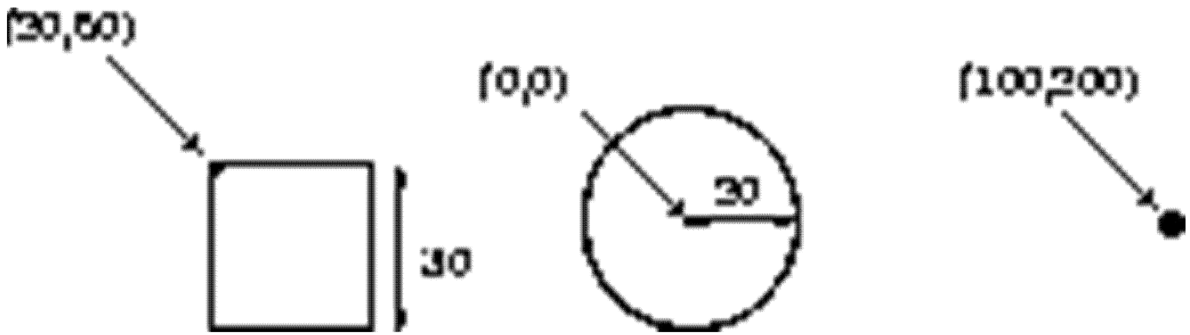
Unions of Classes and Methods

The Drawing Program

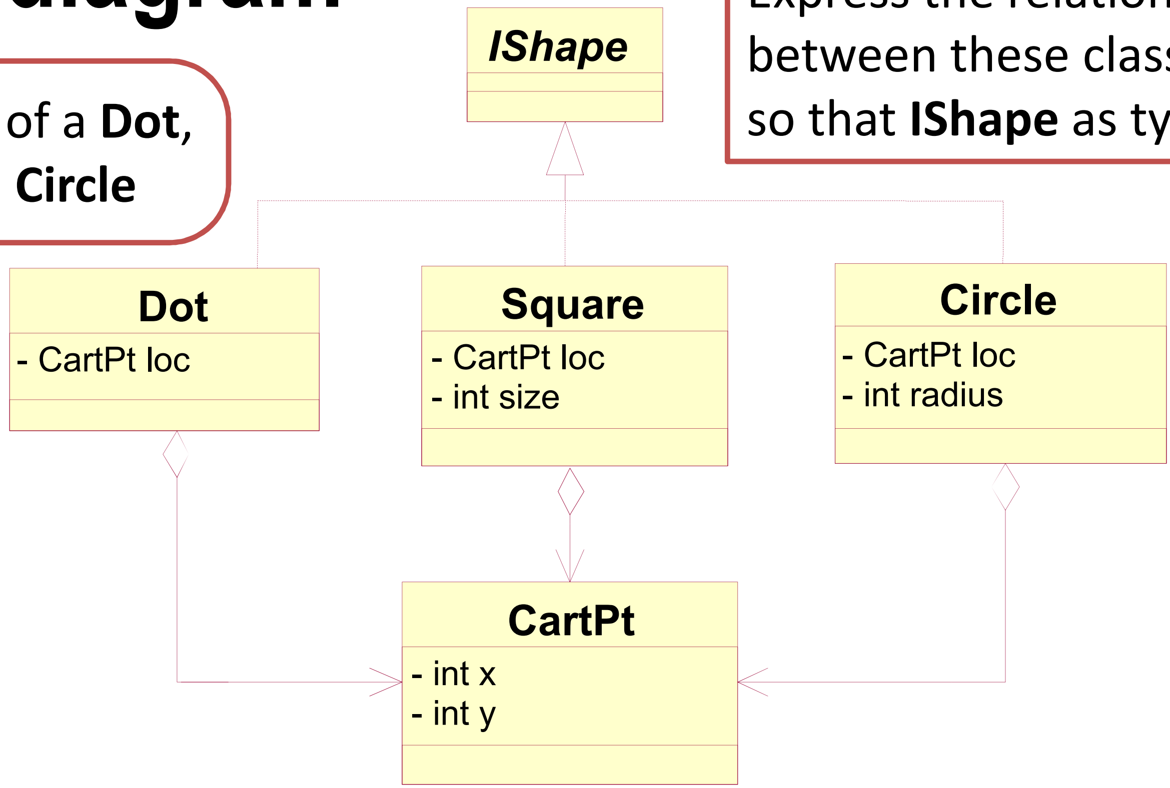
- Develop a drawing program that deals with at least three kinds of shapes: dots, squares, and circles.
  The shapes are located on a Cartesian grid whose origin is in the northwest.
  - A dot is located in the grid and is drawn as a small disk of a fixed size (3 pixels).
  - A square's loc is specified via its north-west corner in the grid and its size.
  - A circle's essential properties are its center point and its radius.



Class diagram

IShape is one of a **Dot**, a **Square**, a **Circle**

Express the relationship between these classes, so that **IShape** as types of data



- **IShape** is the name for a union of variant classes;
  - it doesn't contribute any objects to the complete collection.
  - Its purpose is to represent the complete collection of object

Java data definitions

```
public interface IShape {
}
```

```
public class Dot implements IShape {
   private CartPt loc;
   public Dot(CartPt loc) {
      this.loc = loc;
   }
}
```

```
public class Square implements IShape {
   private CartPt loc;
   private int size;
   public Square(CartPt loc, int size) {
      this.loc = loc;
      this.size = size;
   }
}
```

```
public class Circle implements IShape {
   private CartPt loc;
   private int radius;
   public Circle(CartPt loc, int radius) {
      this.loc = loc;
      this.radius = radius;
   }
}
```

Java data definitions

```
public class CartPt {
   private int x;
   private int y;
   public CartPt(int x, int y) {
      this.x = x;
      this.y = y;
  }
}
```

Test constructors

```
public class ShapeTest extends TestCase {
   public void testConstructor() {
      //test for class CartPt
      CartPt caPt1 = new CartPt(4, 3);
      CartPt caPt2 = new CartPt(5, 12);
      CartPt caPt3 = new CartPt(6, 8);

      // test  for class Dot
      IShape d1 = new Dot(caPt1);
      IShape d2 = new Dot(caPt2);
      IShape d3 = new Dot(caPt3);

      //test for class Circle
      IShape c1 = new Circle(caPt1, 5);
      IShape c2 = new Circle(caPt2, 10);
      IShape c3 = new Circle(caPt3, 12);

      //test for class Square
      IShape s1 = new Square(caPt1,5);
      IShape s2 = new Square(caPt3,10);
      IShape s3 = new Square(caPt3,12);
   }
}
```

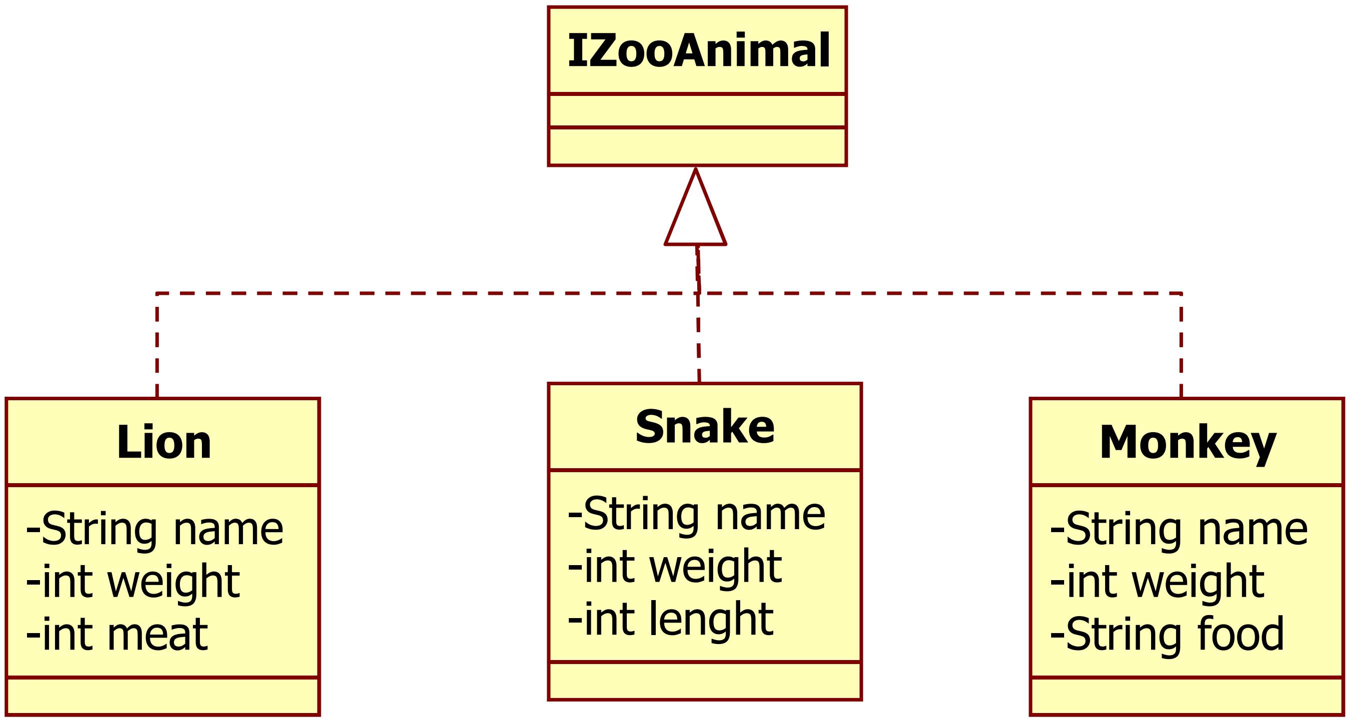
Types vs Classes

- A **class** is a general description of a collection of objects that provides a mechanism for constructing specific objects.
- An **interface** is a **uniform "face"** for several classes, which you sometimes wish to deal with as if it were one.
- A type describes for what kind of objects a variable or a parameter. In Java, a type is either the name of an **interface**, a **class**, or a **primitive type** **(`int`, `double`, `boolean`)**
- When we write: `IShape s;`
  - *s* has type ***`Ishape`***, which means that it is a placeholder for some unknown shape.
- Similarly, when we introduce an example such as `IShape s = new Square(...)`
  - *s* has type ***`IShape`***, even though it stands for an instance of ***`Square`***.

Zoo example

- Develop a program that helps a zoo keeper take care of the animals in the zoo.
- For now the zoo has lions, snakes, and monkeys.
  Every animal has a name and weight.
  The zoo keeper also needs to know how much meat the lion eats per day, the length of each snake, and the favorite food for each monkey
- Examples:
  – The lion Leo weighs 300 pounds and eats 5 pounds of meat every day;
  – The snake Boa weighs 50 pounds and is 5 feet long;
  – The monkey George weighs 150 poundds and loves bananas.
  – The monkey Mina weighs 120 pounds and loves to eat kiwi

Class diagram



Java definitions

```java
public interface IZooAnimal {
}
```

```java
public class Lion
    implements IZooAnimal {
  private String name;
  private int weight;
  private int meat;
  public Lion(String name,
   int weight, int meat) {
    this.name = name;
    this.weight = weight;
    this.meat = meat;
  }
}
```

```java
public class Snake
    implements IZooAnimal {
  private String name;
  private int weight;
  private int length;
  public Snake(String name,
   int weight, int length) {
    this.name = name;
    this.weight = weight;
    this.length = length;
  }
}
```

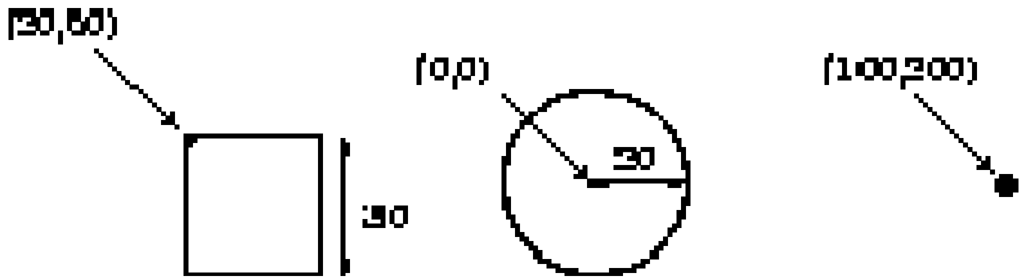
```java
public class Monkey
    implements IZooAnimal {
  private String name;
  private int weight;
  private String food;
  public Monkey(String name,
   int weight, String food) {
    this.name = name;
    this.weight = weight;
    this.food = food;
  }
}
```

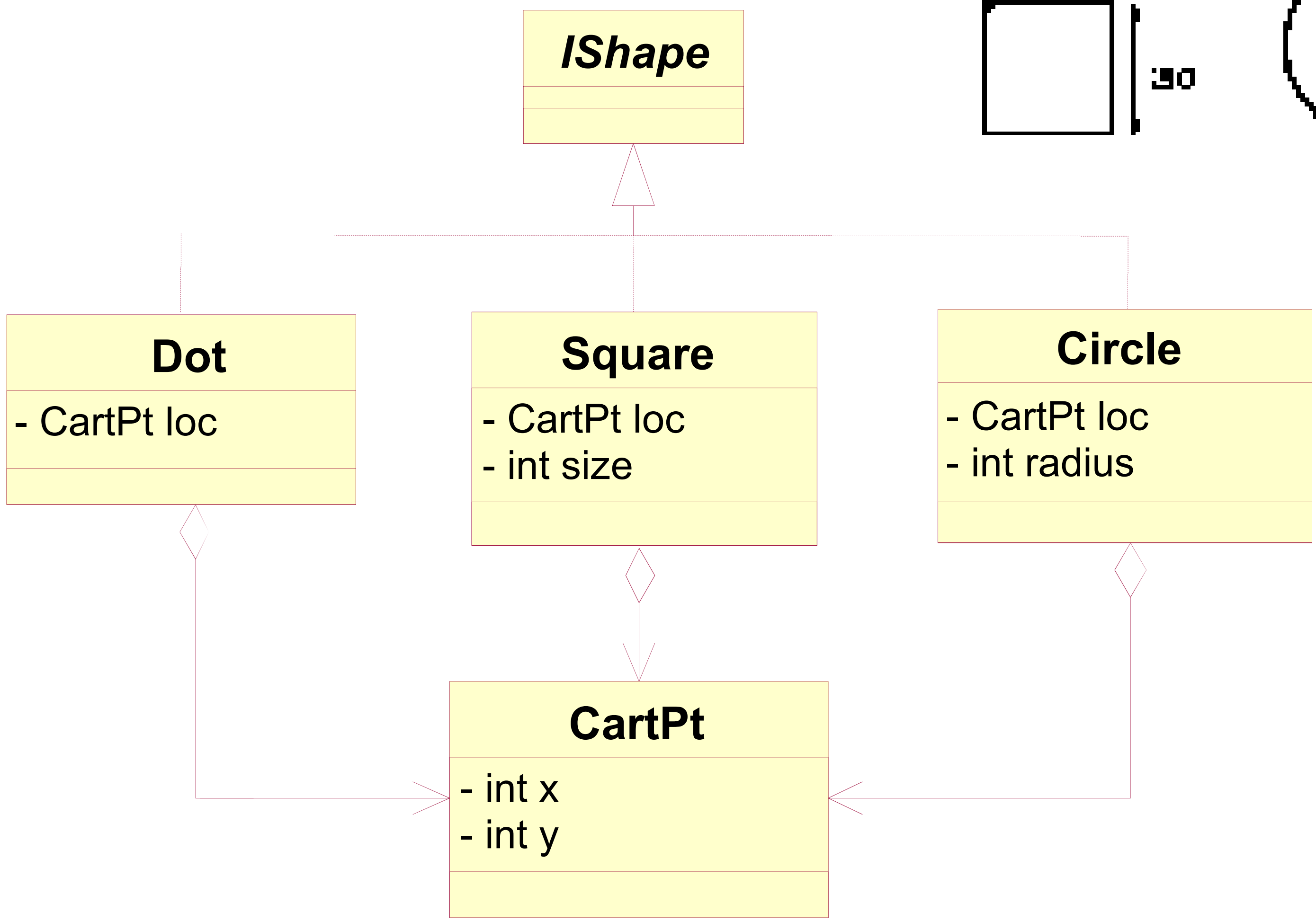
Test constructor

```java
public class AnimalTest extends TestCase {
   public void testConstructor(){
      // test for class Lion
      IZooAnimal leo = new Lion("Leo", 300, 5);
      IZooAnimal samba = new Lion("Samba", 200, 3);
      IZooAnimal Cleopon = new Lion("Cleopon", 250, 5);

      // test for class Snake
      IZooAnimal boa = new Snake("Boa", 50,5);
      IZooAnimal mic = new Snake("Mic", 45,4);
      IZooAnimal bu = new Snake("Bu", 55,6);

      // test for class Monkey
      IZooAnimal george = new Monkey("George", 150, "banana");
      IZooAnimal mina = new Monkey("Mina", 120, "Kiwi");
      IZooAnimal slan = new Monkey("Slan", 100, "Kiwi");
   }
}
```

Design methods for unions of classes

Recall the Drawing Program



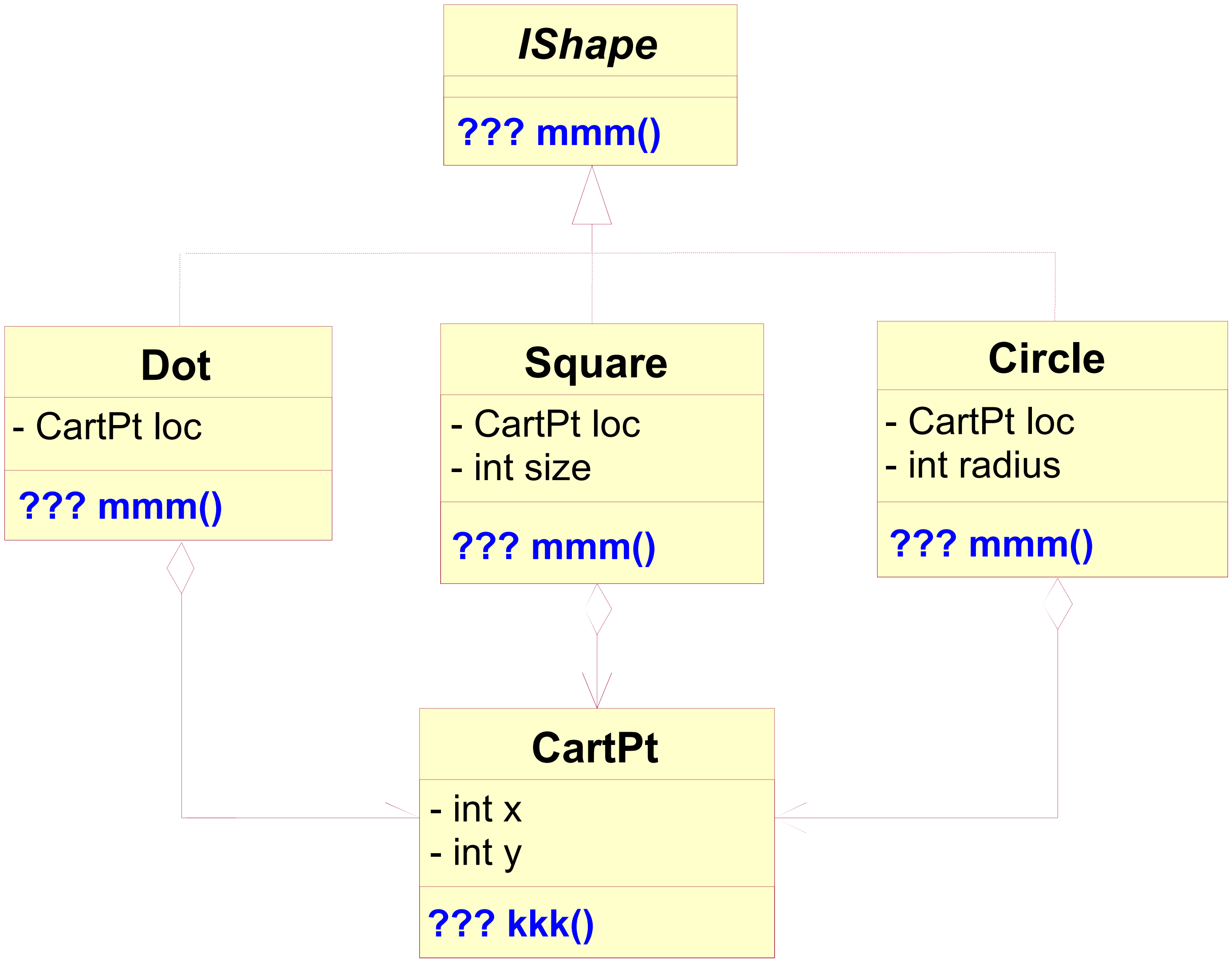


Requirements

1. Compute the *area* of a shape
2. Compute the *distance of a shape to the origin*
3. Determine *whether some point is inside the shape*
4. Compute the *bounding box* of a shape

- All of these methods clearly work with shapes in general but may have to compute different results depending on the concrete shape on which they are invoked
  - For example, a Dot has no true area; a Square's area is computed differently from a Circle's area
- In an object-oriented language, we can express this requirement with the addition of a method signature to the `IShape` interface

Add method for union of Shapes



kkk() Method Template of CartPt

```
public class CartPt {
   private int x;
   private int y;
   public CartPt(int x, int y) {
      this.x = x;
      this.y = y;
  }

  public ??? kkk() {
     …this.x…
     …this.y…
  }
}
```

nnn( ) method Template of Shape

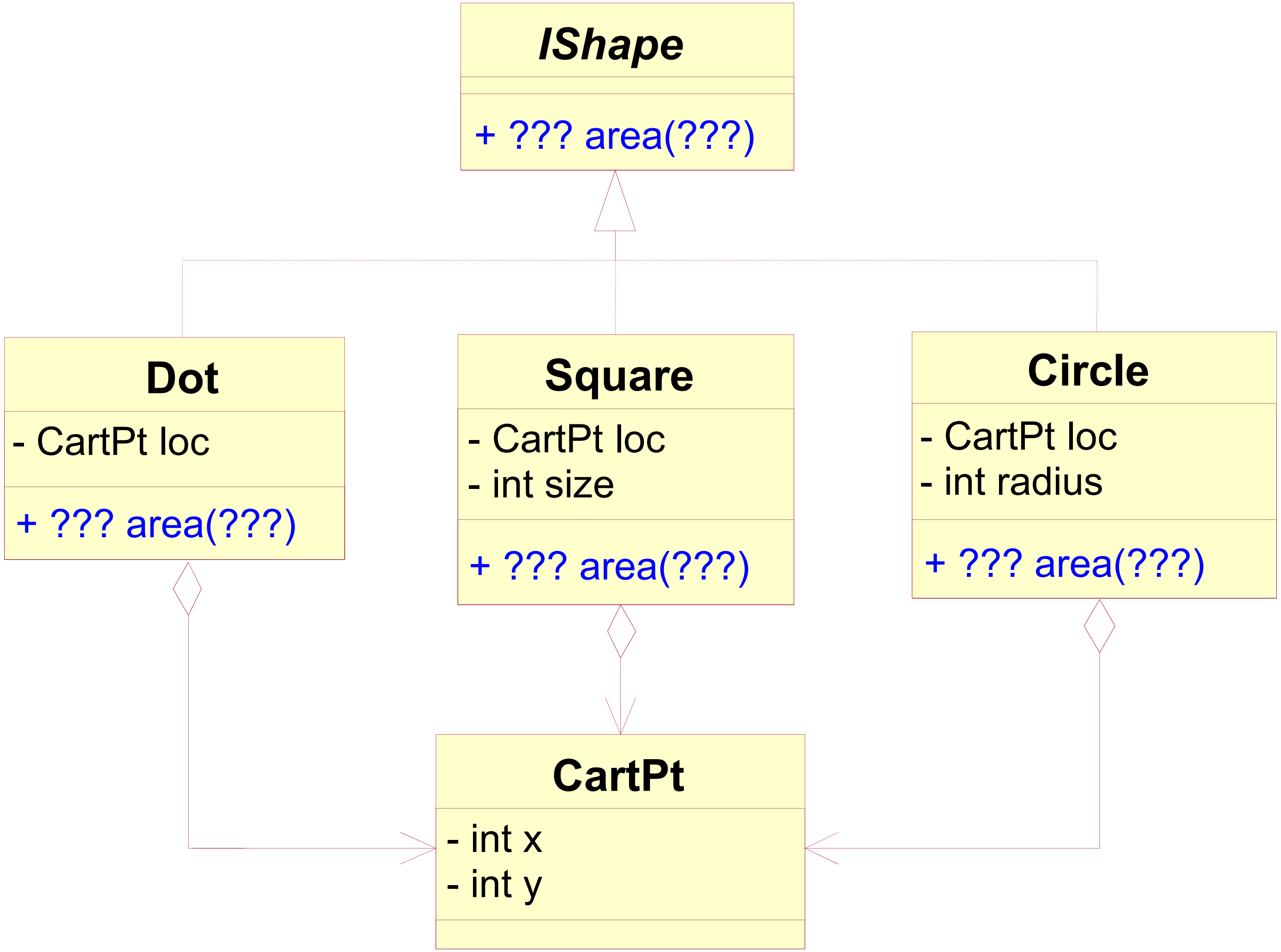
```java
public interface IShape {
   public ??? nnn();
}
```

```java
public class Dot
   implements IShape {
   private CartPt loc;
   public Dot(
         CartPt loc) {
     this.loc = loc;
   }

   public ??? nnn() {
     …this.loc.kkk()…
   }
}
```

```java
public class Square
   implements IShape {
   private CartPt loc;
   private int size;
   public Square(
         CartPt loc,
         int size) {
     this.loc = loc;
     this.size = size;
   }

   public ??? nnn() {
     …this.loc.kkk()…
     …this.size…
   }
}
```

```java
public class Circle
   implements IShape {
   private CartPt loc;
   private int radius;
   public Circle(
         CartPt loc,
         int radius) {
     this.loc = loc;
     this.radius = radius;
   }

   public ??? nnn() {
     …this.loc.kkk()…
     …this.radius…
   }
}
```

1. Computing Area of A Shape



Augmenting IShape
area() purpose and signature

```
public interface IShape {

    // compute area of AShape
    public double area();
}
```

Examples

```
new Dot(new CartPt(4, 3)).area()
// should be 0.0
new Square(new CartPt(4, 3), 30).area()
// should be 900.0
new Circle(new CartPt(5, 5), 20).area()
// should be 1256.6...
```

Q: Implement the body of all `area()` methods

Implement `area()` method

```
// inside of Dot
public double area() {
    return 0.0;
}
```

```
// inside of Circle
public double area() {
    return Math.PI * this.radius * this.radius;
}
```

```
// inside of Square
public double area() {
    return this.size * this.size;
}
```

Unit Testing

```
public class ShapeTest extends TestCase {

   public void testArea() {
      assertEquals(new Dot(new CartPt(4, 3))
           .area(), 0.0, 0.01);

      assertEquals(new Square(new CartPt(4, 3), 30)
           .area(), 900, 0.01);

      assertEquals(new Circle(new CartPt(5, 5), 20)
           .area(), 1256.64, 0.01);
  }
}
```

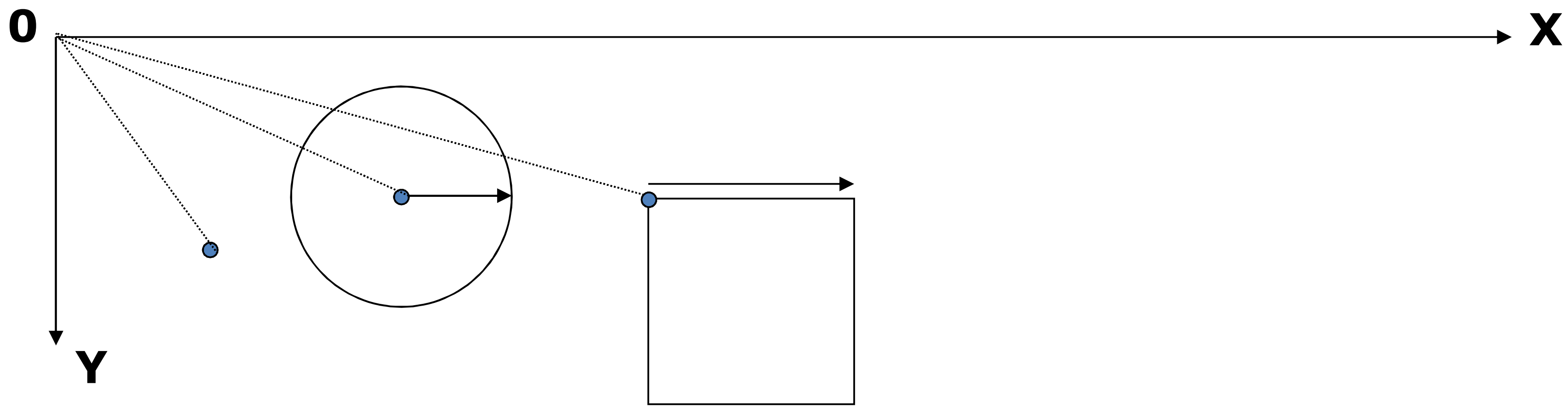
Polymorphism

- With the same call `area()`, but each *concrete subclass deal with it in difference way*.

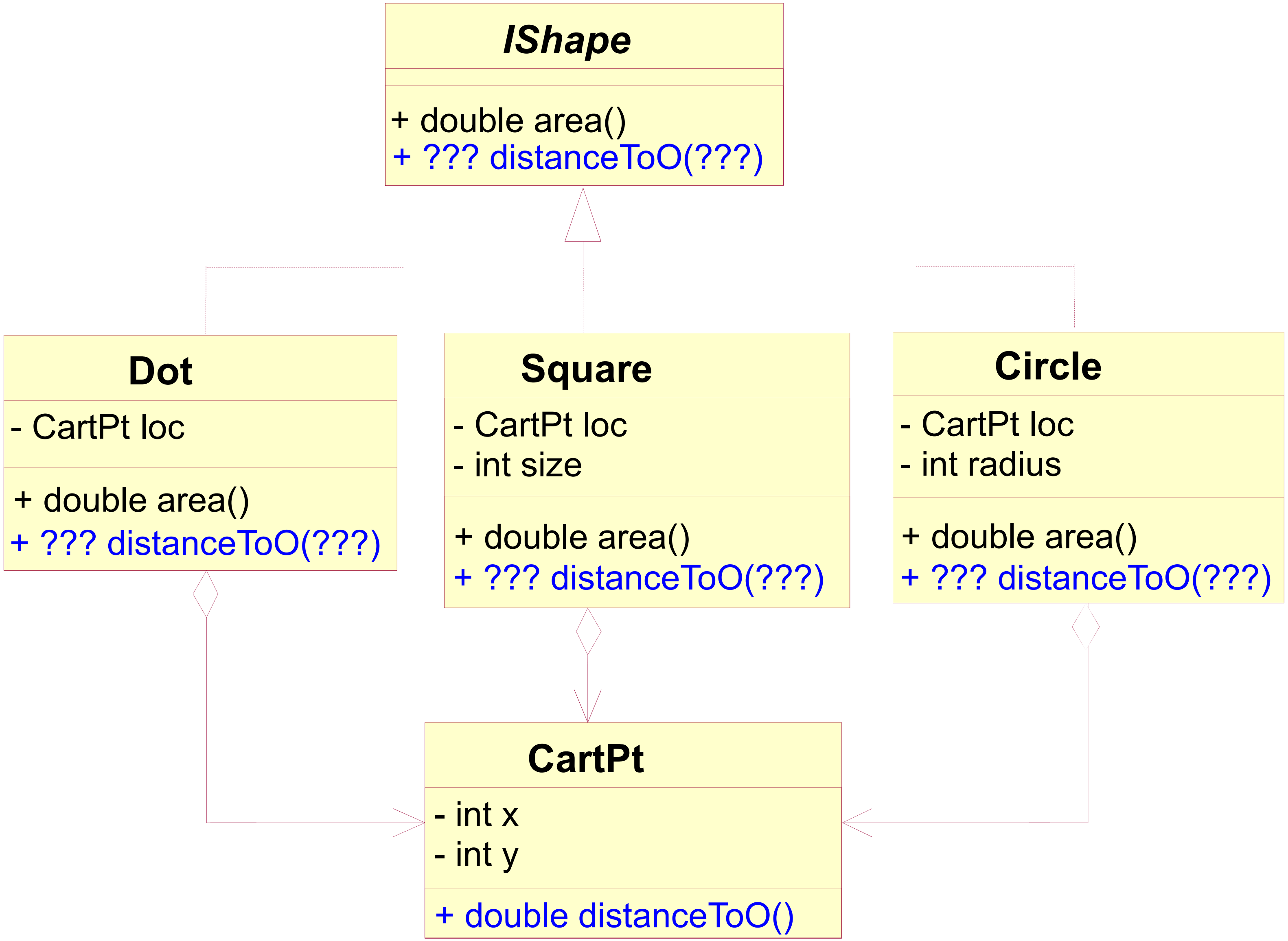
→ Polymorphism

2. Computing the distance of a shape to the origin

- What is the distance between a shape and the origin?



Add method to class diagram



`distanceToO()` purpose and signature

```java
public interface IShape {
   // compute area of AShape
   public double area();

   // to compute the distance of this shape to the origin
   public double distanceToO();
}
```

Same implement `distanceToO()`

```
// inside of Dot
public double distanceToO() {
    return this.loc.distanceToO();
}
```

```
// inside of Circle
public double distanceToO() {
    return this.loc.distanceToO();
}
```

```
// inside of Square
public double distanceToO() {
    return this.loc.distanceToO();
}
```

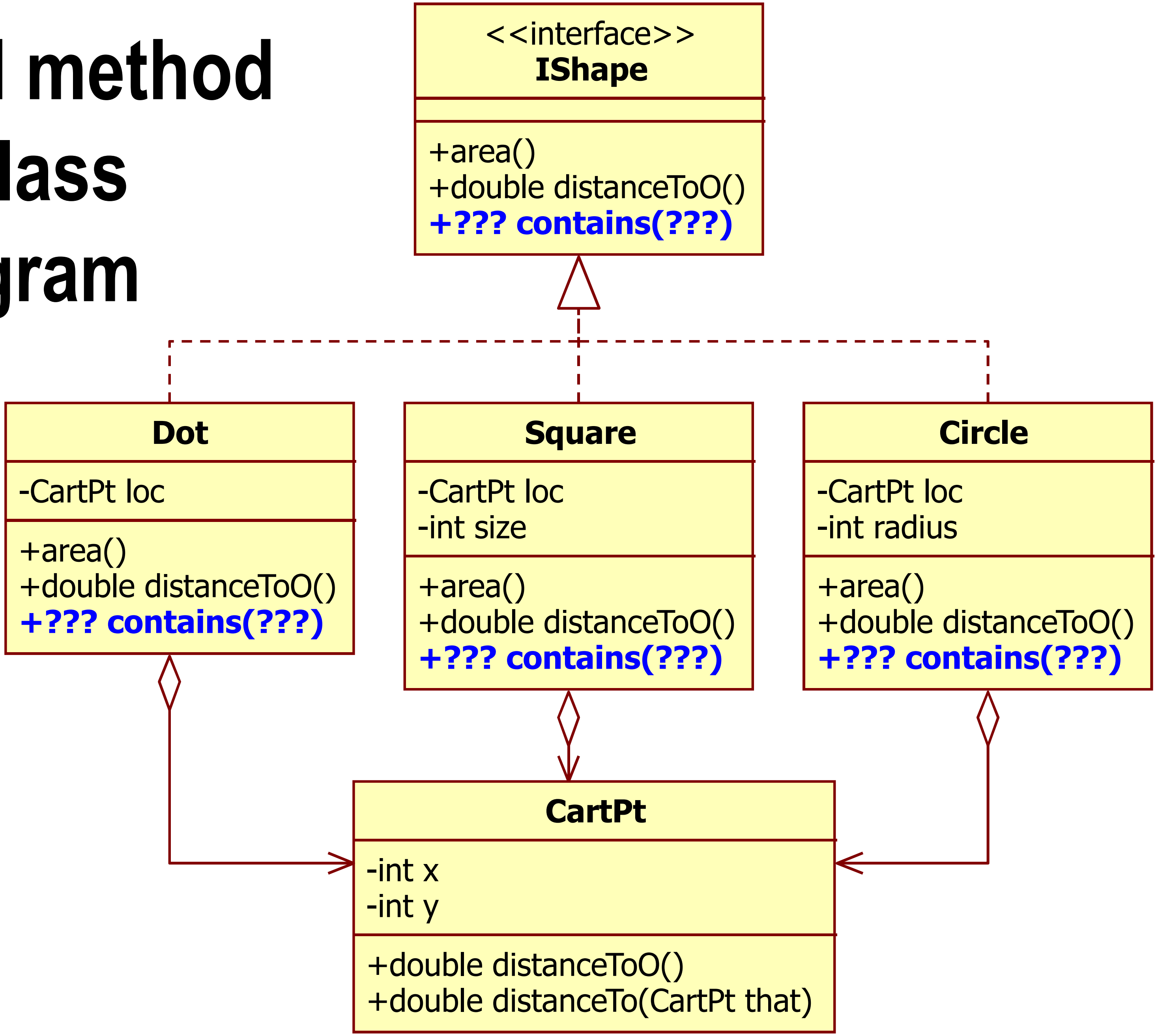
Unit Test

```
public void testDistanceToO() {
   assertEquals(new Dot(new CartPt(4, 3))
      .distanceToO(), 5.0, 0.001);

   assertEquals(new Square(new CartPt(4, 3), 30)
      .distanceToO(), 5.0, 0.001);

   assertEquals(new Circle(new CartPt(12, 5), 20)
      .distanceToO(), 13.0, 0.001);
}
```

3. Determining whether some point is inside the shape

- A given point is inside a DOT when its distance to this DOT is 0.
- a given point is inside a CIRCLE if its distance to the center of the CIRCLE is less than or equal the radius.
- A given point is inside a SQUARE when it is between two pairs of lines.

Add method to class diagram



contains() purpose and signature

```
public interface IShape {
   // compute area of AShape
   public double area();

   // to compute the distance of this shape to the origin
   public double distanceToO();

   // is the given point is within the bounds
   // of this shape
   public boolean contains(CartPt point);
}
```

Examples

```
• new Dot(new CartPt(100, 200))
     .contains(new CartPt(100, 200)) // should be true

• new Dot(new CartPt(100, 200))
     .contains(new CartPt(80, 220))  // should be false

• new Square(new CartPt(100, 200), 40)
     .contains(new CartPt(120, 220)) // should be true

• new Square(new CartPt(100, 200), 40)
     .contains(new CartPt(80, 220))  // should be false

• new Circle(new CartPt(0, 0),20)
     .contains(new CartPt(4, 3))     // should be true

• new Circle(new CartPt(0, 0), 10)
     .contains(new CartPt(12, 5))    // should be false
```

Domain Knowledge

- How to determine whether some point is inside the shape is a kind of knowledge called DOMAIN KNOWLEDGE
- To comprehend the domain knowledge, we sometimes look up in a book and in other situations we gather from experts

Implement `contains()`

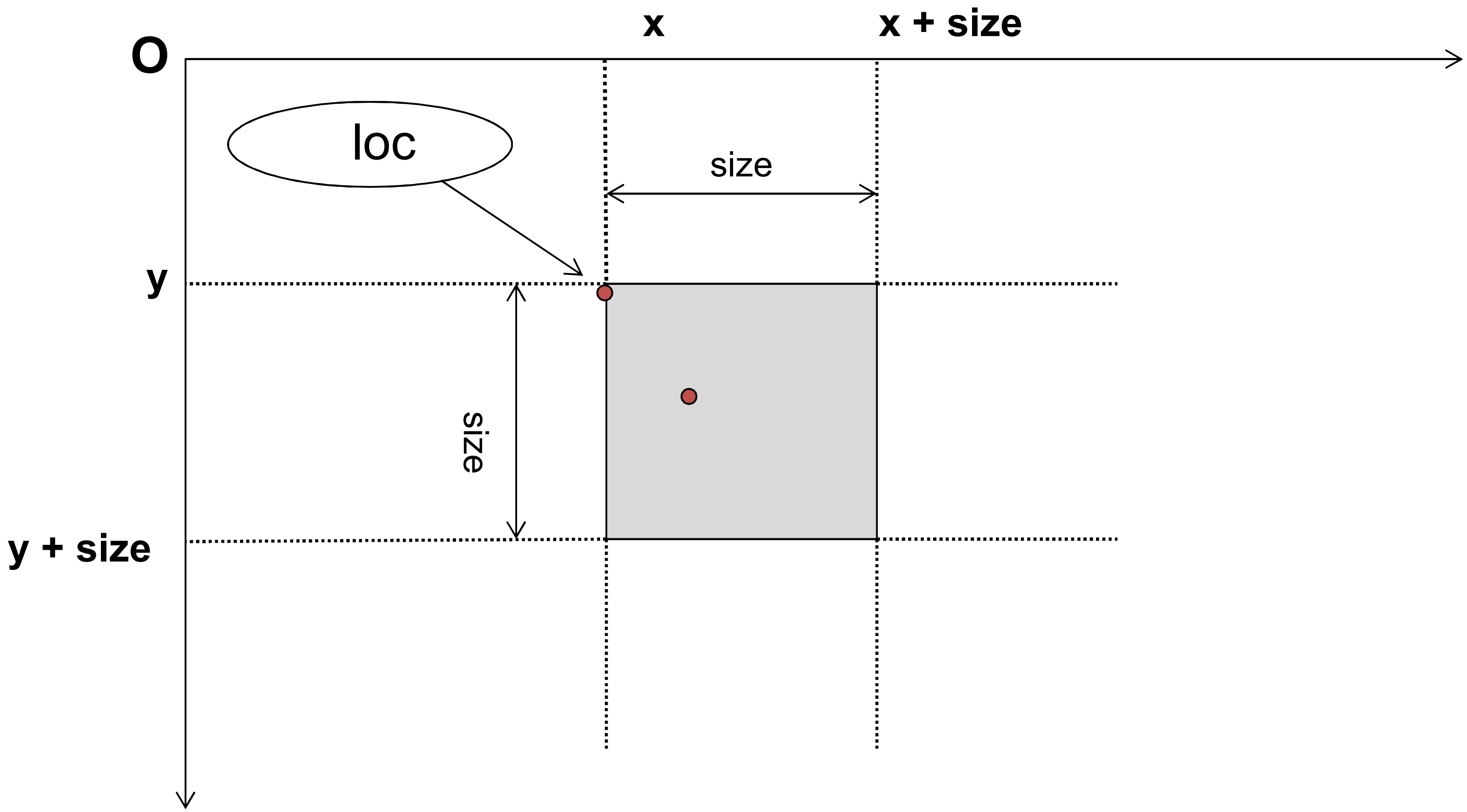
```java
// inside of Dot
public boolean contains(CartPt point) {
   return this.loc.distanceTo(point) == 0.0;
}
```

```java
// inside of Circle
public boolean contains(CartPt point) {
   return this.loc.distanceTo(point) <= this.radius;
}
```

distanceTo() in CartPt

```
public class CartPt {
   private int x;
   private int y;
   public CartPt(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public double distanceToO() {
      return Math.sqrt(this.x * this.x + this.y * this.y);
   }

   // compute distance of this point to another point
   public double distanceTo(CartPt that) {
      double diffX = this.x - that.x;
      double diffY = this.y - that.y;
      return Math.sqrt(diffX * diffX + diffY * diffY);
   }
}
```

Implement `contains()` in `Square`



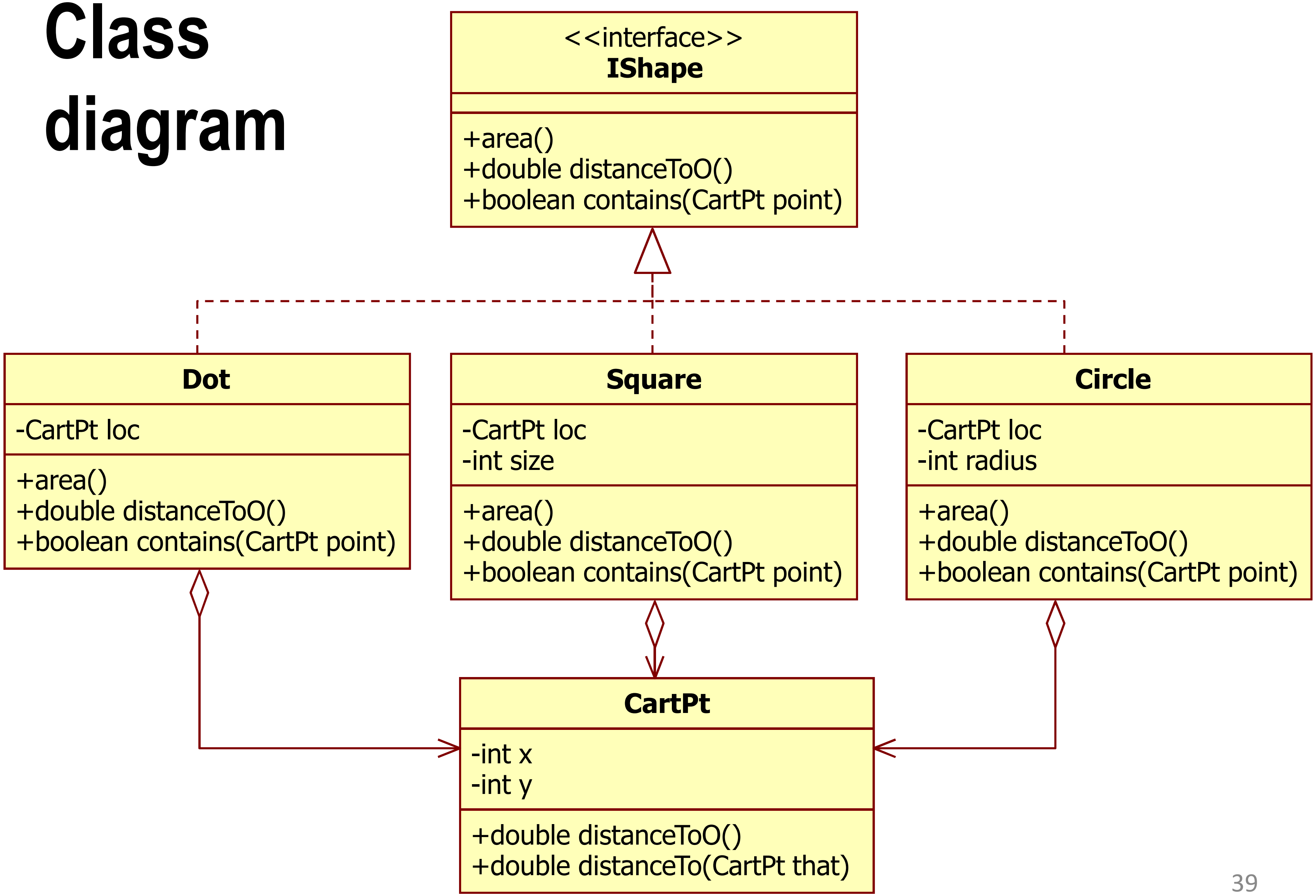
```java
// inside of Square
public boolean contains(CartPt point) {
   int thisX = this.loc.getX();
   int thisY = this.loc.getY();
   return this.between(point.getX(), thisX, thisX + this.size)
       && this.between(point.getY(), thisY, thisY + this.size);
}

//----------------------------------------------------------

private boolean between(int value, int low, int high) {
   return (low <= value) && (value <= high);
}
```

```java
// inside of CartPt
public class CartPt {
   private int x;
   private int y;
   public int getX() { return this.x; }
   public int getY() { return this.y; }
}
```

Unit test

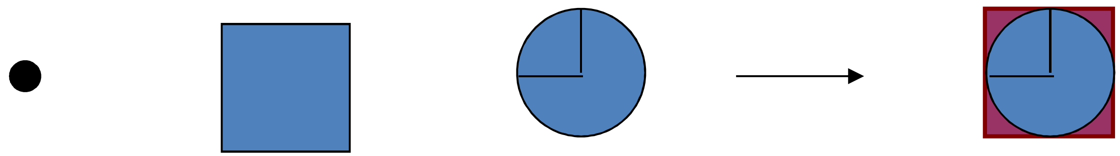
```
public void testContain(){
   assertTrue(new Dot(new CartPt(100, 200))
                  .contains(new CartPt(100, 200)));
   assertFalse(new Dot(new CartPt(100, 200))
                  .contains(new CartPt(80, 220)));

   assertTrue(new Square(new CartPt(100, 200),40)
                  .contains(new CartPt(120, 220)));
   assertFalse(new Square(new CartPt(100,  200),40)
                  .contains(new CartPt(80, 220)));

   assertTrue(new Circle(new CartPt(0, 0),20)
                  .contains(new CartPt(4, 3)));
   assertFalse(new Circle(new CartPt(0, 0),10)
                  .contains(new CartPt(12, 5)));
}
```

Class diagram

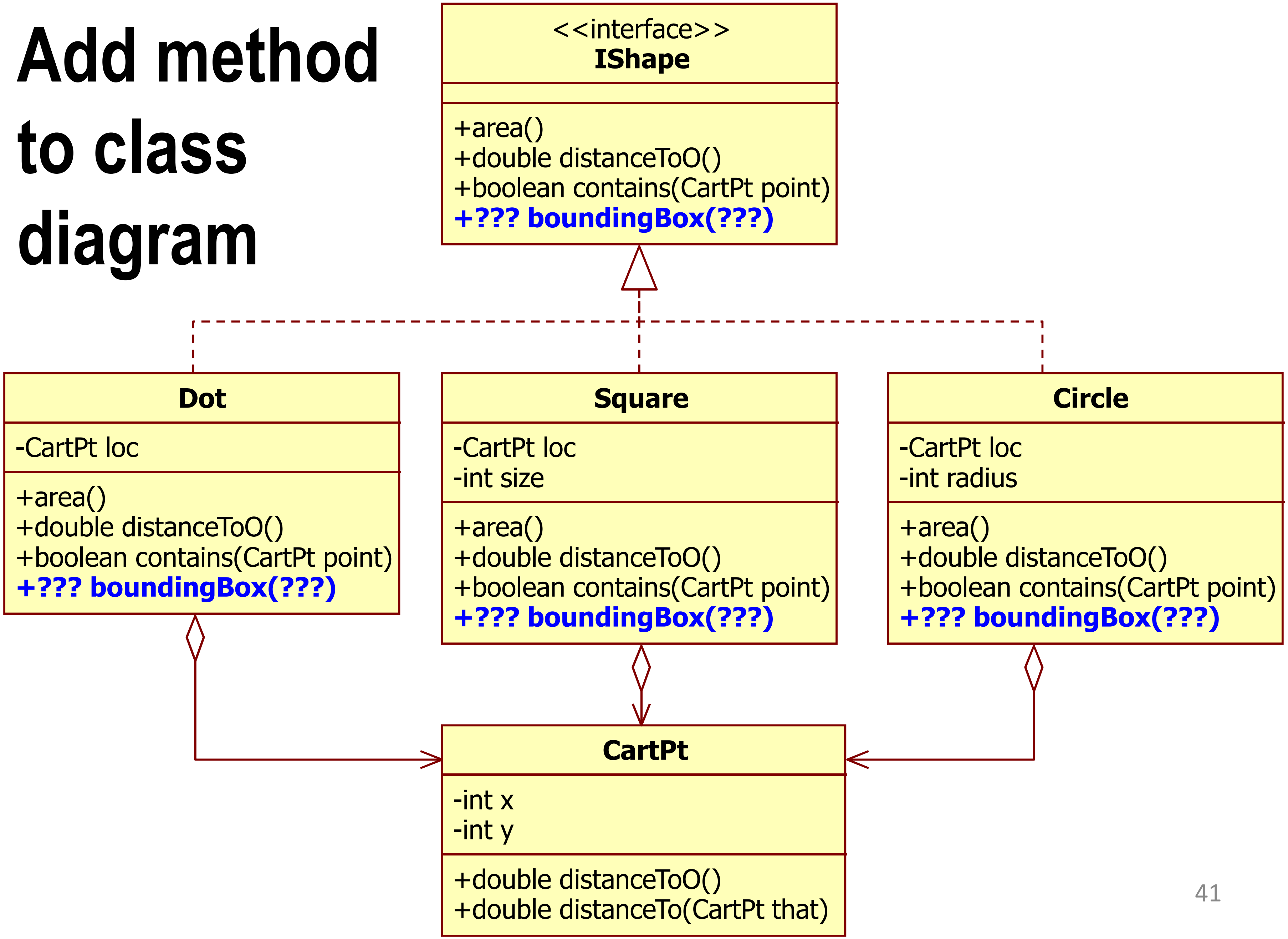


4. Computing the bounding box of a shape

- What is a Bounding Box?
  A bounding box is the smallest square that completely surrounds the given shape
  - The bounding box for a `Square` is the given square itself.
  - For a `Circle`, the bounding box is also a square,
    - its width and height are 2 * radius and
    - its northwest corner is one radius removed from the center of the circle in both directions.
  - The bounding box for a `Dot` is a square with no extent. Mathematicians call this idea a special case



Add method to class diagram



boundingBox purpose and signature

```
public interface IShape {
   // compute area of AShape
   public double area();

   // to compute the distance of this shape to the origin
   public double distanceToO();

   // is the given point is within the bounds
   // of this shape
   public boolean contains(CartPt point);

   // compute the bounding box for this shape
   public Square boundingBox();

}
```

Examples

```
new Dot(new CartPt(100, 100)).boundingBox()
// should be
new Square(new CartPt(100, 100), 0)

new Square(new CartPt(100, 100), 40).boundingBox()
// should be
new Square(new CartPt(100, 100), 40)

new Circle(new CartPt(0, 0), 20).boundingBox()
// should be
new Square(new CartPt(-20, -20), 40)
```

Implement boundingBox() method

inside of **Dot**

```
public Square boundingBox() {
   return new Square(this.loc, 0);
}
```

inside of **Square**

```
public Square boundingBox() {
   return new Square(this.loc, this.size);
}
```

`boundingBox` method in `Circle`

```
public Square boundingBox() {
   return new Square(this.loc.translate(
         -this.radius, -this.radius), this.radius * 2);
}
```

inside of `CartPt`

```
// translate this point to deltaX, deltaY distance
public CartPt translate(int deltaX, int deltaY) {
   return new CartPt(this.x + deltaX, this.y + deltaY);
}
```

Unit test

```
public void testBoudingBox(){
  assertTrue(new Dot(new CartPt(4, 3)).boundingBox()
             .equals(new Square(new CartPt(4, 3), 0)));

  assertTrue(new Circle(new CartPt(5, 5), 5).boundingBox()
             .equals(new Square(new CartPt(0, 0), 10)));

  assertTrue(new Square(new CartPt(4, 3), 30).boundingBox()
             .equals(new Square(new CartPt(4, 3), 30)));
}
```

equals method

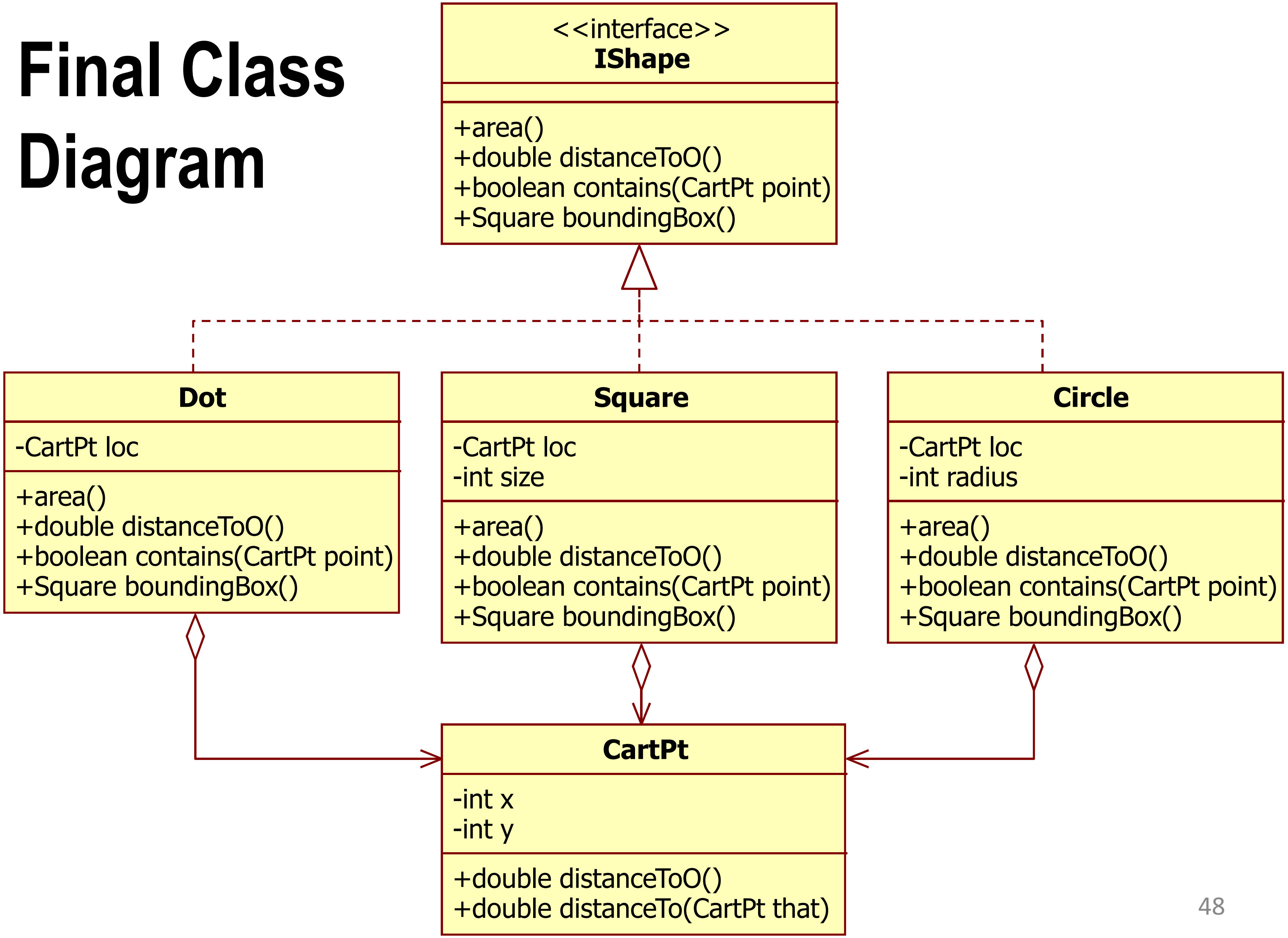
inside of Square class

```
public boolean equals(Object obj) {
   if (null==obj || !(obj instanceof Square))
      return false;
   else {
      Square that = (Square) obj;
      return (this.loc.equals(that.loc)
           && this.size == that.size);
   }
}
```

inside of CartPt class

```
public boolean equals(Object obj) {
  if (null==obj || !(obj instanceof CartPt))
    return false;
  else {
    CartPt that = (CartPt) obj;
    return (this.x == that.x) && (this.y == that.y);
  }
}
```

Final Class Diagram



Abstract with Class Common Data

Similarities in Classes

- **Similarities** among classes are common in unions.
- Several variants often contain **identical field** definitions. Beyond fields, variants also sometimes share identical or **similar method** definitions.
- Our first union is a representation of simple geometric shapes. All three classes implement ***IShape***. Each contains a ***CartPt*** typed field that specifies where the shape is located.

```
public class Dot
  implements IShape {
  private CartPt loc;
  ...
}
```

```
public class Square
  implements IShape {
  private CartPt loc;
  ...
}
```

```
public class Circle
  implements IShape {
  private CartPt loc;
  ...
}
```

Common Fields, Superclasses

- In OOP, classes cannot only inherit from interfaces, they can also inherit from other classes.
- This suggests the introduction of a **common superclass** of *Dot*, *Square*, and *Circle* that represents the commonalities of geometric shapes:

```
public class Shape implements IShape {
  private CartPt loc;
  public Shape(CartPt loc) {
     this.loc = loc;
  }
}
```

- Here the class represents two commonalities: the ***CartPt*** field and the **implements** specification

Inheritance

- If we make *Dot* an extension of *Shape*, it inherits the *CartPt* field and the obligation to implement *IShape*:

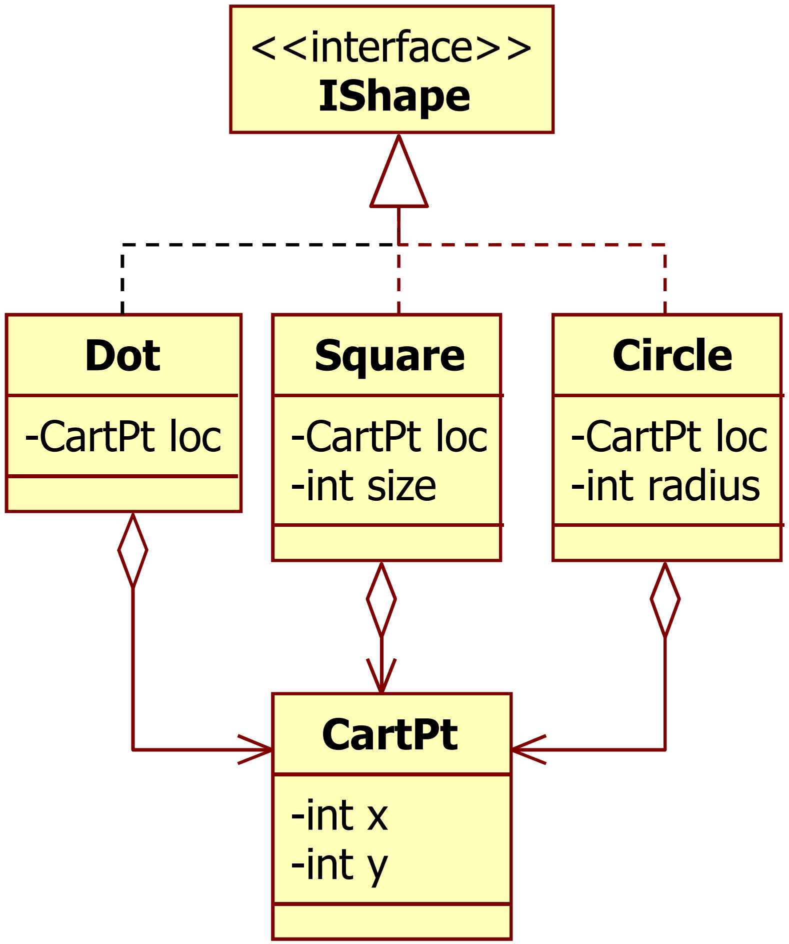
```
public class Dot
   extends Shape {
```

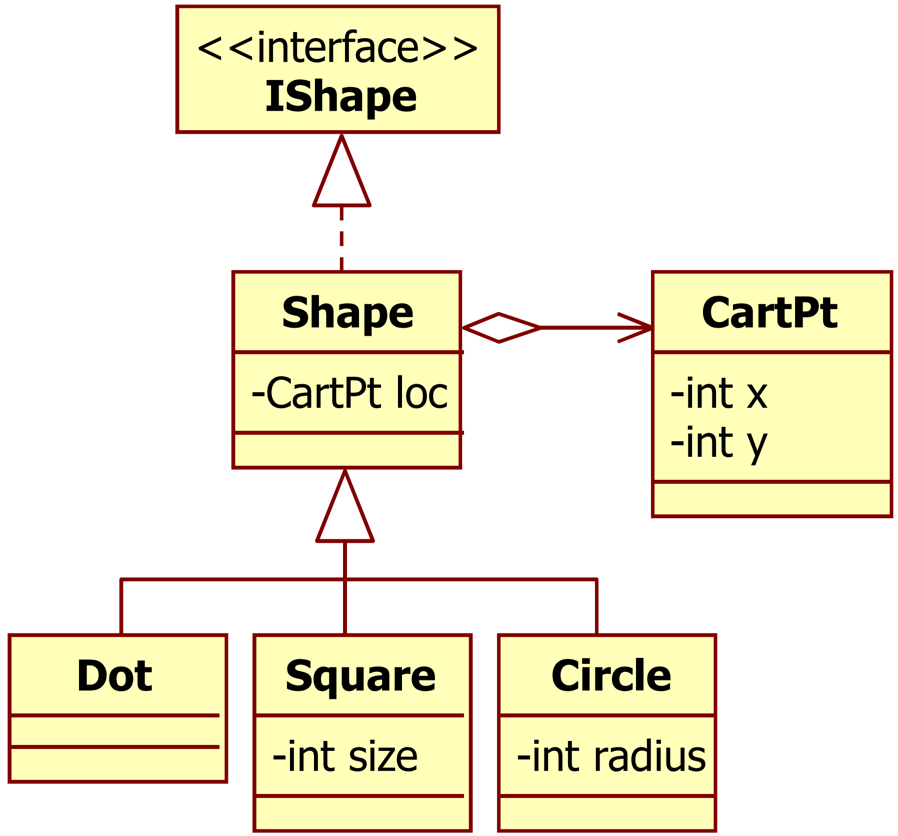
```
public class Square
   extends Shape {
```

```
public class Circle
   extends Shape {
```

- In general, the phrase *A* **extends** *B* says that *B* inherits all of *A*'s features (fields, methods, and **implements** obligations), which include those that *A* inherited.
  - *A* is the SUPERCLASS and *B* is the SUBCLASS;
  - we also say that *B* REFINES *A* or that *B* is DERIVED from *A.*
- Tend to attribute the commonalities to the superclass, and the differences to the subclasses.

Revised class diagram





Subclasses inherits the ***Loc*** field and obligation to implement ***IShape*** from ***Shape*** superclass

Java data definitions

```
public interface IShape {
}
```

```
public class Shape implements IShape {
  private CartPt loc;
}
```

```
public class Dot extends IShape {
}
```

```
public class Square extends IShape {
  private int size;
}
```

```
public class Circle extends IShape {
  private int radius;
}
```

Java data definitions

```
public interface IShape {
}
```

```
public class Shape implements IShape {
  private CartPt loc;
  public Shape(CartPt loc) {
    this.loc = loc;
  }
}
```

```
public class Dot
      extends IShape {
  public Dot(
        CartPt loc) {
    super(loc);
  }
}
```

```
public class Square
      extends IShape {
  private CartPt loc;
  private int size;
  public Square(
        CartPt loc,
        int size) {
    super(loc);
    this.size = size;
  }
}
```

```
public class Circle
      extends IShape {
  private CartPt loc;
  private int radius;
  public Circle(
        CartPt loc,
        int radius) {
    super(loc);
    this.radius = radius;
  }
}
```

Contructor of Shape

Two constructor format of **Square**

```
public abstract class Shape {
   protected CartPt loc;
}
```

access **protected** **loc** field inherits form **Shape**

```
public class Square extends Shape {
   private int size;
   public Square(CartPt loc, int size) {
      this.loc = loc;
      this.size = size;
   }
}
```

```
public abstract class Shape {
   private CartPt loc;
   public Shape(CartPt loc) {
      this.loc = loc;
   }
}
```

Can't to access private **loc** field, so Call constructor of **Shape** superclass

```
public class Square extends Shape {
   private int size;
   public Square(CartPt loc, int size) {
      super(loc);
      this.size = size;
   }
}
```

Final Java data definitions (format 1)

```
public abstract class AShape {
    protected CartPt loc;
}
```

```
public class Dot extends AShape {
    public Dot(CartPt loc) {
        this.loc = loc;
    }
}
```

```
public class Square extends AShape {
    private int size;
    public Square(CartPt loc, int size) {
        this.loc = loc;
        this.size = size;
    }
}
```

```
public class Circle extends AShape {
    private int radius;
    public Circle(CartPt loc, int radius) {
        this.loc = loc;
        this.radius = radius;
    }
}
```

Subclasses can access **protected loc** common field inherits form **AShape**

Final Java data definitions (format 2)

```
public abstract class AShape {
    private CartPt loc;
    public AShape(CartPt loc) {
        this.loc = loc;
    }
}
```

```
public class Dot extends AShape {
    public Dot(CartPt loc) {
        super(loc);
    }
}
```

```
public class Square extends AShape {
    private int size;
    public Square(CartPt loc, int size) {
        super(loc);
        this.size = size;
    }
}
```

```
public class Circle extends AShape {
    private int radius;
    public Circle(CartPt loc, int radius) {
        super(loc);
        this.radius = radius;
    }
}
```

Subclasses call constructor of **AShape** superclass

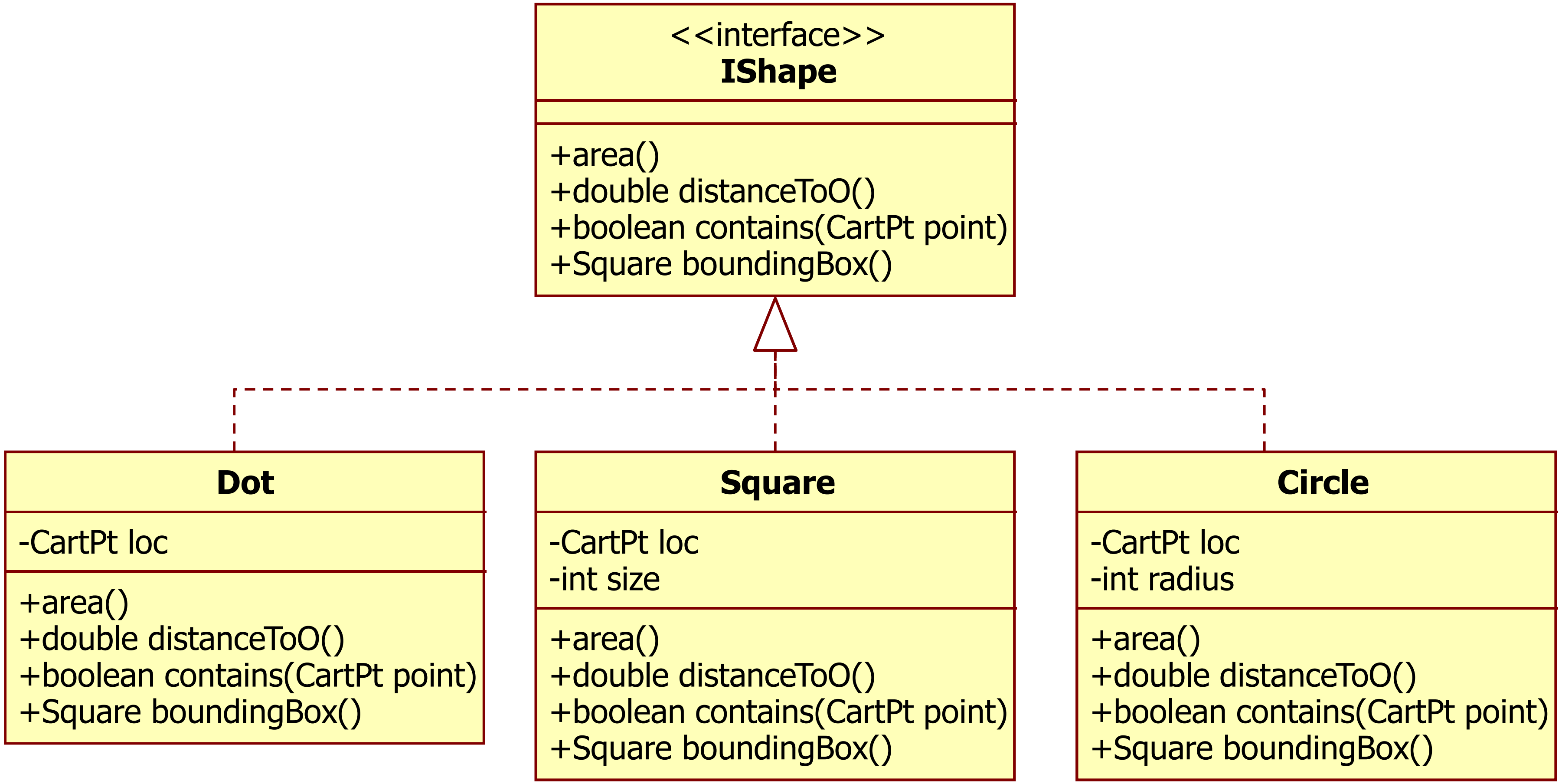
Test constructors

```
public class ShapeTest extends TestCase {
   public void testConstructor() {
      //test for class CartPt
      CartPt caPt1 = new CartPt(4, 3);
      CartPt caPt2 = new CartPt(5, 12);
      CartPt caPt3 = new CartPt(6, 8);

      // test  for class Dot
      AShape d1 = new Dot(caPt1);
      AShape d2 = new Dot(caPt2);
      AShape d3 = new Dot(caPt3);

      //test for class Circle
      AShape c1 = new Circle(caPt1, 5);
      AShape c2 = new Circle(caPt2, 10);
      AShape c3 = new Circle(caPt3, 12);

      //test for class Square
      AShape s1 = new Square(caPt1,5);
      AShape s2 = new Square(caPt3,10);
      AShape s3 = new Square(caPt3,12);
   }
}
```

Abstract Classes, Abstract Methods

Methods of union of shape

- We designed several methods for plain shapes, including *area()*, *distanceToO()*, *contains()*, and *boundingBox()*



Abstract Method

- When we add `AShape` to the class hierarchy for
  - collecting the commonalities of the concrete shapes and
  - implements **`IShape`** so that we wouldn't have to repeat this statement again for all the classes that extend **`AShape`**.
- So `AShape` must implement `area`, `distanceToO`, `contains`, and `boundingBox` what `IShape` specifies
- But implementing methods such as `area()` is different for each subclass.
- ***Solution***: Add `abstract` methods to the `abstract AShape` class

Abstract Method and Class

- An `abstract` method is just like a method signature in an interface, preceded by the keyword `abstract`.
- An `abstract` method ***doesn't have to define the method and shifts the responsibility to subclasses***
- It also makes no sense to create instances of the ***AShape*** class, because:
  - all instances of ***AShape*** are instances of either `Dot`, `Square`, or `Circle`
  - and it doesn't implement all of the methods from its interface yet.
- We make the entire **AShape** class `abstract`.

Abstract Method and Class

```java
public interface IShape {
  // compute area of AShape
  public double area();

  // to compute the distance of
  // this shape to the origin
  public double distanceToO();

  // is the given point is within
  // the bounds of this shape
  public boolean
      contains(CartPt point);

  // compute the bounding box
  // for this shape
  public Square boundingBox();
}
```

It also makes no sense to create instances of the ***AShape*** class, we make the entire class **`abstract`**

```java
public abstract class AShape {
  private CartPt loc;

  public abstract double area();
  public abstract double distanceToO();
  public abstract boolean
      contains(CartPt point);
  public abstract Square boundingBox();
}
```

An ***ABSTRACT*** method is a **method signature** and doesn't define the method, shifts the responsibility to subclasses

Subclasses define methods

```java
public class Dot
  extends AShape {
  ...

  public double area(){
    return 0;
  }

  public double
    distanceTo0() {
    return this.loc
        .distanceTo0();
  }
  ...
}
```

```java
public class Square
  extends AShape {
  private int size;
  ...

  public double area(){
    return this.size *
           this.size;
  }

  public double
    distanceTo0() {
    return this.loc
        .distanceTo0();
  }
  ...
}
```

```java
public class Circle
  extends AShape {
  private int radius;
  ...

  public double area(){
    return Math.PI *
          this.size *
          this.size;
  }

  public double
    distanceTo0() {
    return this.loc
        .distanceTo0();
  }
  ...
}
```

Lifting Methods, Inheriting Methods

- If all concrete subclasses of a union contain identical method definitions, we must lift them to the abstract class.
- Example: `distanceToO()` method in union of shapes

```java
public interface IShape {
  // to compute the distance of
  // this shape to the origin
  public double distanceToO();
  ...
}
```

```java
public abstract class AShape {
  private CartPt loc;
  public abstract double distanceToO();
}
```

```java
public class Dot
  extends AShape {
  public double
    distanceToO() {
    return this.loc
       .distanceToO();
  }
}
```

```java
public class Square
  extends AShape {
  public double
    distanceToO() {
    return this.loc
       .distanceToO();
  }
}
```

```java
public class Circle
  extends AShape {
  public double
    distanceToO() {
    return this.loc
       .distanceToO();
  }
}
```

Lifting `distanceToO()` method to the abstract `AShape` class

- The `distanceToO()` method was designed identically in all the variants of a union with a common `AShape` superclass.
- We can replace the `abstract` method in the superclass with the method definition from the variants:

```
public abstract class AShape {
  private CartPt loc;
  public double distanceToO() {
    return this.loc.distanceToO();
  }
}
```

- Now, we can delete the methods from `Dot`, `Square`, and `Circle` because the lifted `distanceToO` method in `AShape` is now available in all three subclasses

Keyword

- `interface`
  - Type of class
- `abstract` in front of a class indicates
  - The class is abstract.
  - there are no instances of this class.
    - can not use `new` operator to create an object of this class.
- `extends`
  - makes a refinement or an extension of the class, therefore the subclass INHERITS all of the superclass 's fields and methods.
- `implement`
  - makes a class is belong to interface type, and the class must implement all methods that the interface specifies.

`protected` attribute and method

- **`protected`**: ***the class itself and its subclass can access this attribute / method*.**
- **Q**: Review **`public`** and **`private`** modifiers for attribute and method.
  - **`public`**: Classes in all packages can see this attribute method.
  - **`private`**: the class itself can access this attribute and method.
  - **`None modifier`**: Classes in the same package can access attribute and method.

Modifiers for class

- **`None modifier`**: Classes in the same package can see this class.

- **`public`**: Classes in all packages can see this class.

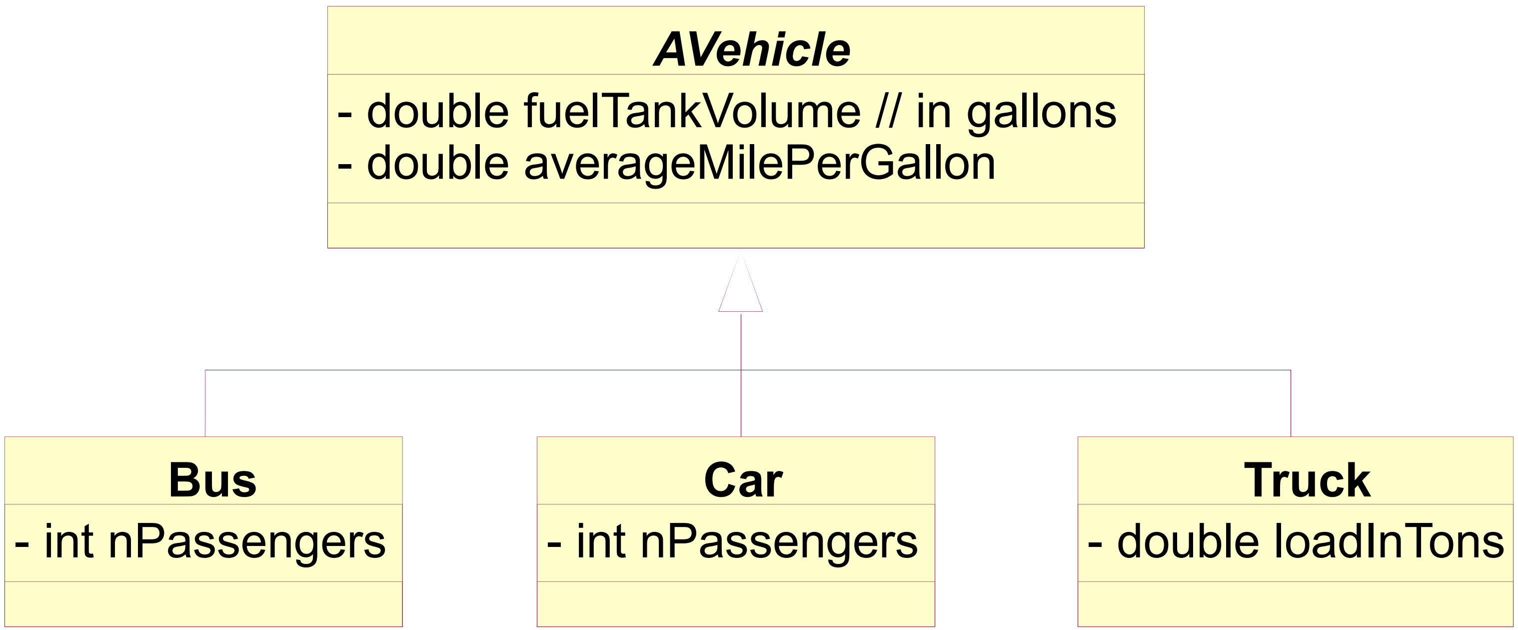
The Fuel Consumption Problem

Problem Statement

- A school district needs to manage the fuel consumption for its fleet of vehicles, which includes school busses, cars, and trucks.
- Each vehicle has a fuel tank of some size (in gallons).
- The district also knows the average mile-per-gallon consumption of fuel per vehicle.
- The fuel consumption for cars and busses depends on the number of passengers the vehicle carries; the fuel consumption of a truck increases with its load (in tons)

Class Diagram

- All vehicles has a fuel tank of some size (in gallons) and average mile-per-gallon consumption of fuel per vehicle.
- We define an **abstract** class `AVehicle` that contains the common fields of all vehicles: `fuelTankVolumn` and `averageMilePerGallon`.



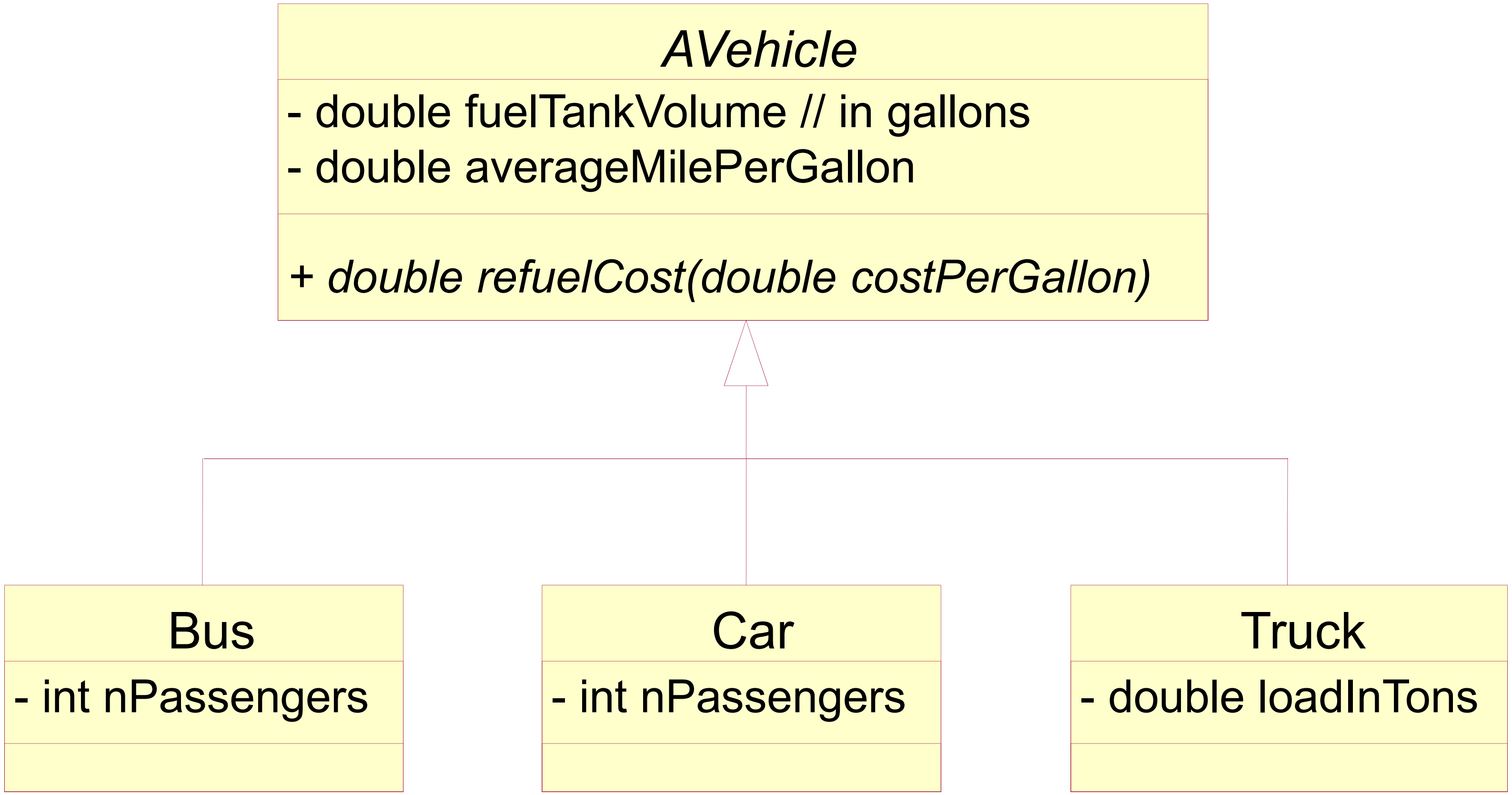
Examples

```
Car c1 = new Car(15.0, 25.0, 1);

Truck t1 = new Truck(120., 6.0, 0.5);

Bus b1 = new Bus(60.0, 10.0, 20);
```

Requirement

- The manager needs to know how much it costs to refuel a vehicle with empty fuel tank at the current fuel prices so that the district can create estimates for the gas budget
- **Q:** Improve the current class diagram and give more examples

Improved Class Diagram



More Examples

```
Bus b1 = new Bus(60.0, 10.0, 20);
Car c1 = new Car(15.0, 25.0, 1);
Truck t1 = new Truck(120., 6.0, 0.5);

b1.refuelCost(2.00)
// should be 120.0

c1.refuelCost(2.00)
// should be 30.0

t1.refuelCost(2.00)
// should be 240.0
```

Implementation of `refuelCost()`

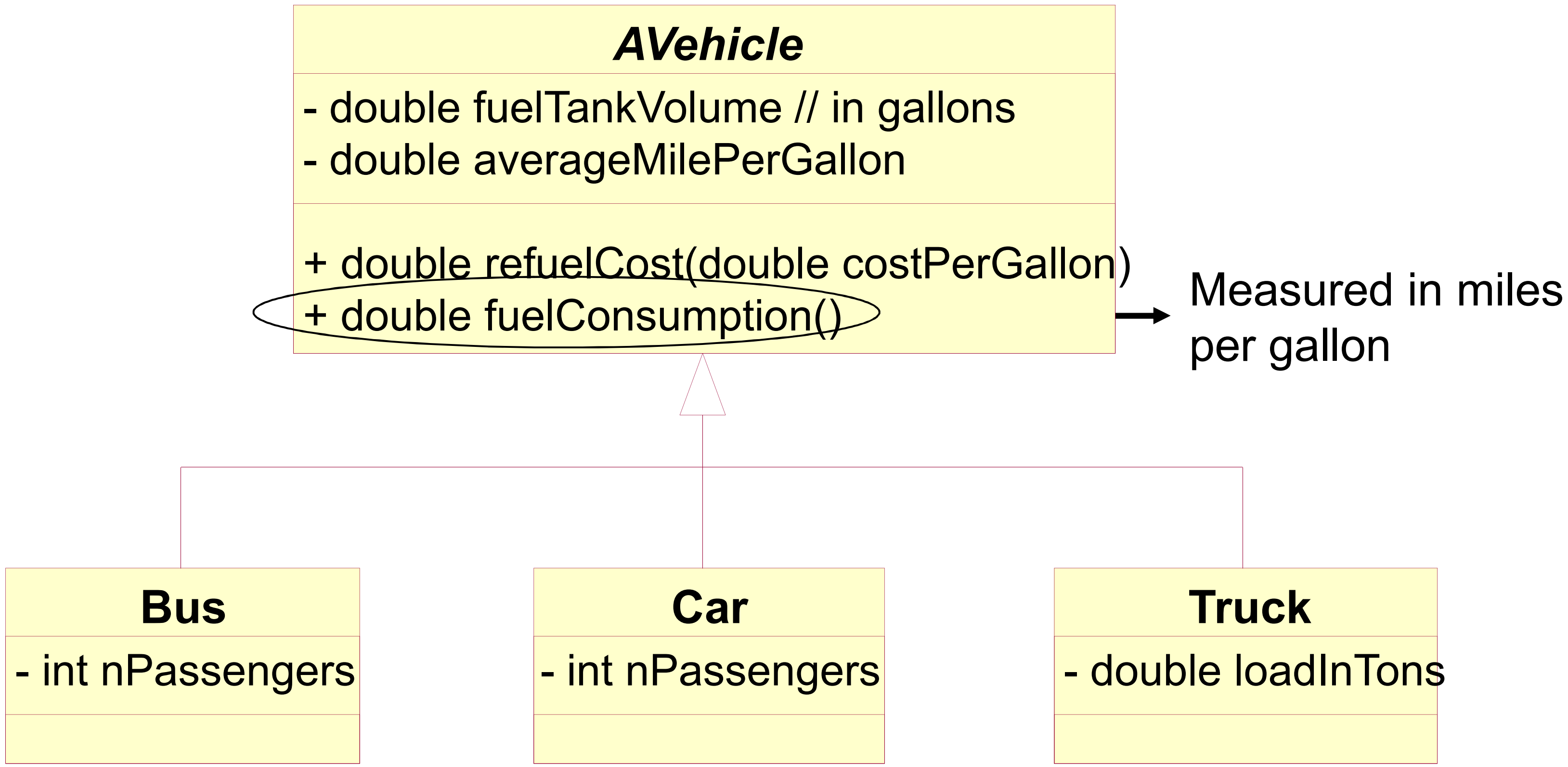
- Since three subclasses have the same implementation for `refuelCost()`, we could move this method to the superclass. This means `refuelCost()` now in the `AVehicle` class will not be abstract any more

```
public double refuelCost(double costPerGallon) {
   return costPerGallon * this.fuelTankVolume;
}
```

One More Requirement

- The manager wants to compute the projected fuel consumption for a specific trip so that the various departments can get a proper projection for the cost of the transportation services. For busses, the fuel consumption increases by 1% with each passenger. For a car, the fuel consumption increases by 10% with each passenger. For truck, one ton of load increases the fuel consumption by 5%
- Notice that the fuel consumption is measured in miles per gallon
- **Q:** Improve the current class diagram and give more examples

Improved Class Diagram



More Examples

```
Bus b1 = new Bus(60.0, 10.0, 20);
Car c1 = new Car(15.0, 25.0, 1);
Truck t1 = new Truck(120., 6.0, 0.5);

b1.fuelConsumption()
// should be 8

c1.fuelConsumption()
// should be 22.5

t1.fuelConsumption()
// should be 5.85
```

fuelConsumption() in AVehicle

```
public abstract class AVehicle {
   protected double fuelTankVolume; // in gallons
   protected double averageMilePerGallon;
   ...

   protected AVehicle(double fuelTankVolume,
                      double averageMilePerGallon) {
      this.fuelTankVolume = fuelTankVolume;
      this.averageMilePerGallon = averageMilePerGallon;
   }

   public double refuelCost(double costPerGallon) {
      return costPerGallon * this.fuelTankVolume;
   }

   public abstract double fuelConsumption();
}
```

`fuelConsumption()` implement

- In this case, all three methods are distinct from each other, and they must be defined for each specific concrete class of vehicles.

```
// inside Bus
public double fuelConsumption(){
   return this.averageMilePerGallon
         * (1 – 0.01*this.nPassengers);
}
```

```
// inside Car
public double fuelConsumption(){
   return this.averageMilePerGallon
            * (1 – 0.1*this.nPassengers);
}
```

```
// inside Truck
public double fuelConsumption(){
   return this.averageMilePerGallon
            * (1 – 0.05*this.loadInTons);
}
```

More Requirement

- Suppose the manager also wants to generate estimates about how far a vehicle can go, assuming it is freshly refueled.
  Design a method that can compute this estimate for each kind of vehicle.

howFar() in AVehicle

```
public abstract class AVehicle {
   protected double fuelTankVolume; // in gallons
   protected double averageMilePerGallon;
   ...
   protected Avehicle(double fuelTankVolume,
            double averageMilePerGallon) {
      this.fuelTankVolume = fuelTankVolume;
      this.averageMilePerGallon = averageMilePerGallon;
   }

   public double refuelCost(double costPerGallon) {
      return costPerGallon * this.fuelTankVolume;
   }

   public abstract double fuelConsumption();
   public abstract double howFar();
}
```

`howFar()` implement

- Since three subclasses have the same implementation for `howFar()`, we could move this method to the parent class.
  This means `howFar()` now in the `AVehicle` class will not be abstract any more

```
public double howFar() {
    return this.fuelTankVolume * this.fuelConsumption();
}
```