Points3D is a Java subtype of Points2D

Under some conditions, Points3D is also a true subtype of Points2D

Subtyping is defined only with respect to specifications – not implementations

- Informally, we often talk about whether an implementation of a specification satisfies the subtyping relationship
- In Java, this usually means interfaces and sometimes means abstract base classes
- In Java, extends is used to define subtypes and subclasses
B is a subtype of A means that a B can always be substituted for an A

- Any property guaranteed by supertype must be guaranteed by subtype (*true subtyping*)
- The subtype is permitted to strengthen and add properties
- Anything provable about an A is provable about a B
- If an instance of subtype is treated purely as supertype – only supertype methods and fields queried – then result should be consistent with an object of the supertype being manipulated

- A Points3D can always be treated as a Points2D
- Points3D adds a property – the z-coordinate
- Invariants over Points2D define the semantics of the type and hold over Points3D – the following invariants on Points3D consider only the components taken from Points2D (that is, treating the subtype purely as its supertype)

  Points2D(α, β).x() = α  
  Points2D(α, β).y() = β  
  Points3D(α, β, γ).x() = α  
  Points3D(α, β, γ).y() = β

- The semantics of Points3D can arbitrarily define semantics of added properties

  Points3D(α, β, γ).z() = γ

  would be the likely expectation

- But the following, albeit weird, would not compromise the subtyping relationship

  Points3D(α, β, γ).z() = α+β+γ
Java subtypes ≠ true subtypes

public class CartesianTwoDPoints
    implements Points2D {
    double xcoord, ycoord;
    public CartesianTwoDPoints(double a, double b) {
        xcoord = a;
        ycoord = b;
    }
    @Override public double x() { return xcoord; }
    @Override public double y() { return ycoord; }
}

public class CartesianThreeDPoints
    implements Points3D {
    double xcoord, ycoord, zcoord;
    public CartesianThreeDPoints(double a, double b, double c) {
        xcoord = a; ycoord = b; zcoord = c;
    }
    @Override x() and y() like in CartesianTwoDPoints
    @Override public double z() { return zcoord; }
}

• These implementations satisfy the true subtyping relationship
  • ex: CartesianThreeDPoints(α, β, γ).y() = β
• Why no subclassing in this example?
Java subtypes ≠ true subtypes

Here, `CartesianThreeDPoints` is a Java subtype of `Points2D` but does not satisfy the true subtyping relationship.
Two questions in class

• What if Points2D defined a distance method (return the distance between two points)?
  – Points3D could redefine the distance method as long as all points in the plane have the same distance as they would if considered as Points2D.

• What if there was a printPoint method in Points2D that printed (say) “x=? y=?” where the question marks show the actual values?
  – The question becomes one of semantics – if the format is constrained by the specification of Points2D, then it would have to be adhered to (perhaps by only printing the x and y coordinates); if it wasn’t constrained, but said something like, “It prints the value of the coordinates,” then Points3D would have more choice.
Subtyping vs. subclassing

```
public class PolarTwoDPoints implements Points2D {
    double r, theta;
    public PolarTwoDPoints(double a, double b) {
        r = Math.sqrt(a*a+b*b);
        theta = 2*Math.atan(b/(a+r));
    }
    @Override
    public double x() { return r*Math.cos(theta); }
    @Override
    public double y() { return r*Math.sin(theta); }
}
```

```
public class AltThreeDPoints extends PolarTwoDPoints implements Points3D {
    double z;
    public AltThreeDPoints(double a, double b, double c) {
        super(a, b);
        z = c;
    }
    @Override
    public double z() { return z; }
}
```

- **AltThreeDPoints** is a subclass of **PolarTwoDPoints** and a Java subtype of **Points2D**
- For this implementation, **AltThreeDPoints** is also a true subtype of **Points2D** – the invariants for Points2D are maintained
- This is true even though an **AltThreeDPoints** is stored as \((r, \theta, z)\)
What if...

• ...we wanted to restrict Points2D to be only in the first quadrant? \( x \geq 0 \land y \geq 0 \)

• What semantics do we want? Here are two possibilities
  – If the client tries to construct a Points2D outside the first quadrant, throw an exception
  – Take the absolute value of \( x \) and of \( y \) before constructing the point
public class FirstQuadrant2DPoints implements Points2D {
    double xcoord, ycoord;
    public FirstQuadrant2DPoints(double a, double b) throws NotFirstQuadrant {
        if ((a <= 0) || (b <= 0)) {
            throw new NotFirstQuadrant();
        }
        xcoord = a;
        ycoord = b;
    }
    @Override
    public double x() {
        return xcoord;
    }
    @Override
    public double y() {
        return ycoord;
    }
}
public class FirstQuadrant2DPoints implements Points2D {
    double xcoord, ycoord;
    public FirstQuadrant2DPoints(double a, double b) {
        xcoord = Math.abs(a);
        ycoord = Math.abs(b);
    }
    @Override
    public double x() {
        return xcoord;
    }
    @Override
    public double y() {
        return ycoord;
    }
}

• Notice, there is no subtyping here (as yet)
• We are still changing the semantics of Points2D (without changing the interface directly)
  Points2D(α, β).x() = |α|
  Points2D(α, β).y() = |β|
public class FirstQuadrant3DPoints implements Points3D {
    double xcoord, ycoord, zcoord;
    public FirstQuadrant3DPoints(double a, double b, double c) throws NotFirstQuadrant {
        if ((a <= 0) || (b <= 0)) {
            throw new NotFirstQuadrant();
        }
        xcoord = a;
        ycoord = b;
        zcoord = c;
    }
    @Override
    public double z() {
        return zcoord;
    }
}

• Now FirstQuadrant3DPoints and FirstQuadrant2D points satisfy the Points3D is a subtype of Points2D relationship
• It could also choose to throw a NotFirstQuadrant exception if z was negative without compromising the subtype relationship
public class FirstQuadrant3DPoints implements Points3D {
    double xcoord, ycoord, zcoord;
    public FirstQuadrant3DPoints(double a, double b) {
        xcoord = Math.abs(a);
        ycoord = Math.abs(b);
        zcoord = c;
    }
    @Override
    public double z() { return zcoord; }
}

• Would this FirstQuadrant3DPoints and FirstQuadrant2DPoints satisfy the Points3D is a subtype of Points2D relationship?