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 Rectangle {

    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[];
```

```java
// 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

}
```

compareTo

Comparable Objects

- many value types have a natural ordering
  - that is, for two objects **`x`** and **`y`**, **`x`** is less than **`y`** is meaningful
    - **`Short`**, **`Integer`**, **`Float`**, **`Double`**, etc
    - **`String`**s can be compared in dictionary order
    - **`Date`**s can be compared in chronological order
    - you might compare points by their distance from the origin
- if your class has a natural ordering, consider implementing the **`Comparable`** interface
  - doing so allows clients to sort arrays or **`Collection`**s of your object

Interfaces

- an interface is (usually) a group of related methods with empty bodies
  - the **`Comparable`** interface has just one method

```
public interface Comparable<T>
{
  int compareTo(T t);
}
```

- a class that implements an interfaces promises to provide an implementation for every method in the interface

`compareTo()`

- Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Throws a **`ClassCastException`** if the specified object type cannot be compared to this object

- suppose that we want to compare points by their distance from the origin

Point2 compareTo

```
public class Point2 implements Comparable<Point2> {
  // fields, constructors, methods...

  @Override
  public int compareTo(Point2 other) {
    double thisDist = Math.hypot(this.x, this.y);
    double otherDist = Math.hypot(other.x, other.y);
    if (thisDist > otherDist) {
      return 1;
    }
    else if (thisDist < otherDist) {
      return -1;
    }
    return 0;
  }
```

`Point2 compareTo`

- don't forget what you learned in previous courses
  - you should delegate work to well-tested components where possible
- for distances, we need to compare two **`double`** values
  - **`java.lang.Double`** has methods that do exactly this

Point2 compareTo

```java
public class Point2 implements Comparable<Point2> {
  // fields, constructors, methods...
```

```java
  @Override
  public int compareTo(Point2 other) {
    double thisDist = Math.hypot(this.x, this.y);
    double otherDist = Math.hypot(other.x, other.y);
    return Double.compare(thisDist, otherDist);
  }
```

Comparable Contract

1. the sign of the returned **`int`** must flip if the order of the two compared objects flip
   - if `x.compareTo(y) > 0` then `y.compareTo(x) < 0`
   - if `x.compareTo(y) < 0` then `y.compareTo(x) > 0`
   - if `x.compareTo(y) == 0` then `y.compareTo(x) == 0`

Comparable Contract

2. `compareTo()` must be transitive
   - if `x.compareTo(y) > 0 && y.compareTo(z) > 0` then `x.compareTo(z) > 0`
   - if `x.compareTo(y) < 0 && y.compareTo(z) < 0` then `x.compareTo(z) < 0`
   - if `x.compareTo(y) == 0 && y.compareTo(z) == 0` then `x.compareTo(z) == 0`

Comparable Contract

3. if `x.compareTo(y) == 0` then the signs of `x.compareTo(z)` and `y.compareTo(z)` must be the same

Consistency with equals

- an implementation of **`compareTo()`** is said to be consistent with **`equals()`** when

```
if x.compareTo(y) == 0 then
   x.equals(y) == true
```

  - and

```
if x.equals(y) == true then
   x.compareTo(y) == 0
```

Not in the Comparable Contract

- it is *not* required that **`compareTo()`** be consistent with **`equals()`**
  - that is

    if **`x.compareTo(y) == 0`** then

    `x.equals(y) == false` is acceptable
  - similarly

    if **`x.equals(y) == true`** then

    `x.compareTo(y) != 0` is acceptable

- try to come up with examples for both cases above
- is **`Point2 compareTo`** consistent with equals?

Implementing **compareTo**

- if you are comparing fields of type **`float`** or **`double`** you should use **`Float.compare`** or **`Double.compare`** instead of **`<`**, **`>`**, or **`==`**

- if your **`compareTo`** implementation is broken, then any classes or methods that rely on **`compareTo`** will behave erratically
  - **`TreeSet`**, **`TreeMap`**
  - many methods in the utility classes **`Collections`** and **`Arrays`**

Mixing Static and Non-Static

`static` Fields

- a field that is **`static`** is a per-class member
  - only one copy of the field, and the field is associated with the class
    - every object created from a class declaring a static field shares the same copy of the field
- static fields are used when you really want only one common instance of the field for the class
  - less common than non-static fields

Example

- a textbook example of a static field is a counter that counts the number of created instances of your class

```
// adapted from Oracle's Java Tutorial
public class Bicycle {
  // some other fields here...
  private static int numberOfBicycles = 0;

  public Bicycle() {
    // set some non-static fields here...
    Bicycle.numberOfBicycles++;
  }

  public static int getNumberOfBicyclesCreated() {
    return Bicycle.numberOfBicycles;
  }
}
```

note: not `this.numberOfBicycles++`

[notes 4.3]

- why does **`numberOfBicycles`** have to be **`static`**?
  - because we really want one common value for all **`Bicycle`** instances

- what would happen if we made **`numberOfBicycles`** non-**`static`**?
  - every **`Bicycle`** would think that there was a different number of **`Bicycle`** instances

- another common example is to count the number of times a method has been called

```java
public class X {

  private static int numTimesXCalled = 0;
  private static int numTimesYCalled = 0;

  public void xMethod() {
    // do something... and then update counter
    ++X.numTimesXCalled;
  }

  public void yMethod() {
    // do something... and then update counter
    ++X.numTimesYCalled;
  }
}
```

- is it useful to add the following to **`Point2`**?

```
public static final Point2 ORIGIN = new Point2(0.0, 0.0);
```

Mixing Static and Non-static Fields

- a class can declare static (per class) and non-static (per instance) fields
- a common textbook example is giving each instance a unique serial number
  - the serial number belongs to the instance
    - therefore it must be a non-static field

```
public class Bicycle {
  // some attributes here...
  private static int numberOfBicycles = 0;

  private int serialNumber;

  // ...
```


[notes 4.3.2]

- how do you assign each instance a unique serial number?
  - the instance cannot give itself a unique serial number because it would need to know all the currently used serial numbers
- could require that the client provide a serial number using the constructor
  - instance has no guarantee that the client has provided a valid (unique) serial number

- the class can provide unique serial numbers using static fields
  - e.g. using the number of instances created as a serial number

```
public class Bicycle {
  // some attributes here...

  private static int numberOfBicycles = 0;
  private int serialNumber;

  public Bicycle() {
    // set some attributes here...
    this.serialNumber = Bicycle.numberOfBicycles;
    Bicycle.numberOfBicycles++;
  }
}
```

- a more sophisticated implementation might use an object to generate serial numbers

```
public class Bicycle {

  // some attributes here...
  private static int numberOfBicycles = 0;

  private static final
    SerialGenerator serialSource = new SerialGenerator();

  private int serialNumber;

  public Bicycle() {
    // set some attributes here...
    this.serialNumber = Bicycle.serialSource.getNext();
    Bicycle.numberOfBicycles++;
  }
}
```

but you would need an implementation of this class

Static Methods

- recall that a **`static`** method is a per-class method
  - client does not need an object to invoke the method
  - client uses the class name to access the method

Static Methods

- a **`static`** method can use only **`static`** fields of the class
  - **`static`** methods have no **`this`** parameter because a **`static`** method can be invoked without an object
  - without a **`this`** parameter, there is no way to access non-static fields
- non-static methods can use all of the fields of a class (including **`static`** ones)

```java
public class Bicycle {
  // some attributes, constructors, methods here...

  public static int getNumberCreated()
  {
    return Bicycle.numberOfBicycles;
  }

  public int getSerialNumber()
  {
    return this.serialNumber;
  }

  public void setNewSerialNumber()
  {
    this.serialNumber = Bicycle.serialSource.getNext();
  }
}
```

static method can only use static fields

non-static method can use non-static fields

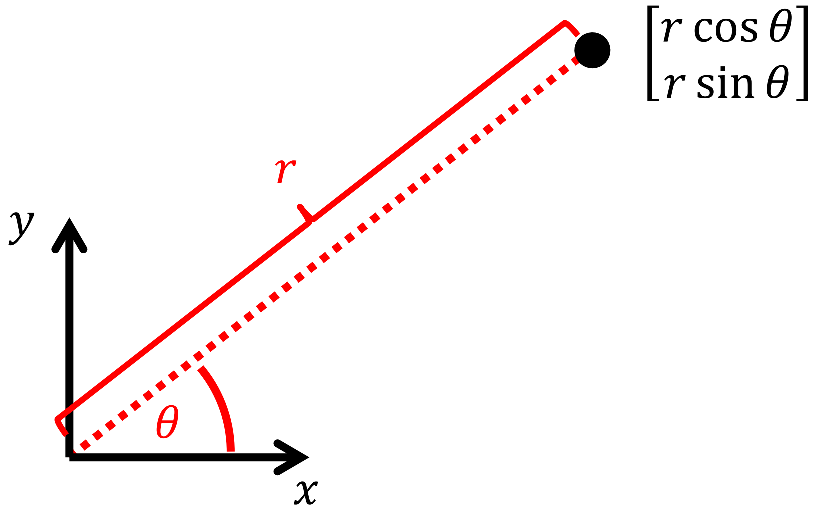
and static fields

Static factory methods

- a common use of static methods in non-utility classes is to create a *static factory method*
  - a static factory method is a static method that returns an instance of the class
  - called a factory method because it makes an object and returns a reference to the object
- you can use a static factory method to create methods that behave like constructors
  - they create and return a reference to a new instance
  - unlike a constructor, the method has a name

Static factory methods

- recall our point class
  - suppose that you want to provide a constructor that constructs a point given the polar form of the point



```java
public class Point2 {

  private double x;
  private double y;

  public Point2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Point2(double r, double theta) {
    this(r * Math.cos(theta), r * Math.sin(theta));
  }
```

Illegal overload; both constructors have the same signature.

Static factory methods

- we can eliminate the problem by replacing the second constructor with a static factory method

```java
public class Point2 {

  private double x;
  private double y;

  public Point2(double x, double y) {
    this.x = x;
    this.y = y;
  }
```

```java
  public static Point2 polar(double r, double theta) {
    double x = r * Math.cos(theta);
    double y = r * Math.sin(theta);
    return new Point2(x, y);
  }
```

Static Factory Methods

- many examples in Java API
  - `java.lang.Integer`

    `public static Integer valueOf(int i)`
    - Returns a **`Integer`** instance representing the specified **`int`** value.
  - `java.util.Arrays`

    `public static int[] copyOf(int[] original, int newLength)`
    - Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
  - `java.lang.String`

    `public static String format(String format, Object... args)`
    - Returns a formatted string using the specified format string and arguments.