Type Hierarchy, Polymorphism and Dispatching

Learning Goals

You must be able to:

- ✓ **extract a type hierarchy from Java source code**
- ✓ **trace how Java source code involving polymorphism, inheritance and dynamic dispatching will execute**
- ✓ **determine if one type is substitutable for another type**
- ✓ **refactor existing code to take advantage of type hierarchies**
- ✓ **write Java code to add a new type into a type hierarchy**

Type Hierarchies

What is a Type?

Many object-oriented programming languages use type hierarchies and polymorphism to simplify the expression of software. Software programs compute and manipulate data. Each piece of data a software program creates or uses has a type that defines the operations that can be performed on that piece of data. Some programming languages infer the type of a piece of data. For instance, in Scheme, you could write a function to calculate the cost of a bus trip if you need to buy `n` tickets as:

```
(define (cost-of-bus-trip n)
   (* 2.50 n))
```

This function attempts to treat the value passed as a parameter (i.e. `n`) as a number because it is used in an operation (`*`) that operates on numbers. If a string is passed, as in `(cost-of-bus-trip "")`, the interpreter will complain that the expected type of `n` is a number.

In other programming languages, such as Java, each variable used in a program must have a type declared by the programmer. In Java, expressions are also typed.[1] The process of type checking ensures that the code written by a programmer meets constraints about how types can be used together. For instance, type checking ensures that a method which expects a `String` parameter is only called with variables or expressions of type `String`. Because Java is a *statically-typed* programming language, this type checking is performed by the compiler (essentially as the program is being written), rather than when the program is executed.

Supertypes and Subtypes

As you saw in the Data Abstraction reading, Java enables a programmer to extend the language through the definition of new data abstractions using the Java `interface` and Java `class` constructs. In that reading, you saw how it was possible to relate newly created types in a hierarchy. Specifically, the reading showed how a specific implementation of a data abstraction provided by a Java `class` can be related to a specification of the data abstraction provided by a Java `interface`. Here is a brief recap of the example shown in the Data Abstraction reading:[2]

```
interface List<E> { … }

class ArrayList<E> implements List<E> {…}
```

We refer to the type introduced by the `List` interface (i.e., the `List` type) as the *supertype* of the type introduced by the `ArrayList` class (i.e., the `ArrayList` type). Sometimes it is helpful to draw a small picture of the type hierarchy of a part of a program. By convention, a supertype is usually drawn above a subtype with an arrow from the subtype to the supertype (Figure 1).

[1] Actually, expressions that result in variables or values are typed. Some expressions are void. We will fuzz this difference somewhat in this reading. Full details are more appropriately covered in a programming languages course.

[2] Liberties have been taken leaving out details of the actual headers of these interfaces and classes.

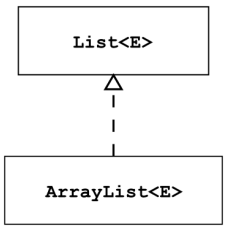

Figure 1: type hierarchy

In the example above, the subtype provides an implementation of the supertype. Subtypes can also extend the behaviour of a supertype. Let's switch back to the `Animal` example also introduced in the Data Abstraction reading. You may recall that we were defining an `Animal` abstraction as part of creating a software system to automate the feeding schedule at the Vancouver aquarium. While we might be able to represent most characteristics of when and what animals are fed in terms of the `Animal` abstraction, there may be some characteristics that depend on the kind of animals. For example, marine mammals may have preferences about being fed on the surface versus under the water whereas insects may have preferences about being fed in the light versus the dark. Figure 3 depicts a type hierarchy that would support capturing the differences between the kinds of animals at the aquarium.

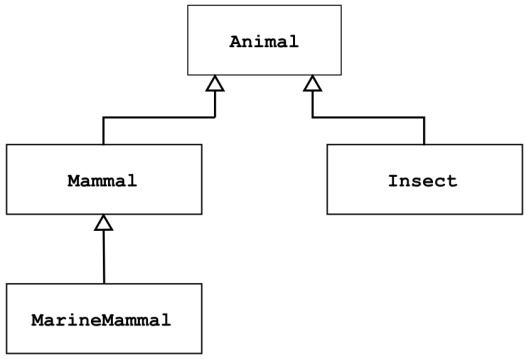

Figure 2: a deeper type hierarchy

Remember that we use types to specify how a variable can be used. In Java, as soon as we declare a variable to have a certain type, we are limiting how that variable can be used to the constructors and methods declared on that type. If a programmer declares a variable of type Animal,

```
Animal anAnimal;
```

then only constructors and methods defined by the `Animal` type apply to `anAnimal`. However, in the Data Abstraction reading, we have already seen that a value corresponding

to a subtype of a type can be assigned to a variable declared to be of the type. For instance, a programmer can write:

```
Animal anAnimal = new Mammal();
```

given the type hierarchy specified in Figure 3. In effect, what this code is stating is that the programmer wants to *substitute* an object of the subtype for an object of the supertype. For such code to work correctly, there must be constraints on how the `Mammal` subtype can extend the `Animal` supertype. These constraints are more involved than simply limiting a subtype to add new operations because of another property of object-oriented software, namely inheritance. Let's look first at what inheritance is before we come back to the topic of *type substitution* at the end of this reading.

Inheritance

Two principles we would like to adhere to when programming are that:

1. We would like the code we write to "look like" the problem domain.
2. We would like to minimize duplication in the code we write.

The data abstraction support provided in an object-oriented programming language helps us to (mostly) achieve the first principle. The principle of inheritance supported in an object-oriented language helps us (mostly) achieve the second principle.

Inheritance is a mechanism provided in object-oriented programming languages that enables a programmer to define a new class in terms of an existing class. Using inheritance, the new class, known as the subclass, inherits all the members, including fields and methods, from the existing class, known as the superclass. For example, in our feeding system example, the code we want to write to represent a mammal might share similarities with the code we write to represent an animal. We can make these similarities explicit by defining the `Mammal` class as a subclass of the `Animal` class. The Java `extends` keyword makes this relationship explicit.

Consider the following partial definition of `Animal`.

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

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

We can declare `Mammal` as a subclass of `Animal` as follows using the Java `extends` keyword.

```
class Mammal extends Animal {
…
}
```

Any method defined on the `Mammal` class has access to the `acceptableFood` field (as long as it is not declared `private` which it is not in this case). Any software using a `Mammal` object can call the `getAcceptableFoodToEat` method without `Mammal` having to provide any declaration or definition of the method. Consider the following code:

```
Mammal aMammal = new Mammal();
List<Food> foodOfMammal = aMammal.getAcceptableFoodToEat();
```

After the first line of code executes, the `aMammal` variable refers to an object that has been created of type `Mammal`. We use the term *apparent type* to refer to the declared type of a variable. In this case, the apparent type of the `aMammal` variable is `Mammal`. We use the term *actual type* to refer to the type of the object assigned to a variable. In this case, the actual type of the object assigned to the `aMammal` variable is also `Mammal`. When the second line of code executes, Java (the virtual machine) uses the actual type of the `aMammal` variable, which is `Mammal`, as a starting point to find a definition for the `getAcceptableFoodToEat` method. As no such definition is found, Java then looks for a definition in the superclass, in this case `Animal`, where a definition is found and the code is executed.

Consider if we had instead written the code as:

```
Animal aMammal = new Mammal();
List<Food> foodOfMammal = aMammal.getAcceptableFoodToEat();
```

In the end, the same code would execute, but the apparent type of the `aMammal` variable is `Animal` instead of `Mammal`.

In Java, a subclass can inherit from only one superclass. (Some languages, such as C++, allow a class to have multiple superclasses.) Every hierarchy of classes you create, though, has a special system class at the very top called `Object`. The `Object` class provides several useful methods to each class, such as `equals` which supports the comparison of two objects. See the Java API documentation for the `Object` class (search Google for "Java API Object") for more details about what the `Object` class provides.[3]

[3] We will see later that there are cases in which you must be careful about the inherited `equals` method from `Object`.

Multiple Interfaces Allowed

Although a Java class can inherit from only one superclass, a Java class can implement many different interfaces. For each interface the Java class implements, the class must provide an implementation for each of the operations provided on that interface. The support for multiple interfaces to be defined for a class allows different parts of a Java software system to use an object from different perspectives. As a high-level example, one interface defined in the standard Java libraries is the `Serializable` interface.[4] You will see examples of the use of this interface in lecture. The purpose of this interface is to allow the saving of an object to disk so that a program can remember data and restart from the data stored on the disk. For instance, if we have an iTunes-like program in Java, we would want to potentially remember the objects representing the music tracks. We could serialize these objects to disk each time the user quit the program and re-read them when the program re-started. We might have a class `MP3Track` declared as follows:

```
class MP3Track implements Serializable, MusicTrack …
```

This class implements a `MusicTrack` interface that declares some general behaviour for a music track. It also implements `Serializable`. The part of the program that plays music will refer to an object of type `MP3Track` through a `MusicTrack` or `MP3Track` variable. The part of the program that saves music tracks might refer to the same object through a variable of type `Serializable`. By typing variables appropriately in our program, we can ensure parts of the software system do not rely on functionality they do not need.

Polymorphism and Dispatching

Generically, the term *polymorphism* means having multiple forms. We have seen how in Java we can declare a data abstraction using the Java `interface` construct and then provide multiple implementations by declaring multiple Java `classes` that implement that interface (i.e., the `List` interface with `ArrayList` and `LinkedList` implementations). This example shows polymorphism in use. An operation defined on the `List` interface is given multiple forms, one in the `ArrayList` implementation and one in the `LinkedList` implementation.

[4] `Serializable` is actually a somewhat odd interface in Java in that it is a Marker interface. It does not require a class to implement any methods (although a class can override special methods for serialization). See the web for many more details!

The mechanism that ensures the desired implementation is called is *dispatching*. Java supports single dispatching, that is it chooses which form to use of an operation based on the actual type of the object on which the operation is being called. Let's make that more meaningful through an example.

If we have an operation `size` declared on `List` for which `ArrayList` and `LinkedList` provide different implementations and we have the following code:

```
List<Animal> aList = new ArrayList<Animal>();
aList.size();
```

Then, as we have seen before in this reading, we use the actual type of `aList` to find the appropriate implementation, which in this case is the definition on `ArrayList`. We refer to this as single dispatch because the selection of which method definition to use is based on the type of the `aList` object. In some languages, such as CommonLisp, the choice of which method to execute is based on the types of the parameters.

Inheritance and dispatching interact in interesting ways to help us write code according to the principles we set out. In effect, inheritance lets us reuse code from a superclass in a subclass. Instead of having to re-type the method definition and all of the method's code in the subclass, we can simply identify that we want to reuse the code from the superclass. This lack of duplication means that if there is an error in the inherited method, fixing it once in the superclass will fix the problem in all of its subclasses.

Sometimes, we want to reuse the implementation from the superclass and add to it. Other times, we want to provide a different implementation than the one inherited. Java allows us to *override* a method definition to make these cases possible. To override a method inherited from a superclass, we re-declare and re-define the method in the subclass with exactly the same signature[5] as it appears in the superclass. Let's revisit our aquarium feeding system. Imagine that the dolphins in the aquarium have all been tagged with a device that emits a sound when it is time for their next feeding; the playing of this sound causes them to come to the appropriate part of their tank for feeding. These devices are being tested out on the dolphins and are not available for any other animal. As a result, we want to augment what happens when we compute the `nextFeedingTime` for dolphins compared to what happens for all other animals. We can achieve the desired effect by overriding the `nextTimeToFeed` method defined on class `Animal`. Here are the partial class definitions.

```
class Animal {
```

[5] By signature, we mean the name of the method, the types of the parameters to the method and the return type of the method.

```
    // When should the animal eat next?
    Date nextTimeToFeed() {
    // Look up the time the animal was last fed, add three hours
    // and return the value
    }
    …
}

class Dolphin extends Animal {

    @Override
    Date nextTimeToFeed() {
          Date timeToFeed = super.nextTimeToFeed();
          // Update device attached to dolphin
    }

}
```

If we now have the following code:

```
Dolphin aDolphin = new Dolphin();
…
Date feedingTime = aDolphin.nextTimeToFeed();
```

When `aDolphin.nextTimeToFeed` executes, Java determines the actual type of `aDolphin` is a `Dolphin` and looks in that class first for a definition for `nextTimeToFeed`. It finds such a definition and starts executing the code. The first line of code contains `super.nextTimeToFeed()`. The `super` keyword in Java is used to refer to the definition of a method in the superclass of the current class. The Java virtual machine then looks for a definition in the superclass of `Dolphin`, which is `Animal`, finds a definition there and executes it. Execution then returns to the next line of the definition of `nextTimeToFeed` in `Dolphin`, which causes the device attached to the `Dolphin` to be updated.

You may have noted the `@Override` annotation provided right before the definition for `nextTimeToFeed` in `Dolphin`. An annotation provides a means of telling the compiler additional information that might be checked. In this case, the `@Override` says to the compiler to check that a method with the same signature exists in a superclass. If not, the compiler will produce a warning. Other annotations exist and a user can define their own kinds of annotations. Annotations provide an additional mechanism for a programmer to express their intent while writing code. Annotations do not change the semantics or execution of the code.

It is not necessary to refer to a superclass' definition of a method in an overriding definition in a subclass. It is likely that different kinds of animals have completely different feeding schedules and it is more likely that the `Dolphin` class would provide its own implementation for `nextTimeToFeed`. To provide such a definition, we do not include a call to `super.nextTimeToFeed()` in the definition of `nextTimeToFeed()` in the `Dolphin` class. For example:

```
class Dolphin extends Animal {

    @Override
    Date nextTimeToFeed() {
        // Return a value two hours from the last feeding time
    }

}
```

The definition of this method could use fields storing relevant data from the superclass. If you refer back to the DataAbstraction notes, the `Animal` class had a `foodEatenAndWhen` field that might be used by `Dolphin` to compute when the next feeding time should be.

A superclass of one class may be a subclass of another and so on. In other words, we can have a linear chain of inheritance. When a subclass references a super object (i.e., calls a method using `super.<methodname>`), Java will look up the inheritance chain until it finds the first suitable definition to use.

Overloading

When we write code, we choose class, method, field and variable names that express the abstraction we are trying to encode. Sometimes, we may have an abstraction to capture, such as a procedural abstraction to print a representation of an object to a screen or file, where we want to use one name to express the abstraction, but we need different implementations. Imagine we want to be able to print information about an object of the `Dolphin` class either to the screen or to a file. The most natural way to do this is to add a `print` method to the `Dolphin` class. However, if we are printing to a screen, we need to provide the method a `Screen` object to print to and if we are printing to a file, we need to provide the method a `File` object to print to. One option might be to define the `print` method to accept both a `Screen` and a `File` object and then determine somehow which one is to be used. This is not a good solution because the code used to print to each device might be very different and we do not need or want to mix it together. Another way in which polymorphism is used is to allow different methods to be defined on a single class with the same name as long as the parameters passed to the method have different types. Which method is called when `print` is invoked on a `Dolphin` object will depend on the types of the parameters passed as part of the method call.

```
class Dolphin extends Animal {

    …
    void print(Screen aScreen) {…}

    void print(File aFile) {…}
}
```

Type Substitution

The ability to define and use a type hierarchy when programming is powerful as it helps achieve abstraction in the code we write. Consider the aquarium feeding system. We can express much of the system in terms of the `Animal` abstraction, such as when to next feed an animal, recording when and what the animal last ate and so on, instead of having to write all of those operations for dolphins and for sloths and for seals and for sea lions and so on. If we add a new animal to the aquarium, we would like to incorporate it into the feeding system by simply adding a new subtype for the animal without having to change the base system that calls all of the appropriate operations in the context of the `Animal` supertype.

To make this more concrete, we would like code in our system that looks something like this:

```
Set<Animal> allAnimals;


// Create an object to represent each animal in the aquarium by
// creating instances of the subtypes of Animal, such as three
// Dolphin objects to represent the three dolphins, six Beluga
// objects, and so on and add each object to the Set allAnimals.
// For example:
Dolphin aDolphin = new Dolphin();
allAnimals.add(aDolphin);



// Determine the next feeding times for each animal
for (Animal anAnimal: allAnimals) {
   Date next = anAnimal.nextTimeToFeed();
   // Schedule that feeding
}
```

Note how the `for` loop is going across each animal in the set of `allAnimals` and asking each animal when the next time is that it should be fed. This `for` loop should not have to change if we introduce a zebra into the aquarium, assuming that `Zebra` is a subtype of `Animal`.

The code above will only work as the programmer expects if an object of a subtype can be substituted for the supertype. To be substitutable, a subtype must support the same operations as the supertype. The Java compiler checks that this fact is true at the (syntactic) method signature level; that is, the subtype has a definition for each operation that matches the definition of the operation in the supertype. What the compiler cannot check is that the behaviour (semantics) of a subtype operation respects the constraints specified for the behaviour of the supertype operation. A careful programmer will do this check by comparing the specifications of the supertype and subtype operations. The following sections entitled "Weakening the pre-condition" and "Strengthening the post-condition" provide details on how this is done.

Weakening the pre-condition

To perform this check, we first need to compare the situations when the supertype method applies with the situations when the subtype method applies. This information is provided by the REQUIRES clause for each method definition. Another way of referring to a REQUIRES clause is as a pre-condition. For a subtype to be substitutable for a supertype, the pre-condition of a method defined in the subtype must be the same as or weaker than the pre-condition specified for the method of the same signature in the supertype. A weaker pre-condition means that the subtype method requires less from its caller than the supertype method.

Consider this partial specification for `Animal` from the Data Abstraction reading.

```
class Animal {

   // pre-condition (Requires): foodEaten is not an empty list
   void recordLastFeeding(List<FeedingRecord> foodEaten) {…}
}
```

If a subtype `Dolphin` also defines this method, the pre-condition for the method must be the same or weaker. For instance, it can accept an empty list.

```
class Dolphin extends Animal {

   // pre-condition: none (i.e., foodEaten may be empty)
   void recordLastFeeding(List<FeedingRecord> foodEaten) {…}
}
```

This is a weakening because if the pre-condition on the `recordLastFeeding` method of the supertype (`Animal` in this case) holds, the pre-condition on the corresponding method in the subtype (`Dolphin` in this case) must also hold.

If we reason about the code in terms of the supertype method, we will ensure the stronger condition that the list is not empty and if the subtype method is called, we are assured it can handle that condition (and more).

Strengthening the post-condition

The second comparison we need to do is on the EFFECTS of the methods. The EFFECTS are sometimes referred to as the post-condition of the method. For a subtype to be substitutable for a supertype, the post-condition of a method defined in the subtype must be the same or stronger than the post-condition specified for the method of the same signature in the supertype. A stronger post-condition on the subtype's method means that if the post-condition on the subtype method holds then the post-condition on the overridden supertype method must also hold, assuming that the pre-condition on the supertype has been met.

Consider this partial specification for `Animal`.

```
class Animal {

   // post-condition (Effects): returns a date (including time)
   // in the future
   Date nextTimeToFeed() {}
}
```

If `Dolphin` defines this method, it must keep the same post-condition or strengthen it. For instance, the `Dolphin` might return dates only between an hour from now and 24 hours from now.

```
class Dolphin extends Animal {

   // post-condition: returns a date an hour from now to 24 hours
   // from now
   Date nextTimeToFeed() {}
}
```

Note that if the post-condition on the subtype's `nextTimeToFeed` method is true then the post-condition on the corresponding overridden method in the supertype method must also be true. All of the dates returned by `nextTimeToFeed` defined in `Dolphin` are within the range specified by `Animal`, but include less of the range. Any code that uses `nextTimeToFeed` that a programmer has reasoned about in terms of an `Animal` will still work correctly if a `Dolphin`'s definition of `nextTimeToFeed` is used.

There are additional properties that can be reasoned about to ensure a subtype is substitutable for a supertype. We will not cover them in this course but you can check out the references at the end of this section if you are interested.

Intermezzo: Some Other Java Stuff

There are a few points about Java you have seen in readings and in class but have not been mentioned explicitly. This and other readings will not try to cover all of them. Please use the web (i.e., the Java Language Tutorials) to search out answers to parts you are unsure about or ask the usual suspects (instructors, TAs, on Piazza, etc.). Here are a few words about a couple of important items.

The Java void type

This reading has discussed how Java is statically typed and variables and expressions have types associated with them. Sometimes, we need to be able to say that an expression has no type. In particular, a method that does not return a value does not need to have a type. The way to indicate the lack of a type is by using the Java `void` type.

The Java abstract class construct

A Java `interface` introduces a new type by declaring methods (operations) supported by the type. A Java `class` introduces a new type by both declaring methods (operations) supported by the type and providing definitions for the methods (i.e., code that describes how the method should behave when executed). A Java abstract class introduces a new type by both declaring methods (operations) supported by the type and providing definitions for some (but not all) of the methods. An abstract class can have a constructor but it is not possible to construct an object of an abstract class since it does not have definitions for all methods. An abstract class is defined using:

```
public abstract class Mammal extends Animal {…}
```

Null values

Variables hold values. A variable may be declared without providing a value. For example,

```
int i;
```

declares the variable `i` but `i` does not yet have a value and is undefined. If the next line of code states

```
i = 1;
```

then `i` has the value of `1`. A variable that references an object may also be undefined as in

```
Animal anAnimal;
```

When we create an object using `new` and assign it to `anAnimal` then `anAnimal` has a value which is a reference to the newly created object. Sometimes, it is useful to indicate explicitly that the variable `anAnimal` is not referencing any object. `null` is used to represent that the variable is not referencing anything as in:

```
Animal anAnimal = null;
```

Simplifications and Omissions

To focus this reading on the main points, we have left out ***many*** details about how these concepts play out in subtle ways in Java. For example, we have not explicitly discussed the ability to define *static methods* as part of a Java class. These methods apply in the context of a class and not an object. These methods can perform such functionality as tracking data across all objects of a class. A subclass can *hide* an implementation of a static class method as opposed to overriding the method. It is hard to learn all of these variations as you are learning a language. As such examples come up in the course of CPSC 210, I will point them out. As you feel more comfortable with the language and

want to learn more, you should read the Java Tutorials or Java language specification in more depth.

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 the potential pitfalls of inheritance to watch out for, check out:

- Effective Java by Joshua Bloch.

The following section of the Java Tutorial provides an overview of the syntax that comes into play when defining interfaces, classes and sub-classes. Read through the tutorial and then read through this document again:

http://download.oracle.com/javase/tutorial/java/landl/index.html