Lecture 4: Methods for unions

Design methods for unions of classes of data. Dynamic dispatch.
Practice using wish lists.

Related files:
IShape.java

Overview

In the last lecture we introduced simple method definitions for classes, method invocation to actually execute those methods, the notion of *delegating* responsibility from one object to another, and introduced how to test our methods. In this lecture, we introduce methods for union data types, and introduce the concept of *dynamic dispatch*.

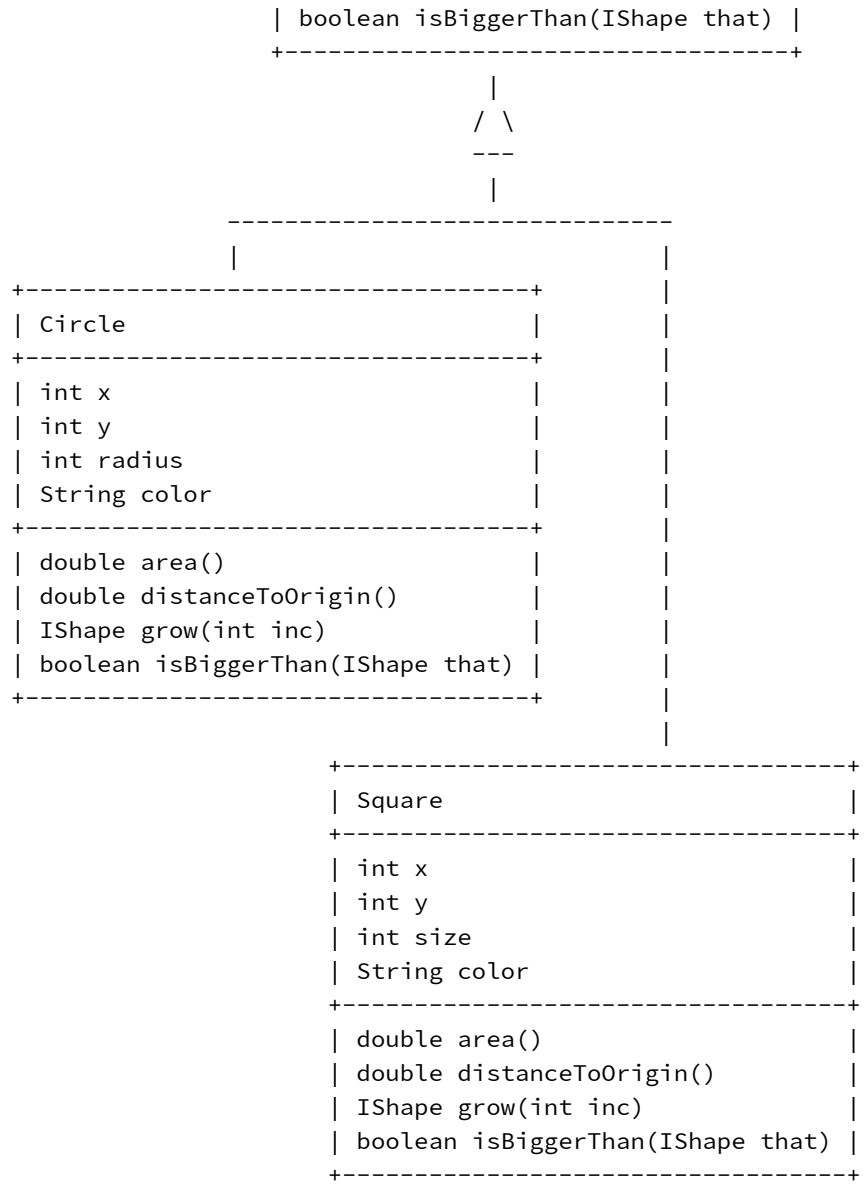
- Designing the `area` method
- Dynamic dispatch
- Designing the `distanceToOrigin` method: Helper classes and methods
- Points: more generalization
- Designing the `isBiggerThan` method: more duplicated code
- Designing the `contains` method: more helper methods

4.1 Designing Methods for Simple Shapes

Here are our two classes that represent shapes - we defined them to *implement* the common `interface IShape`.

The class diagram below shows our ambitious program - of designing four methods for these classes.

```
+-----------------------------------+
| IShape                            |
+-----------------------------------+
+-----------------------------------+
| double area()                     |
| double distanceToOrigin()         |
| IShape grow(int inc)              |
```



We want to design the following methods that would work for any shape - the two we have defined now, and any other shape class we may define in the future (for example a `Triangle` class).

```
// to compute the area of this shape
double area();

// to compute the distance from this shape to the origin
double distanceToOrigin();

// to increase the size of this shape by the given increment
IShape grow(int inc);

// is the area of this shape bigger than the area of the given shape?
boolean isBiggerThan(IShape that);
```

4.2 Designing the `area` method: dynamic dispatch

To compute the area of a shape in DrRacket, the function purpose, signature and header would have been:

```
;; to compute the area of the given shape
;; area : Shape -> Number
(define (area ashape) ...)
```

and the template would have been:

```
;; to compute the area of the given shape
;; area : Shape -> Number
(define (area ashape)
  (cond
       [(circle? ashape) ...]
       [(square? ashape) ...]))
```

In Java the methods that deal with each type of object are defined within the corresponding class definitions for those objects. So the `area` method that computes the area of a circle is defined in the `Circle` class, and the `area` method that computes the area of a square is defined in the `Square` class.

A union of several classes in Java is represented by an *interface* type. We would like to assure that every class that implements our `IShape` interface does indeed define the method `area`. The interface definition, that so far was just an empty shell of code, is now used to define method headers for some methods. Interfaces define a signature: one that must be upheld by every class that claims to implement the interface, and that can be relied upon by every client of the interface. Specifically, every class that implements the interface is required to implement all methods that the interface specifies. In turn, whenever we use any object of the type specified by the interface, we are guaranteed that we can invoke every method defined in the interface on this object.

> For now it seems redundant to have to write `public`; surely Java can simply detect that methods were declared in an interface, so why do we have to write it ourselves? The full answer to that will have to wait until Object Oriented Design...

New keyword: Whenever we define methods in a class that were declared in an interface, we must mark them as `public`. Our code would look like this:

```
// to represent a geometric shape
interface IShape {
  // to compute the area of this shape
  double area();
  // We'll add more methods later
}

// to represent a circle
class Circle implements IShape {
  int x; // represents the center of the circle
  int y;
```

```
  int radius;
  String color;

  Circle(int x, int y, int radius, String color) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.color = color;
  }

  /* TEMPLATE
     FIELDS:
     ... this.x ...                  -- int
     ... this.y ...                  -- int
     ... this.radius ...             -- int
     ... this.color ...              -- String
     METHODS
     ... this.area() ...             -- double
  */

  // to compute the area of this shape
  public double area() {
    return Math.PI * this.radius * this.radius;
  }
}

// to represent a square
class Square implements IShape {
  int x; // represents the top-left corner of the square
  int y;
  int size;
  String color;

  Square(int x, int y, int size, String color) {
    this.x = x;
    this.y = y;
    this.size = size;
    this.color = color;
  }

  /* TEMPLATE
     FIELDS:
     ... this.x ...              -- int
     ... this.y ...              -- int
     ... this.size ...           -- int
     ... this.color ...          -- String
     METHODS:
     ... this.area() ...                 -- double
  */
```

```
    // to compute the area of this shape
    public double area() {
      return this.size * this.size;
    }
  }

  class ExamplesShapes {
    ExamplesShapes() {}

    IShape c1 = new Circle(50, 50, 10, "red");
    IShape s1 = new Square(50, 50, 30, "red");

    // test the method area in the classes that implement IShape
    boolean testIShapeArea(Tester t) {
      return
      t.checkInexact(this.c1.area(), 314.15, 0.01) &&
      t.checkInexact(this.s1.area(), 900.0, 0.01);
    }
  }
```

Do Now!

Define the remaining methods for the classes `Circle` and `Square`:

```
// In IShape
// to compute the distance from this shape to the origin
double distanceToOrigin();

// to increase the size of this shape by the given increment
IShape grow(int inc);

// is the area of this shape bigger than
// the area of the given shape?
boolean isBiggerThan(IShape that);
```

One of these might require more helper methods than have been defined so far.

4.3 Dynamic dispatch

Something should have seemed a bit odd in the examples above. *How* did Java "know" that when we wrote `this.c1.area()`, we intended to invoke the `area` method defined in the `Circle` class, and that when we wrote `this.s1.area()`, we intended to invoke the method defined in the `Square` class? After all, both `s1` and `c1` are declared as having type `IShape` — and interfaces don't define any method implementations at all!

Suppose we made the situation even murkier: suppose we needed to write a helper method in some code that returned twice the area of an object:

```
// somewhere in our code, in some class...
double twiceTheArea(IShape aShape) {
  return 2 * aShape.area();
}
```

Can you tell, just from looking at that snippet of code, what kind of shape this method is given? No! Said another way, all we know *statically* — that is, by looking at the text of our program — is that the argument `aShape` is some `IShape`. However, *dynamically* — that is, when we run the program and this method gets invoked — we must have some specific shape at hand. And it is only at that moment in the program's execution that we could determine which `area` method to invoke.

This process, of *dispatching* to the correct method implementation based on information available only *dynamically*, is called *dynamic dispatch*, and it is one of the cornerstone concepts of object-oriented programming.

> ***Do Now!***
>
> Look back at the Racket template for the `area` function, and compare it to the implementation of the `area` methods in `Circle` and `Square`. Do you see anything missing in the Java version?

Notice that the Java version of this method has *no explicit `cond` expression, or any expressions like `circle?` or `square?` at all*. What was the purpose of that `cond`? It distinguished between the different variants of our union data type. In an object-oriented language, this is *precisely* the purpose of dynamic dispatch: the language itself distinguishes between the variants, and dispatches to the correct method implementation for you, without you having to write any explicit variant-testing code.

If you find yourself wanting to write an expression like `square?` or `circle?`, *stop!* Re-read this portion of the lecture notes, and then redesign your code to use a method on an interface and dynamic dispatch instead.

4.4 Designing the `distanceToOrigin` method

Let's implement the `distanceToOrigin` method, which will return the distance to the edge of a circle, or the distance to the top-left corner of a square. If we naively "just start writing" code, we'll wind up with this poorly-designed result:

```
class Circle implements IShape {
  ...
  /* TEMPLATE
     FIELDS:
     ... this.x ...                    -- int
     ... this.y ...                    -- int
     ... this.radius ...               -- int
     ... this.color ...                -- String
     METHODS
```

```
      ... this.area() ...              -- double
      ... this.distanceToOrigin() ...  -- double
    */
    public double distanceToOrigin() {
      return Math.sqrt(this.x * this.x + this.y * this.y) - this.radius;
    }
  }
  class Square implements IShape {
    ...
    /* TEMPLATE
       FIELDS:
       ... this.x ...                  -- int
       ... this.y ...                  -- int
       ... this.size ...               -- int
       ... this.color ...              -- String
       METHODS
       ... this.area() ...             -- double
       ... this.distanceToOrigin() ...  -- double
    */
    public double distanceToOrigin() {
      return Math.sqrt(this.x * this.x + this.y * this.y);
    }
  }
```

Notice that these two methods are very similar, are operating on nearly identical data, and are producing nearly the same result. If only we didn't have to write out the Pythagorean formula twice – chances are good we'll make a mistake at least once. And what if in the future we add more kinds of `IShape`s: will we have to write the formula again?

Also confusing: while both shapes have `x` and `y` fields, they mean slightly different things: the center of the circle versus the top-left corner of the square.

At this point, the design recipe for abstraction says "make a helper function". But Java has no functions: it only has classes, interfaces, and methods. But that gives us an idea. We can solve all these problems at once, by factoring out the `x` and `y` fields into a *helper class*, which we will call `CartPt` (for Cartesian Point).

```
// To represent a 2-d point by Cartesian coordinates
class CartPt {
  int x;
  int y;
  CartPt(int x, int y) {
    this.x = x;
    this.y = y;
  }
}

class Circle implements IShape {
  CartPt center; // NEW!  And its name is far more helpful
  int radius;
  String color;
```

```
  Circle(CartPt center, int radius, String color) {
    this.center = center;
    this.radius = radius;
    this.color = color;
  }
  ...
}
class Square implements IShape {
  CartPt topLeft; // NEW!  And its name is far more helpful
  int size;
  String color;
  Square(CartPt topLeft, int size, String color) {
    this.topLeft = topLeft;
    this.size = size;
    this.color = color;
  }
  ...
}
```

> *Do Now!*
>
> Revise the class diagram above to include `CartPt` and these changes to `Circle` and `Square`.

Now we can add methods to `CartPt` that might be helpful to us in implementing methods for `IShapes`:

```
class CartPt {
  ...
  // To compute the distance from this point to the origin
  double distanceToOrigin() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}
```

> *Do Now!*
>
> Revise the `distanceToOrigin` methods in `Circle` and `Square` to delegate to this method on `CartPt`.

```
class Circle implements IShape {
  ...
  /* TEMPLATE
     FIELDS:
     ... this.center ...                    -- CartPt
     ... this.radius ...                    -- int
     ... this.color ...                     -- String
     METHODS
```

```
      ... this.area() ...                    -- double
      ... this.distanceToOrigin() ...        -- double
      METHODS ON FIELDS ----- NEW!
      ... this.center.distanceToOrigin() ... -- double
    */
    public double distanceToOrigin() {
      return this.center.distanceToOrigin() - this.radius;
    }
  }
  class Square implements IShape {
    ...
    /* TEMPLATE
      FIELDS:
      ... this.topLeft ...                   -- CartPt
      ... this.size ...                      -- int
      ... this.color ...                     -- String
      METHODS
      ... this.area() ...                    -- double
      ... this.distanceToOrigin() ...        -- double
      METHODS ON FIELDS ----- NEW!
      ... this.topLeft.distanceToOrigin() ... -- double
    */
    public double distanceToOrigin() {
      return this.topLeft.distanceToOrigin();
    }
  }
```

Much better! This is one common example of an important principle in program design: Don't Repeat Yourself. If there is a way to refactor your code, to extract common elements and separate them into reusable abstractions, it's probably a good idea to do that.

In this case, we get to *extend our template* with a new section, methods of fields, that gives us access to exactly the new functionality that we need.

> ***Do Now!***
>
> Design a method `boolean contains(CartPt point)` for `IShape`s that returns `true` if the given point is within the shape.

4.5 What's so special about Cartesian coordinates, anyway?

There is a subtle, but very powerful, benefit to abstracting `x` and `y` into their own class. Notice now that there is no code, outside the `CartPt` class itself, that knows or cares about `x` or `y` fields. So if we wanted to, we could represent our points as `PolarPt`, with `r` and `theta` fields instead.

> ***Do Now!***

> Design the `PolarPt` class. Revise the data definitions we have so far, so that `Square`s and `Circle`s could accept a `PolarPt` instead of just a `CartPt`. (Hint: now that a point is not just a `CartPt`, but *one of* `CartPt` or `PolarPt`, what new construction do you need to inform Java about this relationship?)

> ***Do Now!***
>
> What goes wrong when trying to implement the `contains` method?

4.6 Designing the `isBiggerThan` method

Let's look at the `isBiggerThan` method. It asks whether this shape's area is greater than the given shape's area. If only we had a way to compute the area of shapes, we'd be done. Wait – we are!

> ***Do Now!***
>
> Design the `isBiggerThan` method for `Circle` and `Square`. Did you get a sense of deja vu on implementing the second one?

```
// to represent a circle
class Circle implements IShape {
  ...
  /* TEMPLATE
     FIELDS
     ... this.center ...                    -- CartPt
     ... this.radius ...                    -- int
     ... this.color ...                     -- String
     METHODS
     ... this.area() ...                    -- double
     ... this.distanceToOrigin() ...        -- double
     ... this.isBiggerThan(IShape that) ... -- boolean
     METHODS FOR FIELDS:
     ... this.center.distanceToOrigin() ... -- double
  */
  // is the area of this shape bigger than the area of the given shape?
  public boolean isBiggerThan(IShape that) {
    /*-------------------------------------------------
    // TEMPLATE for this method:
    // EVERYTHING from our class-wide template...
    ... this.center ...                    -- CartPt
    ... this.radius ...                    -- int
    ... this.color ...                     -- String

    ... this.area() ...                    -- double
    ... this.distanceToOrigin() ...        -- double
```

```
    ... this.isBiggerThan(IShape) ...      -- boolean

    ... this.center.distanceToOrigin() ... -- double

    // PLUS methods on the parameters
    ... that.area() ...                    -- double
    ... that.distanceToOrigin() ...        -- double
    ... that.isBiggerThan(IShape) ...      -- boolean
    ------------------------------------------------*/
    return this.area() > that.area();
  }
}
class Square implements IShape {
  ...
  /* TEMPLATE
     FIELDS
     ... this.topLeft ...                   -- CartPt
     ... this.size ...                      -- int
     ... this.color ...                     -- String
     METHODS
     ... this.area() ...                    -- double
     ... this.distanceToOrigin() ...        -- double
     ... this.isBiggerThan(IShape that) ... -- boolean
     METHODS FOR FIELDS:
     ... this.topLeft.distanceToOrigin() ...-- double
  */
  // is the area of this shape bigger than the area of the given shape?
  public boolean isBiggerThan(IShape that) {
    /*------------------------------------------------
    // TEMPLATE for this method:
    // EVERYTHING from our class-wide template...
    ... this.topLeft ...                   -- CartPt
    ... this.size ...                      -- int
    ... this.color ...                     -- String

    ... this.area() ...                    -- double
    ... this.distanceToOrigin() ...        -- double
    ... this.isBiggerThan(IShape) ...      -- boolean

    ... this.topLeft.distanceToOrigin() ...-- double

    // PLUS methods on the parameters
    ... that.area() ...                    -- double
    ... that.distanceToOrigin() ...        -- double
    ... that.isBiggerThan(IShape) ...      -- boolean
    ------------------------------------------------*/
    return this.area() > that.area();
  }
}
```

The implementations of both methods are identical! We'll see in Lecture 9: Abstract classes and inheritance how to eliminate this duplication.

4.7 Designing the `contains` method

Note: this section assumes that you are working with `CartPt` directly, rather than the generalization to `IPoint`. (See What's so special about Cartesian coordinates, anyway?)

Let's solve the problem first for `Circle`s. Determining whether a circle contains a point requires comparing the distances between two points. We already have a method that computes the distance *to a specific point*, the origin, but now we need a more general version that computes the distance *to a given point*:

```
class CartPt {
  ...
  double distanceTo(CartPt that) {
    return Math.sqrt(
      (this.x - that.x) * (this.x - that.x)
      + (this.y - that.y) * (this.y - that.y));
  }
}
```

Now we can implement `contains`:

```
class Circle implements IShape {
  ...
  boolean contains(CartPt point) {
    return this.center.distanceTo(point) < this.radius;
  }
}
```

Solving this for `Square`s is slightly trickier, as we need to compare the actual coordinates of the points:

```
class Square implements IShape {
  ...
  boolean contains(CartPt point) {
    return (this.topLeft.x <= point.x) &&
           (point.x <= this.topLeft.x + this.size) &&
           (this.topLeft.y <= point.y) &&
           (point.y <= this.topLeft.y + this.size);
  }
}
```

4.8 Putting it all together: Lots of Examples

We define several examples of each class we've built so far, and test all the methods on them. Notice two things about these examples: first, we declare the *types* of all our shapes to be `IShape`, rather than `Circle` or `Square`, because we were designing a union type. Second, notice that despite declaring the examples to all have type `IShape`, Java still correctly calls the methods defined in the `Circle` class for examples `c1`, `c2` and `c3`, and the methods defined in the `Square` class for examples `s1`, `s2` and `s3`. This is dynamic dispatch in action.

```
class ExamplesShapes {
  ExamplesShapes() {}

  CartPt pt1 = new CartPt(0, 0);
  CartPt pt2 = new CartPt(3, 4);
  CartPt pt3 = new CartPt(7, 1);

  IShape c1 = new Circle(new CartPt(50, 50), 10, "red");
  IShape c2 = new Circle(new CartPt(50, 50), 30, "red");
  IShape c3 = new Circle(new CartPt(30, 100), 30, "blue");

  IShape s1 = new Square(new CartPt(50, 50), 30, "red");
  IShape s2 = new Square(new CartPt(50, 50), 50, "red");
  IShape s3 = new Square(new CartPt(20, 40), 10, "green");

  // test the method distanceToOrigin in the class CartPt
  boolean testDistanceToOrigin(Tester t) {
    return
    t.checkInexact(this.pt1.distanceToOrigin(), 0.0, 0.001) &&
    t.checkInexact(this.pt2.distanceToOrigin(), 5.0, 0.001);
  }

  // test the method distTo in the class CartPt
  boolean testDistTo(Tester t) {
    return
    t.checkInexact(this.pt1.distTo(this.pt2), 5.0, 0.001) &&
    t.checkInexact(this.pt2.distTo(this.pt3), 5.0, 0.001);
  }

  // test the method area in the class Circle
  boolean testCircleArea(Tester t) {
    return
    t.checkInexact(this.c1.area(), 314.15, 0.01);
  }

  // test the method area in the class Square
  boolean testSquareArea(Tester t) {
    return
    t.checkInexact(this.s1.area(), 900.0, 0.01);
  }

  // test the method distanceToOrigin in the class Circle
  boolean testCircleDistanceToOrigin(Tester t) {
    return
    t.checkInexact(this.c1.distanceToOrigin(), 60.71, 0.01) &&
    t.checkInexact(this.c3.distanceToOrigin(), 74.40, 0.01);
  }

  // test the method distanceToOrigin in the class Square
  boolean testSquareDistanceToOrigin(Tester t) {
```

```
    return
    t.checkInexact(this.s1.distanceToOrigin(), 70.71, 0.01) &&
    t.checkInexact(this.s3.distanceToOrigin(), 44.72, 0.01);
  }

  // test the method grow in the class Circle
  boolean testCircleGrow(Tester t) {
    return
    t.checkExpect(this.c1.grow(20), this.c2);
  }

  // test the method grow in the class Square
  boolean testSquareGrow(Tester t) {
    return
    t.checkExpect(this.s1.grow(20), this.s2);
  }

  // test the method isBiggerThan in the class Circle
  boolean testCircleIsBiggerThan(Tester t) {
    return
    t.checkExpect(this.c1.isBiggerThan(this.c2), false) &&
    t.checkExpect(this.c2.isBiggerThan(this.c1), true) &&
    t.checkExpect(this.c1.isBiggerThan(this.s1), false) &&
    t.checkExpect(this.c1.isBiggerThan(this.s3), true);
  }

  // test the method isBiggerThan in the class Square
  boolean testSquareIsBiggerThan(Tester t) {
    return
    t.checkExpect(this.s1.isBiggerThan(this.s2), false) &&
    t.checkExpect(this.s2.isBiggerThan(this.s1), true) &&
    t.checkExpect(this.s1.isBiggerThan(this.c1), true) &&
    t.checkExpect(this.s3.isBiggerThan(this.c1), false);
  }

   // test the method contains in the class Circle
  boolean testCircleContains(Tester t) {
    return
    t.checkExpect(this.c1.contains(new CartPt(100, 100)), false) &&
    t.checkExpect(this.c2.contains(new CartPt(40, 60)), true);
  }

  // test the method contains in the class Square
  boolean testSquareContains(Tester t) {
    return
    t.checkExpect(this.s1.contains(new CartPt(100, 100)), false) &&
    t.checkExpect(this.s2.contains(new CartPt(55, 60)), true);
  }
}
```

Exercise

Define a new shape, `Combo`, that contains two `IShape`s. Its `distanceToOrigin` should be the minimum of the distances of its two components, its `area` should be the sum of the areas of its two components, it can `grow` by `grow`ing both of its components, and it `contains` a `CartPt` if either of its components `contains` the given point.