public static void cp(Point p) {
    p.translate(2, 3); // add to x, y
    p = new Point(4, 7);
}
// client code of cp
Point pl = new Point(1, 2); // x, y
cp(pl);
System.out.println(pl.toString());

A. (3, 5)
B. (1, 5)
C. (4, 7)
D. (6, 10)
E. (1, 2)

Encapsulation

- **encapsulation**: Hiding implementation details from clients.

  - Encapsulation forces abstraction.
  - Separates external view (behavior) from internal view (state)
  - Protects the integrity of an object's data

Private fields

* A field that cannot be accessed from outside the class

```java
private type name;
```

- Examples:

```java
private int id;
private String name;
```

- Client code won't compile if it accesses private fields:

```java
PointMain.java:11: x has private access in Point
System.out.println(pl.x);
```
Accessing private state

// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}

- Client code will look more like this:
  System.out.println(p1.getX());
  p1.setX(14);

Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an Account's balance.

- Can change the class implementation later
  - Example: Point could be rewritten in polar coordinates \((r, \theta)\) with the same methods.

- Can constrain objects' state (invariants)
  - Example: Only allow Accounts with non-negative balance.
  - Example: Only allow Dates with a month from 1-12.

Point class, version 4

// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }
    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
}

The keyword this

reading: 8.3
The **this** keyword

- **this**: Refers to the implicit parameter inside your class.
  
  *(a variable that stores the object on which a method is called)*

- Refer to a field: `this.field`

- Call a method: `this.method(parameters);`

- One constructor, `this(parameters);`
  can call another:

Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

  ```java
  public class Point {
      private int x;
      private int y;
      ...
      // this is legal
      public void setLocation(int x, int y) {
          ...
      }
  }
  ```
  - In most of the class, `x` and `y` refer to the fields.
  - In setLocation, `x` and `y` refer to the method's parameters.

Fixing shadowing

```java
public class Point {
    private int x;
    private int y;
    ...
    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- **Inside setLocation,**
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

Calling another constructor

```java
public class Point {
    private int x;
    private int y;
    ...
    public Point() {
        this(0, 0); // calls (x, y) constructor
    }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor