COMP 401

Prasun Dewan[1]

IS-A and Multiple Inheritance

It is difficult to talk about inheritance without using the term IS-A. This term is, however, more general than inheritance and also related to interfaces. Here will define IS-A, use it to unite both interfaces and inheritance, and then see that the seemingly straightforward notion of assigning an expression to a variable cannot be properly explained without understanding IS-A in depth.

IS-A Relationships

An inheritance relationship between a subtype and a supertype is a special case of the more general IS-A relationship among entities. Intuitively, we might say:

AStringDatabase IS-A AStringHistory

In ordinary language, this assertion seems right - we say some entity e1 IS-A e2 if e1 has all the properties of e2. Thus, a primate is a mammal since it has all the properties of a mammal. In the context of an object-oriented programming language, the entities are object types (classes and interfaces) and their instances (objects), and the properties we use to determine IS-A relationships among them are their public members (methods and variables).

We can now formally define the IS-A relationship among Java object types and their instances. Given two object types, T1 and T2, and arbitrary instances t1 and t2 of these types, respectively:

T1 IS-A T2

if all public members of T2 are also public members of 1.

From this definition, we can derive, that:

T2 extends T1 => T2 IS-A T1

because every instance of T2 has not only the members declared in its type T2, but also all members declared in the super type of T2, T1. The reverse is not true:

T2 extends T1 => T1 IS-A T2

since T2 can define additional public variables and methods that instances of T1 do not have.

[1] © Copyright Prasun Dewan, 2015.

Inheritance is only one example of an IS-A relationship. The implements relationship between a class and an interface is another example:

> T^2 implements T^1 => T^2 IS-A T^1

because every instance of T^2 has all the members defined in T^1 (plus, optionally, some more since a class is free to define public members not declared in its interface).

> *T^1 **IS-A** T^2 if T^2 can be substituted by T^1, that is, whenever a member of T^2 is expected, a member of T^1 can be used.*

Thus, some IS-A relationships based on our previous examples are:

> StringDatabase IS-A StringHistory
> AStringDatabase IS-A AStringHistory
> AStringHistory IS-A StringHistory
> AStringDatabase IS-A StringDatabse

The IS-A rule is transitive:

> T^3 IS-A T^2 IS-A T^1 => T^3 IS-A T^1

This follows from transitivity of the inheritance relationship. Thus:

> AStringDatabase IS-A StringHistory

By our definition, it is also reflexive:

> T^1 IS-A T^1

Thus:

> AStringHistory IS-A AStringHistory

which should not be surprising!

The IS-A relationship in ordinary life is interesting because it serves as a taxonomy, allowing us to understand the relationship between different kinds of entities. In Computer Science, it allows us to understand the relationships between our designs (interfaces) and implementations (classes). A more practical application is that it gives the basis for type-checking rules in object-oriented languages.

Review of Flexible Typing in Object Assignment

If we use only primitive types, the rules for assigning an expression of type T^2 to a variable of type T^1 are simple: T^1 should be the same as T^2. With object types, we saw more flexibility. Recall the following method we implemented for printing instances of BMISpreadsheet:

```
public void print (BMISpreadsheet aBMISpreadsheet) {
  System.out.println ("Height:" + aBMISpreadsheet.getHeight());
  System.out.println ("Weight:" + aBMISpreadsheet.getWeight());
  System.out.println("BMI:" + aBMISpreadsheet.getBMI());
}
```

We saw that this method is polymorphic –instances of multiple classes can be assigned to its formal parameter:

```
print (new AnotherBMISpreadsheet());
print (new ABMISpreadsheet());
```

Thus, the rule for assigning an object expression of type T^2 to a variable of type T^1 no longer requires that T^1 be the same as T^2. The type-rules are more flexible, thereby making polymorphism possible.

The IS-A relationships determines the exact rules used for assignment.

Type Rules in Assignment

Consider the following declarations:

```
StringHistory stringHistory = new AStringDatabase();
StringDatabase stringDatabase = new AStringHistory();
```

In the first case, we are trying to assign an instance of `AStringDatabase` to a variable expecting `StringHistory`. Since:

```
AStringDatabase IS-A StringDatabase IS-A StringHistory
```

the assignment is legal. On the other hand, in the second case, we are trying to assign an instance of `AStringHistory` to a variable expecting `StringDatabase`. Since `AStringHistory` is not, directly or indirectly, `StringDatabase`, the second assignment is illegal.

To understand what may go wrong in the second assignment, consider the following operation invocation:

```
stringDatabase.clear();
```

Because the type of `stringDatabase` is `StringDatabase`, this invocation will be considered legal at compile time. However, if `stringDatabase` is actually assigned an instance of `AStringHistory`, the instance will not have the `clear` member, and we will get a runtime error.

To understand why the first assignment is safe, consider an operation invocation on `stringHistory`:

```
stringHistory.size();
```

If `stringHistory` has been actually assigned an instance of `AStringDatabase`, the instance is guaranteed to have all the publically accessible members of an instance of `StringHistory`, since indirectly `AStringDatabase` IS-A `StringHistory`.

Given the first assignment:

```
StringHistory stringHistory = new AStringDatabase();
```

should the following be legal?

```
stringHistory.clear();
```

Since `stringHistory` has been assigned an instance of `AStringDatabase`, the instance will have the member, `clear`. However, the compiler will complain. This is because, at compile time, we do not know the exact value of a variable, and have to take the conservative approach of assuming it has only those members that are indicated by its type (and no other member). However, if, at runtime, we are sure about the actual type of the object, we can use a cast, as shown below:

```
((StringDatabase) stringHistory).clear()
```

The cast assures the compiler that the type of the object stored in `stringHistory` is actually `StringDatabase`. Unlike other languages such as C that allow casting, Java keeps the type of a variable at runtime, and will give an error if the actual type, T^2, does not match the type, T^1, given in the cast, that is T^2 is not a T^1. Thus, if we executed:

```
stringHistory = new AStringHistory();
((StringDatabase) stringHistory).clear()
```

we would get a `ClassCastException`.

We can now state the general type rules used by the compiler for object assignment. Assume we assign to some variable v of type T^1 an expression e of type T^2. T^1 and T^2 may be interfaces or classes, or primitive types. The assignment is legal if T^2 IS-A T1.`

Typing an expression is ambiguous if a cast is used:

```
(StringDatabase) stringHistory
```

A cast creates two types for the expression being cast: a static type and a dynamic type. The *static type* is the type used for cast. Thus, in the above example, `StringDatabase` is the static type of the expression. The *dynamic type* is the actual type of the expression being cast, which is determined at runtime. Thus, in the above example, it is determined by the object that has been assigned to `stringHistory`. The type checking rules above use the static type at compile time. A separate type-checking phase occurs at runtime, which uses the actual type, T^2 of the cast expression to ensure it is compatible with static type, T^1, used for casting, that is, T^2 IS-A T^1, as discussed above.

instanceof and IS-A

The instanceof operation can be used to verify the iS-A rules. In general, given an instance I, whose class is C, and a type T

I **instanceof** T

If the CI IS-A T. Thus:

new AStringDatabase() Iistanceof StringHistory → **true**
new AStringHistory() instanceof StringDatabase → **false**

Real-World Analogy

To understand these type rules intuitively, let us consider again the real world. The following is legal:

```
ARegularModel myCar = new ADeluxeModel();
myCar.accelerate();
```

If our rental or buying plan assumes a regular model, and we are upgraded to deluxe model, that is safe, because all operations on a regular model such as accelerate are also applicable on a deluxe model. However, the following is not legal:

```
ADeluxeModel myCar = new ARegularModel();
myCar.setCruiseControl();
```

If our plan assumes a deluxe model, and we are downgraded to a regular model, we will be unhappy, and potentially unsafe, because some operations such as `setCruiseControl` are applicable only to deluxe models. However, the following is safe:

```
ARegularModel myCar = new ADeluxeModel();
ADeluxeModel hisCar = (ADeluxeModel) myCar;
```

In other words, we should be able to perform operations of the upgrade that could not be performed on the car we reserved, as long it can be assured that we did indeed get an upgrade, that is, the cast is successful.

Thus, compile-time type checking is equivalent to checking that our plan for driving the car is consistent with the car we have reserved, while runtime checking is equivalent to checking that it is consistent with the actual car we obtained. It is important to note that both checks occur before we use the car in an inappropriate way, for instance, before we actually try to set the cruise control.

To further re-enforce the relationship with the real-world, consider an example from the natural world. If we want a pet, a cat or a dog will do, since cats are dogs are animals. However, if we need a dog, a cat will not do, since dogs are not cats.

Inheritance and Polymorphism

A consequence of our type rules is that the method we defined for printing `StringHistory`:

```
static void print(StringHistory strings) {
      System.out.println("******************");
      int elementNum = 0;
      while (elementNum < strings.size()) {
            System.out.println(strings.elementAt(elementNum));
            elementNum++;
      }
      System.out.println("******************");
}
```

will also work for printing instances of `StringDatabase`. That is, we can safely invoke:

```
print(stringDatabase);
```

This is because the following assignment is made during parameter passing:

```
StringHistory strings = stringDatabase;
```

which is allowed by the assignment rules. The only member of the argument accessed by `print` is `elementAt`, which is also a member of `stringDatabase`.

As we saw above, creating IS-A relationships via the implements relationships allowed us to write polymporhic methods. Here we see that creating IS-A relationships through inheritance also supports such methods. The type checking rules described above have been designed to support both interface-based and inheritance-based polymorphism.

Heterogeneous Arrays

To further understand the assignment rules, consider the following initializing declaration of arrays:"

Object[] objects = { "Joe Doe", **new** AStringDatabase(), **new** AStringHistory()};
String[] strings = {"Joe Doe", **new** Object()};
In both cases, we are assigning to elements of the array instances of different classes, some of which are not the same as the declared class of the element type, Object and String, respectively. Should these heterogeneous arrays be allowed?

The first array declaration says that each element of the array is an instance of Object. As Object is the superclass of all classes, all objects are its instances. Thus, the first initialization is indeed allowed.

The second declaration says that each element of the array element is an instance of String. The second element is an instance of Object, which is not a String. Thus, the second initialization is not allowed.

Thus, the IS-A relationship allows not only polymorphic methods but also arrays with heterogeneous elements, which in turn, can be processed by polymorphic methods.

Mixing the class Object and Interfaces

Consider the following statements:

```
StringHistory stringHistory = new AStringHistory();
Object object = stringHistory;
```

Should the second statement be allowed? Intuitively, the answer is yes, as every value assigned to a variable is an Object. Yet, interestingly, our type and IS-A rules, given so far, do not allow this statement. Our assignment rules say that this statement is legal as long as StringHistory, the type of the RHS (right hand side) IS-A type of the LHS (left hand side) of the assignment. Our IS-A rules in, turn, say that StringHistory IS-A Object as long as StringHistory (directly or indirectly) extends the class Object. But an interface cannot (directly or indirectly) extend a class!

Java does allow the assignment of an objects typed using an interface to a variable of type Object. To support this feature, it adds an extra rule to the IS-A definition which says T IS-A Object, for all types T. Had Object implemented an interface, there would be no need for this extra rule.

The following example shows an everyday situation in which an object typed using an interface is assigned to a variables typed as Object:

```
StringHistory stringHistory = new AStringHistory();
System.out.println(stringHistory);
```

The type of the formal argument of the println() method called in the second statement is of type Object. Here we are assigning to this variable typed using an interface. You will encounter several other examples assignments to variables of type Object.

Multiple Class Inheritance

In all of the examples we have seen so far, each class has a single super class. In other words, the subclass relationship defines a tree, with Object at its root. Should the relationship define a DAG, that is, should a class be able to inherit from multiple superclasses?

In the real-world, we do see examples of multiple inheritance. In particular, a child inherits genes (and possessions) from multiple parents. In object-oriented programming also, such inheritance makes sense. Assume that we wish to display a string history graphically and provide the capability to move the display in a 2-D space. We may wish to create a new class ALocatableStringHistory that inherits StringHistory methods from our existing AStringHistory class and location manipulation coordinates from a class ALocatable:

public class ALocatableStringHistory **extends** AStringHistory, ALocatable { … }

However, multiple inherirtance the following conceptual problem: When a member (variable or method) is declared in two superclasses, which should be inherited? Such a situation can easily occur. In our example, both AStrngHistory and ALocatable can override the toString() method. The toString() in AStringHistory may print all the elements:

```
public String toString() {
  String retVal = "";
  for (int i = 0; i < size; i++) {
    String separator =
      (i == size- 1)?"":":";
    retVal += separator + contents[i];
  }
  return retVal;
}
```

The version in ALocatable may print the Cartesian coordinates:

```
public String toString() {
   return  x + "," + y;
}
```

Thus, there is a conflict, and it is not clear how it should be resolved. Some languages have rules in the spirit of Mendel's laws to resolve these conflicts. Others, including Java, simply disallow a class from having multiple superclasses. This means we must often duplicate code. In our example, if we make ALocatableStringHistory a subclass of AStringHistory, then we must add all the methods of ALocatable in AStringHistory. We will later see that delegation is a solution to this problem.

Multiple Interface Inheritance

What about interfaces? Should we allow an interface to extend multiple interfaces:

```
public interface LocatableStringHistory extends StringHistory, Locatable { }
```

Again, it is possible to have the same member in multiple interfaces. When the member is a method, it is only a header, not the body. Thus, both of our example interfaces may declare the toString() method header:

```
public String toString();
```

The two headers are equivalent. Thus, there is no ambiguity in inheriting the same header from multiple interfaces. For this reason, Java does allow multiple interface inheritance.

Conflicting Named Constants

Interfaces can also include implementation-independent named constants. These are accessible in all implementations of the interface, which essentially inherit them. These raise the kind of conflicts Java tried to avoid by disallowing multiple class inheritance. Let us identify them and their resolutions.

Suppose the interface StringDatabase defines the SIZE named constant:

```
public interface StringDatabase  extends StringHistory  {
        public final int MAX_SIZE = 20;
```

```
    ….
}
```

Suppose the class AStringHistory also defines this constant:

```
public class AStringHistory implements StringHistory {
   public final int MAX_SIZE = 50;
   ….
}
```

So far so good - AStringHistory does not inherit from AStringDatabse, so it uses the value defined in its interface.

But if we do have a conflict if now create AStringDatabase by extending AStringHistory.

```
public class AStringDatabase
                    extends AStringHistory implements StringDatabase {
  public AStringDatabase() {
      System.out.println(MAX_SIZE);
   }
    ….
}
```

Java will allow the extension, but if the extension refers to MAX_SIZE, it will give a compile-time error saying that the reference is ambiguous. Thus, in Java, it is not possible to make references to named constants inherited from multiple types.

Suppose that we remove the reference from AStringDatabase to resolve the conflict and write the following code:

```
StringDatabase stringDatabase = new AStringDatabase();
System.out.println(stringDatabase.MAX_SIZE);
```

Since the constant is public, it is indeed possible to refer to it outside the class. Should Java allow this reference?

This time there is no ambiguity. Named constant references are resolved at compile time. As far as the compiler is concerned, the type of stringDatabase is the interface StringDatabase. So it uses the definition in StringDatabse. The exact type of the object assigned to the variable at runtime is not relevant to the compiler. We will see that with method calls, the situation is different. The exact method called is determined at runtime.

What if we used AStringDatabase as the type of stringdatabase?

```
AStringDatabase stringDatabase = new AStringDatabase();
System.out.println(stringDatabase.MAX_SIZE);
```

Our course rules preclude such typing, but Java does allow it. This time there is ambiguity, as Java does not know if it should use the constant in AStringHistory or StringDatabase. Therefore, Java does not allow this reference to the constant, giving a compile-time error.

This discussion shows that reasonable, but potentially confusing, techniques can be used to resolve the multiple inheritance problem. Languages such as C++ that allow multiple class inheritance have similar rules, which some do indeed find confusing, but allow more flexibility.

What do we do if we wish to refer to a named constant that has multiple definitions? We can choose the one we want by prefixing its name with the type name:

```
System.out.println(StringDatabase.MAX_SIZE);
System.out.println(AStringHistory.MAX_SIZE);
```

We have seen that a static (class) member can be referenced by using the type name as a prefix. A named constant is always static (as it is final, different instances cannot give it different values), whether it is declared static or not. This is the reason that this material often skips this keyword for named constants.

Multiple Interface Implementation

For the reason that Java allows an interface to extend multiple interfaces, it allows a class to implement multiple interfaces, that is, implement the union of the method headers of its interfaces. If the implemented interfaces have duplicate headers – there is no conflict. If they declare the same named constant, then the constant cannot be used in the class without prefixing it with the type name.

The following is an example of a class that implements multiple interfaces:

```
public class ALocatableStringHistory implements StringHistory, Locatable{
….
}
```

Implementation of multiple interfaces causes issues with our rule that interfaces rather than classes be used for typing variables. In the example above, we can store an instance of ALocatableStringHistory in a variable typed using either StringHistory or Locatable. If we type it using StringHistory, we can only call StringHistory methods on it:

```
StringHistory aStringHistory = new ALocatableStringHistory();
aStringHistory.add(“Joe Doe”);
We cannot invoke Loactable methods on it:
aStringHistory.setX(10);
```

The reverse problem occurs if we type it using Locatable:

```
Locatable aStringHistory = new ALocatableStringHistory();
```

The solution to the problem, of course,is to define a new interface that extends the two implemented interface:

public interface LocatableStringHistory **extends** StringHistory, Locatable { }

Now the class can implement this single interface:

public class ALocatableStringHistory **implements** LocatableStringHistory{
….
}

We can now use this interface to type an instance of the class and reference all of its methods:

LocatableStringHistory aLocatableStringHistory = new ALocatableStringHistory();
aLocatableStringHistory.add("Joe Doe");
aLocatableStringHistory.setX(10);

If we type it using Locatable, we can call only methods defined in that interface. Based on this discussion, we will require that:

A class should implement a single (possibly multiply extended) interface

As before, of course, we require that every public method of a class be declared in an interface it implements (, directly, or indirectly, through inheritance).

Annotation Inheritance?

Consider the following declarations:
@StructurePattern(StructurePatternNames.POINT_PATTERN)
public interface MutablePoint {…}

@StructurePattern(StructurePatternNames.OVAL_PATTERN)
public class ACartesianPoint implements Point {…}

public class AMutablePoint extends ACartesianPoint implements Point {…}

Here the interface and superclass of AMutablePoint define conflicting values for the same annotation? Which one should be used for it?

Java supports inheritance of only variables (including named constants, of course) and methods – not annotations. So it does not address the question. ObjectEditor uses inheritance to interpret the values of annotations, but defines annotation-specific inheritance resolution rules. As these rules are beyond the scope of this material, assume no inheritance occurs, and associate each class and interface with all of the annotations that describe it.

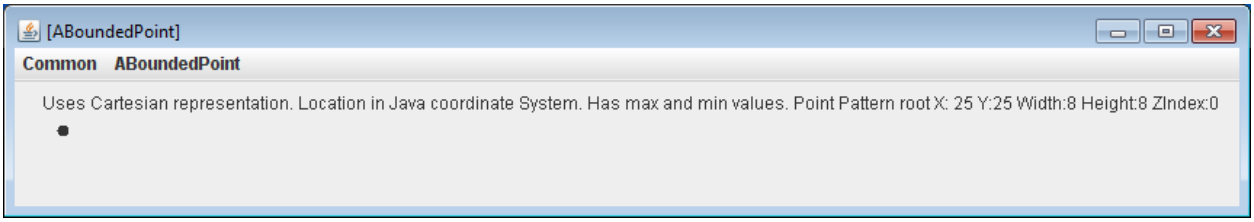
One particular rule of ObjectEditor is useful for illustrating a general approach to solving the multiple inheritance problem. It is defined for the Explanation annotation. Suppose the annotation has different values in different types of an instance:

```
@Explanation("Location in Java coordinate System.")
public interface Point {…}

@Explanation("Uses Cartesian representation.")
public class ACartesianPoint implements Point {…}

@Explanation("Has max and min values.")
public class ABoundedPoint extends AMuatablePoint implements BoundedPoint {…}
```

ObjectEditor will combine the explanations defined in each type of the value, as shown below in the tooltip text displayed for an instance of ABoundedPoint:



Some languages use a similar approach for resolving conflicting method implementations. When the method is invoked, they call all of the implementations!

Cyclic Inheritance?

We have seen that the inheritance relationship defined a tree for classes and a DAG for interfaces. Can we create cycles using this relationship? The answer is, as you expect, no: Conceptually If a type is a special case of another type, it cannot be a general case of that type unless it is equal to the other type.

Summary of Inheritance Benefits

There are several reasons for extending interfaces and classes::

- *Reduced Programming/Storage Costs*: The most obvious reason is that we do not have to write and store on the computer a copy of the code in the base type, thereby reducing programming and storage costs. The programming cost, of course, is minimal if we had a convenient facility to cut and paste. However, the source code of the base type may not always be available, which does not prevent it from being sub typed.

- *Easier Evolution*: Code tends to change. We may decide to change the `MAX_SIZE` constant of `AStringHistory`, in which case, would have to find and change all other classes that are logical but not physical extensions.
- *Polymorphism*: Inheritance allows us to support new kinds of IS-A relationships and hence polymorphism.
- *Modularity*: In the examples so far, we assumed that the base class and interface already existed when we created subtypes of them. For instance, we assumed first that we needed string histories and created appropriate interfaces and classes to support them. Later, when we found the need for string databases, we simply extended existing software. What if we have not had the need for string histories or Cartesian points, and were told to create string databases from scratch? Even in this case, we may want to first create string histories and then extend them rather than create un-extended string databases, because the extension approach increases modularity, thereby giving the accompanying advantages. In this example, it makes us understand, code, and prove correct the two interfaces and classes separately. Moreover, if we later end up needing string histories, we have the interface and class for instantiating them. Similarly, if our need was for bounded points, it is useful to create mutable and Cartesian points. The approach requires us to *design for reuse,* something that is very difficult to do in practice.

How to Get Inheritance Right?

It would be useful, then, to identify general principles for subtyping. One simplistic rule for inheritance could be:

> *If type T1 contains a super set of the method headers in another type T2, then T1 should extend T2.*

This rule is violated by our original definitions of StringHistory and StringDatabase, when we did not use inheritance, as all the methods of StringHistory are duplicated in StringDatabase

```
public interface StringHistory {
      public void addElement(String element);
      public String elementAt (int index);
      public int size();
}

public interface StringDatabase {
    // methods of StringHistory
    public String elementAt(int index);
    public void addElement(String element);
    public int size();
    // additional methods
    public void deleteElement(String element);
    public void clear();
```

```
    public boolean member(String element);
}
```

Of course, it is not violated by our version of the extending StringDatabase. However, it is also not violated by the following rewrites of the two interfaces, which follow the rule but not its spirit.

```
public interface StringHistory {
      public void addHistoryElement(String element);
      public String historyElementAt (int index);
      public int historySize();
}

public interface StringDatabase {
    // methods of StringHistory
    public String databaseElementAt(int index);
    public void databaseSddElement(String element);
    public int databaseSize();
    // additional methods
    public void databaseDeleteElement(String element);
    public void databaseClear();
    public boolean dadtabaseMember(String element);
}
```

Here we have added the term History (Database) to each method in History (Database). This term is unnecessary as the interface name gives the information in the method name. More important, it violates the *uniformity* principle, which says that:

> *Methods that perform the "same" conceptual function, even if they do so differently, should have the same name and order of arguments.*

The term "same" is in quotes because it requires some subjective interpretation, especially in an interface. An interface only gives the method header of an operation – the body of the method is provided by a class that implements the header. It is possible, and in fact, often expected, for two different classes to provide different implementations of the header. Thus, "same" behavior does not mean "identical" behavior. Similarly, addElement() in AStringSet does an extra check that is not performed by the addElement() of AStringHistory() and AStringDatabase(). Yet this operation is the same in all three collections as at an abstract level, it adds an element to a collection. This is why the rule says that the "conceptual" operation rather that the exact operations should be the same. Thus, interpretation of this rule requires experience.

To illustrate the importance of this rule, consider an everyday example. We know that printing a double is different from printing an int, and two different implementations of print() and println() are called to do the printing. Yet these methods have the same name. (As they both take only one parameter, the question of parameter order does not arise.) This principle ensures that we have fewer names and parameter orders to remember. The reason to provide overloading is to support this principle. In the printing example, we do not have to remember a different name for implementation of the two abstract operations.

Thus, the uniformity rule together with the inheritance rule gives us a formal precise bottom-up way to find opportunities for inheritance – we first write our types without inheritance using the uniformity rule and then use the inheritance rule to find opportunities for subtyping.

False Subtyping Example

If only life was so simple. Let us take a couple of examples to illustrate problems with the rule:

```
public interface Point { // mutable
  int getX();
  int getY();
  void setX (int newVal);
  void setY (int newVal)
}
public interface Line {
  int getX();
  int getY();
  void setX (int newVal);
  void setY (int newVal);
  int getWidth();
  int getHeight();
  void setWidth (int newVal);
  void setHeight (int newVal);
}
```

By our inheritance rule, we should remove all the Point methods from Line and make Line a subtype of Point.

```
public interface Line extends Point{
  int getWidth();
  int getHeight();
  void setWidth (int newVal);
  void setHeight (int newVal);
}
```

This approach does reduce code duplication. But it does not make mathematical sense to say that a Line IS-A Point. Yes they have common methods but they are fundamentally different kinds of objects. Recall that:

> T^1 **IS-A** T^2 if T^2 can be substituted by T^1" that, is whenever a member of T^2 is expected a member of T^1 can be used.

A line should not be used where a Point is expected – there is something wrong with our program if we do so.

Correct Subtyping Example

What do we do, then, to make sure Line and Point share code? Again, delegation, discussed later, provides an answer. In this example, even Inheritance does. The key is to abstract what is common in Pont and Line to another type. Whole a Line is not a Point, they both have locations. So we can develop another type, Locatable, and make both Point and Line subtypes of it.

```
public interface Locatable {
  int getX();
  int getY();
  void setX (int newVal);
  void setY (int newVal)
}
public interface Point extends Locatable {
}
public interface Line extends Locatable {
  int getWidth();
  int getHeight();
  void setWidth (int newVal);
  void setHeight (int newVal);
}
```

Our Point interface now becomes empty – which is fine. A type is both a name and a set of members, and in this case, we have created a new type simply to create a new name. In Java, the name of a type is used in type checking rules. So by creating a new name, we are ensuring that wrong things do not get assigned to variables.

Inheritance Rules and Throw-Away Prototypes

Based on this discussion, we can modify our inheritance rule:

> *Type T^2 should extend T^1 only if T^2 IS-A T^1;*
>
> *If type T^2 and T both have a set of common methods C, then there should exist a type T such that T^2 such that T^1 IS-A T,T^2 IS-A T, and T contains all methods in C, some of which may be overridden by T1 and/or T2, modulo the fact the lack of multiple class inheritance in Java.*

Thus, finding inheritance opportunities becomes more challenging as one must sometimes find a types T that is not T^1 or T^2. More important, ideally, we should not take a bottom-up process in which we first create types without inheritance and then change or *refactor* our code to add inheritance – that is more effort than simply not using inheritance. Ideally, we should find inheritance in a top-down process, and add inheritance in the first version of a type we create. Finally, as Java supports only a single superclass, one must decide which of multiple possible logical superclasses is the actual class.

Even expert programmers find this difficult – many believe that when given a new kind of code to develop, first write to the best of your ability, throw away what you wrote, and then rewrite it now that

you understand the problem and can modularize it. This is called the throw-away prototype approach. Naturally, one cannot take this approach in this one-semester course in which so many concepts are covered. By breaking your project into multiple assignments and providing guidance in each assignment, we have identified some of the steps you have to take, and thus reduces the need for a throw-away prototype. Nonetheless, the purists among you will indeed throw away a large amount of code, which is a good sign, as it shows you can distinguish between bad and good code, and more important, have learned new techniques to write good code.

Liskov Substitution Principle and Final Methods and Classes

Just when we thought we understood inheritance and IS-A, let us bring in one more example to show that in some cases, the community disagrees on what is appropriate. Suppose we want to create a Vertical Line, which is a line that has zero width and thus always stays vertical. Is a vertical line a line?

In English, that sounds right. It is very name suggests that it is a special case of a line, just as a String is a special case of an Object. More precisely, the set of vertical lines is a subset of the set of lines, a property we expect from a specialized version of a type. The reason why we might dispute this logic is that a vertical line reduces the functionality provide by a line, as the following code shows

```
public class AVerticalLine extends ALine {
  public AVerticalLine (int initX, int initY, int initHeight) {
    super (initX, initY, 0, initHeight);
  }
  public void setWidth(int newVal) {
  }
}
```

The setWidth() method is now a no-op, it does nothing, to ensure that that the line remains vertical.Our rule for IS-A is that:

> *T^2 **IS-A** T^1 if 1^2 can be substituted by $T^{2,}$ that is, whenever a member of T^1 is expected, a member of T^2 can be used.*

What does substituted mean? Does it mean that the methods that work on T^2 the same way as they do in T^1? Some purists believe that the answer is yes. Barabara Liskov, a famous programming language designer and inventor of the iterator concept, used in Java, proposed the Liskov substitution principle for determining if a type is a special case of another type which says that a subtype should pass all the checks written for its supertype. In our example, this is not the case. For instance, we might write the following check for a line;

```
public static boolean checkLine(Line aLine, int aWidth) {
  aLine.setWidth(aWidth);
  return aLine.getWidth() == aWidth;
}
```

A regular line will pass the check but not our vertical line. The Liskov substitution principle allows very limited overriding of methods in which only the performance of the overridden method can change – its behavior cannot. It would not allow a set to be special case of a database, as now duplicates are removed. More drastic, it would not allow us to ever override an Object method such as toString().

Perhaps that is the right thing to do. Let us take our example to see how we would relate vertical lines and lines while following this rule. We could remove the width property from a vertical line and make a line extend a vertical line by adding this property. Any check written for a vertical line would also pass for a line. But we have now gone from bad to worse. A line is not a vertical line – where a vertical line is expected, we should not pass an arbitrary line. We could define an intermediate class, type, a shape with height, and make line and vertical line extend it, but that seems to be awkward and does not capture the fact that the adjective vertical indicates that a vertical line is a line, and we might want to create polymorphic code that operates on both types and collections that store both types. To address the problem of setWidth() being a no-op, we can give an error message if it is called:

```
public void setWidth(int newVal) {
        System.out.println("Cannot change width of a vertical line");
 }
```

In Java, immutable collections are designed using this philosophy, and give error messages when write operations are called on them.

Of course, programmers might still be confused by such error messages, a problem that does not arise if we follow Liskov's substitution principle. Given these pros and cons, it is up to you if you want to follow the Liskov substitution principle when given no constraints. There will be times when we will ask you explicitly break this rule by overriding the behavior of existing types such as Object.

Recognizing the importance of this principle, at least in some situations, Java designers allow a method declaration, like a variable declaration, to be prefixed with the keyword **final**. A final method cannot be overridden by a subclass. As a result, a subclass cannot compromise the semantics of such a method. Java also allows a class declaration itself to be preceded with the **final** keyword. Such a class cannot be subclassed. The Java String class is an example of a **final** class. We cannot subclass this class to, for instance, override the toString() method of this class to return a String different from the one returned by the String implementation. This is probably a good thing and provides part of the motivation for final classes, which restrict the IS-A relationships we can create.

Implicit and Explicit Casting and Conversion of Primitives

We said above that in the case of primitive types, an expression of type T^2 can be assigned a variable of type T^1 only if T^1 is the same as T^2. Yet some of you have probably seen that the following is legal:

```
int i = 2;
double d = i;
```

This is allowed because Java automatically converts the integer to a double and then applies the type checking rule.

What about the following?
double d = 2.5;
int i = d;
Not surprisingly, this is not allowed. In the first case, there was a double to which the assigned int could be converted, but in the second case, there is no int to which the double can be converted.

We can define the narrow/broader relationship between types: If values of type T^2 are a subset of the values of type T^1 , then $T^2$is narrower than T^1 and $T^1$is broader than T^2.

Java allows a value of primitive type T^2 to be assigned to a variable of type by converting the value of type T^2 to a value of T^1 and then doing the assignment. As you have probably seem from CS-1, in the case of primitives, each type uses a different format for the same value, hence the need for conversion.

You may have also seen that it is possible to a variable of a narrower type a value of a broader type by performing a cast:

double d = 2.5;
int i = **(int)** d;
This cast tells Java to truncate the double and then assign it to the int variable. Naturally, this is a dangerous operation, as there can be loss of information. As in the case of object casts, by putting in the cast, we are telling Java we know these dangers and know what we are doing.

The casts have two different meanings for primitives and objects. In the case of primitives, they indicate a conversion to another type and the danger is that there might be loss of information. In the case of objects, there is no loss of information or conversion, all object variables store pointers. Now the danger is that at runtime, the actual type, T^2, of the cast expression may not be consistent with the types T^1 used as cast, that is, it may not be the case that T^2 IS-A T^1. Thus, the cast in object types arises out of flexible typing rather than conversion issues.

Yet primitive and object types is related in that narrower and IS-A relations are similar: The set of values of a primitive type is the union of the set of values of types that are narrower than it, just as the set instances of an object type is a union of the sets of values of instances of its subtypes. Or to put it perhaps more simply, a value of a primitive type, T2, is also a value of a member of a type T^1 that is broader than T2 (after conversion), and an instance of a subtype, T2 is also an instance of a type T^1, if T^2 IS-A T^1.

It is possible to convert between object types also, but that requires code in the types to do the conversion manually. For example, in Java we can convert an instance of a list to an array, because implementations of List provide a toArray() method that creates a new array containing the elements of the list. More interesting, the class Array provides an asList() static method that provides a list “view” of an array without creating a new list. The type of the returned object is an immutable list, which allows the array elements to be read but not modified.

Mixing Primitives and Objects and Possible NullPointerException

We have seen above two different sets of type-checking rules for primitives and objects. Can we mix the two? That is, can be assign a primitive expression to a primitive variable, and vice versa. Conceptually it does not make sense as a primitive is not an object and an object is not a primitive. Thus, any such assignment will require one type to be converted into another, much as an int is concerted to a double and vice versa. Java does do this conversion. Recall that for each primitive type, Java has defined is a wrapper class. Thus, for the type int, it has defined a wrapper class, Integer. Java converts a primitive type to an instance of the corresponding wrapper class when it is assigned to a primitive variable. When an instance of a wrapper class is assigned to a primitive variable, Java extracts the wrapper value from the wrapper object, and assigns it to the primitive variable.

To understand this mixing of primitives and objects, consider the following assignments.

```
Object anObject = 5;
int i = anObject;
int i = (Integer) anObject;
```

Which of these are legal?

The assignment:

```
Object anObject = 5;
```

Is legal because it is equivalent to:

```
Object anObject = new Integer(5);
```

The assignment:

```
int i = anObject;
```

is illegal as there is no way to convert an arbitrary object to an int.

On the other hand, the assignment:

```
int i = (Integer) anObject;
```

is legal (at compile time). It is equivalent to:

```
int i = ((Integer) anObject).intValue;
```

At runtime, if an Object is found to indeed point to an Integer, then there would be no error. Otherwise there are two cases. First, the variable points to some other kind of object, in which case we will get a

ClassCastException. Second, and more subtle, the variable points to nothing, that is, has the null value. In this case, we will get an NullPointerException. This exception makes sense if we see the code:

int i = ((Integer) anObject).intValue;

However, it may not make sense if we see:

int i = (Integer) anObject;

or more subtle,

2 + (Integer) anObject

as no method call is explicitly called, and NullPointerException is a result of a method call made on a variable holding the **null** value. So you need to be aware of what is under the hood when you mix primitives and objects.

Legal Casts

Casts can, of course, give us runtime errors. They can also give us compile time errors. For example, the following cast is illegal:

```
int i = (int) true;
```

This is not surprising, because in the case of primitives, a cast implies a conversion, and there is no way to convert a Boolean to an int. These two sets are disjoint, so there is no mapping from one to the other. In some languages, false and true values are stored using the values 0 and 1, but that is purely an implementation fact, and conceptually it makes no sense. In general, a primitive expression of type T^2 can be cast to a type T^1, only if T^1 is a subset of T^2 or T^2 is a subset of T^1. Thus one can cast from a primitive type to a broader or narrower type.

Let us now consider object types. Consider the following

ABMISpreadsheet bmiSpreadsheet;
…..
ACartesianPoint cartesianPoint = (ACartesianPoint) bmiSpreadsheet;

Here, we are typing using classes rather than interfaces. Should this cast be allowed? It should be allowed if there is some chance the variable bmiSpreadsheet can be assigned an instance of ACartesianPoint. We know that this variable can be assigned only an object that is ABMISpreadsheet or a subclass of this type. For the cast to be successful, this object must also be ACartesianPoint or its subclass. We know that ACartesianPoint and ABMISpreadsheet do not have a subclass relationship between them. Thus, the object must be a direct or inderect subclass of both classes. But, in Java, a class

cannot have two superclasses (that are unrelated by a subclass relationship between them). Thus, it is never possible for the cast above to succeed, and we will get a compile time error. This leads to the following rule:

> *We can cast an object expression of class C^1 to class C^2 only if C^1 is the same or super or subclass of C^2.*

Interestingly, the following is legal:

```
ABMISpreadsheet bmiSpreadsheet = …; // unspecied assignment
Point cartesianPoint = (Point) bmiSpreadsheet;
bmiSpreadsheet = (ABMISpreadsheet) cartesianPoint;
```

For this to be legal, some subclass of ABMISpreadsheet must also be a Point, and some instance of Point must also be a ABMISpreadsheet. Here is such a subtype:

```
public class ABMISpreadsheetAndPoint
        extends ABMISpreadsheet
        implements Point {
…
}
```

We can now fill in the unspecified assignment to make the subsequent assignments legal:

```
ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheetAndPoint();
Point cartesianPoint = (Point) bmiSpreadsheet;
bmiSpreadsheet = (ABMISpreadsheet) cartesianPoint;
```

When we add interfaces to the mix, the subtype relationship is no longer a tree. It is possible for an instance of any interface to be an instance of any other type, and an instance of any type to also be an instance of an interface. Thus, in the case of interfaces, the rules seem to be:

We can cast an object expression typed by an interface to any object type.

We can cast any object expression to an interface type.

To add to the confusion, the following casts are wrong:

```
String string = “hello”;

Point cartesianPoint = (Point) string;

String string2 = (String) cartesianPoint;
```

For this cast to succeed, we must have a subclass of String that implements Point, But String is a **final** class, which means it cannot be subclassed:

```
public final class String {
…
}
```

Thus, we must modify our rules:

We can cast an object expression typed by any interface to any non-final object type.

We can cast any object whose type is not a final class to any interface type.

Summary

- Inheritance and implementation are examples of IS-A relationships.
- If T2 IS-A T1, then a value of type T2 can be assigned to a variable of type T1.

Exercises