Summary of facts about constructors

1. A constructor in class C has the form shown to the right.

```
/** Constructor: … */
public C (parameters) { … }
```

2. The purpose of a constructor is to initialize the fields of an object so that the class invariant is true.

3. A constructor call can appear in only *two* places:

   - In a new-expression, which has the form **new** <constructor-call>
     An example is **new** C (5, "ab");

   - As the first statement in the body of a constructor, with two possible forms:
     this(*arguments*); // call another constructor in this class
     super(*arguments*); // call a constructor in the superclass

4. If a class C does not have a constructor, this one is inserted: **public** C () {}. It does nothing, but very fast.

5. If a class C does have a constructor, the one mentioned in point 4 is *not* inserted.

6. **Principle**: In a constructor in a subclass, initialize the fields in the superclass first. Java tries to enforce this principle by requiring:

   If a constructor does not begin with a call on another constructor (see point 3), this one is inserted: **super**();

7. *ALMOST NEVER* should a constructor in a subclass attempt to initialize fields of the superclass by storing in them directly or by calling setter methods in the superclass. The smart OO practice is to initialize the superclass fields by *calling a constructor in the superclass, using* **super**(…);

8. Points 5 and 6 can result in a syntax error that is hard to understand. Consider class C shown to the right. It has a one-parameter constructor. Subclass S also has a constructor but the constructor body does not start with a call on a constructor. Therefore, Java inserts a call, and the constructor is like this:

```
public class C {
   public C (int p) {…}
}
public class S extends C {
   public S() {x= 5;}
}
```

   public S() {**super**(); x= 5;} }

   But class C does not have a constructor with no parameters, so the call **super**(); is illegal and there is a syntax error.