Mutability

- `Non-final` variables are **mutable** – they can be reassigned
- `final` variables cannot be reassigned, but they may point to objects whose *fields* can change value
- Objects of an immutable class cannot look different at different times
  - Fields are final and contain primitives or other immutable classes
  - Methods are pure functions
  - Behaves like a primitive type – changing values implies variable reassignment to a new object
- Mutable classes encapsulate state that *can* change
  - Methods may be procedures or functions with side effects

Example: immutable vs. mutable Point

```
public class Point {
    private final double x;
    private final double y;
    public double x() { return x; }
    public double y() { return y; }
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Point shifted(double dx,
                         double dy) {
        return new Point(x + dx, y + dy);
    }
}
```

Pure function

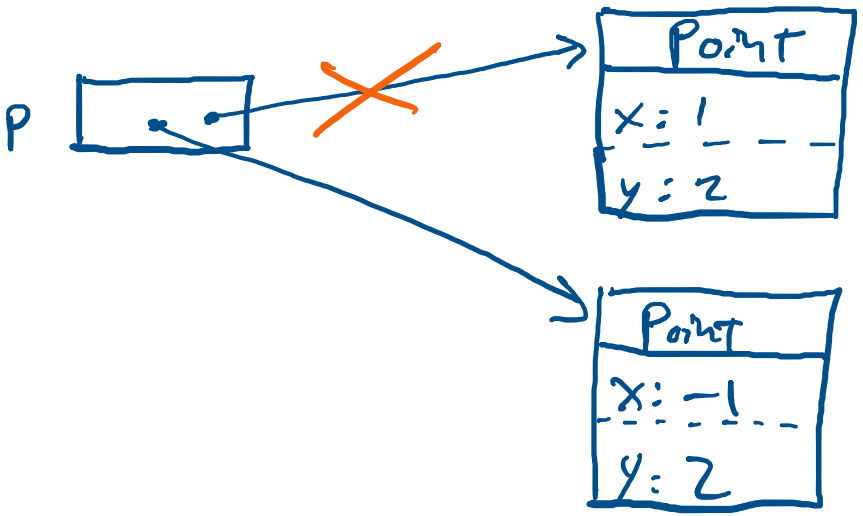
```
public class MPoint {
    private double x;
    private double y;
    public double x() { return x; }
    public double y() { return y; }
    public MPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void shift(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
```

Procedure

Client code: immutable vs. mutable

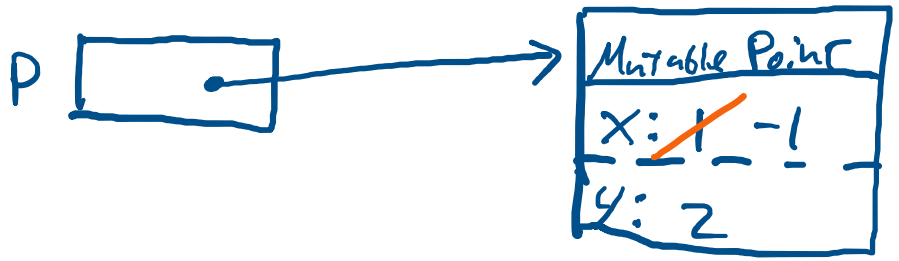
```
Point p = new Point(1, 2);
// Move point left
p = p.shifted(-2, 0);
```

↖ Pure function

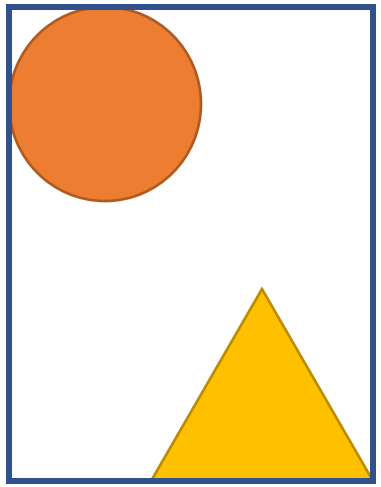



```
                    final
^ MPoint p = new MPoint(1, 2);
  // Move point left
  p.shift(-2, 0);
```

↖ Procedure



Bounding box



Brainstorm: what behavior should a BoundingBox provide?

Count off groups: A, B, C, D

- A & B: **Write interface** for an *immutable* `BoundingBox`
- C & D: **Write interface** for a *mutable* `MutableBoundingBox`
  - Identify preconditions and postconditions where appropriate

- All: **Write client code** to do the following using your interface:
  - Move a `BoundingBox b` so that it is centered on the origin
  - Write a function that returns the area of intersection of two `BoundingBoxes`

Representation

- **Identify at least two** different *representations* (sets of fields) that could be used in a class to implement your interface

- What *class invariants* should be imposed on each representation?

- For each method in your interface, would one representation lead to an easier implementation than the other?

Discussion: representations

- What do most representations have in common?
- Given one representation, could you convert it to another?
  - Could you do so using only the interface?

Implementation: overall objective

- A: Implement an immutable BoundingBox using a two-point (opposite corners) representation
- B: Implement an immutable BoundingBox using a center & extents (width & height) representation
- C: Implement a MutableBoundingBox using a two-point (opposite corners) representation
- D: Implement a MutableBoundingBox using a center & extents (width & height) representation

Implementation I

- Create class and add fields
- Use IDEA to populate method stubs
  - Notice @Override annotation
- Write a `checkInvariant()` method that returns true if the class invariants are satisfied
- Write a constructor for your class
  - Identify and assert preconditions
  - Assert `checkInvariant()` at the appropriate place

Implementation I (continued)

- Implement width, height, centroid, and contains
- Implement shift
- Test client code to shift box to origin

- Class discussion
  - Which representations were easier for which methods?

Implementation II

- Implement `toString()`
- Implement area, intersection
- Test client code to find area of intersection

- Class discussion
  - Which representation was easier?
  - How important is mutable vs. immutable for the client?

Discussion

- Which methods can be implemented by only calling other methods (no additional field access required)?
- Suppose this was done in the interface (possible using "default" methods); what are some pros and cons?