COMP 401

Prasun Dewan[1]

Inheritance and Variables

In the previous chapter, we studied the basics of inheritance in the context of collections. Here, we will use a different example to present subtleties regarding the relationship between variables in subclasses and superclasses. We will see what it means to declare variables in subclasses, how these and inherited variables are initialized by subclass and superclass constructors, and how the variables are stored in memory.

Mutable Point

The examples we present here extend th involve the interface Point, we saw earlier. Recall that Point was immutable; it defined no setters:

```
public interface Point {
  public int getX();
  public int getY();
  public double getAngle();
  public double getRadius();
}
```

We can extend this interface to define MutablePoint, which declares the missing setters;

```
public interface MutablePoint extends Point {
  void setX(int newVal);
  void setY(int newVal);
}
```

As this interface declaration shows, the keyword **public** can be omitted in an interface as it is implied - only public methods can be declared in an interface.

To implement this interface, we should, of course, reuse an implementation of Point. Let us extend ACartesianPoint, reproduced below, to refresh out memory:

```
public class ACartesianPoint implements Point {
  int x, y;
  public ACartesianPoint(int theX, int theY) {
    x = theX;
```

```
    y = theY;
  }
  public ACartesianPoint(double theRadius, double theAngle) {
    x =  (int) (theRadius*Math.cos(theAngle));
    y = (int) (theRadius*Math.sin(theAngle));
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public double getAngle() { return Math.atan2(y, x); }
  public double getRadius() { return Math.sqrt(x*x + y*y); }
}
```

The extension is mostly straightforward:

```
public class AMutablePoint extends ACartesianPoint
                    implements MutablePoint {
  public AMutablePoint(int theX, int theY) {
    super(theX, theY);
  }
  public void setX(int newVal) {
    x = newVal;
  }
  public void setY(int newVal) {
    y = newVal;
  }
}
```

The setters directly access the superclass variables, much as methods in AStringDatabase directly access variables of the superclass. In this and all of the other examples, we will assume that subclasses are declared in the package of the superclass, and thus can see variables in the superclass that have default access.

Calling Superclass Constructors

One subtle issue has to do with the constructor:

```
public AMutablePoint(int theX, int theY) {
    super(theX, theY);
  }
```

It takes the same arguments as the constructor of the superclass. Like the addElement() method in AStringSet, it uses the **super** keyword to call a superclass method. Unlike addElement(), no method name followes the keyword. That is because we are calling a constructors, whose name is derived fro the name of its class. Thus, the constructor of AMutablePoint simply calls the constructor of ACartesianPoint, passing it the arguments it receives.

If AMutablePoint() simply calls the superclass constructor, why did we have to define a constructor in AMutablePoint? Why could we have not simply omitted it, implying that the superclass constructor should be called?

The reason is that when we instantiate a class, we always specify a constructor whose name is the same as the name of the class:

new AMutablePoint (34, 22);

The constructor in the superclass has a different name, and cannot be called implicitly. Hence the need for at least one explicit constructor in each class. It is possible to create a different design in which the instantiating code looks up the superclass chain to find a constructor that can take the supplied arguments. The designers of Java chose not to use this approach, considering a superclass constructor not inheritable but callable from the subclasses.

Bounded Point

To further understand the relationship between subclass and superclass constructors, let us consider an extension of AMutablePoint, ABoundedPoint, which confines the point to a rectangular area whose upper left and lower right corners are defined by two new properties, UpperLeftCorner and LowerRightCorner.

```
public class ABoundedPoint extends AMutablePoint
                        implements MutablePoint {
 MutablePoint upperLeftCorner, lowerRightCorner;
 public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
  super(initX, initY);
  upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
  fixX();
  fixY();
 }
 void fixX() {
  x = Math.max(x, upperLeftCorner.getX());
  x = Math.min(x, lowerRightCorner.getX());
 }
 void fixY() {
  y = Math.max(y, upperLeftCorner.getY());
  y = Math.min(y, lowerRightCorner.getY());
 }
…
```

}

The exact code to implement the class is not important – this is the reason why its complete implementation is not given. What is relevant here is that this class has two extra instance variables, upperLeftCorner and lowerRightCorner. This is the first subclass that declares its own variables - the previous subclasses – AStringHistory, AStringSet and AMutablePoint - simply inherited variables from their superclasses. When a class both declares and inherits variables, both sets of variables may need to be initialized by its constructor. The example above shows how that is done. It first calls the superclass constructor to initialize the inherited variables. It then initializes its own variables. Finally, it calls the methods fixX() and fixY() to fix the position of the coordinates in case they are outside the allowed region.

Order of Constructor Calls

What if changed the order in which the subclass and superclass variables are initialized by making the call to super the last step of the constructor?

```
public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
     upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
     fixX();
     fixY();
     super(initX, initY);
}
```

This is not correct, as the X and Y coordinates are not fixed.

What is we call super in the middle of the constructor:

```
public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
     upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
     super(initX, initY);
     fixX();
     fixY();
}
```

This would actually work. However, Java does not allow either of these alternatives. It requires the call to super be the first step in the constructor. The reason is that the values of subclass variables can depend on those of the superclass variables, which are visible in the subclass. However, the reverse is not true, as the subclass variables are not visible in the superclass. It is important to initialize the independent variables before the dependent variables. We do no harm by making the initialization of

the superclass variables the very first step of a constructor. In many, but not all situations, we do harm by not making it the first step.

Omitting Super Call

What if we omit the call to super completely?

```
public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
     upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
     fixX();
     fixY();
}
```

This means that no constructor is being called to initialize superclass variables. In some situations, there may be no superclass variables to initialize. In this example, the x and y variables are not initialized. We could overcome this specific problem by initializing the superclass variables in the subclass constructor:

```
public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
     x = initX;
     y = initY
     upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
     fixX();
     fixY();
}
```

This solution repeats code in the superclass, requires the subclass designer to know initialization details of the superclass, and most important, does not work if the superclass variables are not visible in the subclass. The seriousness of the problem is highlighted by the fact that class Object, defined by Java, is the superclass of all classes. We nothing about the variables it defined and how they should be initialized. Moreover, the implementation of Object may change with each new version.

For this reason, Java insists that a call to super always occur in a constructor, and that it be the first call. This rule not only prevents mistakes in our code but also ensures that Java classes we extend are initialized correctly.

This rule seems to be at odds with the constructor of ACartesianPoint, which has no super call:

```
public ACartesianPoint(double theRadius, double theAngle) {
   x =  (int) (theRadius*Math.cos(theAngle));
   y = (int) (theRadius*Math.sin(theAngle));
```

```
 }
```

The reason this code compiles is that, if we omit a super class, the Java compiler inserts a call to the parameterless superclass constructor. Thus, the code above is equivalent to:

```
public ACartesianPoint(double theRadius, double theAngle) {
   super();
   x =  (int) (theRadius*Math.cos(theAngle));
   y = (int) (theRadius*Math.sin(theAngle));
 }
```

This superclass constructor is called in class Object, the superclass of ACartesianPoint.

So why did this approach not work when we omitted the superclass constructor in BoundedPoint?

```
public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
     upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
     fixX();
     fixY();
}
```

The reason is that Java first inserts a call to a parameterless superclass constrictor:

```
public ABoundedPoint(int initX, int initY,
      MutablePoint anUpperLeftCorner,
      MutablePoint aLowerRightCorner) {
     super();
     upperLeftCorner = anUpperLeftCorner;
     lowerRightCorner = aLowerRightCorner;
     fixX();
     fixY();
}
```

It then complains that the superclass of ABoundedPoint, AMutablePoint, does not have a parameterless superclass constructor.

Thus, Java insists that the first step in each constructor of a class be a call to some superclass constructor, and if such a call does not occur, it inserts one to the parameterless constructor, which works only if the immediate superclass of the class has such a constructor. One of the reasons for these rules is that Java wants to ensure that the parameterless superclass constructor in Object is always called.

Omitting Constructors

This goal is apparently not met when we completely omit a constructor, as we did in many classes such as AStringSet:

```
public class AStringSet extends AStringDatabase {
  public void addElement(String element) {
        if (member(element)) return;
        super.addElement(element);
 }
}
```

If we do not provide an explicit constructor, the Java compiler inserts a parameterless constructor that calls super():

```
public class AStringSet extends AStringDatabase {
  public AStringSet () {
    super();
  }
 public void addElement(String element) {
        if (member(element)) return;
        super.addElement(element);
 }
}
```

The supeclasses of AStringSet, AStringDatabase and AStringHistory also do not have an explicit constructor. Thus, Java inserts a parameterless constructor in these classes. Thus, the super() call in AStringSet eventuall results in the parameterless constructor in Object being called, through the inserted constructors in AStringDatabase and AStringHistory.

many of the classes we saw earlier, such as A which had no constructors,

We do not harm by making the initializiation of the superclass

This class creates an extension of ACartesianPoint that confines the point to a rectangular region whose upper and lower right corners are defined by two properties. The exact semantics of the class do not matter – this is the reason why the implementation of the methods of this class is not given. What is

important is that this class inherits the instance variables of the super class and can override methods to ensure that the point cannot be moved outside the associated rectangle. Like AStringSet, it uses the **super** keyword to call a method in the superclass.

```
super.setX(newX);
```

In addition, it uses it to call the superclass constructor

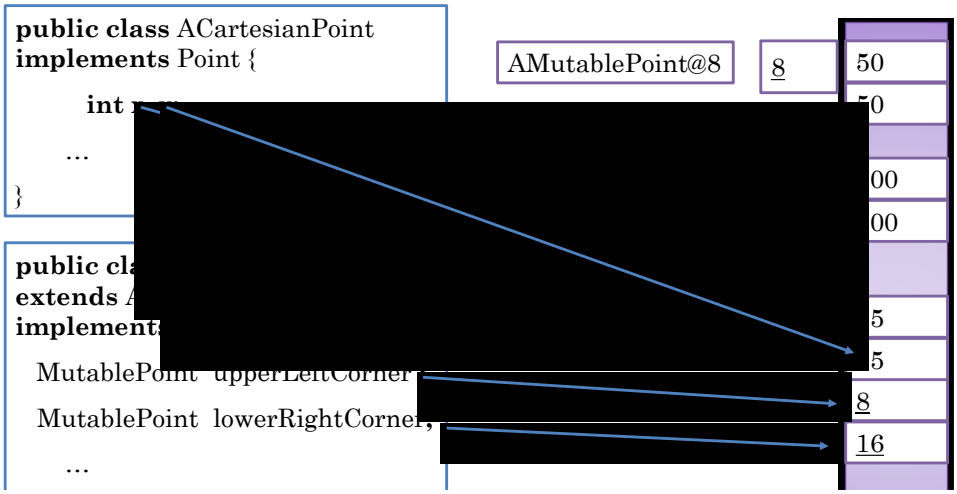
```
super(initX, initY);
```

We will use this class, together with AStringDatabase and AStringSet, to illustrate below some subtle issues with inheritance and the methods of class Object.

Inheritance and Memory Representation

One of these issues is the memory representation of instances of subclasses. Specifically, are the instance variables defined by the subclass and super classes stored in the same or different memory blocks? The following figure provides the answer. It shows the memory representation of the following instance of ABoundedPoint():

```
new ABoundedPoint(75, 75, new AMutablePoint(50,50), new AMutablePoint(100,100) )
```



As it shows, when an instance of a class is crated, Java allocates a single memory block for the instance variables defined by the class and all classes in its superclass chain. In this example, the block consists of the two primitive variables, x and y, defined by the super class of the superclass of ABoundedPoint, and the two new Object variables, upperLeftCorner and lowerRightCorner, defined by the class. Recall that each object instance variable takes exactly one word and that, as the figure shows, a primitive variable directly stores a value while an object variable stores a memory address or pointer. In this figure, pointers are underlined. As it shows, the upperLeftCorner contains the address 8, which points to an

instance of AMutablePoint, which in turn, is a memory block containing two int variables defining the x and y coordinates.

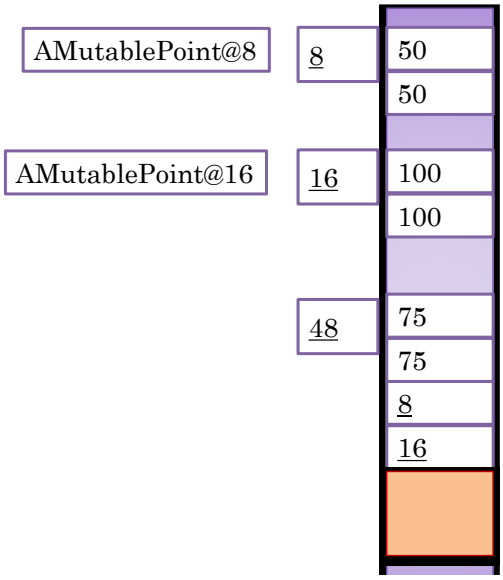
By storing all variables of an object, Java makes it simpler to allocate, copy, access and free up the memory associated with these variables.

Duplicate Variable Names in Subclasses

What if we, accidentally or willfully, declare variables in a class with the same name as variables in a superclass? We know we can create methods with the same name and parameters in a class and its subclasses – that is how methods are overridden. Should we allow variables with the same name in a class and its subclasses? In Smalltalk, the language that popularized object-oriented language, the answer was no. In Java, on the other hand, the answer is yes. Thus, in Java, the following code is legal, even though x and y are declared in the superclass ACartesianPoint:

```
public class ABoundedPoint extends AMutablePoint implements BoundedPoint {
  int x, y;
  MutablePoint  upperLeftCorner ;
  MutablePoint  lowerRightCorner;
     …
}
```

Figure ??? shows the memory representation of an instance of an instance of the modified class:



As we see above, the representation has two extra primitive variables – these are the ones declared in ABounedPoint.

Allowing a variable with the same name V, in a class A and its subclass B, raises the following questions:

1. To which V do methods in class A refer?

2. To which V do methods in class B refer?

For efficiency reasons, these questions have to be resolved when the methods are compiled. The answer to the first question, then, is clear. A class can have multiple subclasses, and these are, in any case, not known in subclasses. Thus, the answer is that methods in A refer to the variable in the superclass, A. The answer to the second question is also clear. If methods in B referred to the variable in the superclass, what would be the point of declaring a duplicate version in the superclass. Thus, consistent with the principle that a more specific scope wins over a more general one, methods in class B refer to the V in class B.

Let us understand these rules in the context of our example:

1. To which x and y are the arguments of the constructor of ABoundedPoint assigned? The assignments occur in the superclass constructor, so the answer is the x and y in ACartesianPoint.
2. To which x and y do the methods fixX() and fixY() refer? As these methods are declared in ABoundedPoint, they refer to the ones in ABoundedPoint.

Thus, our class no longer works it does not fix the variables that are initialized, returned by getters and set by setters.

It is not uncommon to create such duplicate variables in Java when we refactor a class by move some of its code to a superclass. In this process, we may paste some variables in the superclass and forget to delete it from the subclass. Debugging to find his mistake can be a nightmare. Therefore, you should be aware of the possibilities of making such a mistake and be on guard against it.

Sometimes a subclass programmer does not know which variables have been declared in a superclass, especially if the superclass has been written by someone else and not given the subclass access to them. In this situation, the only way to determine the names of superclass variables is to use the debugger. No harm can come out of duplicating the name of a variable that is not visible in the subclass. Even then, it can be confusing to do s. Therefore, we will require that:

> *In a subclass B, do not duplicate the names of variables in B, if you know these names*.

If there are situations in which duplicate variable names is useful – I have not seem them. This example is evidence that Java is a complex language and it is difficult to get everything right about its design, especially when the designer, James Gosling, created the language for himself and his small group of friends.

Why Inheritance?

There are several reasons for extending interfaces and classes, as we have done above, rather than creating new ones from scratch, as we did before:

- *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 polymorphism, as explained below.
- *Modularity*: We assume above that the base class and interface already existed when we created subclasses 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, and were told to create string histories from scratch? Even in this case, we may want to first create string histories and then extend them rather than create unextended 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. The approach requires us to *design for reuse,* something that is very difficult to do in practice.

Inter-Type Reuse Rules

In general, knowing when to use inheritance requires some experience. A simple rule of thumb is that if there is code duplication in different types (interfaces/classes), then you should think either of inheritance or delegation, which we will study later. (Intra-type code duplication involves two methods within the same class duplicating code, and is removed by putting the common code in a common method called by both methods.)

Our previous version of StringDatabase duplicated all the method headers in StringHistory. By examining the two interfaces, it was pretty straightforward to reduce this code duplication by making StringDatabase a subtype of StringDatabase. In the previous versions of AStringHistory and AStringClasses had even more code duplication, as not only were public method headers duplicated in these classes, but also the bodies of these methods, and the variables and non public method accessed by these methods. Again, it was pretty straightforward to reduce the code duplication by making AStringDatabase a subtype of AStringHistory.

This example leads us to our first re-use rule: Use inheritance (and/or delegation) to get rid of method header/body duplication in different types (interfaces/classes).

This rule is not sufficient to eliminate duplication because the same abstract operation can be associated with different method headers. For instance, the addElement() method in (a) StringHistory and

AStringHistory could have been called addToHistory(), and (b) StringDatabase and AStringDatabase could have been called addToDatabase().

This leads us to the second code re-use rule: Ensure that conceptual operations with the "same" semantics (behavior) are associated with same method headers in different types (interfaces/classes).

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, as we will see when we see the implementation of addElement() in AStringHistory, this method 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. It is in the interpretation of this rule where experience really helps.

toString()

To better understand inheritance and Object, the last class in the superclass chain of all classes, let us try and understand in more depth and override the three methods of it identified above: toString(), equals(), and clone().

Let us begin by overriding in `AStringSet` the `toString()` method inherited from class Object:

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

The method returns a ":" separated list of the elements of the collection:

```
stringSet.toString() → "James Dean:John Smith"
```

Recall that the implementation inherited from Object gives us the class name followed by the memory address:

```
stringSet.toString() → "AStringSet@1eed58"
```

Many classes override the `toString` method, since the default implementation of it inherited from `Object` is not very informative, returning, as we saw before, the class name followed by the object address. Recall also that `println` calls this method on an object when displaying it. The reason why

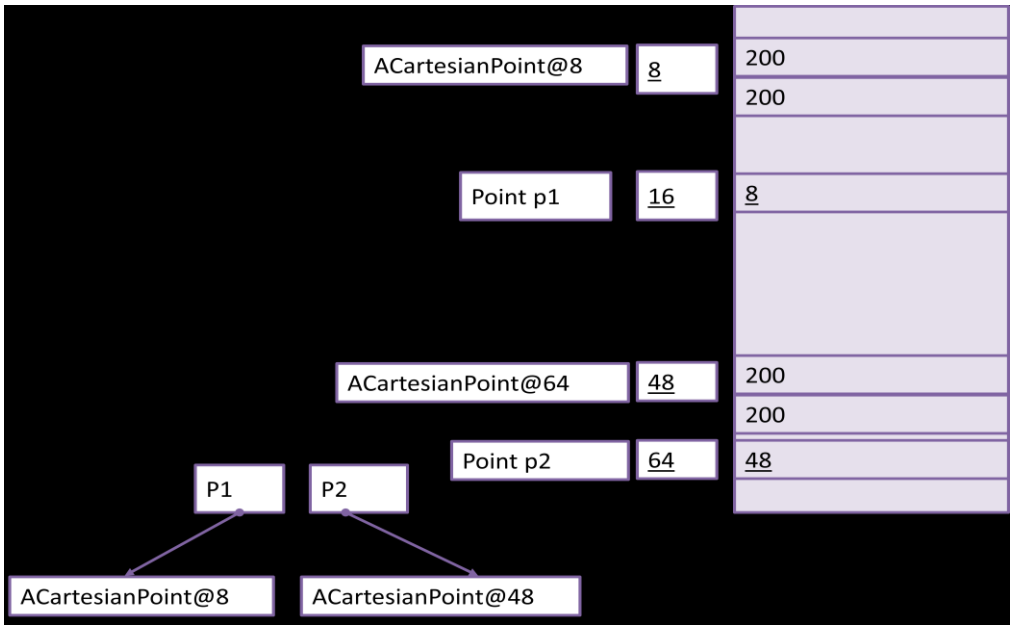
`println` tends to display a reasonable string for most of the existing Java classes is that these classes have overridden the default implementation inherited from Object.

equals()

Java already provides an operator to check for equality, ==, so why do we need a method that, based on its name, seems to do the same thing? To illustrate the difference between == and equals(), consider the following statements:

```
Point p1 = new ACartesianPoint(200, 200);
System.out.println(p1 == p2);
p1 = new  ACartesianPoint (200, 200);
System.out.println(p1 == p2);
```

The == operator dereferences the two pointers, and compares the resulting objects. When the first statement is executed, both p1 and p2 refer to the same object. Therefore, we can expect the first print statement to print "true". But what about the second print statement? Both variables refer to the same logical point in the coordinate space, the point with the coordinates (200,200). However, they refer to different physical objects, as shown below.

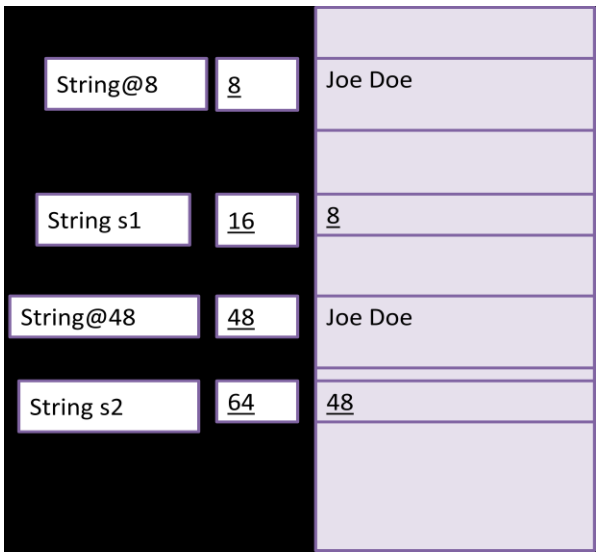


The == operator, in fact, simply checks if its left and right hand side are the same physical object. If not, it returns the false value. It does not understand the concept of two physical objects being the same logical entity. It is the responsibility of each object to define a method that checks if two objects represent the same logical entity. The convention is to call this method, `equals()`. Several predefined classes such as `String` provide such a method.

In String, this method does a character-by-character comparison of the strings that are compared, and returns true if the two strings have the same sequence of characters. The following interaction shows the difference between == and `equals()` for strings:

```
String s1 = "hello world";
String s2 = "hello world";
System.out.println(s1==s2);
System.out.println(s1.equals(s2));
s1 = s2;
System.out.println(s1==s2);
System.out.println(s1.equals(s2));
```

All print statements except the first one will print "true". The reason why the first one returns false is that the two strings are stored in separate memory blocks, as shown in the figure below.



Consider now the following code:

```
Point p1 = new ACartesianPoint(200, 200);
p1 = new  ACartesianPoint (200, 200);
System.out.println(p1.equals( p2));

StringHistory stringHistory1 = new AStringHistory();
StringHistory  stringHistory2  = new AStringHistory();
stringHistory1.equals(stringHistory2);
```

Both println() calls print **false**. In other words, the equals() method, in these two cases, has the same behavior as ==. The reason is that we have not redefined equals() for instances of StringHistory and Point and the default behavior of equals() is the same as ==:

```
//implementation in Object
public boolean equals(Object otherObject) {
        return this == otherObject;
}
```

Each class that declares instance variables whose values influence equality must redefine equals(). To illustrate, suppose users of AStringSet are not happy with the default behavior of equals(). We can add to the class the following to override the default implementation:

```
public boolean equals(Object otherObject) {
    if (otherObject == null || !(otherObject instanceof  AStringHistory))
        return false;
     AStringHistory  otherStringHistory = (AStringHistory) otherObject;
     if (size != otherStringHistory.size)
        return false;
     for (int index  = 0; index < size; index++)
        if (!contents[index].equals(otherStringHistory.contents[index]))
                return false;
     return true;
}
```

The operation o **instanceof** T returns true if the class of o IS-A T. This operation is used to return false when the other object is not a string history. The value of null IS-A T, for all T. Since **null** IS-A StringHistory, an extra check is needed to determine if the otherObject is **null**. If the other object is a non null StringHistory, the method does an element-by-element comparison of the two collections to determine if they are equal. To do so, it accesses the `contents` and `size` variables of the two instances that are compared. These variables are actually defined in the superclass AStringHistory. In fact, AStringDatabase and AStringSet do not define any instance variables. Thus, it is better to move the method to AStringHistory, thereby allowing all instances of AStringHistory and its subtypes to use it.

Accessing arbitrary variables/methods of remote instances

The implementation above illustrates a feature of Java we have not seen before. When a method is called on an instance I of class C, it is possible not only to access arbitrary variables and methods of I but also all other instances of class C. As we saw earlier, to access instances of I, we either omit the target instance:

```
contents[index];
```

or use the keyword **this** to identify it:

```
this. contents[index];
```

To access the variables of some other instance of the class, which we will refer to as a remote instance, we replace this with some variable holding a pointer to the instance. In the example above, otherStringHistory holds a pointer to the remote instance. Hence we use this variable to indicate the target object:

```
otherStringHistory.contents[index];
```

We can use the same syntax to refer to arbitrary methods of the remote instance. In general, in a class definition, all methods and variables of all instances of the class are visible.

Accessing public vs. arbitrary members of remote instances

In the example above, we broke an important rule given before by using a class to type a variable:

```
AStringHistory  otherStringHistory = (AStringHistory) otherObject;
```

Had we used the interface of the class to type it

```
StringHistory otherStringHistory = (StringHistory) otherObject;
```

we would not have been able to access non public members (methods/variables) of the remote instance as these are not defined by interface. Sometimes in a class, it is necessary to access internal members of remote instances. In that case, we are forced to use it for typing. However, in this example, the interfaces exposes the required information, as illustrated by the following rewrite of equals:

```
public boolean equals(Object otherObject) {
    if (otherObject == null ||!(otherObject instanceof  StringHistory))
        return false;
     StringHistory  otherStringHistory = (StringHistory) otherObject;
     if (size != otherStringHistory.size())
        return false;
     for (int index  = 0; index < size; index++)
        if (!contents[index].equals(otherStringHistory.elementAt(index)))
                return false;
     return true;
}
```

This implementation is more polymorphic in that it allows us to compare arbitrary implementations of StringHistory and not only those that are instances of AStringHistory. It is slightly less efficient because the variables of the remote instance are access indirectly, through public methods, rather than directly. In most applications, this inefficiency is not a problem.

Overriding vs. Overloading

An even more elegant implementation of equals() is given below:

```
public boolean equals(StringHistory  otherStringHistory) {
     if (otherStringHistory == null || size != otherStringHistory.size())
        return false;
     for (int index  = 0; index < size; index++)
        if (!contents[index].equals(otherStringHistory.elementAt(index)))
                return false;
     return true;
}
```

This implementation does not have use the **instanceof** operation, which makes the code messy. However, this implementation does not really override the equals() method inherited from Object as its parameter type is different. Hence it overloads rather than overrides the inherited method. As a result, the method does not hide the inherited method; both methods are available for invocation. If we wish to invoke the overloaded method on instances of StringHistory, we must add the signature (header) of

the method to the interface. Once we do that, the syntax for invoking it is exactly the same as the one for the inherited method:

```
StringHistory stringHistory1 = new AStringHistory();
StringHistory  stringHistory2  = new AStringHistory();
stringHistory1.equals(stringHistory2);
```

The equals that accepts the more specific argument type, StringHistory, is called, even though the equals() in objects would also be legal.

On the other hand, the call:

```
stringHistory1.equals("Not an instance of StringHistory");
```

would call the equals() in Object.

It seems that overloading and overriding in this example give the same results. If the argument is an instance of StringHistory, then it seems the more specialized equals() is called. Otherwise, in the overriding case, the specialized method is called, which returns false, and in the overloaded case, the more general method is called, which also return false.

However, they can indeed give different results. The reason is that what matters, when overload resolution is done, is not the type of the actual object assigned to a variable but the type of the variable. The reason is that overload resolution is done at compile time, and at this time, it is not possible to determine the exact type of the objects that will be assigned to it. Thus if we type the second instance of StringHistory as Object:

```
StringHistory stringHistory1 = new AStringHistory();
Object stringHistory2 = new AStringHistory();
stringHistory1.equals(stringHistory2);
```

then the more general equals()will be called and the result will be false. Even if we use StringHistory to type both variables, we can use a cast to call the equals() defined in Object:

```
StringHistory stringHistory1 = new AStringHistory();
StringHistory stringHistory2 = new AStringHistory();
stringHistory1.equals((Object)stringHistory2);
```

Overloading and IS-A

As we saw above, the is-a relation is used in overload resolution: if a call matches multiple overloaded methods, then the one that declared more specific formal arguments is used. In the example above, the formal argument of the equals() in AStringHistory is StringHistory, which is more specific than StringHistory.

To better understand the relationship between overloading and IS-A, let us consider some more examples.

Suppose, we define in AStringSet an overriding equals method with the following signature:

public boolean equals (Object otherObject);

and defined in AStringHistory an overloaded equals method with the following signature:

public boolean equals (StringHistory otherObject);

and executed the following code:

```
StringSet stringSet1 = new AStringSet();
StringSet stringSet2 = new AStringSet();
stringSet1.equals(stringSet);
```

It may seem a bit ambiguous as to which equals() is called in the third statement. The overriding one in AStringSet is in the more specific class, while the overloaded one in AStringHistory has the more specific argument. The one is the more specific class, however, is a different method, even though it has the same name. Therefore, Java will choose the method with the more specific argument types, regardless of which class it is defined in.

As another example, let us consider the following two variations of equals, which could be defined in arbitrary classes:

```
public boolean equals (StringHistory stringHistory1,
                       Object stringHistory2) {
    …
}
public boolean equals (Object stringHistory1,
                       StringHistory stringHistory2) {
    …
}
```

In these implementations, the two instances that are to be compared are both passed as arguments. Hence it does not matter on what objects these methods are actually called (which means they should really be static methods).

Now suppose we make the call:

```
equals(new AStringSet(), new AStringSet());
```

Both overloaded methods can accept the two arguments – which one should be called? Neither class has formal parameters that are of more specific types. Thus, the correct call is ambiguous, and Java will say so at compile time. We can, of course, use casts to disambiguate:

```
equals(new AStringSet(),  (Object) new AStringSet());
```

In this case the first equals() is called.

If we add a third equals method:

```
public boolean equals (StringHistory stringHistory1,
                       StringHistory stringHistory2) {
        …
}
```

Then the call:

```
equals(new AStringSet(), new AStringSet());
```

invokes this new overloaded method as each of its formal parameters is more specific than the corresponding formal parameter of the two other methods. Thus, interesting, adding an overloaded method reduces rather than increases the ambiguity of overload resolution!

Given these three definitions of equals, which one is invoked by the following call?

```
equals(null, null);
```

null can be typed as any object type, so it may seems that the call is still ambiguous. However, as the types of the formal parameters of the third equals () are most specific, there is really no ambiguity, and this method is called.

Suppose we add a fourth equals(), one that takes Point arguments:

```
public boolean equals (Point point1, Point point2) {
        …
}
```

Now the call:

```
equals(null, null);
```

is indeed ambiguous as none of the matching equals() declared more specific types. We can resolve the ambiguity by explicitly casting null:

```
equals ((StringHistory)null, (StringHistory) null);
```

Annotations and Making Override Explicit

As overloaded and overriding can lead to different results, it is important to not accidentally overload when we intended to override, and vice versa. To prevent such mistakes, Java supports override annotations illustrated below:

```
@Override
public boolean equals(Object otherObject) { … }
```

Annotation is a typed "comment" about a method, class, or package that can be processed by some tool such as compiler, Eclipse or ObjectEditor at compile/execution time. It starts with the character, @, as illustrated below. Java supports several types of annotations. The override annotation type is associated with a method and tells Java that the method overrides an existing method. If the method does not in fact do so, the compiler will give an error. In the above example, since the method signature matches that of the equals() method of Object, no error will be given. Similarly, the following annotated method definition in AStringSet will give no error:

```
@Override
public void addElement(String element) { … }
```

On the other hand, the following method declaration in AStringHistory will give an error as it overloads rather than overrides the Object equals() method:

```
@Override
public boolean equals(StringHistory otherStringHistory) { … }
```

Interestingly, the following annotated method declaration in AStringHistory does not give an error:

```
  @Override
  public String elementAt (int index) {  return contents[index]; }
```

Recall that AStringHistory is a direct subclass of Object, which does not define an elementAt() method. In the definition of the override annotation, Java does not distinguish between IS-A and inheritance. An override annotation for method M in class C indicates that M is a(n) (re)implementation of some M declared in some type T, where C IS-A T. In other words, Java considers the implementation of an interface method also as overriding. This is inconsistent will all definitions of overriding in the literature.

Java allows programmers to define their own annotation types to be processed by tools written by them. This extensibility is used in the design of ObjectEditor. For example, the following annotated interface declaration tells ObjectEditor to not consider the interface as an atomic shape type even though it follows all the rules of a point type by having **int** X and Y properties and including the substring Point in its name:

```
public interface NotAPoint {
        int getX();
        int getY();
        …
}
```

As the declaration shows, an annotation can take arguments. In Java, an annotation is actually represented by an interface or class. The arguments to the annotation are passed to a special method defined by the interface or class.

Shallow Copy

Let us finish our exploration of the class Object by looking finally at the method clone(). To motivate it, let us return to the Point example.

```
p1 = new ACartesianPoint(200, 200);
p2 = p1;
p1.setX (100);
System.out.println(p2.getX() == p1.getX());
```

As we saw before, the code above will print **true**. The reason is that assignment simply copies pointers. What if did not want p1 and p2 to share the same object? Instead, we wanted the initial value of *p2*to be a *copy* of the object referenced by *p1;* and later wanted to change the copy without affecting the original object? We might want to do so because we want a backup of *p1* to which we would like to revert later.

Since assignment does not do the job for us, we can extract the information in ACartesianPoint and use this to create a new instance with the same state:

```
p1 = new ACartesianPoint(200, 200);
p2 = new ACartesianPoint (p1.getX(), p1.getY());
p1.setX (100);
System.out.println(p2.getX() == p1.getX());
```

This time the output will be false. However, this approach requires the copier to do the copying, which may not seem much work in this case, but would be more if the object to be copied had several instance variables. Therefore, a better approach is to make the copied object do the work of copying. That is, the copier should simply invoke a method on the object to be copied that returns a copy. The clone() method in Object is such a method. As its name indicates, it returns a copy of the object on which it is invoked. Since the method can be invoked on an object of any type, its return type is Object:

```
// defined in Object
public Object clone() { … }
```

This return type can be cast to the actual type, as shown below:

```
p1 = new ACartesianPoint(200, 200);
p2 = (Point) p1.clone();
p1.setX (100);
System.out.println(p2.getX() == p1.getX());
```

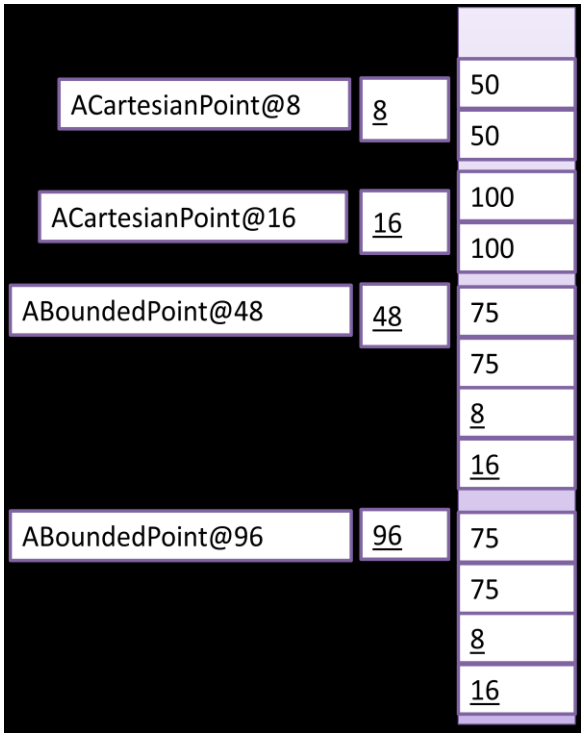
Again, the output is false, as clone() creates a new copy with separate instance variables.

To better understand the notion of copying, let us consider instances of a more complicated type. Consider the following code:

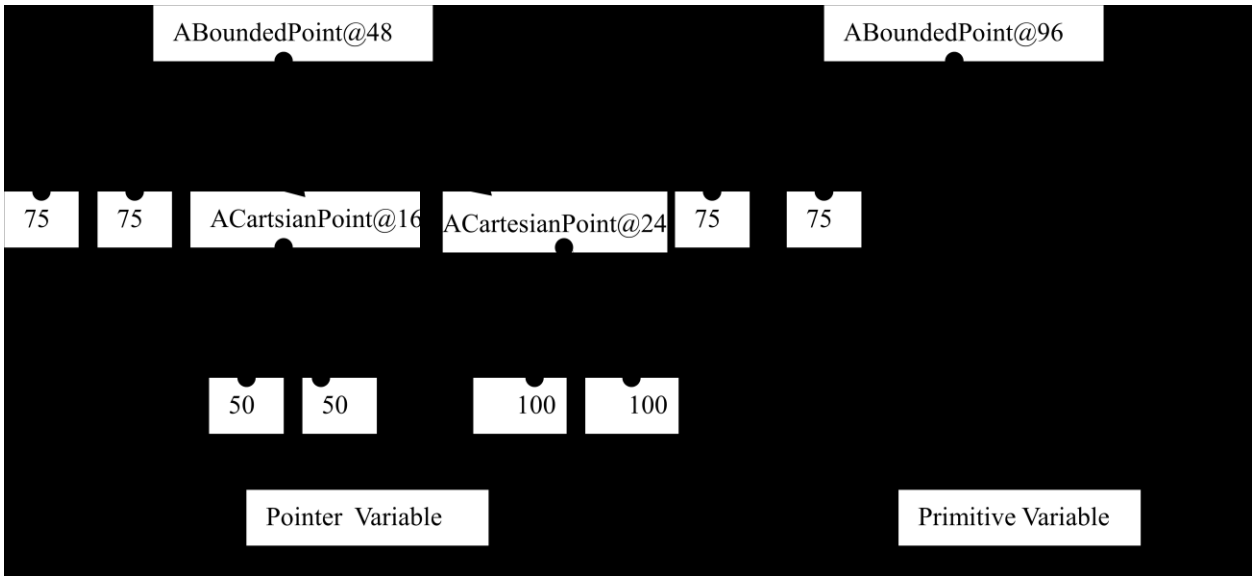
```
p1 = new ABoundedPoint(75, 75,  new ACartesianPoint(50,50), new ACartesianPoint(100,100) );
p2 = (BoundedPoint) p1.clone();
p1.setX (100);
p1.getUpperLeftCorner().setX(200);
System.out.println(p2.getX() == p1.getX());
```

System.out.println (p1.getUpperLeftCorner().getX() == p2.getUpperLeftCorner().getX());

The first output will print false but the second one will, in fact, print true. The reason is that Object implements clone() by simply making a copy of the memory block of the copied object, as shown in the figure below:



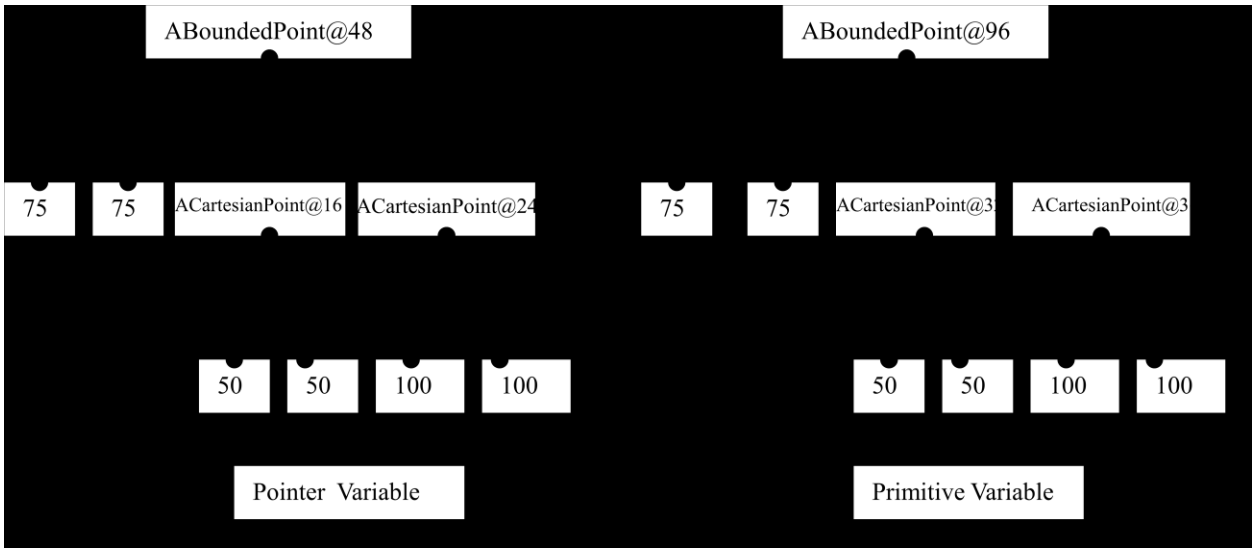
Here, the clone() method, when executed on the object, ABoundedPoint@48, creates a new object, ABoundedPoint@48, whose memory block is a copy of the memory block of the first object. The two primitive instance variables, x and y, of the new copy are assigned copies of the values of the x and y variables of the original object. The two pointer variables in the copy, however are the assigned copies of the *pointers* to the (Point) objects referenced by the upperLeftCorner and lowerRightCorner of the original object. These subobjects of the original object are not themselves copied. This means that object variables in the copy point to the same objects as the corresponding variables in the original object. The second println() call returns true as p1.getUpperLeftCorner() and p2. getUpperLeftCorner() refer to the same object, ACartesianPoint@8. This is shown graphically in the figure below:



Each line in the picture represents an instance variable of the object. A copy that simply duplicates the memory block of the copied object is called a *shallow copy*. The word shallow indicates that it copies only the top level of the physical structure of the object.

Deep Copy

A copy that also copies memory blocks of components of the copied object is called a *deep copy.* The following figure illustrates deep copy of our example object:



Such a copy is not provided by Java, so we must override the clone() method to implement it:

```
public Object clone() {
        return new ABoundedPoint (x, y, (Point) upperLeftCorner.clone(),
                                        (Point) lowerRightCorner.clone());
```

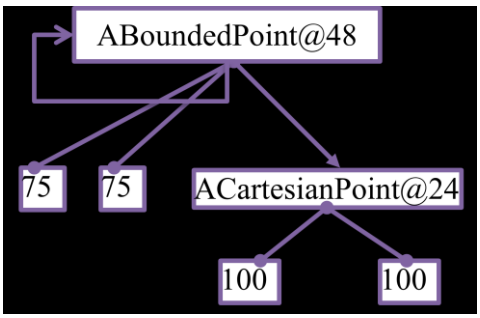
};

Here we construct a new instance of ABoundedPoint that has a copy of the objects to which the upperLeftCorner and lowerRightCorner instance variables point.

This copy, however, does not always work. The problem is illustrated by the example below:

```
p1 = new ABoundedPoint(75, 75,  new ACartesianPoint(50,50),  new ACartesianPoint(100,100) );
p1.setUpperLeftCorner(p1);
p2 = (BoundedPoint) p1.clone();
```

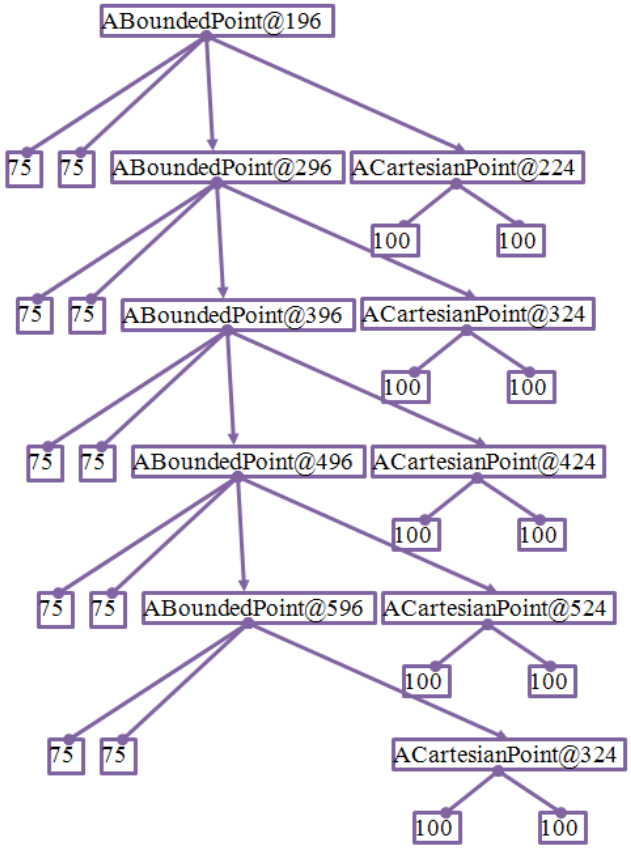
Here, we have created a recursive structure, that is, a structure in which a child component points to its ancestor:



As a result, when the clone method is invoked on this object, the following call leads to an infinite recursion:

```
(Point) upperLeftCorner.clone()
```

Each time the call is made, a new copy of ABoundedPoint@48 is made, and the call is made again, which makes another copy of the object, and so on, leading to the creation of an infinite number of copies of the object, some of which are shown below:



Thus, we must be careful in either how we implement deep copy or the kind of objects on which we try to invoke this operation. Recursive structures cannot be banned as they are very useful. A class of problems called graph problems require such structures.

The Java Object class avoids this problem in its implementation of clone() by providing a shallow copy.

Supporting Multiple Copy Operations

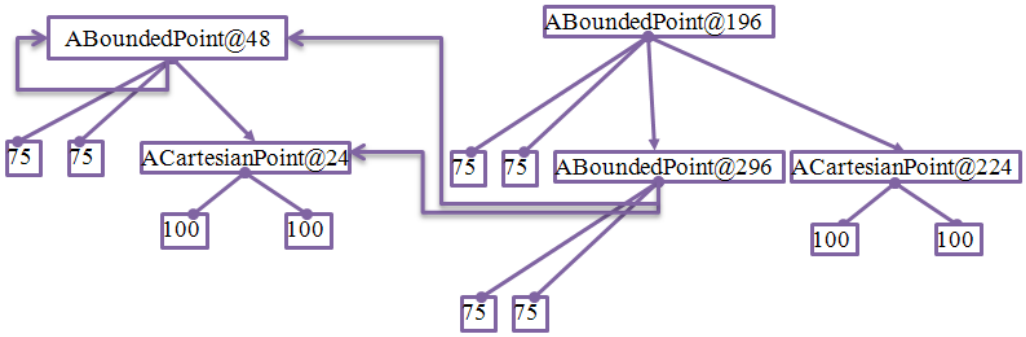
One way to reduce the problem of copying recursive structures is to associate each object with *both* a shallow and deep copy, and allow the deep copy to determine if shallow or deep copies of components are made. This solution is illustrated below for our example class, *ABoundedPoint*:

```
public Object shallowCopy() {
        return new ABoundedPoint (x, y, upperLeftCorner, lowerRightCorner);
};

public Object deepCopy() {
        return new ABoundedPoint (x, y, (Point) upperLeftCorner.shallowCopy(),
                                        (Point) lowerRightCorner.shallowCopy());
};
```

The “deep” copy in this solution is not a full deep copy as it does not copy all levels in the physical structure of the copied object – it is simply a deeper copy than a shallow copy. However, a call to it will never lead to infinite recursion.

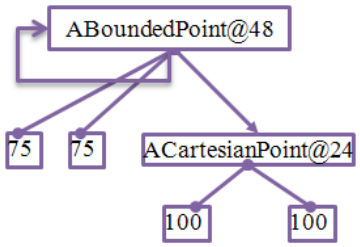
An extension of the approach of providing both a shallow and deep(er) copy is taken in Smalltalk. Each Smalltalk object provides three copy methods: shallow, deep, and regular copy. The shallow copy is like the Object clone() method: It creates a new object and assigns instance variables of the copied object to corresponding instance variables of new object. The deep copy creates a new object and assigns a *regular copy* of each instance variable of the copied object to corresponding instance variable of new object. The semantics of the regular copy of an object of some class C is defined by that class. It is expected to either be a shallow or deep copy – the programmer of each class defines which of these two choices is taken. By default, the regular copy is a shallow copy. In our cyclic example, if the regular copy is a shallow copy, then a deep copy of the cyclic structure on the left would result in the structure on the right:

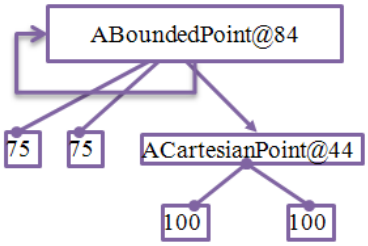


Here the deep copy creates a new instance of ABoundedPoint (ABoundedPoint@196), and then does a copy of the two object pointers in it. As this copy is a shallow copy, we get new instance of ABoundedPoint (ABoundedPoint@296) and a new instance of ACartesianPoint (ACartesianPOint@224) whose memory content are copies of the objects representing the upper left corner and lower right corner of the original object. This results in the two object pointers in ABoundedPoint@296 pointing back to the original object.

Detecting Recursion

The multiple copy solution does not, of course, work when a full deep copy is needed. In this situation, recursive structures can be handled by detecting recursion while performing the copy operation, that is, before copying a component, detecting if the component has already been copied in the operation. When recursion is detected, we can either not copy the component, or give an error, or create an identical or isomorphic structure. In the example above, we could create an isomorphic structure y creating another instance of a BoundedPoint whose upper left corner points to it, as shown below.





While the Java clone method supports shallow copy, called serialization, which makes object copies that are written to files or sent across the network, supports such isomorphic copies.

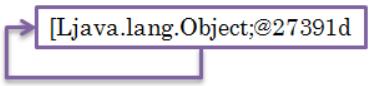
Why no Recursive Print?

The fact that recursive structures can be created is probably the reason that Java println() does not print the elements of arrays, instead simply displaying the id of the array and the type of the elements of the array:

```
[Ljava.lang.Object;@27391d
```

It is possible to use an array to create a recursive structure:

```
Object[] recursive = new Object[1];
recursive[0] = recursive;
```



A println() that recursively printed each element of a recursive array such as the one above would recurse forever.

ObjectEditor faces a similar problem when creating a widget structure for the logical structure of an object. The current version detects recursion and does not create a widget for a component for which it has already created a widget earlier.

Other Object Methods

The discussion above helps us better understand three operations provided by class Object: toString(), equals(), and clone(). This class provides several other methods:

- hashCode():This is relevant to hashtables, which you will learn in depth in data structures. Think of a hashCode as the internal address of object.
- Various versions of wait() and notify():These are relevant to threads – you will study them in depth in an operating systems course.

- getClass(): This method returns the class of an object, on which one can invoke “reflection” methods, which are beyond the scope of most undergrad courses. These methods allow one to determine and invoke the methods of a class. ObjectEditor uses these methods.
- Finalize(): This method is called when the object is garbage collected, discussed below.

Objects and Interfaces

As we have seen above, it is possible to invoke an Object method on any object variable. If the variable is typed by a class, this makes sense, as it is possible to invoke any method declared in the class or its super class chain; and Object is the last type in any superclass chain. However, if the variable is typed by an interface, this rule does not make sense, as Object is a class and not an interface, and thus cannot be on the supertype chain of any interface. Yet, if we type a variable by an interface:

```
StringHistory stringHistory1, stringHistory2;
```

we can indeed call Object methods on that variable:

```
stringHistory1.equals(stringHistory2);
```

The reason this is legal is that Java uses a special rule to allow all interfaces to “inherit” Object methods. The cleaner solution would have been to associate Object with an interface, and make this interface the last interface in the super type chain of all interfaces. Unfortunately, the designers of the language and libraries have had a schizophrenic attitude towards interfaces, using them in some situation and not in others. Had they followed our rule of making every class implement one or more interfaces, we would not have this fundamental problem with Object methods.

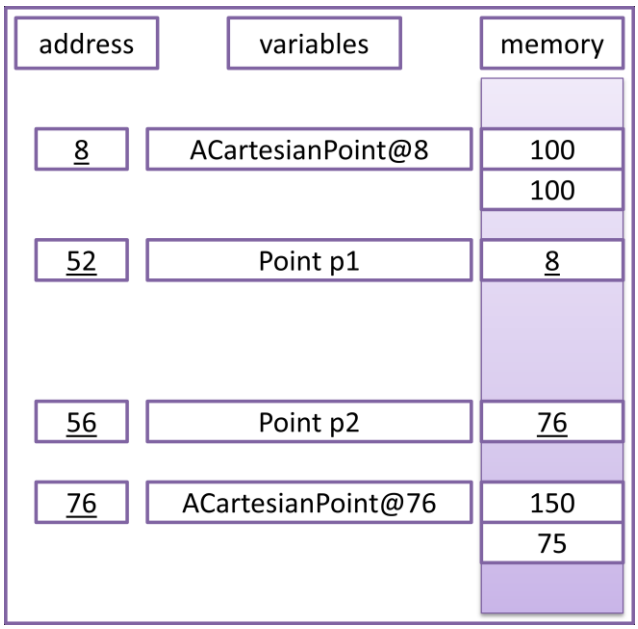
Garbage Collection

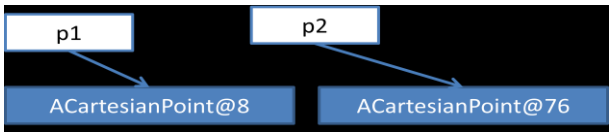
Suppose we execute the following code:

```
Point p1   = new  ACartesianPoint(100,100);

Point p2   = new  ACartesianPoint(150,75);
```

Two new variables are created, which point to different objects, as shown in the figures below:

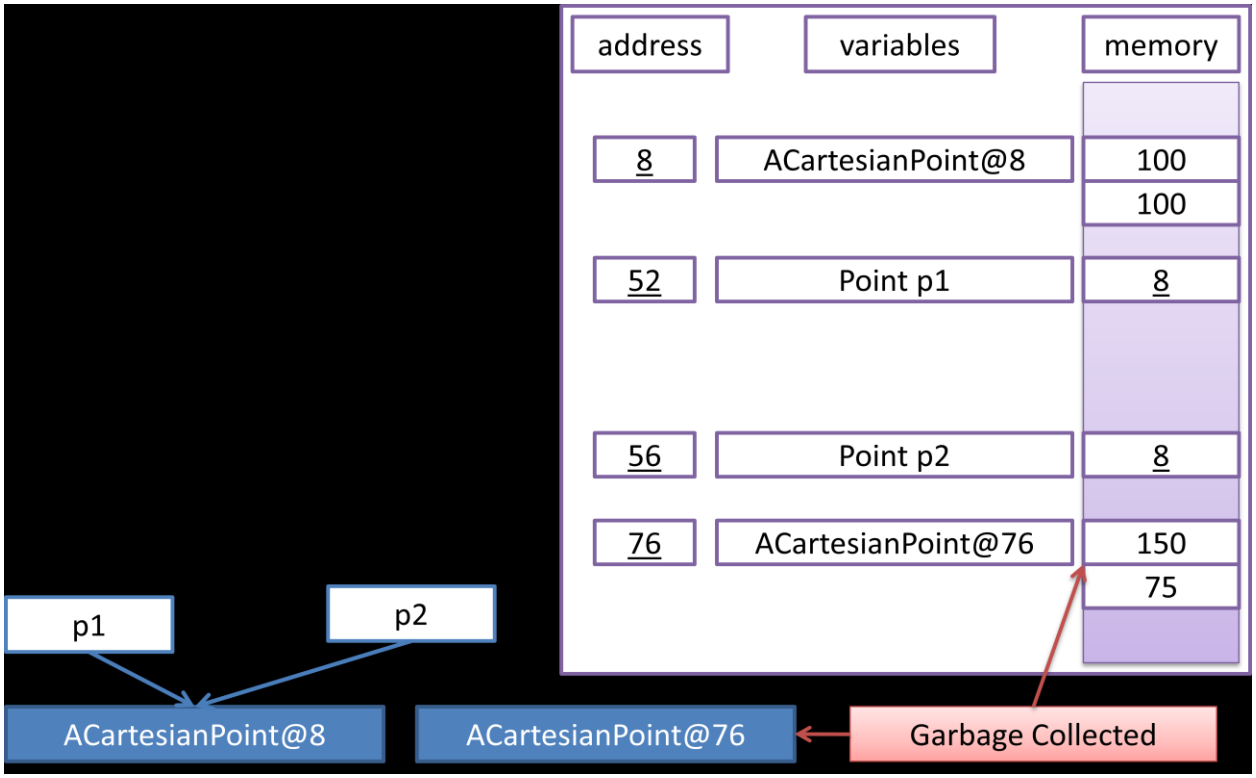




What if we now execute the following code:

p2 = p1;

Now both variables point to the same object, and the object to which p2 pointed cannot ever be accessed. This is shown below:

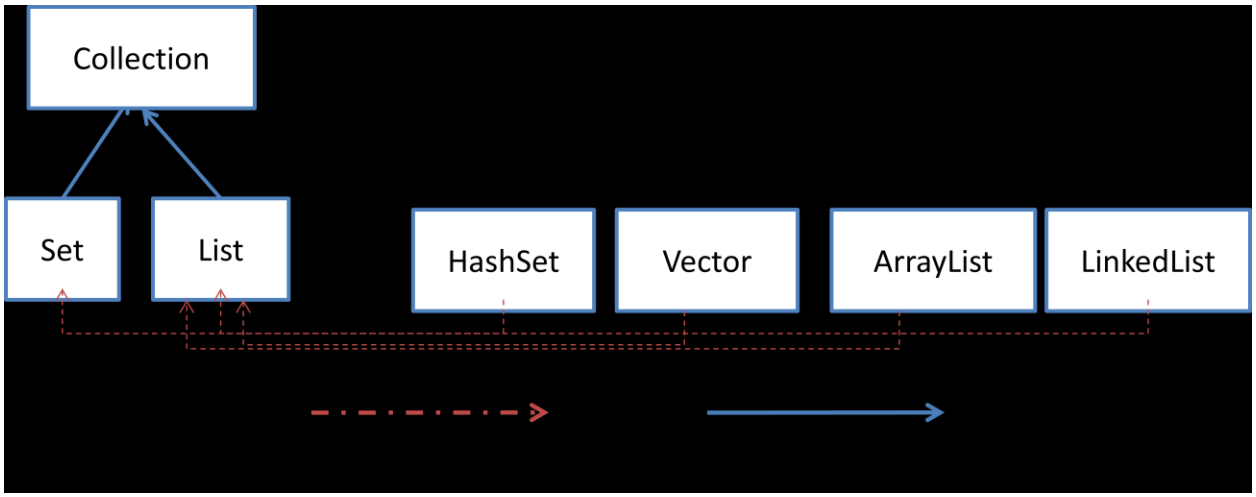


As no variable refers to the object, it is *garbage collected.* With each object, Java keeps a count, called a *reference count*, that tracks how many object variables store pointers to it. When this count goes to zero, it collects the object as garbage, since no other variable will ever be able to point to it again.

Automatic garbage collection is a really nice feature of Java since in most traditional languages such as C, the programmer is responsible for deleting objects. In such languages, the danger is that we may accidentally delete something that is being used, thereby creating *dangling pointers* to it, or forget to delete something that is not being used, thereby creating a *memory leak* that keeps wasting memory. On the other hand, garbage collection makes Java slower, since Java must interrupt our program execution to do garbage collection.

Collection Inheritance in Java

In this chapter, we have seen how inheritance can be used while defining collection classes. The Java API also addresses this issue. The hierarchy it creates has some similarity with the one we created. Part of the hierarchy is given below:



It defines two interfaces, Set and List, which are both subtypes of Collection. This is analogous to StringDatabase and StringSet being subtypes of StringHistory. Vector, ArrayList, and LinkedList provide three different implements of the List interface. In earlier versions, Vector did not implement any interface. When Java programmers felt the need for additional implementations with the same functionality, they created the List interface. Both Vector and ArrayList use arrays to create a variable-sized collection – the main difference is in how they grow when new elements are added. LinkedList uses a different data structure, called a linked list, which you will study later. Similarly, there are a variety of classes that implement the Set interface, we see only HashSet in this figure.

Summary

- Java allows classes and interfaces to inherit declarations in existing classes and methods, adding only the definitions needed to extend the latter.
- An inherited method can be overridden by a new method.
- 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.
- All variables of an object, including its inherited variables, are stored in a single memory block.
- Overriding != overloading
- If two overloaded method match a call, the one with more specific parameter types is chosen.
- Casts may be needed to disambiguate between overloaded methods.
- equals() != ==
- Copies can be shallow or deep depending on whether components of the copied object are themselves copied or not.
- Recursive structures interfere with deep copy.
- Unreferenced objects are garbage collected.

Exercises