COMP 210 - Lecture Notes - 05 - Class Extension

January 31, 2016

In these notes we address class extension and implementation inheritance. In doing so we'll learn about *abstract* and *final* classes in Java and the use of the *protected* and default access modifiers.

Shared Implementation and Class Extension

When designing and implementing class unions it is not uncommon to find that several variants in the union have some shared implementation. For example, all the classes in our *Shape* union from lecture notes 4 had a *Loc* field. What's more the *distanceTo* and *moveTo* methods were dependent only on the location field and therefore looked the same in each variant in the union. This kind of repetition is a sure sign that our design could be improved by some abstraction.

In some sense the use of a contained object, the Loc, protects us from some repetition. If the core logic for the repeated methods is done by Loc, then the implementation of those methods at the Shape level can be a simple call to the appropriate Loc method. This also means that changes to the core implementation of that method need only take place in a single location, the implementation within *Loc*. These points are all well and good, but they don't prevent the repetition of code across the hierarchy.

The real fix to this problem comes from a new kind of Class relationship: Class Extension. When one class extends another then a hierarchy is established in the same fashion as interface implementation. The extender, called the subclass is a specific variant of the extendee, the superclass. Increased specification comes from overriding behavior or adding functionality and not through hiding or restricting existing implementation. In the case of shapes, we'd like to extract all the pin location details up into a more generic *AbstractShape* so that we can then extend that class with the elements unique to each shape.

The AbstractShape class is halfway between an interface and a concrete class like Rectangle. It captures concrete implementation details, i.e. that all shapes contain a pin location and *distanceTo* and *moveTo* can be completely specified in this context, and in that way is like the concrete shape classes. On the other hand, an AbstractShape is a less specified, more generic type we can associate with our class union.[1].

[1] The AbstractShape type is the union of Dot, Rectangle, and Circle

The Old Shape Hierarchy

Recall our previous design as diagrammed in figure 1. Here we've repeated class method declarations to indicate the point of actual implementation. We see evidence of shared implementation in this design by the repeated containment relationship with *Loc* and the repetition of location related functionality. The repetition of the interface declarations is not necessarily evidence of shared implementation[2]. Only if the implementation of an interface method is based solely on shared implementation will we be able to abstract it out of the concrete classes.

[2] it obviously means shared behavior!

A New Shape Hierarchy

If we go and lift out all of the shared implementation from the Shape hierarchy and place it in the *AbstractShape* abstract class, then we'd end up with the design diagrammed in figure 2. Once again, *Shape* interface methods are restated in the class in which they are implemented. With that in mind, we can see that some of the interface methods can be implemented in the abstract class and some cannot. This lack of a complete implementation of the interface is precisely what makes the class abstract in this case.

Abstract Methods and Classes in Java

The keyword *abstract* can be applied to methods and Classes in Java. An abstract method is a method without an implementation, it's simply a declaration. All the methods declared in an Interface are implicitly *abstract* and public. Figure 3 shows how you could rewrite the declaration of the *area* method to include these implicit keyword[3].

[3] as a matter of style, we don't do this

An abstract class is a class with abstract methods. That's it. These methods can either be introduced by the class or, as is the case in our example, the abstract methods are interface methods that still lack an implementation. If you're working from the perspective of lifting out shared implementation, then it's likely you'll end up with an abstract class. An important feature of abstract classes is that you cannot directly construct objects of that type. It's a compile time error. The reason for this is simple. As an abstract class its implementation is necessarily incomplete. Some of the methods are abstract. This means the compiler cannot guarantee that a all the methods in that class can be executed. Just like procedural C++ code can't compile without definitions for declared procedures, Java code will not compile if you attempt to instantiate an object from a class with undefined or abstract methods.

To declare a class abstract simply tack in the *abstract* keyword as

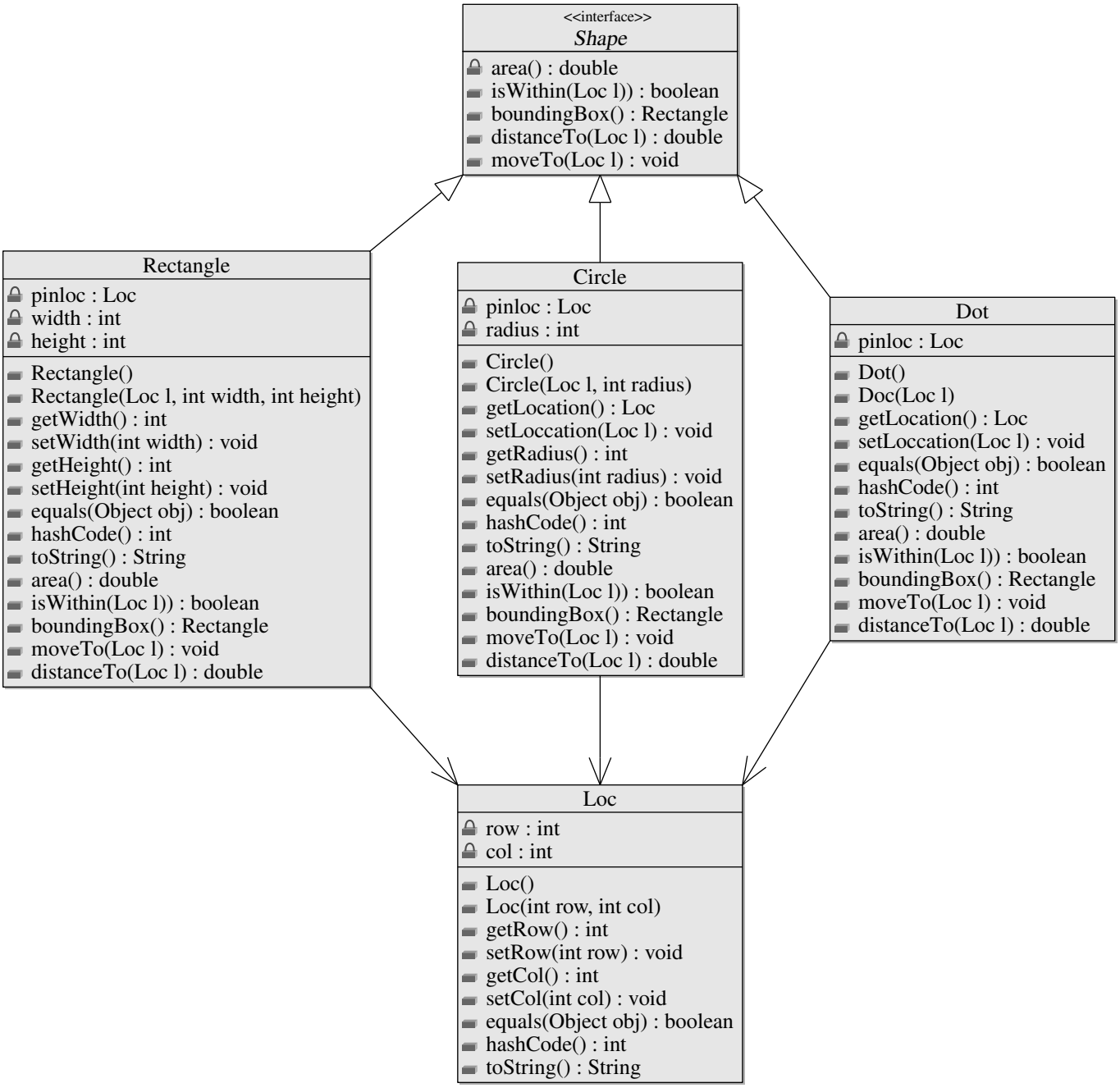

Figure 1: Shapes Class Hierarchy

shown in figure 4. Eclipse has a checkbox for *abstract* in the New Class wizard.

Finally, to indicate class extension we use the *extends* keyword prior to the interface declarations. Figure 5 shows the new Circle class definition header line. Technically, the interfaces of the super-

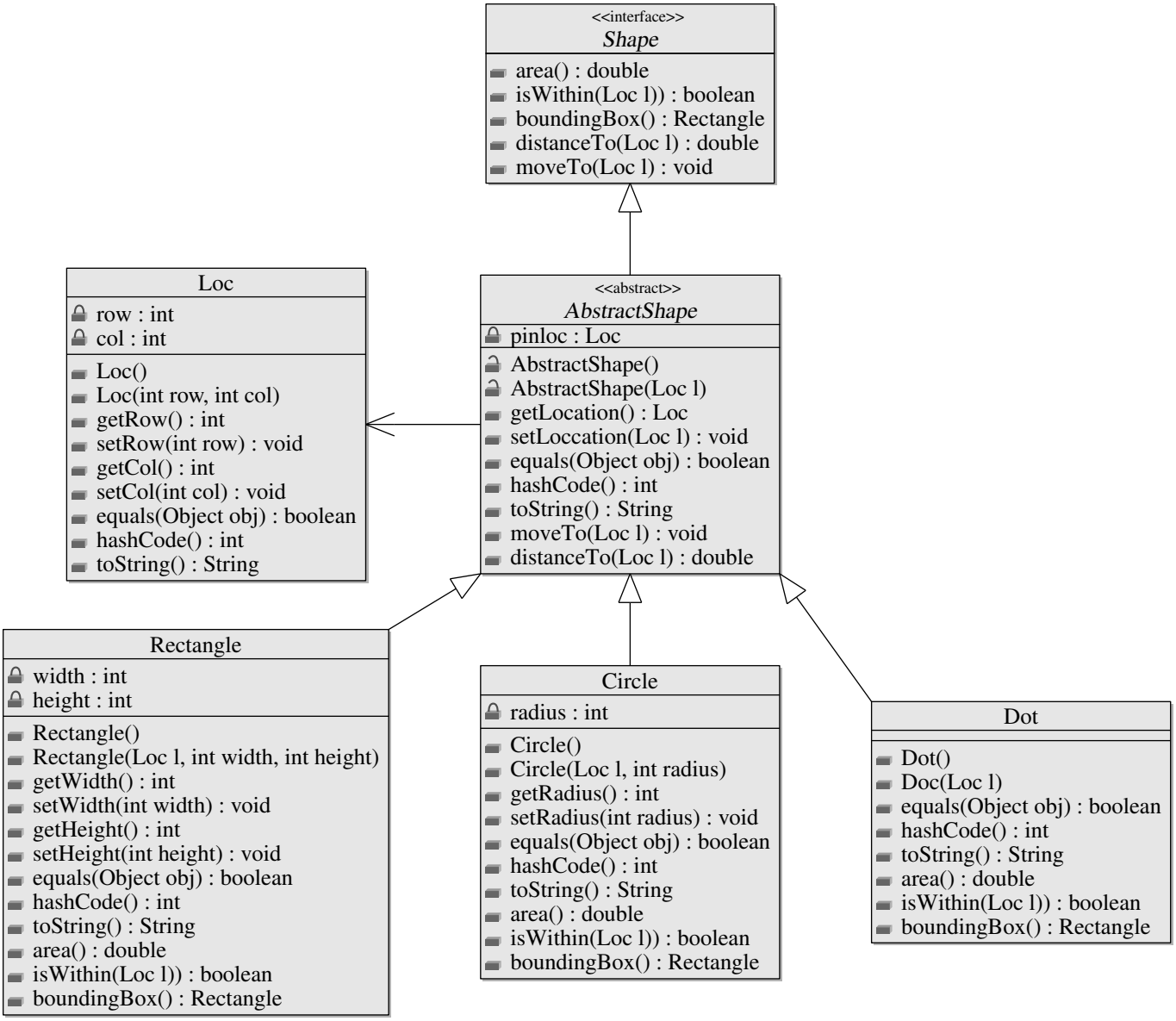

Figure 2: Shapes with an Abstract Class

```
//in Shape.java
abstract public double area();
```

Figure 3: Complete declaration of *area* in *Shape*

class are inherited by the subclass, so the redeclaration of Shape isn't needed, but OK if you need to remind yourself its there.

```
//in AbstractShape.java
public abstract class AbstractShape implements Shape{
  // Field and Method declaration will go here.
}
```

Figure 4: Stub Declaration of the *AbstractShape* abstract class

```
//in Circle.java
public class Circle extends AbstractShape implements Shape{
  // Field and Method declaration will go here.
}
```

Figure 5: New header line for the *Circle* class

Access Modifiers and Class Extension

As we start using abstract classes and inheritance hierarchies involving class extensions, we'll need to think more closely about the access modifiers. Thus far *public* has been used for the the user-facing parts of our design. The stuff that we need to solve our problem. The *private* modifier has been used to hide implementation details like object data fields and occasionally helper methods.

These basic permissions still apply between subclass and superclass. The private elements of a superclass are not accessible to the subclass or anywhere else. Public methods and fields in a supper class are not only accessible within the subclass but can be access outside of the hierarchy through the subclass. So while *AbstractShape* will contain the implementation of *distanceTo*, we can still invoke that method from an object of type Circle. After all, all subclass are also instances of their superclass with added functionality.

```
Circle acirc = new Circle(new Loc(5,2),4,3);
// even though there is no listed
... acirc.distanceTo(new Loc()) ...
```

Figure 6: Superclass public method invocation from subclass objects

Essentially what happens is when a method is invoked on an object, the runtime system first checks the object's class for an implementation. If none is found, the superclass is checked and so on up the hierarchy. Eventually an implementation will be found. To see why, consider the fact that you cannot instantiate an object if it's class is abstract and that any class with unimplemented methods must either declare those methods abstract or implement them before the code can compile. So, the only way to compile and instantiate an instance of a class is if you're dealing with a concrete class in which

all declared methods are implemented. While its not always easy to predict which implementation gets executed, it is guaranteed that an implementation exists.

There is a middle ground between *public* and *private*. If you'd like a method or field to be accessible within a subclass but not outside of the hierarchy, then it should be declared as PROTECTED. In UML this is indicated with a # symbol[4]. Declaring protected access is helpful when fields and methods are needed for subclass implementation and only subclass implementation. In this case, public would open up the fields and methods for access outside the implementation code and private would restrict the fields and methods to the class in which their declared. Protected is exactly the access you'd need. We'll always declare the abstract class constructors as protected because the only place you're allowed to use them is the direct subclass anyway.

[4] which shows up as a partially open lock in the diagram

The super keyword

If, within a class method, you wish to refer to the object on which the method was invoked, then you use the keyword *this*. If instead you wish to refer directly to a public or protected field or method from that class' superclass, then you use the keyword *super*. The most immediate application of this is in constructors. Notice the construct for the Circle class as shown in figure 7.

```
public Circle(Loc center, int radius) {
  super(center);
  this.radius = radius;
}
```

Figure 7: Circle construct that invokes the AbstractShape constructor through *super*

The center location is a property of the abstract super class and is therefore initialized by that construct. We can invoke this through *super(...)*. A more this-like invocation of *super* can be seen in the Eclipse generated *hashCode* for the Shape subclasses as shown in figure 8.

```
public int hashCode() {
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + radius;
  return result;
}
```

Figure 8: Circle hashCode implementation that invokes super class hashCode constructor through *super*

Notice that rather than initialize *result* to 1, it's initialized to the hash code of the superClass. This captures the hash of the center location and leaves only the radius left to account for. Most importantly, this exposes an important idiom for dealing with superclass. Just like we let contained classes manage their own data, we let superclasses manage their own. You see this again in *equals* as shown in figure 9.

Figure 9: Circle equals that incorporates proper handling of super class data

```
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  // compare at the superclass level
  if (!super.equals(obj)) {
    return false;
  }
  // if you get this far, then obj
  // must be an AbstractShape pinned
  // to the same place as this

  if (getClass() != obj.getClass()) {
    return false;
  }
  Circle other = (Circle) obj;
  if (radius != other.radius) {
    return false;
  }
  return true;
}
```

When superclass data is private and you wish to incorporate it in your toString output, then you'll need to explore the *Inherited Methods* menu and select an appropriate constructor as shown in figure 10.

Figure 10: Circle toString that uses the public super accessor

```
public String toString() {
  StringBuilder builder = new StringBuilder();
  builder.append("Circle [radius=");
  builder.append(radius);
  builder.append(", center=");
  // get the location of this actually drops
  // down the getLocation definied in super
  builder.append(getLocation());
  builder.append("]");
  return builder.toString();
}
```

Testing and AbstractClasses

The good news is that because you cannot directly instantiation objects from your abstract classes then you cannot directly test the class. That's also the bad news. Public methods from the abstract class should be tested in its subclasses. Protected and private methods should be tested implicitly through the subclass public methods. Basically the usual rule of thumb applies: *test the public facing interface*.