The **hashCode** method

hashCode

- if you override **`equals`** you *must* override **`hashCode`**
  - otherwise, the hashed containers won't work properly
    - recall that we did not override **`hashCode`** for **`SimplePoint2`**

```
// client code somewhere
SimplePoint2 p = new SimplePoint2(1f, -2f);

HashSet<SimplePoint2> h = new HashSet<>();
h.add(p);
System.out.println( h.contains(p) );        // true

SimplePoint2 q = new SimplePoint2(1f, -2f);
System.out.println( h.contains(q) );        // false!
```


[notes 3.3.5]

Arrays as Containers

- suppose you have a list of unique **SimplePoint2** points
  - how do you compute whether or not the list contains a particular point?
    - write a loop to examine every element of the list

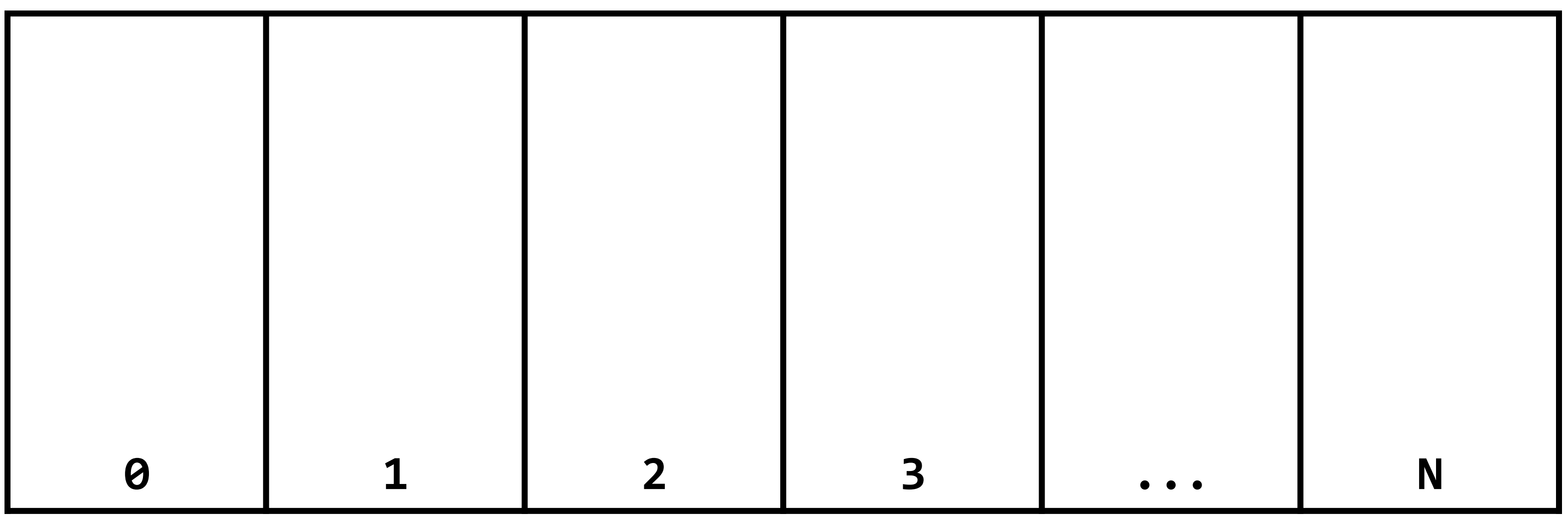
```
public static boolean
       hasPoint(SimplePoint2 p, List<SimplePoint2> points) {

  for( SimplePoint2 point : points ) {
    if (point.equals(p)) {
      return true;
    }
  }
  return false;
}
```

- called *linear search* or *sequential search*
  - doubling the length of the array doubles the amount of searching we need to do
- if there are **`n`** elements in the list:
  - best case
    - the first element is the one we are searching for
      - 1 call to **`equals`**
  - worst case
    - the element is not in the list
      - n calls to **`equals`**
  - average case
    - the element is somewhere in the middle of the list
      - approximately **`(n/2)`** calls to **`equals`**

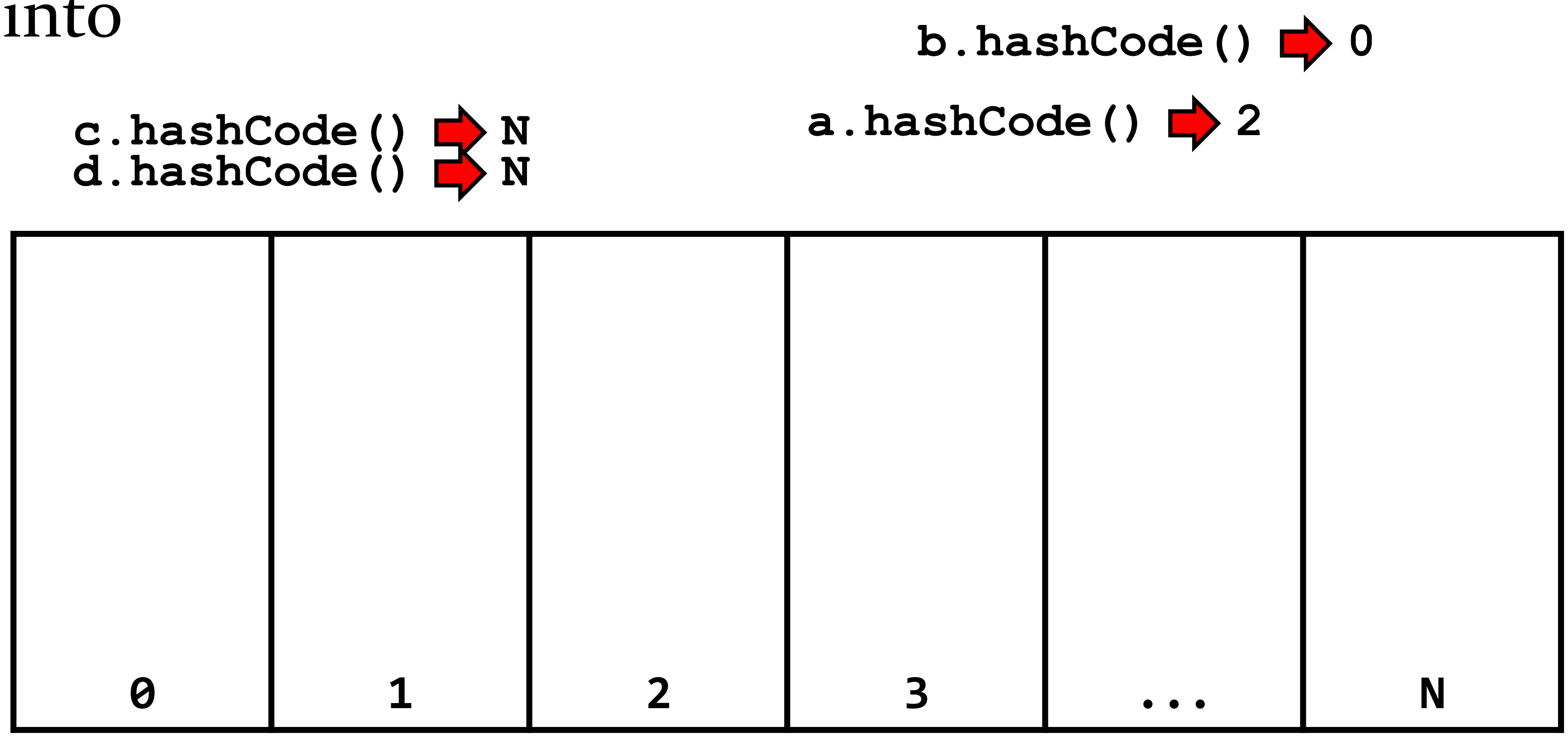
Hash Tables

- you can think of a hash table as being an array of buckets where each bucket holds the stored objects



Insertion into a Hash Table

- to insert an object **a**, the hash table calls **`a.hashCode()`** method to compute which bucket to put the object into



means the hash table takes the hash code and does something to it to make it fit in the range **`0—N`**

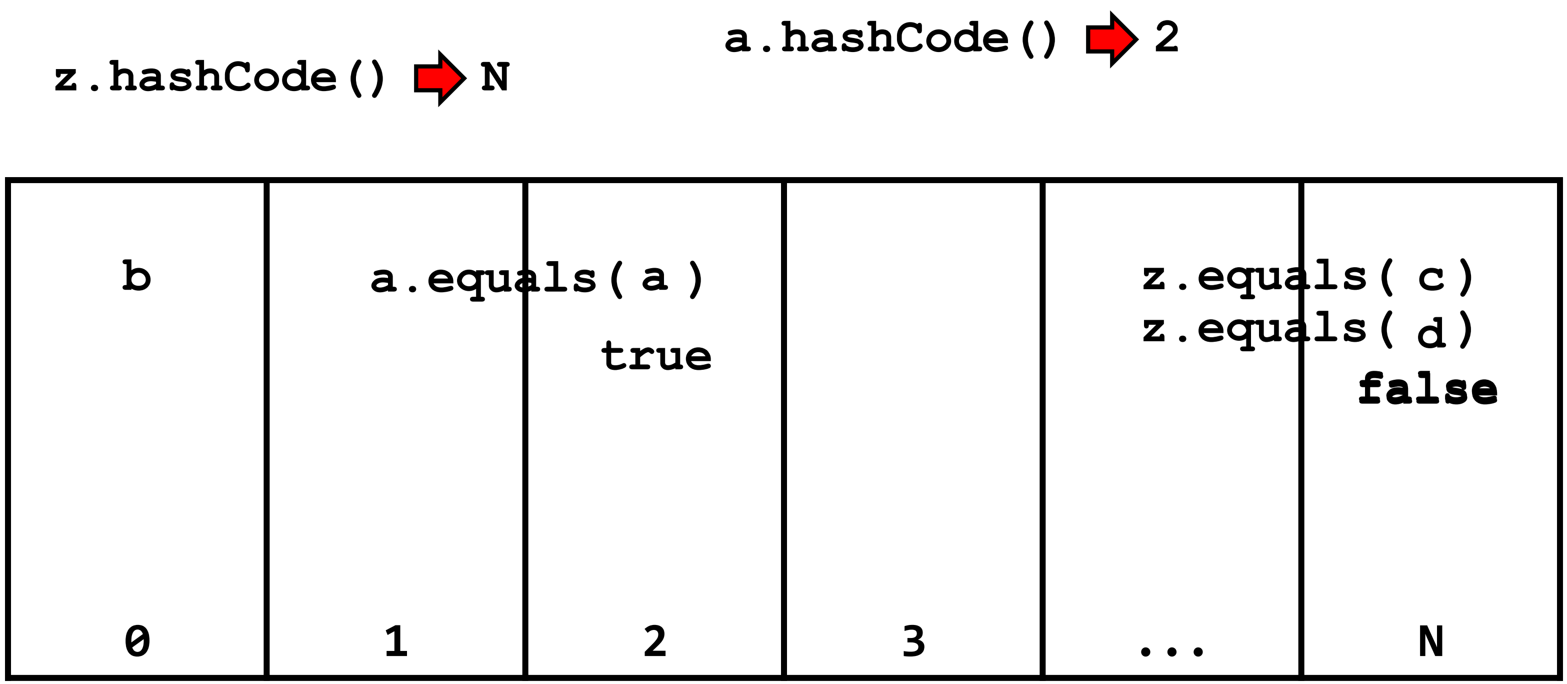
Insertion into a Hash Table

- to insert an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to put the object into

`b`		`a`			`c` `d`
`0`	`1`	`2`	`3`	`...`	`N`

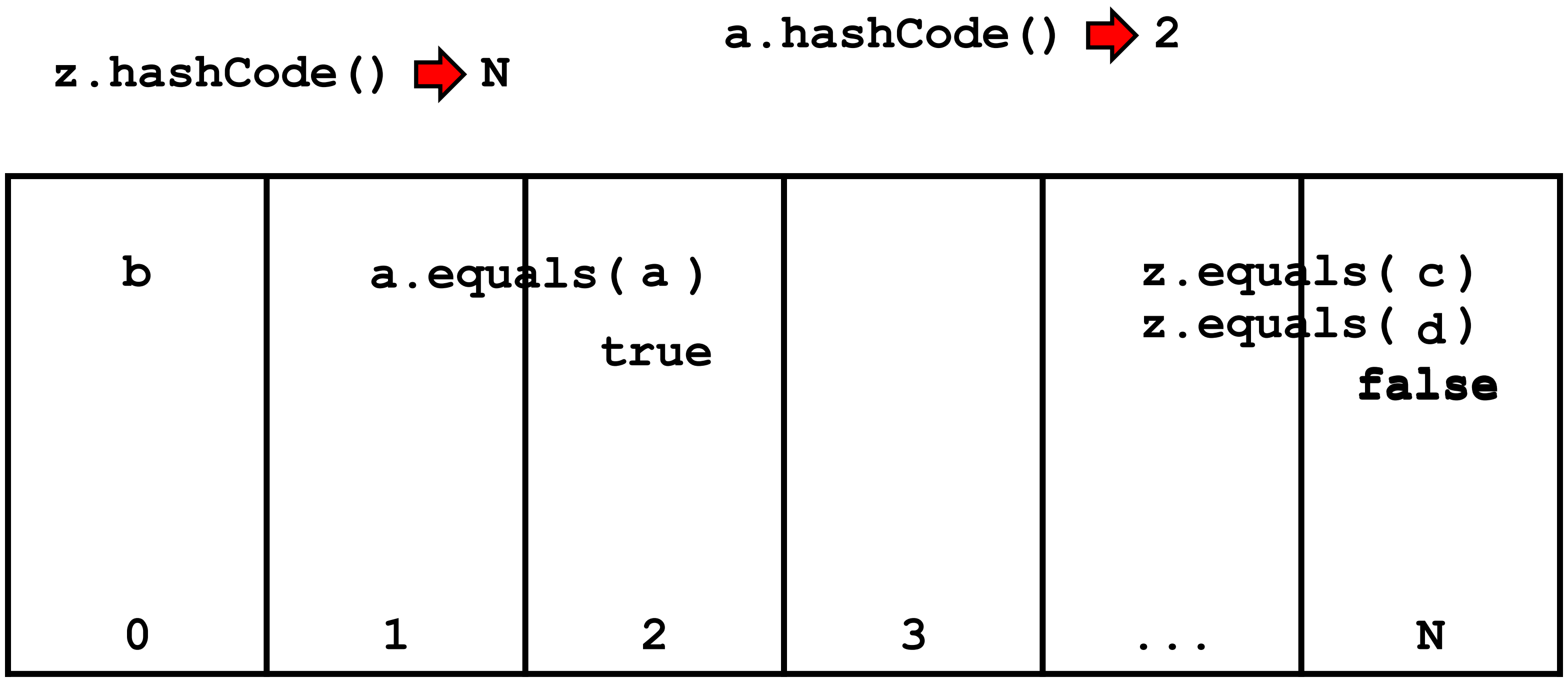
Search on a Hash Table

- to see if a hash table contains an object **a**, the hash table calls **a.hashCode()** method to compute which bucket to look for **a** in



Search on a Hash Table

- to see if a hash table contains an object **a**, the hash table calls **a.hashCode()** method to compute which bucket to look for **a** in



- searching a hash table is usually much faster than linear search
  - doubling the number of elements in the hash table usually does not noticably increase the amount of search needed
- if there are **`n`** elements in the hash table:
  - best case
    - the bucket is empty, or the first element in the bucket is the one we are searching for
      - 0 or 1 call to **`equals`**
  - worst case
    - all **`n`** of the elements are in the same bucket
      - n calls to **`equals`**
  - average case
    - the element is in a bucket with a small number of other elements
      - a small number of calls to **`equals`**

`Object.hashCode`

- if you don't override **`hashCode`**, you get the implementation from **`Object.hashCode`**
  - **`Object.hashCode`** uses the memory address of the object to compute the hash code

```java
// client code somewhere
SimplePoint2 p = new SimplePoint2(1f, -2f);

HashSet<SimplePoint2> h = new HashSet<>();
h.add(p);
System.out.println( h.contains(p) );        // true

SimplePoint2 q = new SimplePoint2(1f, -2f);
System.out.println( h.contains(q) );        // false!
```

- note that **p** and **q** refer to distinct objects
  - therefore, their memory locations must be different
    - therefore, their hash codes are different (probably)
    - therefore, the hash table looks in the wrong bucket (probably) and does not find the complex number even though **p.equals(q)** is **true**

Implementing **hashCode**

- the basic idea is generate a hash code using the fields of the object
- it would be nice if two distinct objects had two distinct hash codes
  - but this is not required; two different objects can have the same hash code
- it is required that:
  1. if `x.equals(y)` then `x.hashCode() == y.hashCode()`
  2. `x.hashCode()` always returns the same value if `x` does not change its state

A bad (but legal) **hashCode**

```
public class SimplePoint2 {
  public float x;
  public float y;

  @Override
  public int hashCode() {
    return 1;
  }
```

- this will cause a hashed container to put all points into the same bucket

A slightly better **hashCode**

```
public class SimplePoint2 {
  public float x;
  public float y;

  @Override
  public int hashCode() {
    return (int) (this.x + this.y);
  }
```

A good **hashCode**

```
public class SimplePoint2 {
  public float x;
  public float y;

  @Override
  public int hashCode() {
    return Objects.hash(this.x, this.y);
  }
```

eclipse **hashCode**

- eclipse will also generate a hashCode method for you
  - Source → Generate hashCode() and equals()…

- it uses an algorithm that
  - "… yields reasonably good hash functions, [but] does not yield state-of-the-art hash functions, nor do the Java platform libraries provide such hash functions as of release 1.6. Writing such hash functions is a research topic, best left to mathematicians and theoretical computer scientists."
    - Joshua Bloch, *Effective Java 2nd Edition*

Information hiding

The problem with **`public`** fields

- recall that our point class has two **`public`** fields

```
public class SimplePoint2 {
  public float x;
  public float y;

  // implementation not shown
}
```

The problem with **`public`** fields

- clients are expected to manipulate the fields directly

```
public class BoundingBox {

    private SimplePoint2 bottomLeft;
    private SimplePoint2 topRight;

    public float area() {
        float width = topRight.x - bottomLeft.x;
        float height = topRight.y - bottomLeft.y;
        return width * height;
    }
}
```

The problem with **`public`** fields

- the problem with public fields is that they become a permanent part of the API of your class
- after you have released a class with public fields you:
  - cannot change the access modifier
  - cannot change the type of the field
  - cannot change the name of the field

without breaking client code

Information hiding

- information hiding is the principle of hiding implementation details behind a stable interface
  - if the interface never changes then clients will not be affected if the implementation details change

- for a Java class, information hiding suggests that you should hide the implementation details of your class behind a stable API
  - fields and their types are part of the implementation details of a class
  - fields should be private; if clients need access to a field then they should use a method provided by the class

```java
/**
 * A simple class for representing points in 2D Cartesian
 * coordinates. Every <code>Point2D</code> instance has an
 * x and y coordinate.
 */
public class Point2 {

    private double x;
    private double y;
```

```java
// default constructor
public Point2() {
    this(0.0, 0.0);
}

// custom constructor
public Point2(double newX, double newY) {
    this.set(newX, newY);
}

// copy constructor
public Point2(Point2 other) {
    this(other.x, other.y);
}
```

Accessors

- an accessor method enables the client to gain access to an otherwise private field of the class
- the name of an accessor method often, but not always, begins with **`get`**

```java
// Accessor methods (methods that get the value of a field)

// get the x coordinate
public double getX() {
    return this.x;
}

// get the y coordinate
public double getY() {
    return this.y;
}
```

Mutators

- a mutator method enables the client to modify (or mutate) an otherwise private field of the class
- the name of an accessor method often, but not always, begins with **`set`**

```java
// Mutator methods: methods that change the value of a field


// set the x coordinate
public void setX(double newX) {
    this.x = newX;
}

// set the y coordinate
public void setY(double newY) {
    this.y = newY;
}

// set both x and y coordinates
public void set(double newX, double newY) {
    this.x = newX;
    this.y = newY;
}
```

Information hiding

- hiding the implementation details of our class gives us the ability to change the underlying implementation without affecting clients
  - for example, we can use an array to store the coordinates

```java
/**
 * A simple class for representing points in 2D Cartesian
 * coordinates. Every <code>Point2D</code> instance has an
 * x and y coordinate.
 */
public class Point2 {

    private double coord[];
```

```
// default constructor
public Point2() {
    this(0.0, 0.0);
}

// custom constructor
public Point2(double newX, double newY) {
    this.coord = new double[2];
    this.coord[0] = newX;
    this.coord[1] = newY;
}

// copy constructor
public Point2(Point2 other) {
    this(other.x, other.y);
}
```

```java
// Accessor methods (methods that get the value of a field)

// get the x coordinate
public double getX() {
    return this.coord[0];
}

// get the y coordinate
public double getY() {
    return this.coord[1];
}
```

```java
// Mutator methods: methods that change the value of a field


// set the x coordinate
public void setX(double newX) {
    this.coord[0] = newX;
}

// set the y coordinate
public void setY(double newY) {
    this.coord[1] = newY;
}

// set both x and y coordinates
public void set(double newX, double newY) {
    this.coord[0] = newX;
    this.coord[1] = newY;
}
```

Information hiding

- notice that:
  - we changed how the point is represented by using an array instead of two separate fields for the coordinates
  - we did not change the API of the class
- by hiding the implementation details of the class we have insulated all clients of our class from the change

Immutability

Immutability

- an immutable object is an object whose state cannot be changed once it has been created
  - examples: **`String`**, **`Integer`**, **`Double`**, and all of the other wrapper classes
- advantages of immutability versus mutability
  - easier to design, implement, and use
  - can never be put into an inconsistent state after creation
  - object references can be safely shared
- information hiding makes immutability possible

Recipe for Immutability

- the recipe for immutability in Java is described by Joshua Bloch in the book *Effective Java**

1. Do not provide any methods that can alter the state of the object
2. Prevent the class from being extended
3. Make all fields **`final`**
4. Make all fields **`private`**
5. Prevent clients from obtaining a reference to any mutable fields

revisit when we talk about inheritance

revisit when we talk about composition

*highly recommended reading if you plan on becoming a Java programmer

An immutable point class

- we can easily make an immutable version of our **`Point2`** class
  - remove the mutator methods
  - make the fields **`final`** (they are already **`private`**)
  - make the class **`final`** (which satisfies Rule 2 from the recipe)

```java
/**
 * A simple class for immutable points in 2D Cartesian
 * coordinates. Every <code>IPoint2D</code> instance has an
 * x and y coordinate.
 */
public final class IPoint2 {

    final private double x;
    final private double y;
```

```java
// default constructor
public IPoint2() {
    this(0.0, 0.0);
}

// custom constructor
public IPoint2(double newX, double newY) {
    this.x = newX;
    this.y = newY;
}

// copy constructor
public IPoint2(Point2 other) {
    this(other.x, other.y);
}
```

```java
    // Accessor methods (methods that get the value of a field)

    // get the x coordinate
    public double getX() {
        return this.x;
    }

    // get the y coordinate
    public double getY() {
        return this.y;
    }

    // No mutator methods

    // toString, hashCode, equals are all OK to have
}
```

Class invariants

Class invariants

- a class invariant is a condition regarding the state of a an object that is always true
  - the invariant established when the object is created and every public method of the class must ensure that the invariant is true when the method finishes running
- immutability is a special case of a class invariant
  - once created, the state of an immutable object is always the same
- information hiding makes maintaining class invariants possible

Class invariants

- suppose we want to create a point class where the coordinates of a point are always greater than or equal to zero
  - the constructors must not allow a point to be created with negative coordinates
  - if there are mutator methods then those methods must not set the coordinates of the point to a negative value

```java
/**
 * A simple class for representing points in 2D Cartesian
 * coordinates. Every <code>PPoint2D</code> instance has an
 * x and y coordinate that is greater than or equal to zero.
 *
 * @author EECS2030 Winter 2016-17
 *
 */
public class PPoint2 {

    private double x;   // invariant: this.x >= 0
    private double y;   // invariant: this.y >= 0
```

```java
/**
 * Create a point with coordinates <code>(0, 0)</code>.
 */
public PPoint2() {
    this(0.0, 0.0); // invariants are true
}


/**
 * Create a point with the same coordinates as
 * <code>other</code>.
 *
 * @param other another point
 */
public PPoint2(PPoint2 other) {
    this(other.x, other.y);  // invariants are true
                             // because other is a PPoint2
}
```

```java
/**
 * Create a point with coordinates <code>(newX, newY)</code>.
 *
 * @param newX the x-coordinate of the point
 * @param newY the y-coordinate of the point
 */
public PPoint2(double newX, double newY) {
    // must check newX and newY first before setting this.x and this.y
    if (newX < 0.0) {
        throw new IllegalArgumentException(
            "x coordinate is negative");
    }
    if (newY < 0.0) {
        throw new IllegalArgumentException(
            "y coordinate is negative");
    }
    this.x = newX;  // invariants are true
    this.y = newY;  // invariants are true
}
```

```java
/**
 * Returns the x-coordinate of this point.
 *
 * @return the x-coordinate of this point
 */
public double getX() {
    return this.x;  // invariants are true
}

/**
 * Returns the y-coordinate of this point.
 *
 * @return the y-coordinate of this point
 */
public double getY() {
    return this.y;  // invariants are true
}
```

```java
/**
 * Sets the x-coordinate of this point to <code>newX</code
 *
 * @param newX the new x-coordinate of this point
 */
public void setX(double newX) {
    // must check newX before setting this.x
    if (newX < 0.0) {
        throw new IllegalArgumentException("x coordinate is negative");
    }
        this.x = newX;  // invariants are true
    }

/**
 * Sets the y-coordinate of this point to <code>newY</code>.
 *
 * @param newY the new y-coordinate of this point
 */
public void setY(double newY) {
    // must check newY before setting this.y
    if (newY < 0.0) {
        throw new IllegalArgumentException("y coordinate is negative");
    }
        this.y = newY;  // invariants are true
}
```

```java
/**
 * Sets the x-coordinate and y-coordinate of this point to
 * <code>newX</code> and <code>newY</code>, respectively.
 *
 * @param newX the new x-coordinate of this point
 * @param newY the new y-coordinate of this point
 */
public void set(double newX, double newY) {
    // must check newX and newY before setting this.x and this.y
    if (newX < 0.0) {
        throw new IllegalArgumentException(
            "x coordinate is negative");
    }
    if (newY < 0.0) {
        throw new IllegalArgumentException(
            "y coordinate is negative");
    }
        this.x = newX;  // invariants are true
        this.y = newY;  // invariants are true
}
```

Removing duplicate code

- notice that there is a lot of duplicate code related to validating the coordinates of the point
  - one constructor is almost identical to **`set(double, double)`**
  - **`set(double, double)`** repeats the same validation code as **`setX(double)`** and **`setY(double)`**

- we should try to remove the duplicate code by delegating to the appropriate methods

```java
/**
 * Create a point with coordinates <code>(newX, newY)</code
 *
 * @param newX the x-coordinate of the point
 * @param newY the y-coordinate of the point
 */
public PPoint2(double newX, double newY) {
    this.set(newX, newY); // use set to ensure
                          // invariants are true
}
```

```java
/**
 * Sets the x-coordinate of this point to <code>newX</code>.
 *
 * @param newX the new x-coordinate of this point
*/
public void setX(double newX) {
    this.set(newX, this.y); // use set to ensure
                            // invariants are true

}

/**
 * Sets the y-coordinate of this point to <code>newY</code>.
 *
 * @param newY the new y-coordinate of this point
 */
public void setY(double newY) {
    this.set(this.x, newY); // use set to ensure
                            // invariants are true

}
```