COMP 210
Lecture Notes 04
Basic Classes and Hierarchies

January 24, 2017

These notes are more or less a reorganization and retelling of content from *How to Design Classes* chapters 1-4 and 10-14. Through a problem with geometric shapes we'll explore the design and implementation of OO code using Object Containment and Class Unions.

Some 2D Geometry

Let's think about the following problem[1]:

> You've been tasked to develop some libraries for managing basic shapes laid out on a graphical canvas. The canvas is just a whole-valued, positive only coordinate system addressed in row then column order with the origin, $(0, 0)$ in the upper left hand corner.[2]. For starters, your library should support three shapes: *Circles*, *Rectangles*, and *Dots*. All shapes have an associated location. For circles that location is the center of the circle. For rectangles, that location is the upper left hand corner of the rectangle. For dots, that location is, essentially, the dot. Circles also have an associated radius. Similarly, rectangles have a length and width where length is the number of rows covered by the rectangle and width is the number of columns. All shapes must have the following functionality: compute and return its area, determine if a given point is within the bounds of that shape, compute the distance of the shapes "pin" location to a given location, modify the location of the shape, and compute and return the bounding box of a shape as a Rectangle.

[1] See page 27 and 155 in HtDC

[2] The standard coordinate system for 2D computer graphics

The Objects for Consideration

The first design question to ask yourself is, "What concrete objects are implied by the problem description, what are their classes, and how are the represented?" You need to establish the types/values/objects that will make up the problem domain so that problem funcitonality can be attributed to the most appropriate type. In this problem we see 3 concrete classes:

1. Dots which are represented by a location
2. Rectangles which are represented by a location, width, and length
3. Circles which are represented by a location and radius

There is another logically implied class as well:

1. 2D Canvas locations, i.e. (row,column) addresses for shape pin locations

If each shape type *has a*n associated location then we're dealing with OBJECT CONTAINMENT. You might be tempted to drop the location class all together and simply use a row and column field in each shape type. You'd be wrong. For starters, there is little reason to flatten the problem logic like this. You're likely to think and talk about "locations" and "points" and there's not reason to not reflect that thinking in your design. Additionally, encapsulating location specific functionality and logic, creates code independence between the shapes and the implementation of their locations. *As long as the interface to a 2D point never changes, you're free to tinker around with the implementation of some or all of it's behaviors without breaking the Shape's own implementation.*[3]. Finally, by moving location logic into a separate class we get to reuse code. If every Shape does it's own location logic, then you'll end up repeating some logic verbatim within each shape class.

[3] this is referred to as *Separation of Concerns*

Some of these things are a lot like the others

Dots, rectangles, and circles are not unrelated. In fact, we keep referring to them generically as *shapes*. A shape is not an object type in the same sense as a Rectangle. The later is tangible, concrete, while the former is abstract and conceptual. We can understand and think in terms of the Shape abstraction, but have not yet seen how to encode this abstract relationship into our programs.

If a Shape isn't a concrete object, then there can't be a Shape class. What shape really describes is the CLASS UNION[4] of the Dot, Rectangle, and Circle classes. Any object of type Dot, Rectangle, or Circle should also be viewed as an instance of the Shape type. The question we must ask now is, "what makes a shape a shape?"

[4] union here is the same union discussed in the mathematics of sets

Once again we turn to behavioral abstraction. We'll define shapes not by specific data, or physical properties by by behavior. In programming terms, we're talking about a COMMON PUBLIC **interface**. The old saying, "walks like a duck, quacks like a duck, must be a duck" is appropriate here. It defines ducks by what they do, not by the presence of a beak, feather, webbed feet, etc.

Our problem statement clearly lists the expected behavior for shape objects:

- Shapes can compute their own area
- Shapes can determine if a given point falls within the shape
- Shapes can determine the distance from the shape's pin location to

another point

- Shapes can compute their own bounding boxes
- Shape can change their location

Notice that two of these behaviors could be restated as physical properties and envisioned as data fields, not methods:

- Shapes have an area
- Shapes have a bounding box

Both properties can be computed from other inescapable properties of a Shape[5], and so they are also secondary attributes. If they were fixed fields, then they'd need to be computed after the appropriate primary attributes were given. More importantly, committing to representation via data, making these fixed fields not computed values, means committing to a specific implementation. If area is something returned by a shape, then we have not committed ourselves to any one means of achieving that return value. This same principle underlies the usage of basic getters and setters. By restricting access to data fields to methods, to abstractions, then you're not committing to any one representation of that data.

[5] radius,width,height, etc

The combination of the Shape type with the concrete variants Dots, Rectangles, and Circles, is our first CLASS HIERARCHY. When a set of classes can be viewed as subsets of some larger, logical class, then we can create a CLASS UNION by identifying the expected behavioral interface for all members of that union. Defining types purely in terms of an interface[6] is an important mode of thinking to work in. It affords you maximum implementation flexibility by specifying what makes a type a type without committing to any interface. It gives rise to a whole school of OO design and analysis in which you *program to interfaces, not implementations*. Clearly you must eventually implement something, but recognizing the ability to first establish a concrete abstraction like the interface and then implement that interface in a variety of ways is game changer.

[6] the public methods

From Hierarchy Design to Java Implementation

A class union defined solely in terms of object behavior[7] is defined using a Java *interface*. An interface provides a series of method declarations and documentation. Methods in an interface must always be public and we therefore do not need to specify the access modifier. Concrete instances of that type, i.e. the subclasses of the union, then declare that they *implement* that interface. Classes can implement any

[7] methods

number of interfaces in Java. Once a class declares an interface, then it must implement that interface[8].

[8] In the next notes we'll see there is a way around this

Before we get to the coding, we'll sketch out a visual diagram of our Class hierarchy using **UML**[9]. This lets us write down everything but actual method implementation details. In doing so we see all the "what" details of our design without getting bogged down by the "how". This is nice in part because Java doesn't typically let us separate this like we did in C++[10]. It also let's you devise an implementation plan. Eclipse can do a lot of the coding grunt work for us, but we need to be able to direct it. It can help you implement a design, but I won't do the design work for you. By diagramming your hierarchy, you're writing down your ideas in such a way that doesn't commit you to any code.

[9] Unified Modeling Language

[10] Header + Implementation

In UML, all classes and interfaces are designated with a box. At the top of the box is the class/interface name. Interfaces are labeled as such to avoid confusion with concrete classes. Object containment relationships are diagrammed using an arrow with an arrowhead shaped point that points from container to containee. Subclasses of a class union have an arrow with a triangular point that points from subtype to supertype.

Figure 1 shows the basic structure of the Shape hierarchy design. An instance of the *Loc* class is contained within each of the Shape subclasses[11]. The Dot, Circle, and Rectangle classes form the Shape type when joined by a union. This means that they each inherit that type[12].

[11] Dot,Rectangle,and Circle each **has-a** Loc

[12] Dot, Circle, and Rectangle each **is-a** Shape

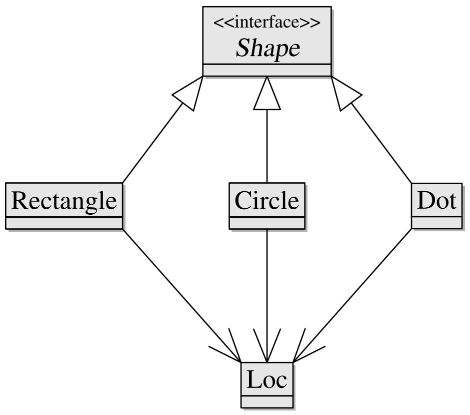


Figure 1: Class and Interface Structure for Shape Hierarchy

The class union relationship is then translated to Java through the **implements** keyword in the class header line. This is illustrated in figure 2.

All interface methods are declared under the interface name. Methods are declared with the following syntax:

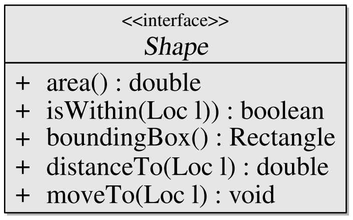
```
name(parameter list) : return type
```

```java
// In Shape.java
public interface Shape{
  // Method declarations here...
}

// In Rectangle.java
public class Rectangle implements Shape{
  // Data and Method Definitions here
}

// In Dot.java
public class Dot implements Shape{
  // Data and Method Definitions here
}

// In Circle.java
public class Circle implements Shape{
  // Data and Method Definitions here
}
```

Figure 2: Connecting Shape Subtypes to their Supertype

Public and private methods can be designated as such using + for public and − for private[13]. In figure 3 we see the complete UML diagram for the *Shape* interface.

[13] In the diagrams used in these notes you'll see an open or closed lock



<<interface>>
Shape

+ area() : double
+ isWithin(Loc l)) : boolean
+ boundingBox() : Rectangle
+ distanceTo(Loc l) : double
+ moveTo(Loc l) : void

Figure 3: The Shape Interface

For classes, we first list the data fields under the class name using the following syntax:

```
name : type
```

Once again, a + or − can be used to designate a field public versus private. We typically make all data fields private. Doing so forces a layer of abstraction between the abstract behavior, as represented by public methods, and concrete implementation, the private fields and methods. Class methods are listed below the fields with the same syntax as interface methods. However, we do not have to restate any methods inherited from a superclass or interface. We will, for the time being, restate the methods from *java.lang.Object* that we're overriding[14]. In addition to these methods we'll create two constructors, a default constructor and one to initialize all the fields, as well as

[14] equals,toString,hashCode

a suite of field accesssors and mutators. In practice you can design whatever interface you need for you class. The style we're using is pretty standard. So much so that eclipse will write most of it for you.

Figure 4 provides the complete UML diagram for our Shape hierarchy. Take a moment to give it a scan. Look at the classes individually, then examine the containment and union relationships.

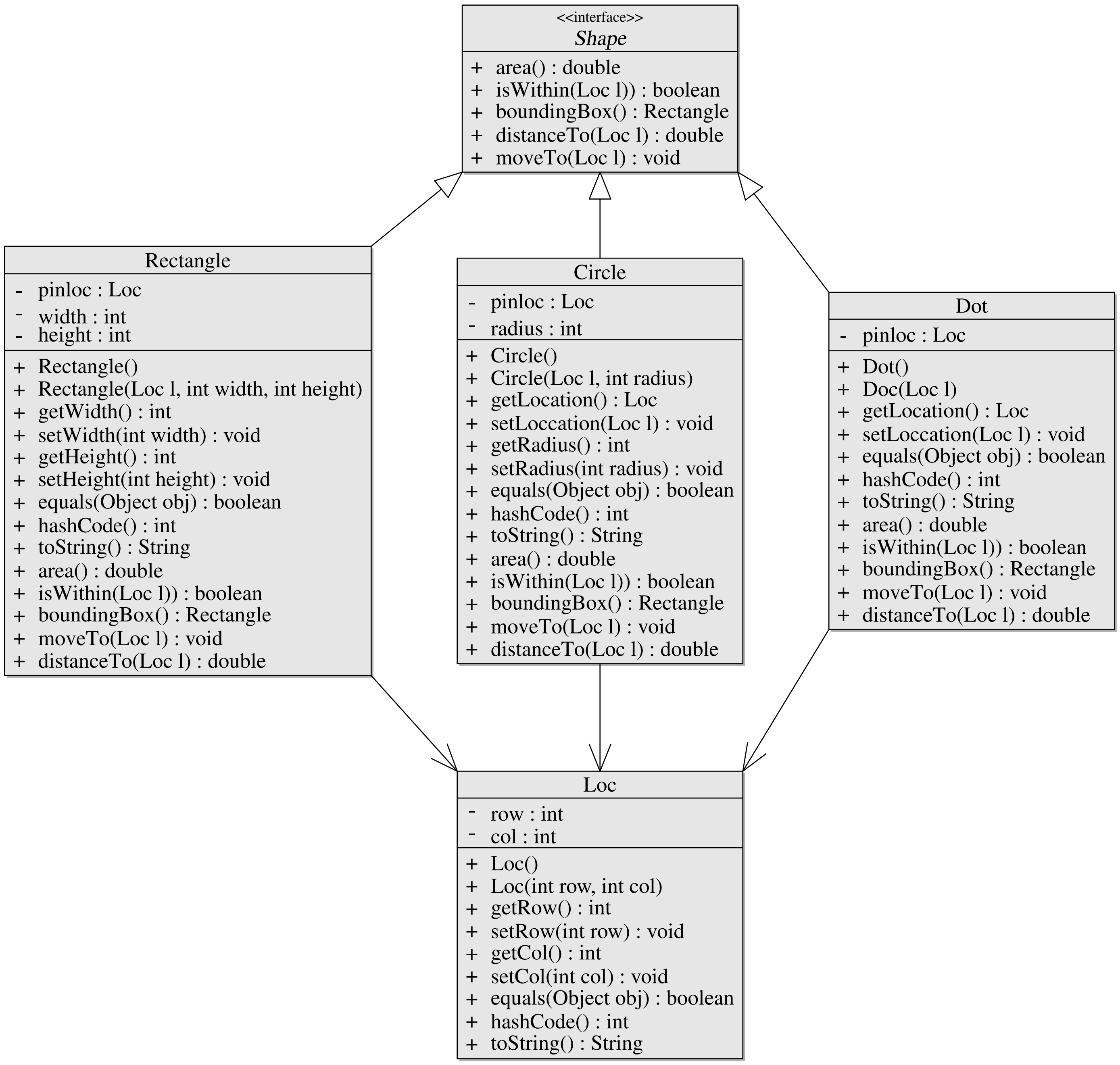


Figure 4: Shape Class Hierarchy

There's a lot more to UML diagramming. We're just employing some basics in order to clearly state the core details of our class design.

Completing the Implementation in Java

Eclipse has several features that help you to get Classes and Class Hierarchies setup quickly:

- Automatic stubs for inherited methods by declaring interfaces[15] in the new class wizard.
- Launching the new Class/Interface wizard from unresolved type errors as a Quick fix option.
- Automatic generation of Class getters and setters based on declared fields
- Automatic generation of Class constructors based on declared fields
- Automatic generation of *equals*, *hashCode*, and *toString* overrides based on class fields
- Automatic generation of stubs for undeclared methods

[15] and super classes

Basically, most of your boilerplate can be automatically generated for you and the methods that all objects inherit from the *Object* class can be automatically overridden using accepted Java best practices.

To make the best use of Eclipse's automatic code generation features, we'll work the implementation of our hierarchies from most to least abstract. This typically means interface on down.

1. Complete interfaces by declaring and documenting all their methods.

   Whenever you need to reference a Class or Interface that is a part of your program but is not yet implemented, just go ahead and state the type. Eclipse will highlight it as an error. Ignore it until the interface is complete.

2. For any undeclared interfaces or classes in your completed interface, use the *Create Class* or *Create Interface* quick fix[16] to have Eclipse stub out the Class/Interface. Be sure to add interfaces to the classes from within the wizard so that eclipse can stub out the methods for you.

[16] hover the mouse over the error

3. Repeat the above steps until you have rough stubs for the complete hierarchy. All that should be left at this point are classes[17] of contained objects. Repeat the above stubbing process for those classes/hierarchies as well.

[17] or hierarchies

4. For each class, add the data field declarations at the top of the class, above stubbed out methods.[18].

[18] We'll list fields then methods

5. For each class use the *Source* menu to auto generate:

   - A no-arg/default constructor and a field initialization construction[19]
   - Getters and Setters
   - An *equals* and *hashCode* method
   - A *toString* method

   [19] one argument per field

   Eclipse lets you specify the entry point for all the code it generates. For our Java classes, we want all the above stubs and generated code listed in the following order: fields, constructors, getters and setters, equals, hashCode, toString, and finally inherited methods.

6. All the boilerplate and stubs for all the methods in all the classes should now be in place and we can now generate JUnit Test cases for each class and begin writing your tests.

   In practice you should test any method that is public. Constructors can be tested implicitly through the methods[20]. We may loosen our testing practices up for auto-generated code eventually, but to start out, writing tests is an excellent way to familiarize yourself with the methods and their functionality.

   [20] if it behaves the way its supposed to, then it was constructed correctly

7. Verify your auto-generated code then begin implementing your core methods.

   If, while implementing your design, you decide you need a helper method, go ahead and call it. Eclipse generates an undeclared method error with a quick fix option of adding that method to the appropriate class. If you're confident in the helper's purpose, then finish the method that caused you to need the helper. If you're unsure about the helper, then stop work on the current method in order to document, write tests for, and implement the helper.

It often makes sense, and is convenient, to run the tests for multiple classes at once. For example, you might like to run all the tests for all the classes that implement a particular interface. This is accomplished by a *JUnit Tests Suite*. To create one go to *New* > *Other* > *JUnit* > *Test Suite* and use the wizard to create a suite.

Polymorphic Method Dispatch

While you cannot instantiate a Shape object[21]. You can have variables of that type.That variable can take on values[22] of any of the classes

[21] nor any Interface type

[22] reference objects

that implement it. We call this variable POLYMORPHIC as its actual value takes on many different forms, or types. There is often significant advantage in doing this. It's important to understand how computation proceeds when working with these abstractly typed variables.

Consider the following test: The type of *aShape* is *Shape*. The Shape

```java
// A 5x5 square at row 14, column 10
Shape aShape = new Rectangle(new Loc(14,10),5,5);
assertEquals(aShape.area(),25.0,0.000001);
```

Figure 5: Testing a Rectangle with a Shape Variable

type is an interface and has no concrete implementation for area. So, determine how to compute the area of *aShape* cannot be as simple as, "run the method implementation for the type of the variable."

The key, of course, is that the type of the variable is less important than the type of the value. In our example, the value stored in *aShape* is of type *Rectangle*. This type has a clear and unambiguous implementation for area. The only question is how does the computer make this connection? The answer is SUBSTITUTION. First the computer evaluates the variable itself and effectively substitutes the name for the value. In our example this produces the unambiguous statement[23]:

[23] the assigned expression is the value

```java
assertEquals(new Rectangle(new
    Loc(14,10),5,5).area(),25.0,0.000001);
```

Figure 6: Name-Value Substitution to resolve ambiguity

What if this came next?

```java
aShape = new Dot(new Loc(0,0));
assertEquals(aShape.area(),0.0,0.0000001);
```

Figure 7: Assigning a new value to *aShape*

Now the exact same expression[24] produces the execution of different code.

[24] aShape.area()

This is an entirely new phenomenon for us. The reason is clear, the expression maybe the same but the type of the object referenced by aShape has changed and it's that type that determines which implementation of area is executed. This feature of OOP and class hierarchies is called POLYMORPHIC METHOD DISPATCH[25]. Recall that the metaphor of operation is message passing. Calling *area* is done by passing a message, or dispatching a message, to the appropriate object which then executes the code. The dispatch described by

[25] aka Dynamic Dispatch

```
assertEquals(new Dot(new Loc(0,0)).area(),25.0,0.000001);
```

Figure 8: Once again, object type, not variable type, determines method call

this expression can takes on many different forms[26] within a single program based on the type of the object referenced by *aShape*.

[26] polymorphic

Examining Eclipse's Code

You can learn a lot about Java and OOP programming idioms by looking at the Eclipse generated code. It's also worth playing with the different options available to you. Here we'll look at an instance of *equals* and *toString* from the Shape hierarchy to point out some key styles and practices.

equals for Circle

The *equals* method generator really on has one option that isn't purely stylistic and that's one to "Use *instanceof* to compare types". We'll start with this option unchecked and briefly discuss the difference when its checked. I always use the blocks in if statements option, you can choose whichever style you prefer.

What is most important to realize is that the equals generated by Eclipse allows you to compare an instance of your class to *any other datum in Java*. This happens for two reasons. First and foremost all Classes implicitly extend the *java.lang.Object* class[27] and therefore inherit the type *Object*. That means any object can be passed as input to *equals*. The second reason this equals works on all data is that primitive types can and will get auto-converted to a logically equivalent built in Class through a process called *autoboxing*[28]. It's this flexibility and complexity that makes the Java *equals* an interesting method to study.

[27] more of CLASS EXTENSION in the next set of notes

[28] `https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html`

The *equals* implementation for *Circle* is a good representative of the *equals* method logic because it has both primitive and Class-based fields. So, combined with the other equals boilerplate, it shows you everything you need to know really. In figure 9 we see *equals* for *Circle* as generated by Eclipse.

This *equals* implementation begins by checking for pointer equality with the *if* on line 3. It then carries out a series of tests that, for the most part, check for conditions that immediately rule out equality. If all of those checks fail, then the argument *obj* must be an equivalent circle. The *if* on line 7 checks to see if the Object reference *obj* is null, if it is then we're comparing an allocated Circle to nothing. These first two conditions clearly illustrate the fact that object variables are

Figure 9: Eclipse *equals* for *Circle*

```
public boolean equals(Object obj) {
   // test for address equality
  if (this == obj) {
    return true;
  }
  // check that obj isn't just null
  if (obj == null) {
    return false;
  }
  // ensure that obj is a Circle
  if (getClass() != obj.getClass()) {
    return false;
  }
  // Ok. obj is a Circle. Compare them by fields.
  Circle other = (Circle) obj;
  if (center == null) {
    if (other.center != null) {
      return false;
    }
  } else if (!center.equals(other.center)) {
    return false;
  }
  if (radius != other.radius) {
    return false;
  }
  return true;
}
```

really pointer-like references to object values. Testing two variables directly with == compares addresses, just like with C++ pointers. As a reference to object, a variable can also refer to nothing, *null*, and so we must always be wary of using the dot operator of a variable that we cannot guarantee isn't a null reference.

Once we've dealt with the pointer-like conditions, we move on to the type checking. To be equal to a circle the Object *obj* must be a circle. The *if* statement on line 11 verifies this. Notice that *getClass()* is implicitly *this.getClass()*. This method, inherited from Object, returns the type of the object on which its invoked and so this comparison is only false when *obj* is of type *Circle*. When this is not the case, the method returns false.

When *obj* is a circle we begin the process of verifying that the circle *this* and the Circle *obj* have equivalent field values. To do this we need to stop looking at *obj* as an Object and start looking at it as a *Circle*. On line 15 we type cast *obj* to Circle then assign that value to the Circle variable *other*. The compiler recognizes *other* as a Circle and allows us to access Circle fields and methods through the dot operator and *other*.

The *if* starting on line 16 checks to see if the contained locations are different and returns false if they are. Because, in general, Object variables might be null, Eclipse does the safe thing and first null checks all of the *center* fields starting with *this.center*[29]. Take the time to see how this code covers the standard four cases for two references: both are null, only this.center is null, only other.center is null, neither are null. Finally, when neither are null we hand off the responsible of equality checking to the contained Class *equals* method. This is a vital strategy in managing CLASS CONTAINMENT. As much as possible, let the contained object manage all the computing relative to its own value. A good gut check is if you find yourself selecting a contained object's fields and then computing with that value, step back and see if you can rethink the problem as a method for the contained object and write it as such[30].

[29] again, the *this* is implicit

[30] notice this only works if you can modify the contained class' definition

The *if* on line 23 compares the primitive typed fields. These variables store their datum directly, not by reference. That means they cannot be null and a direct comparison with == will compare the actual value, not the address of the value. This case is straight forward and when it fails the method moves on to the final line. All the situations that would make these object not equivalent have been ruled out, and so we return true.

This implementation demonstrates best practices for managing contained objects as well as primitive data. In the later case, we're save to proceed with basic computation. When we're working with objects, we should work through and with that object's interface rather than attempt to select field values and do the work ourselves. The other part of our reality that this code highlights is the fact that variables used to store objects are pointer-like in nature. They can be null and their direct comparison is a check on sameness[31] not value equality.

[31] as in same address

Had we selected to use *instanceof*, then rather than comparing types with *getClass* we'd be checking the type of *obj* with the instanceof operator as shown in figure 10 The big difference here is

Figure 10: Type checking with *instanceof*

```
if( !(obj instanceof Circle) ){
  return false;
}
```

that *instanceof* will return true of the object on the left is from the class on the right *or one of its subclasses*. If in the future we extended Circle somehow and *obj* was an instance of that subclass, then this code wouldn't recognize that the most specific, descriptive type of obj didn't directly match that of *this*. This may or may not be significant depending on the problem and classes. We'll stick to the *getClass()* style to avoid the potential problem.

The many faces of toString

The only option for *toString* generation that we need to look at right now is *Code Style*. The four options you can choose from really highlight some old ideas as well as some newer OO ideas.

The first option is the purely functional one. By using the immutable String class as shown in figure 11 we can often write a simple one liner. What's important to notice is the the value of *center* and *radius* are implicitly converted to strings. In the case of *center*, this is done through the Loc class' *toString* method. This auto-conversion of value makes generating strings[32] easy.

[32] and similarly output on System.out or a stream

Figure 11: *toString* using Strings

```
public String toString() {
  return "Circle [center=" + center + ", radius=" + radius + "]";
}
```

If functional programming isn't your thing, then you can use the mutable string class *StringBuilder* and construct the string via a series of append mutations as seen in figure 12. Notice we still get the auto-conversion of non-String values to strings.

If the stop and go, multi-statement style of figure 12 isn't your thing, then you can CHAIN METHOD CALLS together and do all the

```
public String toString() {
  StringBuilder builder = new StringBuilder();
  builder.append("Circle [center=");
  builder.append(center);
  builder.append(", radius=");
  builder.append(radius);
  builder.append("]");
  return builder.toString();
}
```

Figure 12: *toString* with StringBuilder

appends in a single statement as seen in figure 13. This third style

```
public String toString() {
  StringBuilder builder = new StringBuilder();
  builder.append("Circle [center=").append(center).append(",
      radius=").append(radius).append("]");
  return builder.toString();
}
```

Figure 13: *toString* with StringBuilder and chained calls

merits some closer examination as it exposes a useful style for OOP mutators that we're likely to employ soon. Go look at the documentation for StringBuilder's append[33]. It returns a StringBuilder. What this code shows us is that the returned StringBuilder is the freshly modified StringBuilder on the left side of the dot operator. This works efficiently because of the pointer-like nature of Java object variables. The return value of the mutation is effectively the address of the modified object. The subsequent method call then acts on the same object rather than a copy. In C++ we accomplish this declaring a reference-type return value using & then return **this* at the end of the method. In Java we'll have to jump through significantly fewer syntactic hoops.

[33] `https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html`

The final option uses format strings which have their roots back in Fortran. To read all about format strings you need to look at the documentation for the Formatter class[34]. This style of string construction is useful to know because it lets you control things like the width of printed values, the number of decimal places, and many other things that we previously managed with stream manipulators in C++. We'll return to this topic later, so do take some time to go through the Formatter documentation and learn about format strings.

[34] `https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html`

```java
public String toString() {
  return String.format("Circle [center=%s, radius=%s]", center,
      radius);
}
```

Figure 14: *toString* with printf style Formatting