Constructors

A constructor is a method that is declared with the basic form:

<access modifier> <class-name> (<parameter-declarations>) { … }

where the <class-name> is the name of the class in which the method appears. Note that neither keyword **void** nor a return type appears in the declaration.

The purpose of a constructor is to initialize the fields of a new object of class <class-name> so that the class invariant is true when the object is created.

An example of a constructor appears in class Time, to the right. The constructor with parameter t stores t in field time. Because of the precondition, the constraint 0 <= time < 24*60 in the class invariant will be true.

```
public class Time {
  int time; // time of day, in minutes,
            //  (0 <= time < 24*60

  /** Constructor: instance with time t.
      Precondition: 0 <= t < 24*60. */
  public Time() { time= t; }

  /** Constructor: instance with time 0.*/
  public Time() {}
}
```

The constructor is called in a new-expression

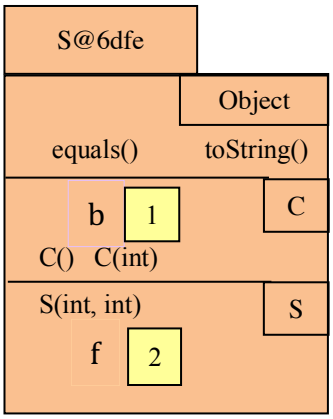
Evaluation of the new-expression **new** Time(60) first creates a new object of class Time, with default values for the fields. Next, it executes the constructor call that appears in the new-expression: Time(60) . In this case, we see that execution stores 60 in field time of the newly created object. (The final step of evaluation of the new expression is to yield as its value the name of (pointer to) the newly created object.)

If the default value is appropriate for a field, the constructor need not store anything in it. For example, in the above class Time, the second constructor, with no parameters, need not store 0 in field t because 0 is the default.

Why have more than one constructor? To provide flexibility and take care of different situations. Class JFrame, an object of which is associated with a window on your monitor, has at least 10 constructors.

Rule: Assign superclass fields first

To the right is an object of class S. S is a subclass of C, so the object has partitions for classes S, C, and Object. S and C each have a field. Java has this rule:



> **Rule Superclass fields first:** When creating an object, initialize superclass fields before subclass fields.

Thus, when S's constructor is called, it should fill in field b before field f. But b is probably private, so the only way to assign to b within S's constructor is to call a constructor in C.

To reinforce the rule of filling in superclass fields first, Java has the following rule —we explain the notation "**super**();" in a moment.

> **Syntactic rule**: The first statement of any constructor you write must be a call on another constructor. If the first statement is not a constructor call, Java inserts this one: **super**();

Calling a superclass constructor

To the right, we give a possible constructor in class S. The first statement is a call on a superclass constructor. Your first inclination is to write the call as C(bb), since the superclass name is C. But that's not the Java syntax; the Java syntax requires keyword **super** instead of the class name to call a superclass constructor.

```
public S(int bb, int ff) {
   super(bb);
   f= ff;
}
```

Presumably, the call **super**(bb) stores bb in field b —but it doesn't have to; at this point, we don't know what C's constructor does.

If the call **super**(bb) is omitted, Java uses instead **super**(); which calls the constructor in C that has no parameters. That's OK. But if that no-parameter constructor were missing from C, a syntax error would occur: There would not be a no-parameter constructor to call in class C.

Calling another constructor in this class

To the right are two constructors to go in class C. The one-parameter constructor does the obvious: store parameter bb in field b.

```
public C(int bb) {
   b= bb;
}

/** Constructor: an instance
    with 1 in its field */
public C() {
   this(1);
}
```

The second constructor, with no parameters, is supposed to store 1 in field b. This could be done with the assignment b= 1. But we want to show how to call another constructor in the same class. You might think this would be done using the call C(1), but that is not Java syntax. Instead, use keyword **this**:

To call another constructor in the same class, use **this**(…);

As usual, when a method name is overloaded, the number of arguments of the call, along with their types, will determine which of the several methods will be called.

Why call another constructor? In the case shown, there is no need for this; just use the assignment b= 1. But in general, it's good to avoid duplicate code, and if another constructor does a lot of the work of initializing fields, then have a call on that constructor followed by initialization of the rest of the fields.

What if no constructor is declared in a class?

If no constructor is declared in a class C, Java inserts this one —it does nothing, but very fast:

C() {}

If a constructor *is* declared in class C, this one is not inserted.