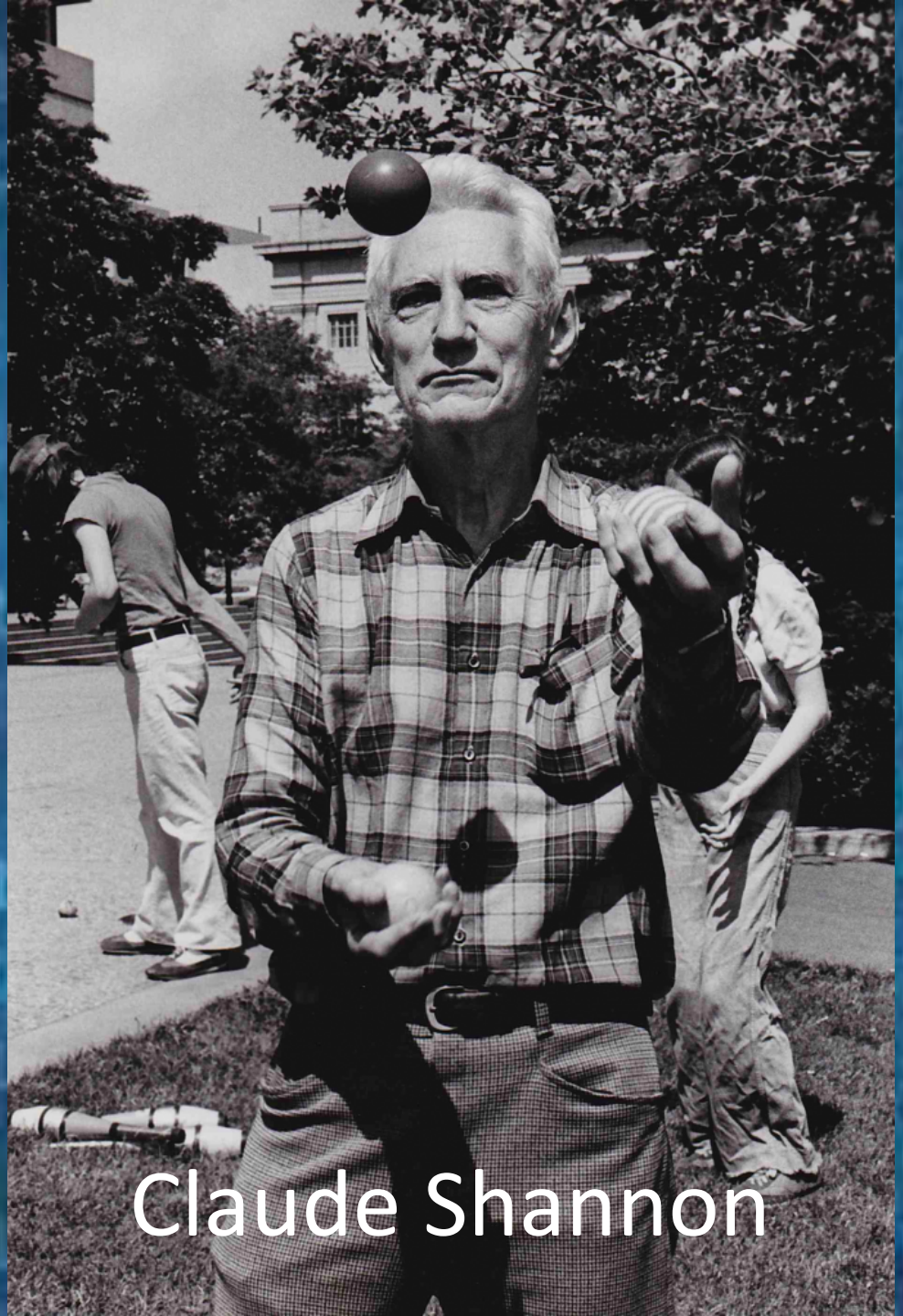
CSCI 1102 Computer Science 2

Meeting 16: Thursday 3/25/2021

More on Order & Equality; Huffman Coding



Claude Shannon

Reprinted with corrections from *The Bell System Technical Journal,*
Vol. 27, pp. 379–423, 623–656, July, October, 1948.

A Mathematical Theory of Communication

By C. E. SHANNON

INTRODUCTION

THE recent development of various methods of modulation such as PCM and PPM which exchange bandwidth for signal-to-noise ratio has intensified the interest in a general theory of communication. A basis for such a theory is contained in the important papers of Nyquist[1] and Hartley[2] on this subject. In the present paper we will extend the theory to include a number of new factors, in particular the effect of noise in the channel, and the savings possible due to the statistical structure of the original message and due to the nature of the final destination of the information.

The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point. Frequently the messages have *meaning*; that is they refer to or are correlated according to some system with certain physical or conceptual entities. These semantic aspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual

Order & Equality in Java

== versus equals

- The == operator compares bits;
- new Integer(6) == new Integer(6) is false because the two heap-allocated integers are stored in separate locations; == is comparing addresses.
- == is fine for values of type int, long, short, ...

== and boolean

- boolean a, b;
- a == b or a != b are ok;
- Instead of a == true, just use a
- Instead of a == false, just use !a

Don’t use == on Strings or Floats

- “Mei” == “Mei” is true
- new String(“Mei”) == new String(“Mei”) is false
- new String(“Mei”).equals(“Mei”) is true
- Trouble with == and floats discussed below.

equals in Java

- equals – should define an equivalence relation, for items x and y of the same type.
- Defined in class Object so it is inherited by every reference type
- When defining a new ADT usually want to @Override equals

equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

compareTo in Java

- compareTo – should define a total order;

- compareTo should be *consistent* with equals: for items x and y of the same type, it should be the case that:

  x.compareTo(y) == 0 if and only if x.equals(y)

Ordered Colors

```java
@Override
public boolean equals(Object other) {
  if (other == null || other.getClass() != this.getClass())
    return false;
  return this.compareTo((OrderedColor) other) == 0;
}

// Ordered by amount of red
public int compareTo(OrderedColor other) {
  if (other == null)
    throw new RuntimeException("compareTo - you gave me null, wanted color");
  return Integer.compare(this.getRed(), other.getRed());
}
```

The usual proviso on Mutation

x.equals(y) is true

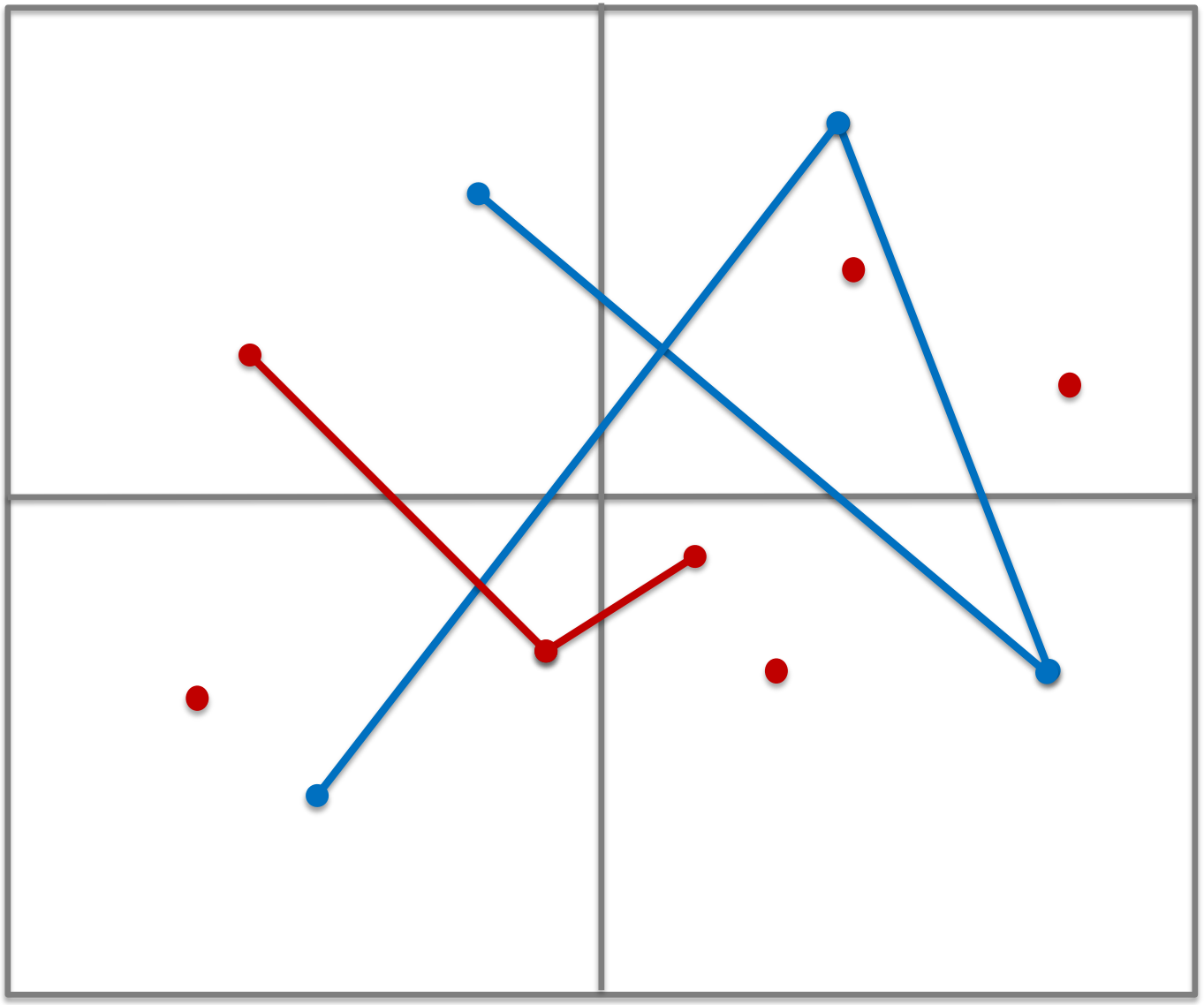
…

p(x)

…

x.equals(y) is false, p mutated x

Ordered Points & Lines



Ordered Points & Lines

- Can't use == operator on floating point numbers:

$$0.1 + 0.2 =/= 0.3$$

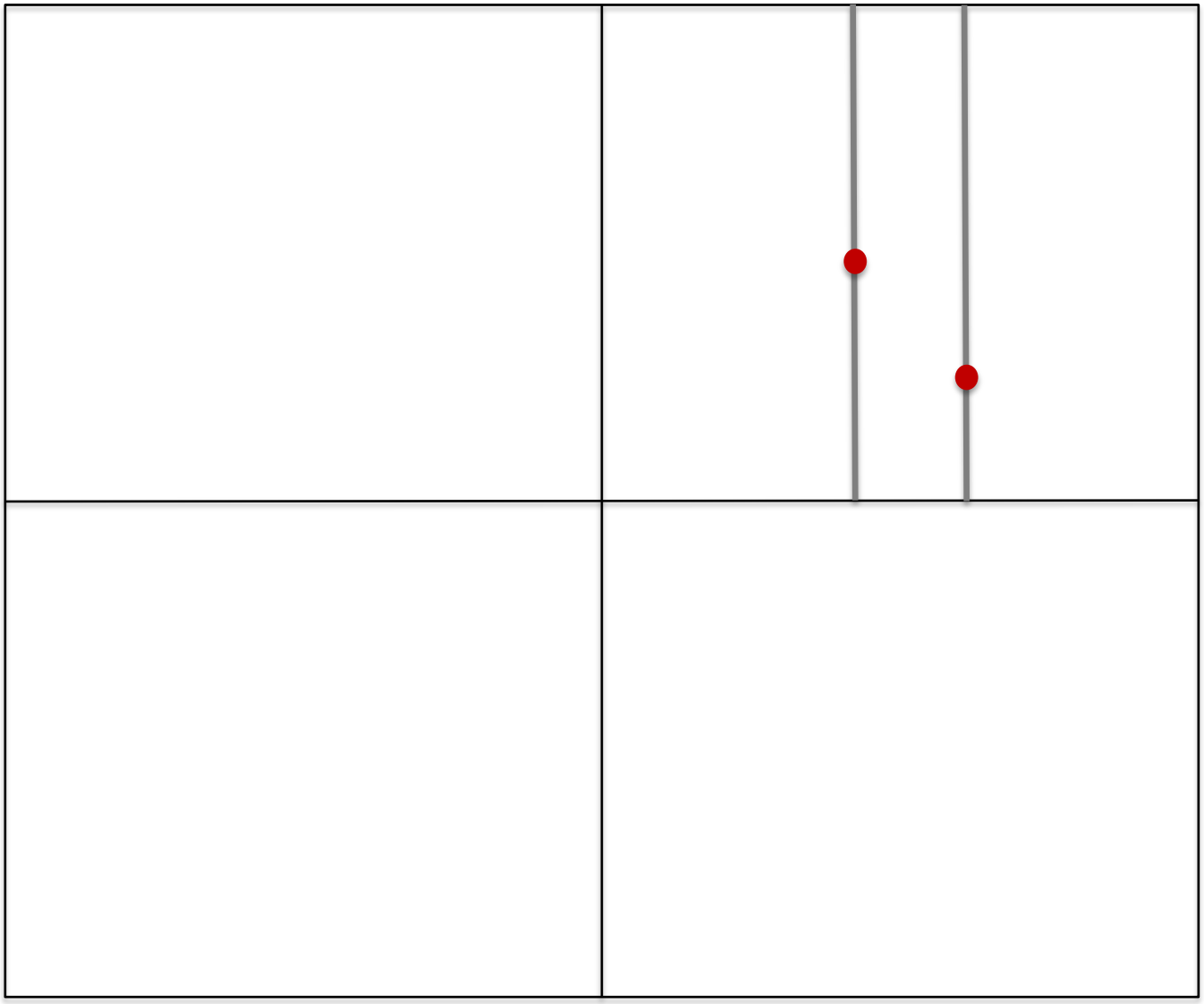
```
private static final double EPSILON = 1.0 * Math.pow(10.0, -6.0);

private boolean closeEnough(double a, double b) {
  return Math.abs(a - b) < EPSILON;
}
```

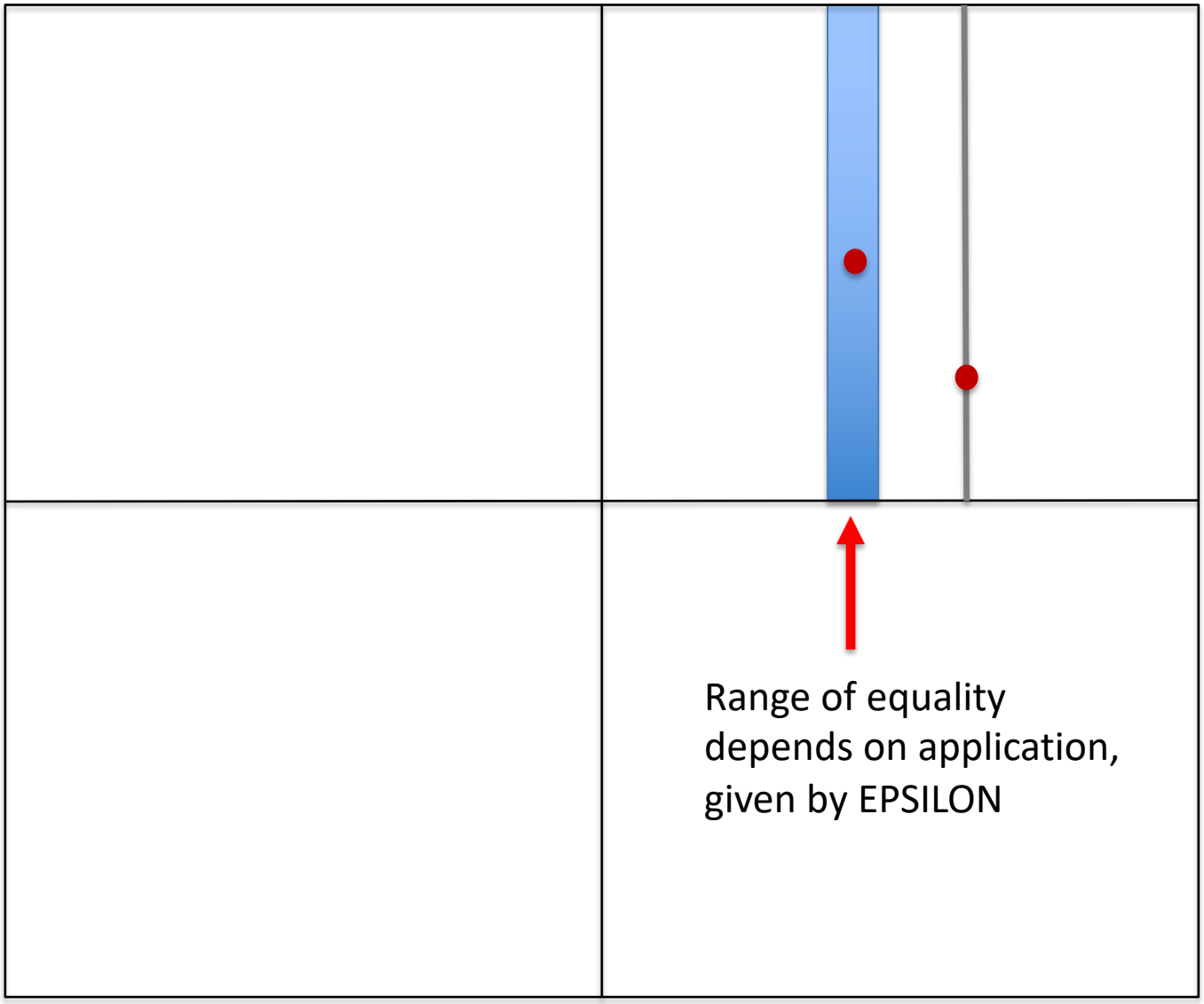
Ordered Points & Lines

- Build Line orderings on Point ordering
- Change Point ordering, all orderings remain synchronized

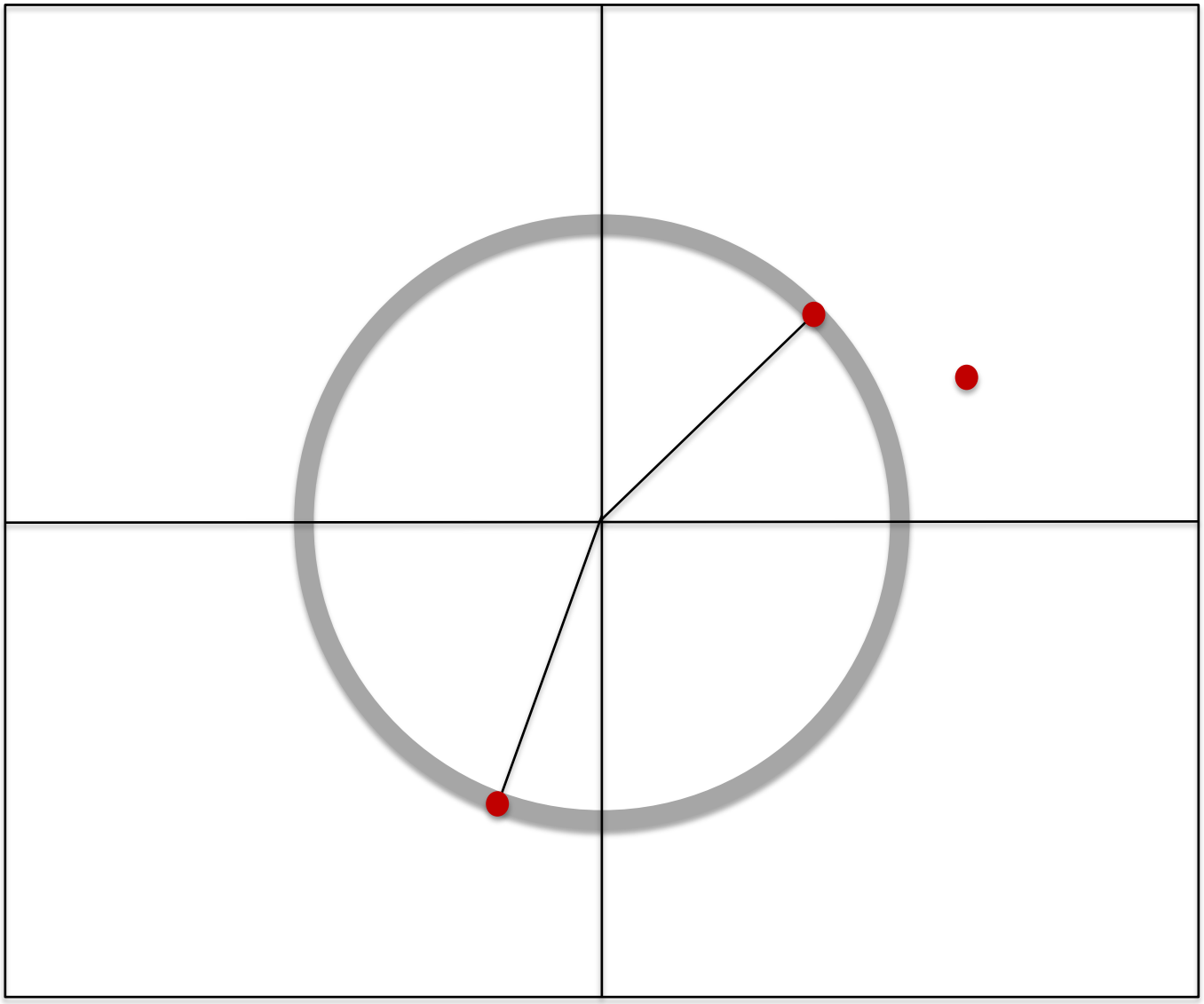
E.g., Order Points by x component alone



E.g., Order Points by x component alone



E.g., Order Points by distance from origin

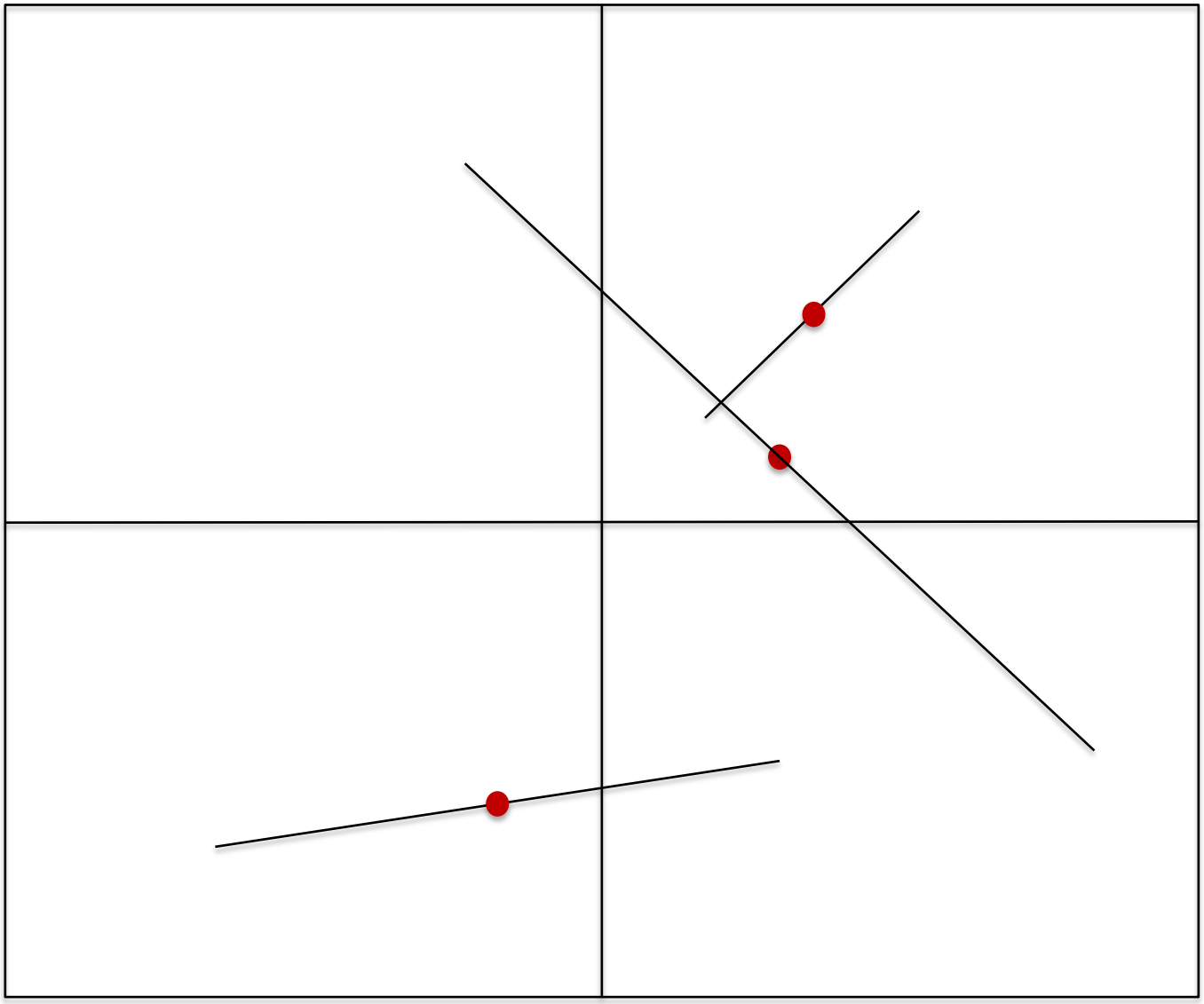


```java
private boolean closeEnough(double a, double b) {
  return Math.abs(a - b) < EPSILON;
}

// Our natural ordering compares distance to the origin. This equates Points
// in rings around the origin.
//
public int compareTo(Point other) {
  if (other == null)
    throw new RuntimeException("compareTo: you provided null, I needed a Point");
  if (closeEnough(this.distance(), other.distance()))
    return 0;
  else
    return Double.compare(this.distance(), other.distance());
}

// The most direct way to ensure that equals is consistent with compareTo.
@Override
public boolean equals(Object other) {
  if (other == null || (other.getClass() != this.getClass()))
    return false;
  return this.compareTo((Point) other) == 0;
}
```

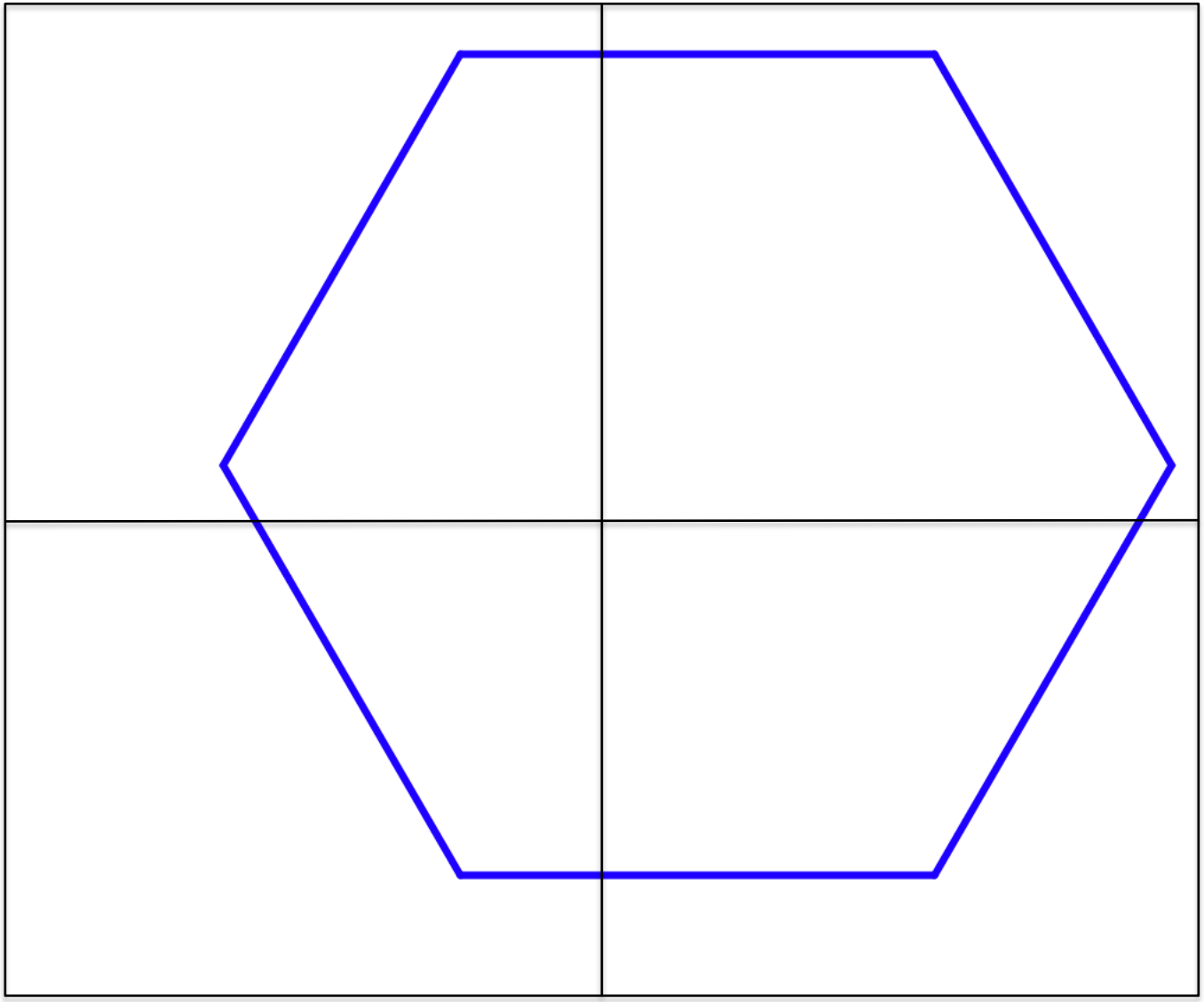
Ordering Single-Segment Lines by Midpoint



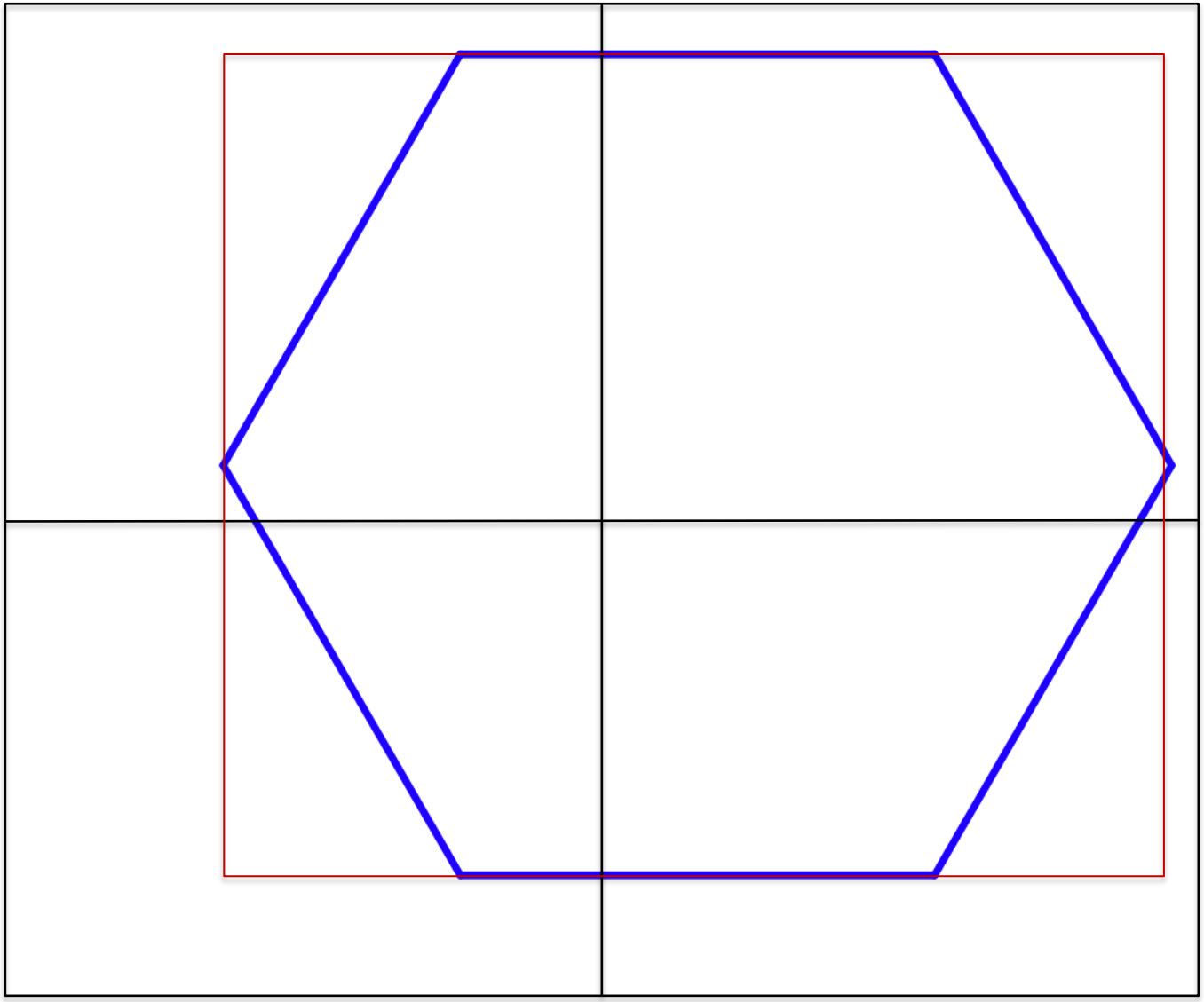
```java
// A natural ordering comparing midpoints.
//
public int compareTo(Line other) {
  if (other == null)
    throw new RuntimeException("compareTo: you provided null, I needed a Line");
  return this.midpoint().compareTo(other.midpoint());
}

@Override
public boolean equals(Object other) {
  if (other == null || other.getClass() != this.getClass())
    return false;
  return this.compareTo((Line) other) == 0;
}
```

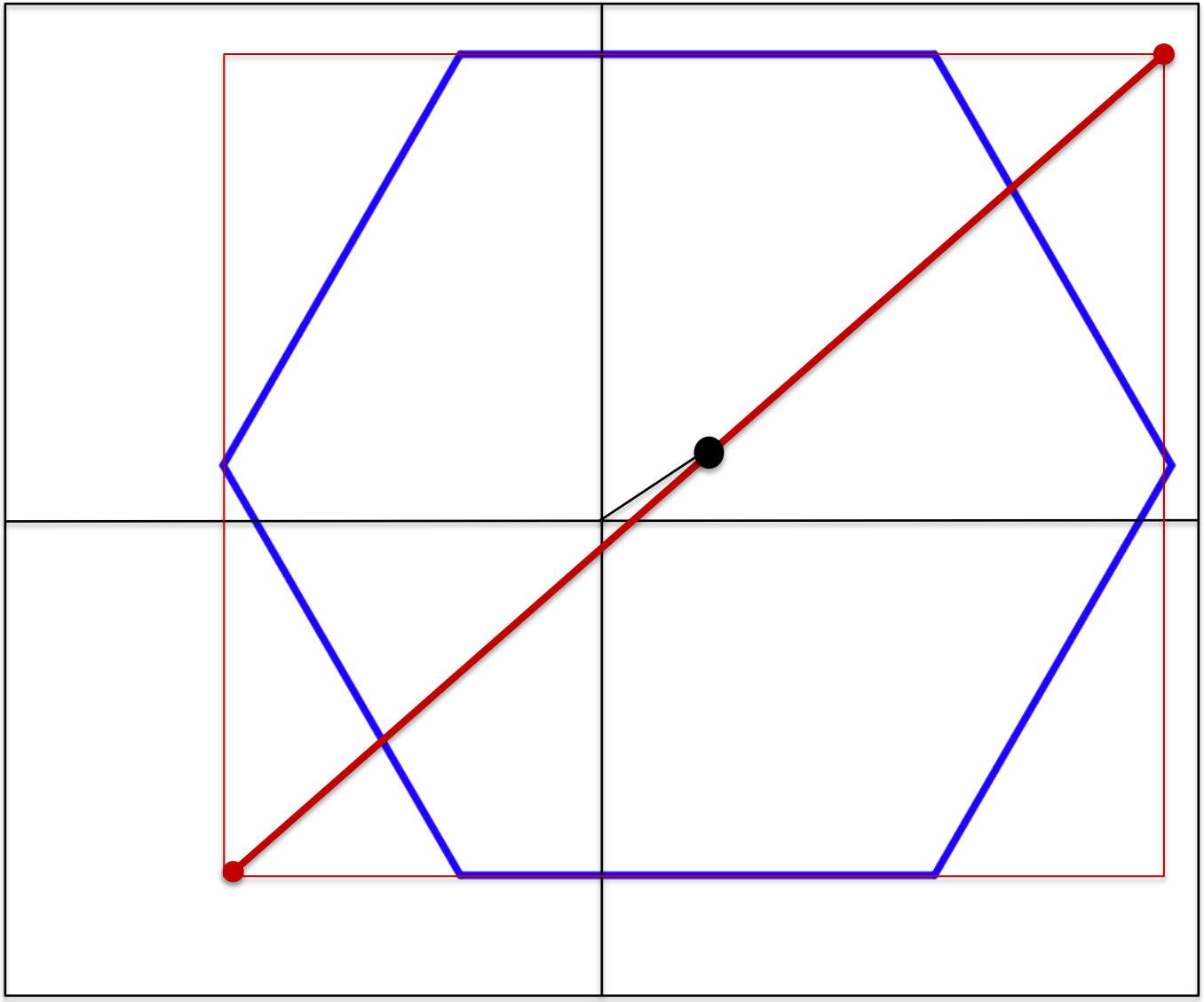
Ordering Multi-Segment Lines by Midpoint



Bounding Box



Bounding Box Midpoint



```java
public Point center() {
  return new LineC(this.min(), this.max()).midpoint();
}

// compare by center of bounding boxes
public int compareTo(MultiSegmentLine other) {
  if (other == null)
    throw new RuntimeException("null MultiSegmentLine");
  return this.center().compareTo(other.center());
}

@Override
public boolean equals(Object other) {
  if (other == null || other.getClass() != this.getClass())
    return false;
  return this.compareTo((MultiSegmentLine) other) == 0;
}
```

An aside on Streams in Java

- Introduced in Java version 1.8, java.util.stream
- “A sequence of elements supporting sequential and parallel aggregate operations.”
- “Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source.”

Using Streams to find the Bounding Box

```
private Point min() {
  double x =
          Arrays.stream(this.lines) Stream<Line>
                .map(Line::min) Stream<Point>
                .mapToDouble(Point::getX) DoubleStream
                .reduce((x1, x2) -> Double.compare(x1, x2) < 0 ? x1 : x2) OptionalDouble
                .getAsDouble();
  double y =
          Arrays.stream(this.lines) Stream<Line>
                .map(Line::min) Stream<Point>
                .mapToDouble(Point::getY) DoubleStream
                .reduce((y1, y2) -> Double.compare(y1, y2) < 0 ? y1 : y2) OptionalDouble
                .getAsDouble();
  return new PointC(x, y);
}
```