Why provide more than one constructor?

Doing so is better for the user.
From A1 (we assume you remember the specs):

```java
public Organism(int lev, int m, String nn) { ... }

public Organism(int lev) { ... }
```

So, the user can write `new Organism(4)` instead of `new Organism(4, 0, null)`.

For the programmer, it’d be great to have the one-parameter constructor call the other:

```java
public Organism(int lev) {
    Organism(lev, 0, null);
    this(lev, 0, null);
}
```

We wish we could say this!

But you HAVE to do it “this” way. Here, `this` refers to the other constructor.
Issues related to sub-classes

The ability to extend existing subclasses to reuse/refine existing behavior is a **terrific** aspect of object-oriented programming.

Example: modeling **sulfur bacteria** as photosynthesizing organisms that come in purple or green variations. (Thank you, Wikipedia.)

We want `sb.eat(victim)` to have the effect appropriate for sulfur bacteria (i.e., nothing happens) rather than the effect appropriate for generic Organisms.

[Program and tester will be posted to the course website.]
For the call `sb.eat(v)`, which method `eat` is called?

**Overriding rule or bottom-up rule:**
Start at the bottom of the folder and search upward until a matching method is found.

**Terminology.** `SBact` **inherits** methods and fields from `Organism`. `Sbact` **overrides** `eat` and `toString`.

<table>
<thead>
<tr>
<th>Method/Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equals(Object)</code></td>
<td><code>Object</code></td>
</tr>
<tr>
<td><code>toString()</code></td>
<td></td>
</tr>
<tr>
<td><code>level</code></td>
<td><code>Organism</code></td>
</tr>
<tr>
<td><code>Organism(int)</code></td>
<td><code>Organism(int, int, String)</code></td>
</tr>
<tr>
<td><code>eat(Organism)</code></td>
<td><code>toString()</code></td>
</tr>
<tr>
<td><code>getColor()</code></td>
<td></td>
</tr>
<tr>
<td><code>SBact(String)</code></td>
<td><code>getLevel()</code></td>
</tr>
<tr>
<td><code>eat(Organism)</code></td>
<td><code>toString()</code></td>
</tr>
</tbody>
</table>
Purpose of super and this

Suppose we’re overriding a method in order to modify it just a bit; so, we’d like to refer to the overridden method. The word `super` refers only to components in the partitions above it.

```java
/** = String like A3 requires for Organism, but with "<green/purple> SBact." in front. */
public String toString() {
    return getColor() + " SBact. " + super.toString();
}
```

this refers to the name of the object in which it appears. We could have written `this.getColor()`, but it wasn’t necessary to.
public class SBact extends Organism {
    private String color;
    /** Constructor: A sulfur bacterium of color c [etc. Full program will be posted.]*/
    public SBact(String c) {
        super(0); /* default Org. values, * lowest level */
        color= c;
    }
}

The first (and only the first) statement in a constructor has to be a call to a constructor of the superclass. If you don’t put one in, then this one is automatically used:

    super();

This corresponds to a natural principle: Fill in superclass fields first.
Non-abstract vs. Abstract

```java
/** Instances of subclasses of Car represent cars. */
public class Car {
    private String make; // a make for a car

    /** Constructor: a Car with make x.
     * Precondition: x is the car make as a String. */
    public Car(String x) {
        make = x;
    }

    /** = the make of this car */
    public String getMake() {
        return make;
    }

    /** "ob is a Car that is made by a competing company." */
    public boolean isCompetitor(Object ob) {
        // Make sure ob is a car.
        if (!(ob instanceof Car)) {
            return false;
        }
        // Is the make of ob different from
        // the make of this car?
        return ((Car)ob).getMake() != getMake();
    }
}

/** Instances of subclasses of Car represent cars. */
public abstract class Car {
    private String make; // a make for a car

    /** Constructor: a Car with make x.
     * Precondition: x is the car make as a String. */
    public Car(String x) {
        make = x;
    }

    /** = the make of this car */
    public String getMake() {
        return make;
    }

    /** "ob is a Car that is made by a competing company." */
    public abstract boolean isCompetitor(Object ob); // ;
```
Non-abstract vs. Abstract

```java
/** An instance is a Volkswagen. */
public class VW extends Car {
    /** Constructor: a new VW. */
    public VW()
    {
        super("Volkswagen");
    }
}
```

```java
/** An instance is a Volkswagen. */
public class VW extends Car {
    /** Constructor: a new VW. */
    public VW()
    {
        super("Volkswagen");
    }

    /** = "ob is a Car that is made by a competing company." */
    public boolean isCompetitor(Object ob)
    {
        // Make sure ob is a car.
        if (!(ob instanceof Car)) {
            return false;
        }

        // Is the make of ob a subsidiary?
        if (((Car)ob).getMake().equals("Audi")
            || ((Car)ob).getMake().equals("Bentley")
            || ...) {
            return false;
        }

        // If not, check as before.
        return ((Car)ob).getMake() != getMake();
    }
}
```