Constructors and Encapsulation
Constructors

- Constructors allow us to specify how to create a new object with some specified initial state.
- **Ex.:** `Point p1 = new Point(8, 4);`
- The constructor gets executed when the client uses the `new` keyword.
- The constructor parameters specify which fields must have values passed from the client during initialization.
Constructors

• Constructors look a bit like methods.
• Header must start with the public keyword, followed by the class name and any parameters.
• You do NOT specify a return type.
• The parameters usually will define the initial state of the new object.
public class Point {
    int x;
    int y;
    public Point (int x0, int y0) {
        x = x0;
        y = y0;
    }
}
Constructor Execution

• Constructors execute on a particular object, just like instance methods.

• Executed on the object being created with the new keyword. Can refer to fields and methods of that object directly.

• What exactly happens when we reach a statement that uses a constructor?

  - Point p1 = new Point(8, 4);
Constructor Call - Point

• First, a new Point object is created and allocated in memory.
• The constructor is called on this new object, passing 8 and 4 as the x0 and y0 parameter values.
• A Point reference variable named p1 is created and set to refer to the newly created object.
Default Constructor

• When a class doesn't have a constructor (like first version of our Point), Java automagically gives us a default constructor with no parameters.

• Java does not supply default constructor when we write our own, so in new version of Point, we cannot do this anymore:

  - Point p = new Point();
The “this” keyword

• We know that instance methods of an object can refer to its other methods and fields (it knows what it's operating on)

• Idea of an “implicit parameter”: a special reference that is set each time an instance method is called.

• Access this special reference using this keyword.
"this"

- When we referred to the field $x$ in our Point code, that was actually shorthand for `this.x`
- Using "this" explicitly can make for clearer code.
- When you call an instance method inside the same class, that's also shorthand for `this.methodName()`
Shadow variables

- Shadowing occurs when a field is obscured by another variable (method parameter or local variable) with the same name.

```java
public Point(int x, int y) {
    x = x;
    y = y;
}
```

- The parameter “x” here shadows the field x.
Unshadowing using “this”

- We can instead do this:
  
  ```java
  this.x = x;
  this.y = y;
  ```

- Some programmers like this style of naming a constructor parameter the same as the field.

- “this” lets us do that without ambiguity.
Multiple Constructors

• A class can have multiple constructors. Allows us more than one way to create a new type of object.

• Each constructor must have a different signature (i.e. a different number and/or type of parameters).

• We could, for instance, add back a parameter-less default constructor to Point.
Encapsulation

- Def. - Hiding the implementation details of an object from the clients of the object.
- Think about an iPod – almost everyone knows how to use an iPod, but few people know how to build one or understand the hardware/software inside it.
- This is a benefit of the iPod design; we don't need to know how it works to use it.
- Like a “black box”.
Encapsulation

- From the outside, we just want to see behavior.
- From the inside, we see internal state that is used to accomplish that behavior.
- This is an example of an important concept in IT/CS – abstraction.
- To achieve encapsulation in our programs, we want to hide internal state from outside view.
Point Example

• So far, we've built a class that has good behavior on outside, but we can still see internal state.

• Encapsulate the fields of a Point by declaring them to be private:

```java
private int x;
private int y;
```
Private fields

- Private fields are visible to all code inside the class, but not anywhere else.

- We need to provide a way for client code to access a Point's fields – accessor/getter methods
  - public int getX() {return x;}
  - public int getY() {return y;}