This method declaration is unfortunately somewhat complex, but it accurately reflects all requirements that must be fulfilled for the method to work. The type `T` must be comparable. The `Node` must belong to the `BinarySearchTree` with type `T`, and the visitor must be an instance of `Visitor<T>` with the same `T`.

Actually, that's still not right. As mentioned in the section on wildcards, there is nothing wrong with using a visitor for a supertype of `T`. The most general form is

```
private static <T extends Comparable<? super T>> void inorder(
      BinarySearchTree<T>.Node parent, Visitor<? super T> v)
```

With generics, the implementor must give precise specifications so that the programmers using the generic construct can do so under the most general circumstances.

worked_example_1/TreeTester2.java

```
/**
   This class demonstrates the advanced techniques in BinarySearchTree2.
*/
public class TreeTester2
{
   public static void main(String[] args)
   {
      BinarySearchTree2<Student> students = new BinarySearchTree2<>();
      // Can form BinarySearchTree2<Student> even though Student
      // implements Comparable<Person> and not Comparable<Student>
```

```

      students.add(new Student("Romeo", "Art History"));
      students.add(new Student("Juliet", "CS"));
      students.add(new Student("Tom", "Leisure Studies"));
      students.add(new Student("Diana", "EE"));
      students.add(new Student("Harry", "Biology"));

      class PrintVisitor implements Visitor<Object>
      {
         public void visit(Object data)
         {
            System.out.println(data);
         }
      }

      // Can pass a Visitor<Object>, not just a Visitor<Student>
      students.inorder(new PrintVisitor());
   }
}
```

worked_example_1/BinarySearchTree2.java

```
/**
   This class implements a binary search tree whose
   nodes hold objects that implement the Comparable
   interface for an appropriate type parameter.
*/
public class BinarySearchTree2<E extends Comparable<? super E>>
{
   private Node root;

   /**
      Constructs an empty tree.
   */
   public BinarySearchTree2()
   {
      root = null;
   }

   /**
      Inserts a new node into the tree.
      @param obj the object to insert
   */
   public void add(E obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) { root = newNode; }
      else { root.addNode(newNode); }
   }

   /**
      Tries to find an object in the tree.
      @param obj the object to find
      @return true if the object is contained in the tree
   */
```

```
   public boolean find(E obj)
   {
      Node current = root;
      while (current != null)
      {
         int d = current.data.compareTo(obj);
         if (d == 0) { return true; }
         else if (d > 0) { current = current.left; }
         else { current = current.right; }
      }
      return false;
   }

   /**
      Tries to remove an object from the tree. Does nothing
      if the object is not contained in the tree.
      @param obj the object to remove
   */
   public void remove(E obj)
   {
      // Find node to be removed

      Node toBeRemoved = root;
      Node parent = null;
      boolean found = false;
      while (!found && toBeRemoved != null)
      {
         int d = toBeRemoved.data.compareTo(obj);
         if (d == 0) { found = true; }
         else
         {
            parent = toBeRemoved;
            if (d > 0) { toBeRemoved = toBeRemoved.left; }
            else { toBeRemoved = toBeRemoved.right; }
         }
      }

      if (!found) { return; }

      // toBeRemoved contains obj

      // If one of the children is empty, use the other

      if (toBeRemoved.left == null || toBeRemoved.right == null)
      {
         Node newChild;
         if (toBeRemoved.left == null)
         {
            newChild = toBeRemoved.right;
         }
         else
         {
            newChild = toBeRemoved.left;
         }

         if (parent == null) // Found in root
         {
            root = newChild;
```

```
         }
         else if (parent.left == toBeRemoved)
         {
            parent.left = newChild;
         }
         else
         {
            parent.right = newChild;
         }
         return;
      }

      // Neither subtree is empty

      // Find smallest element of the right subtree

      Node smallestParent = toBeRemoved;
      Node smallest = toBeRemoved.right;
      while (smallest.left != null)
      {
         smallestParent = smallest;
         smallest = smallest.left;
      }

      // smallest contains smallest child in right subtree

      // Move contents, unlink child

      toBeRemoved.data = smallest.data;
      if (smallestParent == toBeRemoved)
      {
         smallestParent.right = smallest.right;
      }
      else
      {
         smallestParent.left = smallest.right;
      }
   }

   /**
      Prints the contents of the tree in sorted order.
   */
   public void inorder(Visitor<? super E> v)
   {
      inorder(root, v);
   }

   /**
      Prints a node and all of its descendants in sorted order.
      @param parent the root of the subtree to print
   */
   private static <T extends Comparable<? super T>> void
      inorder(BinarySearchTree2<T>.Node parent, Visitor<? super T> v)
   {
      if (parent == null) { return; }
      inorder(parent.left, v);
      v.visit(parent.data);
      inorder(parent.right, v);
   }
```

```

   /**
      A node of a tree stores a data item and references
      of the child nodes to the left and to the right.
   */
   class Node
   {
      public E data;
      public Node left;
      public Node right;

      /**
         Inserts a new node as a descendant of this node.
         @param newNode the node to insert
      */
      public void addNode(Node newNode)
      {
         int comp = newNode.data.compareTo(data);
         if (comp < 0)
         {
            if (left == null) { left = newNode; }
            else { left.addNode(newNode); }
         }
         else if (comp > 0)
         {
            if (right == null) { right = newNode; }
            else { right.addNode(newNode); }
         }
      }
   }
}
```