Data Abstraction

Learning Goals

You must be able to:

- ✓ **describe how data abstraction makes possible the construction of larger software systems**
- ✓ **specify a data abstraction for a Java class, including the use of procedural abstraction**
- ✓ **test a data abstraction**
- ✓ **implement a data abstraction**
- ✓ **recognize and understand the consequences of mutable and immutable data abstractions**
- ✓ **understand how Java interfaces relate to the concept of data abstraction**

Abstraction

Object-oriented software uses data abstraction to model the domain that the software addresses. Abstraction is a common and central concept in computer science. Abstraction enables the commonality between similar but different things to be represented while details that vary between the things are factored out. In this way, we can use a single abstraction to reason about many things (i.e., a one-to-many mapping).

In software development, abstraction helps a developer manage the complexity of building a large software system. As a simple example, consider a developer who wants to determine if a given year is a leap year. If the developer has access to an appropriate existing method, such as the `isALeapYear` method described in the Control and Data Models reading, the developer can simply call the existing method to achieve the desired result, focusing on what needs to be done rather than the details of how the computation is performed.

There are three different kinds of abstraction used in the Java programming language: *procedural* abstraction, *data* abstraction and *iteration* abstraction.[1] In this reading, we will focus on what a data abstraction is and how it is used in Java. As our means of

[1] Other programming languages also use some or all of these kinds of abstraction.

describing data abstraction will rely on the use of procedural abstraction, this reading will cover both of these concepts (despite the readings title!). We will look at iteration abstraction later in the course.

Intermezzo: Models Revisited

Before we dive into the topic of data abstraction, let us take another quick look at the various roles of models in software development.

In the last reading, we considered different kinds of models—for instance, flow charts, call graphs and data definitions—that can be extracted from an existing software system to help a software developer understand how the system works. As Figure 1 shows, these models can be extracted from the source code, by reading it, or from the executing system, by inspecting the system in a debugger.

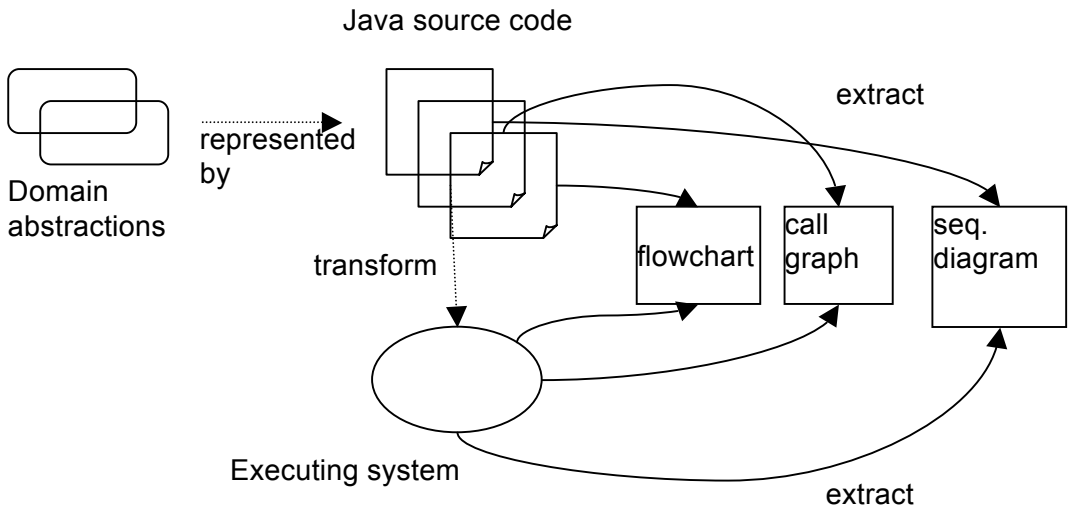

Figure 1 Models everywhere

In this reading, we will be looking at how the source code itself is a model of the domain it addresses. For example, if you are building a software system to help manage the feeding of animals at the Vancouver aquarium, your software will need to have a representation of that domain, such as what animals the aquarium has, how aquariums work and so on. As you read in the Control and Data Models reading, the data associated with a system often changes less often than the computations the software system performs. Structuring the software around the data can thus provide a (relatively) stable base for describing a software system. Object-oriented software systems take this approach using the concept of a class (or object) to represent data abstractions. In Java, it is the `class` construct that provides a means to represent a data abstraction. Figure 1 shows how data from the domain is represented by constructs in the software system's source code.

In reality, the interaction between the models used for understanding the system and the models used to construct the system are not so clearly separated. With some programming systems, it is possible to generate a system or source code from the model. When models are used for generation, the approach is called model-oriented software engineering. A model-oriented approach is of particular interest to manufacturers of products who produce many variations of similar products, such as car manufacturers and cell phone makers, as the manufacturer can tweak the model and regenerate the typically large and complicated software system powering the device. We have not depicted these intricacies in Figure 1 for clarity.

Data Abstraction

Returning to our main topic for this reading, we are considering how source code models a domain. When we are writing object-oriented software, the primary abstraction approach used to support the modeling is data abstraction.

A data abstraction enables the description of a data object[2] in terms of how the object behaves rather than the details of how the data is represented. This abstraction enables other parts of the software to use a data object without knowing all of the details about how the data object is represented. If the data object's representation changes, the software dependent upon the object need not change.

Consider the software system mentioned above to help manage the Vancouver Aquarium. Presumably, such a system would need to include support for determining the feeding schedules of various animals and notifying personnel when a feeding is due. Each kind of animal will have unique needs for feeding: dolphins are fed different food at different times than sloths. However, with respect to feeding, the scheduling software can treat all animals similarly, this software just needs to determine from an animal what it can eat and when it should next be fed. The system should also record for the animal what it was fed as this may effect when the animal is next fed.

The code below shows the beginnings of a data abstraction *specification* for an animal in the feeding scheduling software system.[3] A specification provides a definition for the abstraction of interest. For simplicity, we will specify data abstractions in this course using Java constructs and stylized comments, similar to [1]. In the coming sections, we will refine this specification to be more complete and consider how the specification may be implemented in Java.

[2] We are using the term data object here in terms of an object in the domain rather than a Java object per se.
[3] While we are using Java to specify the data abstraction, this is not a syntactically correct class definition (yet!).

```
// An animal at the aquarium.
class Animal {

   // Construct an animal. We need to say what food the animal
   // eats
   Animal( List<Food> acceptableFoodToEat ){…}

   // What can this animal eat?
   List<Food> getAcceptableFoodToEat() {…}

   // When should the animal eat next?
   Date nextTimeToFeed() {…}

   // Remember what the animal last ate
   void recordLastFeeding( List<FeedingRecord> foodEaten ) {…}
}
```

Specifying Operations in a Data Abstraction

The behaviour of a data abstraction is provided through operations. In Java, we use constructors[4] and methods of a class to define the operations of the data abstraction. In the code shown for the `Animal` class, the specifications for the operations of `Animal` are limited to *method signatures*, which provide only the name of the method, the parameter types and the return type. The information in a method signature is all *syntactic*; we can use it determine if software invoking the method is well-formed but it gives us only informal hints about what the method does. When we are specifying a data abstraction, we need to provide more *semantic* information about the behaviour of the operations; that is, what is the meaning of an operation.

In defining the meaning of an operation, we need to capture three pieces of information:

- any constraints on when the operation may be used,
- which, if any, inputs to the operation are modified by the operation, and
- what the operation does in terms of output values produced and how any modified inputs are changed.

We specify these three pieces of information using informal language in comments associated with each constructor and method. We'll consider each piece of information in turn in the context of operations from the Animal class.

[4] A constructor is a special operation in Java that returns a new object of the Java class. Multiple constructors can be defined for a class; this allows constructors to be defined that take in different parameters and initialize an object of the class differently.

REQUIRES Clause

We describe constraints on when an operation may be used through a REQUIRES clause. This clause specifies for which values of the operation's parameters the operation is defined. This clause also specifies any constraints on the data object on which this operation will be invoked. We use the special word `this` to represent the data object.[5] Occasionally, an operation may even depend upon data that is not specified in the parameters to the operation but may come from access to external resources, such as the file system. In this section, you will see examples of REQUIRES clauses that mention input parameters that are explicit to an operation. We will look at examples that cover the case of constraints on the data object in class. We will not consider clauses involving implicit parameters to the operation (other than the data object) at this time.

If an invoker of the method does not meet the constraints of the REQUIRES clause, the behaviour of the operation is undefined. If the operation is defined for all values of the operation's parameters and for all states of a data object, the REQUIRES clause is not needed.

Let's look at two operations from the `Animal` data abstraction to see how to specify a REQUIRES clause. First, let's consider the `nextTimeToFeed` operation.

```
// Requires: nothing
Date nextTimeToFeed() {
}
```

The `nextTimeToFeed()` operation has no parameters, requires no special state of `this` and thus should work in every situation. We would typically just elide the REQUIRES clause for this specification, but include it here for completeness.

What about the `recordLastFeeding` operation?

```
// Requires: foodEaten must be a non-empty list
void recordLastFeeding( List<FeedingRecord> foodEaten ){
}
```

This operation takes as a parameter a list of `FeedingRecord`s, where a `FeedingRecord` is a data abstraction that records what the animal last ate and at what time. The meaning of a `FeedingRecord` would only be evident by looking up the specification for the data abstraction and is not evident in what has been given in this

[5] `this` is also a keyword in Java used to refer to the data object providing the context for a particular method invocation. We'll see some examples of this used in Java code in the coming weeks.

reading. For this operation to be well-defined, the REQUIRES clause captures that the list provided when the operation is called must contain actual values, it cannot be a list that contains no values.

In some cases, the REQUIRES clause may specify that an operation is well-defined only if some relationship between the values of different parameters holds. Consider this variant specification of a `recordLastFeeding` method:

```
// Requires: feedingTimes.size() = foodEaten.size() and
//    for all i: feedingTimes[i] represents time of foodEaten[i]
void recordLastFeeding( Date[] feedingTimes, Food[] foodEaten ) {
}
```

In this specification of the operation, the parameters are two arrays of values: one array holds the times at which feedings occurred and the other array holds the food eaten during a feeding. The REQUIRES clause specifies that the size of the two arrays must be the same and states that a given feeding is to be recorded at the same index position in each array. In general, we would prefer the first definition of the `recordLastFeeding` operation given above to this one with two arrays, as the first abstracts the details away about how the times and food eaten at a feeding are represented.

Specialized languages exist for specifying REQUIRES and similar clauses. In future computer science classes, you may be introduced to some of these languages. More precision in the specification can enable better checking of errors and can ease communication between the multiple people involved in a software development. For now, we will stick with an informal and natural language for requires clauses. When writing or critiquing a REQUIRES clause, you should strive for your description to be as clear as possible. You may also consider the lack of a REQUIRES clause to mean there are no constraints on the use of an operation.

MODIFIES clause

A MODIFIES clause states which input values are modified by an operation. This clause does not describe how the values are modified; this is part of the role of the effects clause which we will describe next. If a change to an input value persists after the operation executes, we say that the operation has a *side effect*.

An input value may be a parameter to the operation or the data object to which the operation is applied. In the first case, we can explicitly refer to the parameter in the clause. In the second case, we use `this` to represent the data object.

Let's start looking at examples of MODIFIES clauses by considering two operations from our partial `Animal` data abstraction specification: the constructor and the `recordLastFeeding` operation. In neither case does the operation need to modify the input values and the specification should make this clear by not listing the parameters as part of the MODIFIES clause. On the other hand, a constructor always modifies a data object through the sheer act of creating and initializing it. As a matter of convention, because this is always the case, we do not list `this` as being modified by a constructor.

However, the `recordLastFeeding` operation must remember the last time a feeding occurred when the operation is invoked on a specific data object. *(See the Aside: Method Invocations at the end of this reading if you would like more detail about the calling of a method defined on a Java class.)* As a result, the MODIFIES clause for this operation lists `this`.

```
// Requires: foodEaten must be a non-empty list
Animal(List<Food> foodEaten) {…}

// Requires: foodEaten must be a non-empty list
// Modifies: this
void recordLastFeeding( List<FeedingRecord> foodEaten ) {…}
```

The lack of the `foodEaten` parameter in the MODIFIES clause above means that the `recordLastFeeding` operation does not modify its input parameter. This information assures the caller of this operation that the parameter values passed are unchanged after the operation executes. What if instead the `recordLastFeeding` operation was specified as follows:

```
// Requires: foodEaten must a non-empty list
// Modifies: this and foodEaten
void recordLastFeeding( List<FeedingRecord> foodEaten ) {…}
```

If this was the specification then the software which uses this operation cannot assume that `foodEaten` will contain the same values in the list after the operation completes as when the operation was called. We will see in the next section how to specify how `foodEaten` is changed by the operation.

For convenience, we will consider an absent MODIFIES clause as a specification that the operation does not modify any values (except in the case of a constructor where we assume that `this` is modified).

EFFECTS clause

The EFFECTS clause is used to specify the behaviour of the operation given that the REQUIRES clause is met when the operation is called. This clause must specify what outputs are produced by the operation and how any input values modified are altered. Let us pick up with our version of the `recordLastFeeding` operation in which the list of food eaten might be altered by the operation. The EFFECTS clause states that the operation will alter `this` to remember the food eaten and states how the given list of food is altered by the operation. Note that in specifying the effect of the operation on `this`, we only informally state how `this` changes as a more specific description would require a description of how the data abstraction is realized, which would defeat the point of the data abstraction.

```
// Requires: foodEaten must be a non-empty list
// Modifies: this and foodEaten
```

```
// Effects: this remembers foodEaten and foodEaten is empty
void recordLastFeeding( List<FeedingRecord> foodEaten ) {…}
```

The EFFECTS clause above captures that, as the operation processes the input list of food eaten, it empties out the list. For this operation, it is hard to imagine a reason why we would want to alter the list of food eaten that the operation is passed, which is why the version in which the list is not modified is likely preferred.

You might be wondering when it is ever desirable to alter the input values you are passed. Such situations do occur and we may see some later in term.

> *If you want to look into this, consider Java implementations of the quicksort searching approach (search google for "java quicksort").*

The EFFECTS clause is also used to specify the return values provided by an operation, provided that the return type is not void.[6] For instance, the EFFECTS clause for the `nextTimeToFeed` operation specifies that the date returned must be in the future.

```
// Requires: nothing
// Modifies: nothing
// Effects: The date and time returned must be in the future
Date nextTimeToFeed() {…}
```

Specification for the Animal data abstraction

We can now provide a more complete specification for the `Animal` data abstraction for the Aquarium feeding system. As described earlier, you can assume missing REQUIRES and MODIFIES clauses specify that there are no constraints on the running the operations or that no modifications to input values occur.

[6] Returning `void` indicates that no value is returned from the method. `void` is a Java keyword.

```
// An animal at the aquarium.
class Animal {

   // Construct an animal.
   // Requires: acceptableFoodToEat is not an empty list
   // Effects: this updated with acceptableFoodToEat
   Animal( List<Food> acceptableFoodToEat ){…}

   // What can this animal eat?
   // Effects: returns a non-empty list of food
   List<Food> getAcceptableFoodToEat() {…}

   // When should the animal eat next?
   // Effects: returns a date (including time) in the future
   Date nextTimeToFeed() {…}

   // Remember what the animal last ate
   // Requires: foodEaten is not an empty list
   // Modifies: this
   // Effects: this updated with foodEaten
   void recordLastFeeding( List<FeedingRecord> foodEaten ) {…}
}
```

In general, a specification for a data abstraction includes operations that can create instances of the data abstraction, we refer to these operations as *constructors*. There exists one constructor for the `Animal` data abstraction that takes as a parameter a list of `Food`. Other operations can alter an instance of the data abstraction. We refer to these operations as *mutators*. Mutators can be recognized as they have a MODIFIES clause that specifies `this`. The `Animal` data abstraction has one mutator, `recordLastFeeding`. The `getAcceptableFoodToEat` and `nextTimeToFeed` operations are both considered observers as they do not modify the state of the data object.

The operations specified on a data abstraction are uses of *procedural abstraction*, one of the three kinds of abstraction referred to in the introduction of this reading. Procedural abstraction abstracts steps to performing an action. The fact that we can give a name to a sequence of steps (as when we define a method containing a sequence of statements) is procedural abstraction in use.

Data abstraction is the means by which we can extend Java (or other similar programming languages) with a new data type. When we are building a software system that includes the `Animal` class, we can declare variables to be of the type `Animal` and the operations that are available on `Animal` are then defined by the methods defined in the `Animal` class. The ability to add new types to the programming language is powerful. It allows us to write software in terms of types that are relevant to the domain of the problem we are trying to address so that the software can more directly model the domain.

Testing a Data Abstraction

Before using or releasing for use an implementation of a data abstraction, we need to ensure the implementation satisfies the specification. If we use a formal approach to specifying a data abstraction—for instance, the more formal and complete approach presented in [1]—we can use formal reasoning techniques to assess whether an implementation satisfies a specification. As most data abstractions in use in programs are specified informally and incompletely or rely on data types for which formal reasoning is not easily defined (e.g., strings as opposed to integer values), the approach more commonly used for determining if an implementation satisfies a data abstraction specification is testing.

In this description, we assume you have been exposed to the basic ideas of testing in your first computing course, namely specifying expected output for given inputs to a piece of software and then executing the software to see if it produces the expected results with those inputs.

A common starting point for testing an implementation of a data abstraction is to use a black-box testing approach. In this approach, we determine the test cases to use for testing based on the specification of the data abstraction, and not any details about its implementation. One benefit of this approach is that we can use the same test cases to test different implementations of the same data abstraction. Another benefit is that the test cases should exercise the implementation in ways that the abstraction has been specified to support without making any assumptions about what the implementation may allow.

A test case consists of source code to create an instance of the data abstraction and then zero or more invocations of operations defined on that instance. A test case must also choose data values to pass to operations that require input values. In general, we want to keep the test cases as focused on one operation of the data abstraction as possible.

The test cases should also take different paths through the operations defined by the specification. For example, if the REQUIRES clause of one operation on the data abstraction requires the EFFECTS provided by another operation, we would want to have a test case that exercises this dependency. We would want a separate test case for every such path.

Finally, when choosing the data to use as input values, we want to choose both typical input values as well as values at the *boundaries* defined by the input values data type and the REQUIRES clause. For example, if an operation takes an integer parameter that can range between 0 and 10, we would want one test case that uses typical values for this parameter, such as 5. We would also want a test case at each boundary point, such as one test case using a 0 value and another test case using a value of 10.

To ease the writing and running of tests for a data abstraction, a popular tool used in Java programming is JUnit. You will be introduced to JUnit in an upcoming lab. More information is available on the web at junit.org.

Implementing a Data Abstraction

Specifying a data abstraction is not enough to produce running software. We must also provide an implementation for the abstraction that describes how the data abstraction will be represented and how the operations will perform the desired functionality.

To represent a data abstraction in Java, we must use the Java construct of a *field*.[7] A field captures a piece of state for an object. Each field is of a defined type. A field can hold a value that matches the defined type.

When representing a data abstraction, there are often many choices one might make for the field comprising the representation. One choice might result in slower performance for some operations compared to others. The appropriate choice typically depends on how the data abstraction is likely to be used. In some cases, the data abstraction might be implemented in different ways through different Java classes so that the user of the software can choose the appropriate trade-offs. We will see some examples of this later in this reading.

An implementation of the `Animal` data abstraction needs to track what food the animal can eat and when and what the animal has been fed. There are many different ways we can represent this information through fields on the Java `Animal` class.

As one example, we could use two fields. The `acceptableFood` field remembers in a list all types of food that the animal can eat. The definition of this field relies on access to a data abstraction for Food that would be defined through another Java class. The `foodEatenAndWhen` field remembers when and what the animal ate and also relies on another data abstraction, `FeedingRecord`. This example is shown below in Implementation A.

```
// Implementation A
class Animal {
    // Food the animal can eat
   List<Food> acceptableFood;

   // Feeding record for the animal
   List<FeedingRecord> foodEatenAndWhen;
   …
}
```

A different choice would be to split up what the animal ate and when it ate into two separate lists that we would need to keep in synch as shown below in Implementation B.

[7] A field is also known as a *member variable* of a class. In other object-oriented languages, fields may also be known as *attributes* or *instance variables*.

This choice would be odd given that the input parameter to the `recordLastFeeding` already uses a data abstraction of a `FeedingRecord`.

```
//Implementation B
class Animal {
   // Food the animal can eat
   List<Food> acceptableFood
   // Feeding record for the animal
   List<Food> foodEaten:
   List<Date> timeFoodEaten;
}
```

In general, you want to choose the simplest and most expressive means for representing the data. In this case, Implementation A is the better choice.

Java visibility modifiers

A data abstraction is effective for programming when software that uses the abstraction does not depend upon the representation of the data. Many programming languages provide explicit language support to help ensure an abstraction is used appropriately. In Java, this support is provided through access modifiers to class, method and field declarations. An access modifier determines what other software can see and use the Java construct being modified. We will consider just two access modifiers at present: `public` and `private`.

If a Java class, method or field is declared with the `public` modifier, the class, method or field may be used by any other Java class or method. If a Java class, method or field is declared with the `private` modifier, the class, method or field can only be accessed from the class in which it is declared.

In most cases, when you declare a data abstraction in Java using a Java class, you will want that class to use the `public` access modifier so that other software in the system can use the abstraction. (We will see cases later in term where a `private` or other access modifier is more suitable.) Similarly, the methods corresponding to operations on the data abstraction should be declared `public` so that software using the data abstraction can use the abstraction through the operations.

The fields representing the data abstraction should use the `private` access modifier. Any software using the abstraction should not have access to these fields directly, instead the desired behaviour should be achieved by calling operations of the data abstraction.

The Java class definition corresponding to the `Animal` data abstraction is thus more correctly declared as follows (with comments and specifications of operations removed for simplicity of presentation):

```
public class Animal {
```

```
   private List<Food> acceptableFood;
   private List<FeedingRecord> foodEatenAndWhen;

   public Animal( List<Food> acceptableFoodToEat ) {…}

   public List<Food> get AcceptableFoodToEat() {…}

   public Date nextTimeToFeed() {…}
}
```

Software (other than what is defined within the `Animal` class) that attempts to use the instance variables of the Animal class will not compile and cannot execute.

Mutability

A data abstraction is either mutable or immutable. The objects of a mutable data abstraction have values that may change. A mutable data abstraction can be recognized because at least one operation will have a MODIFIES clause specifying that `this` is modified. Objects of an immutable data abstraction have values that do not change once the object is constructed.

The `Animal` data abstraction used as an example in this reading is mutable. Objects of type `Animal` have state that changes as operations are executed.

An example of an immutable data abstraction in Java is the Java `String` class defined in the standard Java libraries. Below is a partial specification of the Java String class:

```
public class String {

   // an empty string is created
   public String( ) …

   // Effects: returns a new string with the specified string
   // appended to the existing string and this is unchanged
   public String concat( String stringToAppend ) …
}
```

Note that this specification states that when the `concat` operation is called on an object of type `String`, a new `String` object is returned. The `String` object on which the `concat` operation was called is unchanged. All operations on the `String` class follow this approach and hence the `String` class is immutable.

There are many advantages to defining data abstractions as immutable. We will see examples of why immutability is a good idea over term. (In actuality, we'll probably see more examples of why mutability makes programming hard rather than how immutability

makes it easy!) From your existing programming experience, you probably know that immutable objects are easier to define, test and use. If you want to dive ahead and learn more about why immutability is desirable, read the appropriate chapter of Effective Java by Joshua Bloch.

Using a Data Abstraction

Conceptually, using a data abstraction for which a specification is available is simple. Find an implementation for the data abstraction, use one of the constructors to create an object for that abstraction and then call (invoke) operations defined for the abstraction to achieve a desired goal. The order in which you invoke operations will be constrained by the specification of the operations. For instance, as we saw when testing a data abstraction, one operation may require the effects of another operation.

In practice, using a data abstraction defined in Java is not simple because most Java classes do not provide even as complete a specification as we have shown for the `Animal` class. Instead, a user of the software must rely on the documentation provided with the abstraction that is typically in the form of Javadoc comments. As it is not always possible to discern how to achieve desired functionality through such comments, developers will often rely upon software that provides an example of use of the abstraction. Developers search through existing code bases and on the web to find suitable examples.

The situation is better when using data abstractions defined in standard Java libraries. The Java Collections Framework (JCF) is a standard library that provides abstractions for collections of objects, such as lists of objects. In addition to being carefully designed, data abstractions in the JCF include documentation that is closer in nature to the specifications we have looked at in this reading.

You have already seen a use of an abstraction from the JCF in the code shown in the readings and in lecture. The declaration:

```
List<Food> acceptableFood;
```

from the `Animal` class uses the `List` data abstraction from JCF. As its name implies, the `List` abstraction provides a list of the kinds of objects specified in the angle brackets. More specifically, a list is an ordered collection of objects. A user of this abstraction can access objects either at a specific position in the list or by searching for whether a particular object exists in the list.

In JCF, the `List` abstraction is declared using the Java `interface` construct rather than the `class` construct. Like a Java class, a Java interface defines a new data type. A Java interface can include declarations of operations, but not definitions of those operations. In other words, a Java interface can define the method signature of the operations, but not how the operation is implemented. A Java interface can also not include the definition of constructors or fields.

As a Java interface defines a new data type, a variable in Java can be declared to be of the new type, just as you can see from the line of code above where the `acceptableFood` variable is declared to be of type `List`.[8]

While it is helpful to declare the variable, we also need an implementation of the `List` abstraction. The JCF provides several different implementations of this abstraction, including `ArrayList` and `LinkedList`. Each of these implementations varies in particular details. `ArrayList` provides fast access to specific locations in a list, but can be slow when adding new elements to the list. `LinkedList` provides fast additions of new elements, but can be slow when accessing an element at a particular location.

When we need to use the List abstraction, we have to pick the implementation with the characteristics we want. We can make this choice for each list object we construct. For instance, we would create an `ArrayList` for `acceptableFood` as follows:

```
List<Food> acceptableFood = new ArrayList<Food>();
```

This code can compile because the `ArrayList` class `implements` the `List` abstraction. Specifically, the definition of the `ArrayList` class includes a header that says (in addition to more):

```
public class ArrayList<T> implements List<T> …
```

The `T` acts similar to a parameter for a method. The `T` says we don't want to specify what kinds of items to put in the list until we declare a variable that will reference a list. When we declare the variable `acceptableFood` above, we specify that the `List` will hold values of type `Food`.

When `implements` is specified as part of a class definition, the Java compiler checks that all of the operations defined in the interface (i.e., in this case `List`) are implemented by the class .The compiler can only check the syntax of these operations, meaning that the method defined in the class has a method signature that matches the method signature of the corresponding operation in the interface. The compiler cannot check that the implementation of the method in the class corresponds to any specification provided of the corresponding operation in the interface. As we described earlier, testing is commonly used to ensure a specification provided by an interface is satisfied by an implementation.

We will be discussing more about JCF abstractions and interfaces and classes as we move through the course. For now, you need to be comfortable with the role of a Java

[8] The actual type of the variable is List<Food>. We'll get to the meaning of the angle brackets soon!

interface in Java source code, how an interface relates to a data abstraction and what it means for a Java class to implement an interface.

Data Abstraction Benefits

Data abstractions provide many benefits when programming:

- Data abstractions can be used to define new data types to extend a programming language.
- Data abstractions enable the expression of the intent of a variable without committing to a particular implementation. The implementation can vary upon the characteristics needed by the program.
- Data abstractions simplify programming because the user of an abstraction need not know all the details of how that abstraction is represented.
- The representation of a data abstraction can change without having to change all code that is dependent upon the abstraction.
- Data abstractions can enable different programming tasks to be undertaken simultaneously. One group can implement code assuming the data abstraction while another group implements the abstraction.

Method Invocations

As we saw in the Control and Data Models reading, classes provide a template for defining objects (or instances of a class). The class construct provides a means for expressing a data abstraction in Java. The operations available on the class can be invoked on objects of the class. That is, methods (as we have seen thus far) cannot be called except within the context of an object. As a result, a method call is of the form:

```
variable.method();
```

where variable is a Java variable that can refer to objects of a specified class and method is an operation defined on the specified class. Constructors are called differently using the `new` keyword. Consider the following code:

```
Animal aDolphin = new Animal( acceptableFood );
aDolphin.recordLastFeeding( foodEatenAtFeeding );
```

The first line declares a variable, `aDolphin`, which can refer to objects of the `Animal` class. The first line goes on (after the = sign) to construct a new `Animal` object by invoking a constructor of the `Animal` class. In this case, the constructor requires a list of the food the animal will eat. We do not show the creation of the list of acceptable food but just pass a variable referring to an object that is a list of food as required by the specification of the constructor operation. The second line calls (invokes) the `recordLastFeeding` operation specified on an `Animal` class in the context of the `aDolphin` object. This operation is invoked with a list of the food eaten at the last feeding; the declaration and initialization of this variable is not shown for simplicity.

References and Further Reading

Material in this reading is based on:

[1] Program Development in Java: Abstraction, Specification and Object-oriented Design by Barbara Liskov with John Guttag. Addison-Wesley, 2001.

Liskov is the 2008 winner of the ACM A.M. Turing Award which is the most prestigious award in computer science. She was awarded this honour for foundational innovations in programming language design.

This book provides a more comprehensive description of abstraction mechanisms and the kinds of abstractions if you are interested in gaining a deeper understanding of this material.

For further reading on (im)mutability, see:

- Effective Java by Joshua Bloch.

The following sections of the Java tutorial describe some of the syntax that you've seen in this reading. Read through these parts of the tutorial and then read through this document again.

- Arrays – read through the opening section only
  http://download.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html