PHAS3459: Scientific Programming Using Object Oriented Languages

Module 6: Inheritance, Abstract Methods and Interfaces

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 6

The aim of this module is to give you a working knowledge of the very powerful concepts of *inheritance*, *abstract classes* and *interfaces* within the Java language. Effective use of these language constructs is essential for good object-oriented program design. By the end of this module you should:

- Be able to recognise when an inheritance relationship between a superclass (or "base-class") and corresponding subclasses is appropriate.
- Be able to understand abstract classes and interfaces and when it is appropriate to use one or the other (or indeed both) in your program design.
- Understand the syntax of inheritance, abstract classes and methods, and interfaces.
- Have a deeper understanding of polymorphism attained through the use of superclasses and interfaces.

2. Extending Classes: Inheritance and Polymorphism

Inheritance and *polymorphism* are two of the "three pillars of object-oriented programming" alongside *encapsulation*, which we have already encountered.

2.1 Inheritance

Inheritance is a mechanism for *extending* one class to create a new class with different, usually more specialised functionality. The new class is said to be a *subclass* of the original, which is a *superclass* of the new one. The subclass can benefit from the functions already provided by the superclass, while adding its own or replacing some or all of the originals. This is often a useful way of modelling real-world entities.

For example, different types of *particle* have many properties in common: momentum, energy, charge, spin, mass etc. (Some of these properties may have the value zero, but the particle still has the property!) However, more specific types of particle may also have their own properties or behaviour that they do not share with other types: a *muon* may decay; an *electron* undergoes bremsstrahlung.

We specify that a class inherits the behaviour of a parent class using the keyword `extends` in the line where we define the class name, e.g. instead of:

```
public class Electron {
  public Electron() {}
}
```

we have:

```
public class Electron extends Particle {
  public Electron() {}
}
```

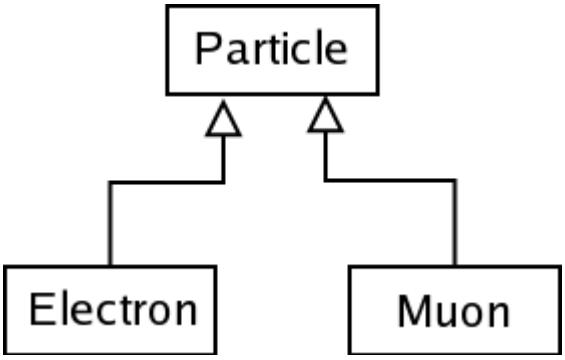

Figure 6.1: A simple example of two classes, `Electron` and `Muon` both extending a single superclass, `Particle`.

Any number of subclasses can extend the same superclass, but in Java any one subclass can only extend a single superclass. Figure 6.1 illustrates a simple case with one superclass and two subclasses. We are not limited to one layer of inheritance: the inheritance hierarchy, or tree, can be as deep as needed. Generally, the further down the hierarchy a class appears, the more specialized its behaviour. Figure 6.2 shows a multi-level hierarchy where a general `Particle` class is extended by two classes, `Lepton` and `Hadron`, which could implement behaviour specific to these classes of particle. These are then extended by classes such as `Electron` to add the details that are really specific to only one species of particle.

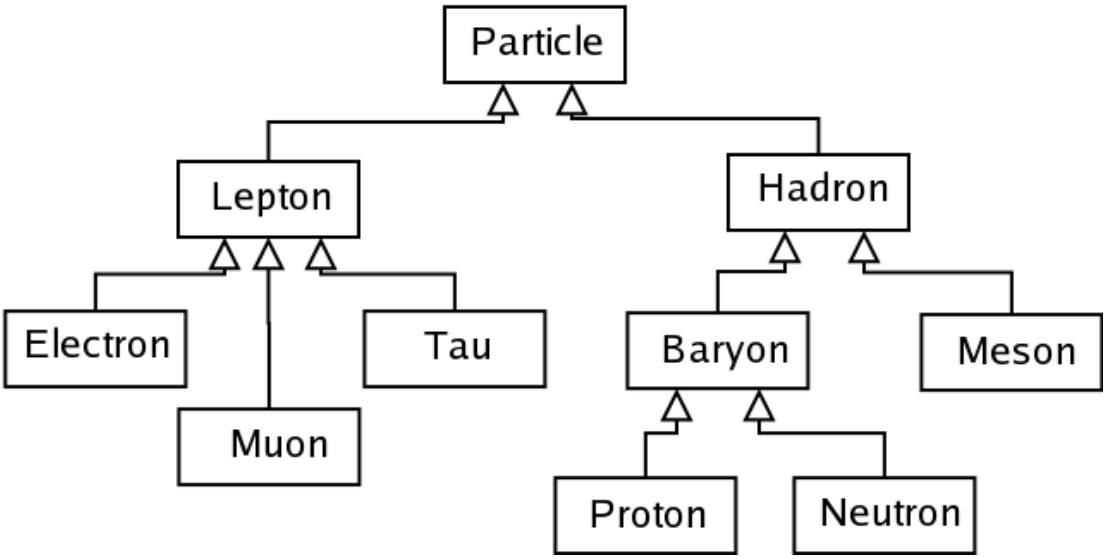

Figure 6.2: A more extended example of inheritance, illustrating a multi-level hierarchy of classes.

To instantiate an `Electron` object we use the same syntax as usual:

```
Electron anElectron = new Electron();
```

2.1.1 Inheriting Methods and Variables

When we create an `Electron` class we are implicitly creating a `Particle` object first, and then adding the specific details that make this particular `Particle` into an `Electron`. The `Electron` object inherits all the variables and methods of the `Particle` class, provided they are declared `public` or `protected` in the `Particle` class. Variables declared `private` in `Particle` are hidden to the subclass, although they are still present and can be accessed via `public` or `protected` methods.

Suppose we create a simple class:

```
public class SuperClass {
  protected String name = "SuperClass";

  protected void print(String text) {
    System.out.println("SuperClass print: "+text);
  }

  protected String className() {return name;}
}
```

Then we create a subclass that extends `SuperClass` but defines none of its own methods or variables:

```
public class SubClass extends SuperClass {}
```

Now we can see how the class `SubClass` inherits its behaviour from `SuperClass`:

```
SuperClass a = new SuperClass();
a.print("a.print()");
System.out.println("a.className() = "+a.className());

SubClass sub = new SubClass();
sub.print("sub.print()");
System.out.println("sub.className = "+sub.className());
```

We can of course add behaviour to the subclass that is not provided by the superclass:

```
public class SubClass extends SuperClass {
  protected int square(int i) {return i*i;}
}
```

Methods and variables declared `protected` are less tightly enclosed in a class than those that are `private` in that they are accessible both to subclasses of the class in which they are declared, and to other classes in the same package. The use of *packages* to organize classes in Java will be covered in more detail in Module 7. The use of `protected` members is often a useful compromise between `private` ones, which can only be accessed in the class in which they are declared, and `public` ones, which are accessible to all classes in any package and thus provide no encapsulation at all. The fourth option is the *default* or

package-private access level: if you do not specify any of the keywords `private`, `protected` or `public` the member is accessible to any class in the same package as the class in question, but not to any class outside that package, even if it is a subclass.

2.1.2 Overriding Methods and Member Variables

It may be that the implementation of a method provided by the superclass is not what is really needed by a subclass. In this case the subclass can *override* the method by simply declaring a new version with the same name and taking the same arguments. We can also *hide* a member variable by declaring a new one with the same name in a subclass.

Note that *overriding* is different from *overloading*, as described in Module 2, where the methods involved have the same name but take different arguments.

For example, we can create two new subclasses of the class `SuperClass` given above:

```
public class SubClass1 extends SuperClass {
  protected String className() {return "SubClass1";}
}
```

```
public class SubClass2 extends SuperClass {
  protected String name = "SubClass2";
  protected void print(String text) {
    System.out.println("SubClass2 print: "+text);
  }
}
```

Now we can see what happens when we use these new subclasses:

```
SubClass1 sub1 = new SubClass1();
sub1.print("sub1.print()");
System.out.println("sub1.className = "+sub1.className());

SubClass2 sub2 = new SubClass2();
sub2.print("sub2.print()");
System.out.println("sub2.className = "+sub2.className());
```

Make sure you see what happens when each method is called, and understand why.

As a further example of overriding, we will revisit the `SimpleCounter` class from Module 2 and adapt it slightly. We start with a simple version of the class with no maximum allowed value:

```
public class SimpleCounter {
  protected int counter;

  public SimpleCounter() {}
  public SimpleCounter(int value) { counter = value; }

  public int getCounter() {return counter;}
  public void setCounter(int val) { counter = val; }
}
```

Note that the member variable `counter` is now `protected` instead of `private` to allow for extension of the class.

Now instead of modifying the existing class to impose a limit on the allowed values of the counter, we can create a new class that extends the `SimpleCounter` class. It will keep the existing `counter` variable but add a new static variable `max`, representing the maximum allowed value of the counter. The original `setCounter` method is now no longer adequate and must be replaced by a version that checks the new value before changing the value of the counter:

```
public class LimitedCounter extends SimpleCounter {
  protected static int max=1000;

  public void setCounter(int val) {
    if (val<=max) counter=val;
  }
}
```

There are some restrictions on what can be done when overriding a method. One is that you cannot *reduce the visibility* of a superclass method: e.g. you can't override a `public` method with a `protected` or `private` one. Another is that the method cannot throw any type of exception that is not declared using a `throws` declaration in the superclass method. So the `setCounter` method of `LimitedCounter` cannot throw an exception if the new value is out of range unless we add the `throws Exception` declaration to the `SimpleCounter` class. After reading the Section on Polymorphism you may be able to see why these restrictions exist. They mean that you need to be quite careful when designing a class that you may later want to extend.

2.1.3 Constructors

Like any class, a subclass needs a constructor to tell the computer how to create and initialize an object of that class. The first step in this process is always to initialize the object as if it were an instance of the superclass. This is done by calling the superclass constructor.

The way this is done in general is to use the keyword `super` with the arguments to be passed to the superclass constructor. For an example, look at this class, which also extends `SimpleCounter` by adding a name to each counter object:

```
public class NamedCounter extends SimpleCounter {
  protected String n;

  public NamedCounter(String name) {
    super();
    n = name;
  }

  public NamedCounter(String name, int value) {
    super(value);
    n = name;
  }

  public String getName() {return n;}
}
```

The `NamedCounter` class has two constructors. Each of them first calls a constructor of the superclass, `SimpleCounter`, to initialize the member variables of that class, in this case `counter`. They then carry out the initialization specific to this subclass, in this case setting the name variable `n`. `super()` calls the `SimpleCounter` constructor that takes no arguments, while `super(value)` calls the version that takes an integer argument.

In fact the first constructor doesn't need an explicit call to `super()`. If the first line of a subclass constructor is not a call to `super` the compiler will add one automatically. This will always be a call to the default (no-argument) constructor of the superclass, so if there is no such default constructor, or you need to pass an argument, you need to put in the `super` call yourself.

In our `LimitedCounter` class above, we have not defined a constructor at all. You may recall from Module 2 that in this case a default constructor will automatically be generated that will "do nothing". In this case it will still automatically call the default constructor of the `SimpleCounter` class. This also "does nothing", meaning it leaves `counter` with its default value of zero. If we want to have the option of setting a non-zero initial value for our `LimitedCounter` objects we need to put in two constructors:

```
public LimitedCounter() {}

public LimitedCounter(int val) {
  if (val<=max) counter=val;
}
```

2.2 Polymorphism

Deciding when to use inheritance is not always simple. A rule of thumb is that inheritance is used to model *Is-A* relationships: an electron *is a* particle (a more specific type of particle) and so the `Electron` class can inherit from the `Particle` class. However, a more precise rule is that an object of the subclass should be *substitutable for* an object of the superclass. This brings us to the topic of *polymorphism*.

An instance of the subclass can be used as such, or it can be used anywhere that an instance of the superclass is expected. We can say that an object of the subclass should be *substitutable for* an object of the superclass. This is referred to as *polymorphism*: the same

object can effectively take on *multiple forms*.

In other words we can use variables of type `SimpleCounter` (the superclass) to refer to objects of one of its subclasses, `LimitedCounter` or `NamedCounter`:

```
SimpleCounter a = new SimpleCounter(2);
SimpleCounter b = new LimitedCounter();

System.out.println(a.getCounter());
b.setCounter(1001);
System.out.println(b.getCounter());
```

Here we use the `SimpleCounter` references `b` to manipulate a `LimitedCounter` object. Note that the `setCounter` method used is the one belonging to the actual class of the object. In this case it is the `setCounter` method of the `LimitedCounter` class, so it checks the new value against `max` and does not change the value of the counter.

However, we can only use methods that exist in the `SimpleCounter` class, even if they are overridden in the subclass. The following code snippet will not work:

```
SimpleCounter c = new NamedCounter("protons",0);
System.out.println(c.getName());  // Doesn't compile!
```

If we are sure that the `SimpleCounter` variable is in fact referring to a `NamedCounter` and want to call one of the methods that belongs only to the subclass, we have to cast the variable first:

```
NamedCounter d = (NamedCounter) c;
System.out.println(d.getName());
```

This should be done with caution since if it were to turn out that on some occasion the object referred to by `c` was *not* in fact a `NamedCounter` object, the cast would fail and the program would crash when it was run.

Now it may be clearer why looking for *Is-A* relationships can be misleading when deciding whether to us inheritance, unless we use *Is-A* more specifically to mean *Is-Substitutable-For*. In the example used above, an `NamedCounter` is indeed substitutable for a `SimpleCounter`. Similarly an *electron* is substitutable for a *particle*: any action that can be performed on a particle, such as asking its momentum, can also be performed on an electron as a specific type of particle.

We might be tempted to think that since a square *is a* specific type of rectangle it would be reasonable to make a `Square` class inherit from a `Rectangle` class. However, this example fails the *Is-Substitutable-For* test: a *rectangle* can have its height changed without affecting its width, but doing the same to a *square* does not make sense since the “square” would no longer be an object of the same type.

We can also now see why a subclass method must allow at least as much access (`public`,

`protected` or `private`) as the superclass method it is overriding, and cannot throw an exception that is not declared by the superclass method. Otherwise a programmer using a superclass variable would not know that their code might fail when using an instance of the subclass.

2.3 The `Object` Class

In previous modules you have in fact already used inheritance, perhaps without knowing it. All classes in Java inherit from the built-in `Object` class either directly or indirectly. If you do not explicitly extend a different class, your class will implicitly extend `Object`.

The `Object` class implements several methods, including `toString`, `clone` and `equals`. When you add a `toString` method to a class you are overriding the version provided by `Object`. If you don't do so then when you print one of your objects the `Object` version is used, which prints a string containing the name of the class and a *hash code* for the object. Similarly in Module 5 you learned how to override the `equals` and `clone` methods.

2.4 Exceptions

A common use of inheritance is in defining exception classes to represent specific types of event. This is done as follows:

```
public class TooLazyException extends Exception {
  public TooLazyException() {}

  public TooLazyException(String message) {
    super("TooLazyException ERROR: "+message);
  }
}
```

Eclipse will give you a warning: “The serializable class TooLazyException does not declare a static final SerialVersionUID field of type long”. This does not stop the code from working, but you can get rid of the warning by accepting Eclipse's proposal to “add default serial version ID” or “Add @SuppressWarnings...”.

Our new exception class can be thrown like this:

```
public static double lazyLog(double x) throws TooLazyException {
  if (x==1.0) {return 0;}
  else {
    throw new TooLazyException(
    "Can't be bothered to calculate logarithm of "+x);
  }
}
```

And we can catch it like any other exception:

```
try {
  double d = DodgyCode.lazyLog(3.5);
}
catch (TooLazyException e) {
  System.out.println("Something went wrong!");
  System.out.println(e.getMessage());
}
```

3. Abstract Classes and Interfaces

These two concepts overcome some of the limitations with the basic structure of inheritance that we have visited in previous sections. These limitations include:

- A subclass can only inherit directly from one base-class. What if we want to inherit two sets of functionality, e.g. a `Motorbike` would like to extend the definitions of both `Bicycle` and `MotorisedVehicle`?
- What if we want to *force* subclasses to implement their own versions of certain methods, rather than leaving it optional?
- What if we want to provide a rigid template for how a class is to implement certain functionality, e.g. storing data?

3.1 Abstract Classes

Often we want to define a super-class that doesn't represent anything real (i.e. is not something we would ever instantiate), but which encapsulates in some or all of its methods the features of its subclasses.

For example the `Integer` and `Float` classes represent real-life numbers that we use in programs. Their features are modeled on the abstract "`Number`" class, yet we can never use or instantiate a `Number` object directly; for example the compiler would not know how many bytes to set-aside to represent a `Number` object. Such things are only defined for the subclasses such as `Integer` and `Float` that can really be instantiated.

To take another example, a program that made use of geometrical entities might define an abstract "`Shape`" class. An object of this type can never be instantiated, only its subclasses such as "`Circle`", "`Rectangle`" etc.

Referring back to the physics example illustrated in Figure 6.2, it might make sense to make "`Particle`" an abstract class. It doesn't make sense to instantiate a generic particle in, say, a detector simulation package since such a program needs to know the specific kind of particle in order to correctly model its behaviour. Part of the declaration of the abstract `Particle` class might look like this:

```
public abstract class Particle {

  public abstract String name();

  // Other methods, abstract or not
}
```

Here are some of the rules for abstract classes:

- Any class with an abstract method is automatically abstract itself and must be declared as such. Note the syntax of the keyword "`abstract`" in both the method and the class declaration above.
- An abstract class cannot be instantiated. Note the error that is generated when you try to compile the line of code "`Number aNumber = new Number();`"
- A subclass of an abstract class can only be instantiated if it overrides each of the abstract methods of the superclass and provides an implementation (i.e. method body) for all of them. Such a class is often called a "concrete subclass" to emphasize the fact that it is not abstract.
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract.
- `static`, `private` and `final` methods cannot be abstract, since these attributes prevent such methods from being overridden by a subclass. Similarly, a `final` class cannot contain any abstract methods.
- A class can be declared abstract even if it does not have any abstract methods. This signifies that the class is somehow incomplete: it cannot be instantiated and serves only as a superclass for subclasses to complete the implementation.

More than one class in an inheritance hierarchy can be abstract. For example, again referring to Figure 6.2, it would make sense for "`Lepton`" to be an abstract class since it represents a family rather than a specific type of particle:

```
// This class extends an abstract class,
// and in addition is itself declared
// abstract:
public abstract class Lepton extends Particle {

  // All leptons must be able to report
  // which generation they belong to:
  public abstract int generation();

}
```

Finally, the concrete subclasses can actually be instantiated and used in a program:

```
public class Electron extends Lepton {

  public int generation() {
    // Electrons belong to the
    // first generation of leptons:
    return 1;
  }

  public String name() {
    return "Electron";
  }

}

// ...
// Somewhere else in the program:
Electron e1 = new Electron();
// ...
```

3.2 Interfaces

What if we want to go one step further and have a class where every method is abstract, and moreover we want this "blue-print" to apply to any class and not just subclasses in a class hierarchy? Such a blue-print is called an *interface*. It defines *a protocol of behaviour that can be implemented by any class anywhere in the class hierarchy*.

An interface definition is not a class definition, but is quite similar in appearance. For example here are the definitions for two interfaces:

```
public interface SimulableParticle {

  // An interface defining the methods
  // that all particles that can be
  // simulated must implement. The
  // method declarations are just
  // like those in a class definition,
  // but with no implementations:

  boolean depositsIonisationInTracker();
  boolean depositsEnergyInCalorimeter();

}

public interface ReconstructableParticle() {

  // An interface defining the methods
  // that all particles that can be
  // reconstructed must implement.

  double energyResolution();
  double angularResolution();

}
```

Here is an explicit list of the rules for interfaces:

- An interface contains no implementation whatsoever. All of the methods are

implicitly abstract (the keyword "`abstract`" does not appear anywhere however).

- All methods of an interface are implicitly public. It is an error to declare methods in an interface definition as `private` or `protected`.
- No data members can be defined in an interface except constants that are declared `static` and `final` (and by definition, public).
- An interface cannot be instantiated, hence it does not define a constructor.
- If a class implements an interface but does not provide an implementation for every interface method, then the class must be declared abstract and hence cannot be instantiated.
- A class can implement any number of interfaces.
- An interface can itself extend (i.e. inherit from, or be a "subclass" of) any number of other interfaces (i.e. the restriction that a class can only extend one other class is relaxed for interfaces).

Common uses of interfaces are:

- Defining families of objects.
- Defining Application Programmer Interfaces (API's) that all users must follow; e.g. if a given functionality can be implemented in a variety of ways then the interface defines the template that all implementers must adhere to. This allows additional implementations to be easily added.
- Defining constants; i.e. an interface with no methods but containing a list of "`static final`" data definitions. Any class that implements the interface then has access to these constants.
- As a safe way of achieving the C++ functionality of multiple-inheritance (see Module 10).

You can immediately see that some of these reasons are similar to those that would motivate the use of an abstract class. It is often the case that either an abstract class or an interface will do the job. Some pros and cons to keep in mind are:

- Any class can implement a certain interface, while only subclasses can extend an abstract class.
- A class can implement any number of interfaces, while a class that extends an abstract class cannot extend any other class.
- Each class that implements an interface must contain method code to satisfy the interface. This can be tedious if there are many implementing classes and the code is very similar. By contrast, an abstract class might contain a partial (non-abstract) implementation that will be available to all subclasses and who need only then implement their smaller "specialised" behaviour.
- If you add an extra method to an interface you have to go and add this to all the implementing classes; by contrast non-abstract methods can safely be added to abstract classes without breaking subclasses.

A common design choice is to use *both*: that is, declare classes that extend an abstract class *and* implement certain relevant interfaces.

Let's return to our particle physics example. Particles which it will be necessary to simulate

should implement the `SimulableParticle` interface. Particles which can be reconstructed using detector data should implement the `ReconstructableParticle` interface. So for example our `Electron` class might now look something like this:

```
// Electron extends Lepton and implements
// both interfaces. Note the use of the
// keyword "implements", which is similar
// to a class becoming a subclass using
// the keyword "extends":
public class Electron extends Lepton
  implements SimulableParticle, ReconstructableParticle {

  public int generation() {
    return 1;
  }

  public String name() {
    return "Electron";
  }

  public boolean depositsIonisationInTracker() {
    return true;
  }

  public boolean depositsEnergyInCalorimeter() {
    return true;
  }

  public double energyResolution() {
    return 0.01;
  }

  public double angularResolution() {
    return 0.1;
  }

}
```

while other particles may implement any number of the defined interfaces:

```
// We don't simulate neutrino interactions
// in our detector, but they are (indirectly)
// reconstructable:

public class Neutrino extends Lepton
  implements ReconstructableParticle {

  // Class implementation ...

}

// We need to simulate neutron interactions,
// but they cannot be individually reconstructed
// based on the detector response:

public class Neutron extends Baryon
  implements SimulableParticle {

  // Class implementation ...

}
```

3.2.1 Marker Interfaces

Empty interfaces are sometimes called *marker interfaces*. For example:

```
public interface Cloneable { }
```

A class can implement this interface merely by naming it in its "`implements`" clause without having to implement any of its methods (because it has none!). The "`instanceof`" operator can then check whether a given object is using this interface — this is often used to provide additional information about an object or flag that something is then possible:

```
SomeObject o; // initialised elsewhere
SomeObject copy;

// If this object extends Cloneable,
// I expect to be able to call its
// clone() method in order to copy
// its internal state into another
// object:
if (o instanceof Cloneable) {
  copy = o.clone()
} else {
  copy = null;
}
```

The contract to provide a "`clone()`" method is not enforced by the interface directly (since it is empty), but in the documentation for the "`Cloneable`" interface.

3.2.2 Polymorphism Using Interfaces

The remarks in the Section on Polymorphism are largely applicable to the case of different classes that implement the same interface. As in the case of inheritance, code can be written in terms of the "interface" and can be substituted by any object that implements that interface. For example:

```
// ...
ArrayList<ReconstructableParticle> reconstructableParticles =
    new ArrayList<ReconstructableParticle>();

Electron e1 = new Electron();
Neutrino n1 = new Neutrino();
reconstructableParticles.add(e1);
reconstructableParticles.add(n1);

Iterator<ReconstructableParticle> it =
    reconstructableParticles.iterator();
while (it.hasNext()) {
  ReconstructableParticle rp = it.next();
  System.out.println("Energy resolution = "+rp.energyResolution());
}
// ...
```

3.3 The Collections Framework Revisited

3.3.1 Interfaces

In Module 5 we used two classes, `ArrayList` and `HashMap`, from the Java collections framework, and said that these were types of *list* and *map* and that there were two other types of collection in the framework, *set* and *queue*. In fact each type of collection corresponds to an *interface* in the collections framework. There is also an even more general `Collection` interface which is extended by the `Set`, `List` and `Queue` interfaces.

The `ArrayList` class implements the `List` interface, which in turn extends the `Collection` interface. Whenever we use a container class we should aim to refer to it using the most general (highest-level) interface possible. If the important thing in a given application is just to have a collection of objects and their order is not significant, we should use the `Collection` interface:

```
Collection<String> languages = new ArrayList<String>();
languages.add("Java");
languages.add("C++");
languages.add("Smalltalk");
boolean rubySupported = languages.contains("Ruby");
```

If the order of elements is significant then we should use the `List` interface:

```
List<String> names = new ArrayList<String>();
names.add("Einstein");
names.add("Fermi");
names.add("Bose");
Collections.sort(names); // Sort alphabeticallly
```

The reason for this is to make our code as general as possible, so that we are not tied to a particular implementation unnecessarily, and can more easily interoperate with code that uses a different container class. A method that takes a `Collection` as an input will work when given a `ArrayList`, `LinkedList`, `HashSet` or any of a number of other types of object. For example the method `Collections.max()` can find the maximum element contained in any of these classes. The method `Collections.sort()` on the other hand sorts the elements of a collection into increasing order, and only makes sense when applied to a collection where the elements are kept in a given order, so it takes a `List` as an argument and can therefore act on an `ArrayList` or `LinkedList` but not on a `HashSet`, which does not implement the `List` interface.

By using the `List` interface we can also more easily adapt our code to use a different implementation if this is needed. We might find that in a particular program we often need to insert new elements in the middle of a list and therefore a `LinkedList` provides better performance.

Similarly we should use the `Map` interface when referring to the `HashMap` objects we created in Module 5. Although `HashMap` is the most efficient implementation for most purposes, there are some cases where a `TreeMap` or `LinkedHashMap` might be better, and by using the `Map` interface wherever possible we can make our code deal with these cases too for little extra effort.

3.3.2 Abstract Classes

The Java collections framework also includes examples of abstract classes, although you are unlikely to need them in this course. For example the `AbstractList` class provides a skeleton implementation of the `List` interface, but a concrete implementation needs to implement the abstract `get` method and will normally also need to override some or all of the other methods of the class. In fact the `Vector` and `ArrayList` classes extend this abstract class.

4. Summary

In this module you have learned the details of some very important object-oriented concepts: *inheritance*, *abstract classes* and *interfaces*. You should think carefully during your program design which of these it makes most sense to employ, although there may well be more than one adequate solution for your particular programming situation.