Chapter 7

Polymorphism

Previously, we developed two data structures that implemented the *list* abstract data type: linked lists and array lists. However, these implementations were unsatisfying along two dimensions:

1. Even though a linked list and array list provide the same functions to a user, they are not *interchangable* in user code. If the user wanted the flexibility of choosing between a linked list or array list (because they have different performance characteristics), they would need to duplicate code who's only difference would be the type of the list it uses.
2. The *carrier type* of either list implementation is fixed. If we wanted a list that held integers as well as a list that contained strings, we would need two separate implementations of the list that only differed in the type of the value at each node.

To solve both these problems, Java provides us mechanisms to write *polymorphic* code. Polymorphic code is code that can operate over multiple types. In particular, the two problems described above are addressed with two types of polymorphisms: subtype polymorphism and parametric polymorphism.

7.1 Subtype Polymorphism

The list abstract data type defined a number of operations that all "list-like" objects ought to implement:

- `int size()`,
- `void add(int x)`,
- `void insert(int x, int index)`,
- `void clear()`,
- `int get(int index)`, and
- `int remove(int index)`.

Our linked list and array list classes implemented these methods. However, there was no enforcement by the compiler that these classes actually implemented these operations. Furthermore, even though the two list implementations provided the exact same set of methods to the user, we could not interchange one list for another because they are different types. For example, if we have a method:

```
public int sumAll(LinkedList l) {
    int sum = 0;
    for (int i = 0; i < l.size(); i++) {
        sum += l.get(i);
    }
    return sum;
}
```

A programmer cannot pass an array list to this method even though it would work perfectly fine with one.

To this end, we need a programming mechanism for capturing the shared behavior between linked list and array list, enforcing that they indeed share this common behavior as well as allowing us to interchange them in our code. This mechanism in Java is called the *interface*.

An interface looks like a class declaration except that:

1. It uses the `interface` keywords in its definition rather than `class`.
2. It only contains (`public`) methods.
3. It does not provide any implementation of these methods; in other words, the interface only contains *method signatures*.

For example, here is the realization of our list ADT as a Java interface called `List`. Note that like classes, a `public` interface must be declared in its own file whose name matches the interface's name:

```
// In List.java
public interface List {
    public int size();
    public void add(int x);
    public void insert(int x, int index);
    public void clear();
    public int get(int index), a;
    public int remove(int index);
}
```

Also like a class, the interface declares a type that our local variables can take on:

```
List l;
```

However, unlike a class, we can't instantiate an interface as it is not a class (*i.e.*, blueprint for an actual object):

```
/* Compiler error: List is an interface; it cannot be instantiated. */
List l = new List();
```

This alone is not particularly helpful. We must combine this interface declaration with classes that would like to *implement* the operations defined by this interface. To do this, we note in the relevant class's declaration that it implements the list interface:

```
// In LinkedList.java
public class LinkedList implements List {
    // The usual definition of the linked list class
}
```

Now, the compiler will check the `LinkedList` provides implementations for each of the methods described by the `List` interface. If the `LinkedList` class does not possess one of these methods, then the compiler will raise an error.

By going through this process, the programmer establishes a *subtyping* or *is-a* relationship between the `List` interface and the `LinkedList` class. Because the `LinkedList` class implements the `List`, it must possess all the methods declared by the `List` interface. Therefore, we can use a `LinkedList` in any context where a `List` is expected. In particular, we can assign a `LinkedList` to a `List` variable:

```
List l = new LinkedList();     // Ok!
```

Or we can pass a `LinkedList` to a method expecting a `List`:

```
// Note: the parameter to sumAll is now a List
public int sumAll(List l) {
    int sum = 0;
    for (int i = 0; i < l.size(); i++) {
        sum += l.get(i);
    }
    return sum;
}
// ...
LinkedList l = new LinkedList();
sumAll(l);    // Ok!
```

If our `ArrayList` class also implements the `List` interface:

```
// In ArrayList.java
public class ArrayList implements List {
    //The usual definition of the array list class
}
```

Then we can also pass an `ArrayList` to `sumAll`:

```
ArrayList l = new ArrayList();
sumAll(l);    // Also ok!
```

We say that `sumAll` is *polymorphic* over the type of list it accepts; in fact, it can take any class that implements the `List` interface!

7.1.1 Static and Dynamic Types

The variable declaration:

```
List l = new ArrayList();
```

deserves some study because it is the first instance where the type of the variable does not coincide with the actual type of the object that it is referencing. As a running example, consider the following toy interface and classes:

```
public interface I {
    public void f();
}

public class A implements I {
    public void f() { System.out.println("A.f"); }
}

public class B implements I {
    public void f() { System.out.println("B.f"); }
    public void g() { System.out.println("B.g"); }
}
```

Before interfaces, the only variable declarations and assignments we could perform were:

```
A a = new A();  // Ok!
B b = new B();  // Ok!
```

Through `a`, we can call `A`'s `f` method and through `b`, we can call `B`'s `f` and `g` methods. In contrast, we know that the variable assignment `A a = new B();` fails at compile time because `A` and `B` are different types.

With interfaces and subtype polymorphism, we can now perform the following assignments

```
I i1 = new A();  // Ok!
I i2 = new B();  // Ok!
```

because both `A` and `B` implement the `I` interface. When we call the `f` method on each of `i1` and `i2`, the program produces the following output:

```
i1.f();   // A.f
i2.f();   // B.f
```

`I` itself does not provide a implementation for `f`; instead, the concrete classes `A` and `B` do so. Therefore, when we invoke `f`, the program uses the type of the *actual object* that the variable references to determine what method should be invoked. For `i1`, the type of object it refers to is `A` and for `i2`, the type of the object it refers to is `B`. We assign the term *dynamic type* to the actual type of the underlying value of a program and the *static type* to the type that the compiler knows about during compilation. The static type for both `i1` and `i2` is `I` which comes from the type of the declared variable.

This seems rather pointless with variable declaration and assignment. However, the nuances become more pronounced when we look at a method:

```
public static void test1(I i) {
    i.f();
}
```

This method simply calls `f` on its argument, but now it is not clear which `f` will be invoked. Indeed, we don't have enough information to determine this by looking at the method alone as all we know is the static type of `i` (`I`)! We must consider particular method calls and the actual values passed to those calls:

```
test1(new A())   // a.f()
test1(new B())   // b.f()
```

In summary, when we are checking the type of the variable during compilation, we only consider the static type of the variable. When we perform method lookup, we look at the dynamic type of the object to determine which method to invoke. By doing this, we will never try to call a method (or look up the field) that an object does not possess. To see this, note that the static type only specifies a *subset* of the overall functionality of an object. That is, an object must implement all the methods of an interface it implements but is free to implement others. Therefore, if an object has a particular static type, we know that it at least has the behavior that the static type guarantees.

This leads to one counter-intuitive situation with subtyping

```
I i1 = new A();
I i2 = new B();
i1.g();   // Fails at compilation!
i2.g();   // Also fails at compilation!
```

It makes sense that `i1.g()` should fail because `A` does not provide a `g` method. But `i2.g()` should work because `B` does provide a `g` method. Nevertheless, the compiler rejects both method invocations. Why?

Appealing to our rules about type checking, we note that the compiler uses the *static* type of `i2` during the type checking phase of compilation. Therefore, when checking to see if `i2.g()` is a valid method call at compilation time, the compiler checks to see if `I` has a `g` method, rather than `B`! It does not, so type checking fails. In this sense, type checking is not a *complete* procedure. Type checking will also ensure that a program does not mix up its types. However, type checking will rule out some perfectly sensible programs.

To see why this must be the case, consider a variation on the `test1` method from before:

```
public static void test2(I i) {
    i.g();    // Also fails to compile.
}
```

Unlike the variable declaration case, it is more obvious here that we must rule out `i.g()` because we do not know when type checking the definition of `test2` what values may be passed to it.

7.1.2 Use of the Interface Type

With our toy example, we have two possible choices when declaring a variable that will reference a `B`:

```
I i = new B();
B b = new B();
```

With `b`, we can invoke `B`'s `g` method but with `i`, we cannot. It seems like a no-brainer to always favor the concrete type over the interface type.

However, it turns out that the opposite is true! We want to prefer using the type that is the *most general* type possible that still lets us perform the operations we want on the object. This has two effects:

1. Less importantly, if we need to be more flexible with our types, *e.g.*, we want to pass either a linked list or array list to a method, the interface type accommodates this need.
2. More importantly, if we only need the methods of the interface type and not the additional methods of the concrete type, then using the interface type documents this fact.

Concretely, it is rare that you will need to use a list but then require either linked list- or array list-specific methods. Because of this, you should use the interface type `List`—more generally, the interface type—whenever declaring variables (locals, fields, or parameters) unless your code requires functionality from a particular concrete type.

Exercise (Subtyping as Approximation) Consider the following interface and class declarations:

```
public interface I { public void foo(); }
public class C implements I {
    public void foo() { System.out.println("C.foo"); }
    public void bar() { System.out.println("C.bar"); }
}
```

And consider the following code snippet:

```
I i = new C();
i.foo();    // (A)
i.bar();    // (B)
```

For lines (A) and (B) above, answer the following questions:

1. Does the line of code type check?
2. If so, what is the output?

Recall that the static type of `i` is `I` and the dynamic type is `C`. What does this exercise tell you about how *precise* subtyping is? In particular, is there code that is *safe* but *rejected* by Java's type system?

7.2 Parametric Polymorphism

The other pain point concerning our list implementations was the inflexibility surrounding the *carrier type* of the list, *i.e.*, the type of elements the list held. In our original example, we required, *e.g.*, our array list hold integers:

```
public class ArrayList implements List {
    private int[] data;
    // The rest of the class...
}
```

However, if we wanted to have an array list of some other type, say strings, we needed a whole different class to do this:

```
public class ArrayStringList implements StringList {
    private String[] data;
    // The rest of the class...
}
```

This class would be a complete duplicate of the original `ArrayList` class except for the carrier type! Furthermore, with our interfaces, we would need to create a new interface type that handles lists of strings. This is clearly an unacceptable situation.

The key insight to solving this problem is realizing that the list doesn't *care* about its carrier type. All of our list implementations don't depend on having an `int` or a `String`. The type might as well be a *variable* that the implementation does not know anything about, just like how a method does not know anything about its parameters (other than its type).

In Java, classes that are parameterized by types in this manner are called *generic classes*. Here is a non-generic version of our linked list's `Node` class specialized to strings:

```
// In Node.java
public class Node {
    public String value;
    public Node next;
    public Node(String value, Node next) {
        this.value = value;
        this.next = next;
    }
}
```

And here is the generic version of the `Node` class:

```
// In Node.java
public class Node<T> {
    public T value;
    public Node<T> next;
    public Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
```

Note the differences:

1. Next to the name of the class, we declare the common-separated list of *type parameters* that the class takes in angle brackets. By convention, we usually use the letters `T` and `U` for these type parameters.
2. Where ever we mentioned `String`, we now replace it with our type variable `T`.
3. Where ever we mention the type of `Node`, we must now provide a concrete type to pass along for the class's type parameter. Note that inside the class, the occurrence of `T` is a *use* of the type variable `T`, not a declaration (which occurs at the top of the class).

Now that our node class is generic, we must also make our `LinkedList` list class generic:

```
// In LinkedList.java
public class LinkedList<T> {
    private Node<T> first;
    public LinkedList() { first = null; }
    public void add(T v) {
        if (first == null) {
            first = new Node<T>(v, null);
        } else {
```

```
            Node<T> cur = first;
            while (cur.next != null) { cur = cur.next; }
            cur.next = new Node<T>(v, null);
        }
    }
    // Additional implementation...
}
```

Our generic linked list uses the generic node class, threading along the type variable `T`. Note that when instantiating a generic class, like the `Node`, the class name also includes the instantiating type. This is annoying to do, so Java provides a little amount of *type inference*—inferring the type to pass in based on context. This allows us to instantiate a node with the following syntax instead:

```
first = new Node<>(v, null);
```

where the angle brackets are empty rather than containing the instantiating type.

Now the user of the linked list must specify a concrete type that the linked list ought to hold:

```
LinkedList<String> l1 = new LinkedList<String>();
l1.add("hello");
l1.add("world");
```

The act of passing a concrete type to a generic class is called *instantiating* the class.

By specializing `l` to be a list of strings, we can not add values of other types to this particular linked list. The concrete type only applies to `l`, so we can create another list to hold some other type:

```
  LinkedList<Color> l2 = new LinkedList<Color>();     // Holds Color objects.
```

Finally, we must also make our interface generic to match our `LinkedList`:

```
// In List.java
public interface List<T> {
    public int size();
    public void add(T x);
    public void insert(T x, int index);
    public void clear();
    public T get(int index), a;
    public T remove(int index);
}
```

We then need to change the declaration of our `LinkedList` class thusly:

```
// In LinkedList.java
public class LinkedList<T> implements List<T> {
    // Implementation...
}
```

Note that the first `<T>` (attached to `LinkedList`) is the *declaration* of the type parameter `T`. The second `<T>` (attached to `List`) is a *use* of the type parameter, passing it along to the `List` interface.

7.2.1 Autoboxing and Unboxing

The following code produces a surprising compiler error:

```
List<int> l = new List<>();   // error: cannot instantiate a generic
                              // with a primitive type.
```

It appears that it is impossible to instantiate generics with primitive types, and it is because a generic type parameter must be instantiated to a type variable or a reference type.

Recall that a reference type means that variables of that type are references to objects rather than objects themselves. Because of this, implementing, *e.g.*, a generic field is trivial to do because pointers are the same size no matter what sort of object they are pointing to. This means that the *layout* of the object is identical, irrespective of what the generic type is instantiated to. However, primitives have different sorts of sizes, *e.g.*, a `double` is likely a different size than `int`, and so objects that differ by these types have different layouts.

The workaround is to wrap a primitive in a class. That way, this wrapper can be the type that our generic class holds. Java provides a number of these *wrapper classes* for every primitive:

Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`short`	`Short`
`char`	`Character`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

(These wrapper classes also provide helper methods for working with these primitive data, *e.g.*, `Integer.parseInt` for parsing a string as an `int`.)

To use these wrapper classes, we instantiate them with a value of an appropriate type which can then be held in our generic list:

```
List<Integer> l = new LinkedList<>();
l.add(new Integer(5));   // Now ok!
int n = l.get(0).intValue();
```

This is also cumbersome to do, so Java provides facilities for automatically putting a primitive type in a wrapper and then unwrapping it when necessary, called *autoboxing* and *unboxing*. With autoboxing, the code to use integers with a generic list is quite natural:

```
List<Integer> l = new LinkedList<>();
l.add(5);           // Auto-boxed int -> Integer
int n = l.get(0);   // Auto-unboxed Integer -> int
```

7.2.2 Generics and Arrays

Finally, there is one additional detail to consider with respect to generics and arrays. Suppose that we were writing our array list class with generics. The following code does not compile!

```
public class ArrayList<T> implements List<T> {
    private T[] data;
    private int last;
    public ArrayList() {
        data = new T[16];   // Error: cannot instantiate generic array!
        last = 0;
    }
    // Rest of implementation...
}
```

It turns out that a (non-)feature of Java generics, *generic type erasure*, does not play well with arrays. Because of this, we cannot instantiate arrays of generic type. To get around this, we must proceed in a roundabout way:

```
public class ArrayList<T> implements List<T> {
    private Object[] data;
    private int last;
    public ArrayList() {
        data = new Object[16];
        last = 0;
    }
    public void add(T v) {
        ensureCapacity();
        data[last++] = v;
    }
    public T get(int n) {
        if (n < 0 || n >= last) { throw new IllegalArgumentException(); }
        @SuppressWarnings("unchecked")
        T ret = (T) data[n];
        return ret;;
    }
    // Rest of implementation...
}
```

Rather than instantiating an array of generic type, we instantiate the array to be an array of type `Object`. We'll discuss the `Object` class later in the course, but for now, we can think of `Object` as a type that everything is an instance of. Because a generic `T` is an `Object`, then `add` can put `v` in the array directly. However, `get` must *cast* the result of fetching data from the array since not every object is a `T`. When performing this cast, the compiler reports a warning saying that the cast is unchecked; we can suppress this warning with an *annotation* above the offending statement. The annotation `SuppressWarnings` tells the compiler to ignore warnings of a particular type that are generated by the decorated statement.

Exercise (Generalizing Code) Consider the following code snippet:

```
public class Box {
    private int value;
```

```
    public Box(int value) { this.value = value; }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
}
```

Change the definition of Box so that it can hold any type rather than just integers.