The garbage collector finishes by scanning the heap for heap variables still marked as inaccessible: *a*, *c*, *e*, *g*, and *i*. These really *are* inaccessible, so the garbage collector deallocates them.

□

Mark–sweep garbage collection may be seen to be a simple graph algorithm. The heap variables and the stack are the nodes of a directed graph, and the pointers are the edges. Our aim is to determine the largest subgraph in which all nodes can be reached from the stack node. The *mark–sweep garbage collection algorithm* can be expressed recursively as follows:

Procedure to collect garbage:
    mark all heap variables as inaccessible;
    scan all frames in the stack;
    add all heap variables still marked as inaccessible to the free list.

Procedure to scan the storage region *R*:
    for each pointer *p* in *R*:
        if *p* points to a heap variable *v* that is marked as inaccessible:
            mark *v* as accessible;
            scan *v*.

For this algorithm to work, it must be able to visit all heap variables, it must know the size of each, and it must be able to mark each as accessible or inaccessible. One way to meet these requirements is to extend each heap variable with the following hidden fields: a size field; a link field (used to connect all heap variables into a single linked list, to permit them all to be visited); and a 1-bit accessibility field. These hidden fields are used by the garbage collector but invisible to the programmer.

Another requirement is that pointers must be distinguishable from other data in the store. (This is a requirement not only for garbage collection, but also for heap compaction as described in Section 6.6.2.) This is an awkward problem: pointers are represented by addresses, which typically have exactly the same form as integers. Some clever techniques have been devised to solve this problem – see Wilson (1992).

6.7 Run-time organization for object-oriented languages

Object-oriented (OO) languages give rise to interesting and special problems in run-time organization. An object is a special kind of record. Attached to each object are some methods, each method being a kind of procedure or function that is able to operate on that object. Objects are grouped into classes, such that all objects of the same class have identical structure and identical methods.

We shall assume the following more precise definitions:

- An ***object*** is a group of instance variables, to which a group of instance methods are attached.
- An ***instance variable*** is a named component of a particular object.
- An ***instance method*** is a named operation, which is attached to a particular object and is able to access that object's instance variables.
- An ***object class*** (or just *class*) is a family of objects with similar instance variables and identical methods.

In a pure OO language, all instance variables would be private, leaving the instance methods as the *only* way to operate on the objects. In practice, most OO languages (such as Java and C++) allow the programmer to decide which of the instance variables are public and which are private. Anyway, this issue does not affect their representation.

An instance-method call explicitly identifies a particular object, called the *receiver object*, and a particular instance method attached to that object. In Java, such a method call has the form:

$$E_0.I(E_1, \ \ldots, \ E_n)$$

The expression E_0 is evaluated to yield the receiver object. The identifier I names an instance method attached to that object. The expressions E_1, ... , E_n are evaluated to yield the arguments passed to the method.

Although an object is somewhat similar to a record, the representation of an object must reflect the close association between the object and its instance methods. From an object we must be able to locate the attached instance methods. In turn, each instance method must somehow 'know' which object it is attached to.

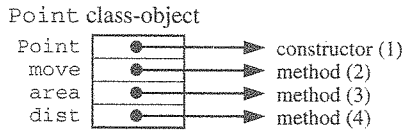
Example 6.32 Java object representation (single class)

Consider the following Java class:

```
class Point {
   // A Point object represents a geometric point located at (x, y).

   protected int x, y;

(1) public Point (int x, int y) {
      this.x = x; this.y = y;
   }

(2) public void move (int dx, int dy) {.
      this.x += dx; this.y += dy;
   }

(3) public float area () {
      return 0.0;
   }
```

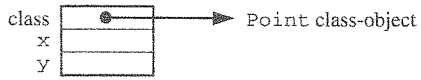
```
(4) public float dist (Point that) {.
        int dx = this.x - that.x;
        int dy = this.y - that.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

Associated with class `Point` is a unique *class-object* that looks like this:



The `Point` class-object contains the addresses of the class's instance methods named `move`, `area`, and `dist`, as well as its constructor.

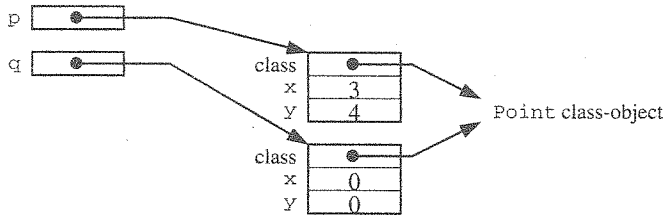
An object of class `Point` looks like this:[5]



A `Point` object consists of the instance variables named x and y, together with a *class-tag*. The class-tag is just a pointer to the `Point` class-object. Class-tags serve to distinguish objects of different classes. (Later we shall see why this is necessary.) They also serve to link each object to the methods of that object's class.

Objects are created by the allocator `new`. (They are heap variables in the terminology of Section 6.6.) Each variable declared with class `Point` may be assigned a pointer to a `Point` object:

```
Point p = new Point(2, 3);
Point q = new Point(0, 0);
p.move(1, 1);
```



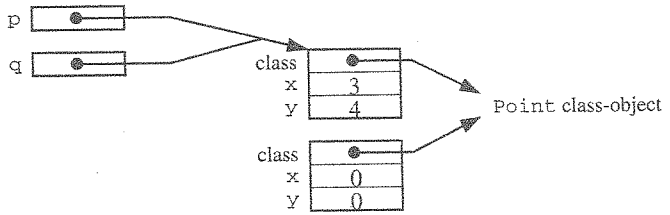
Now the method call '`q.dist(p)`' would return 5.0.

Object assignment copies only the pointer (not the object itself[6]):

[5] Note that throughout this section, we assume that a class only has a single constructor.

[6] The Java `clone` method can be used to copy an object.

```
q = p;
```



Now p and q point to the same object. The method call '`q.dist(p)`' returns 0.0. The method call '`p.move(1, 1)`' would update the object that both p and q point to.

The method call '`p.move(1, 1)`' works as follows. The receiver object is the object that p points to. The address of the called method is found by following the pointer from the receiver object to the corresponding class-object, where the address of the instance method named `move` is found. As well as the explicit arguments (both 1 in this case), the receiver object is passed as an implicit argument, and bound to the keyword `this`. In this way the instance method 'knows' which object it is attached to.

Note that a side effect of the assignment '`q = p;`' above was to leave one of the objects inaccessible. A garbage collector is needed to deallocate inaccessible objects.

□

A ***subclass*** of class *C*, say *S*, is a family of objects similar to objects of class *C*, but possibly with extra instance variables, extra methods, and/or overridden methods. *C* is known as the ***superclass*** of S.

Any object of subclass *S* can be treated like an object of class *C*, simply by ignoring its extra instance variables and methods.

By default, each instance method of class *C* is ***inherited*** by subclass *S*. In other words, the same method code is attached to objects of class *C* and to objects of class *S*. (The language's scope rules must ensure that the body of the inherited method accesses only the instance variables of *C*, not those of *S*.)

Alternatively, an instance method of class *C* can be ***overridden*** by a method of the same name in class *S*. In other words, different method code is attached to objects of class *C* and to objects of class *S*. (The overriding method may access the instance variables of *S*, not just those of *C*.)

Here we shall assume that the OO language enforces ***single inheritance***, i.e., each class has at most one superclass. This is (more or less) the case in Java.[7]

[7] C++ supports *multiple inheritance*, whereby a class may have several superclasses. Java actually supports a limited form of multiple inheritance, *via* interfaces.

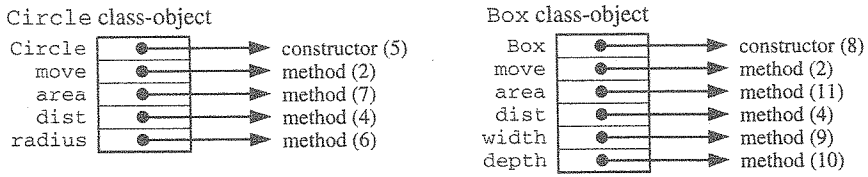
The superclass of Point in Example 6.32 is Object, by default. A consequence of this is that Point inherits the methods of class Object (but for simplicity we have omitted these inherited methods in the Point class-object).

Example 6.33 Java object representation (class and subclasses)

Consider the following Java classes, both of which are subclasses of Point:

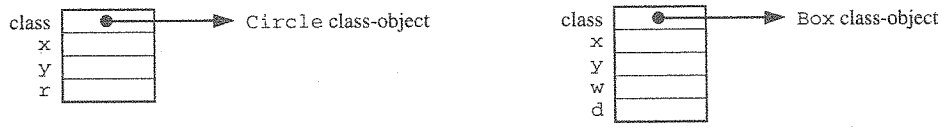
```
class Circle extends Point {
   // A Circle object represents a circle of radius r, centered at (x, y).

   protected int r;

(5) public Circle (int x, int y, int r) {
      this.x = x; this.y = y;  this.r = r;
   }

(6) public int radius () {
      return this.r;
   }

(7) public double area () {
      double pi = 3.1416;
      return pi * this.r * this.r;
   }
}

class Box extends Point {
   // A Box object represents a rectangle of width w and depth d,
   // centered at (x, y).

   protected int w, d;

(8) public Box (int x, int y, int w, int d) {
      this.x = x; this.y = y; this.w = w; this.d = d;
   }

(9) public int width () {
      return this.w;
   }

(10) public int depth () {
      return this.d;
   }

(11) public double area () {
      return (double) (this.w * this.d);
   }
}
```

The Circle and Box class-objects look like these:



The Circle class-object is an extension of the Point class-object in Example 6.32. It starts with the instance methods named move (which is inherited from the Point class), area (which is overridden), and dist (which is inherited); these are followed by the extra method named radius. The Box class-object is likewise an extension of the Point class-object, but its extra instance methods are named width and depth.

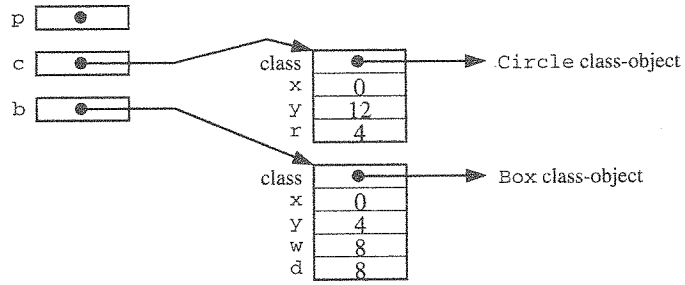
Objects of classes Circle and Box look like these:



Each Circle object starts with a class-tag and the instance variables x and y; these are followed by the extra instance variable named r. Its class-tag is a pointer to the Circle class-object. A Box object is similar, but its extra instance variables are named w and d. Its class-tag is a pointer to the Box class-object.

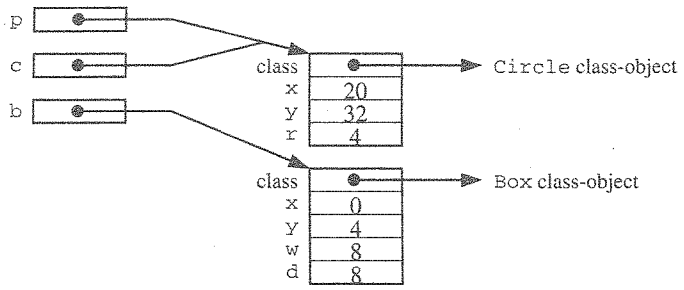
The following code illustrates the behavior of Circle and Box objects:

```
int s    = 4;
Point p  = null;
Circle c = new Circle(0, 3*s, s);
Box b    = new Box(0, s, 2*s, 2*s);
```



Now the method call 'c.dist(b)' would return 8.0, since dist is inherited by class Circle (and by class Box); the answer is in fact the distance between the *centers* of c and b.

```
p = c;
p.move(20, 20);
```



Now the `Point` variable `p` points to an object of class `Circle`, which is legal. The program can still access `p.x` and `p.y`, but any attempt to access `p.r` would be a type error. In this context the object pointed to by `p` is treated like a `Point` object. However, the method call '`p.area()`' would return 50.3, not 0.0, because it is actually the overriding method (7) that is called.

In detail, the method call '`p.area()`' works as follows. The receiver object is the object that `p` points to. The address of the called method is found by following the pointer from the receiver object to the corresponding class-object, which in this example is the `Circle` class-object, and there the instance method named `move` is (7) rather than (3). Thus method overriding works as required.

□

This example illustrates several important points:

- Each instance variable has a fixed offset relative to the base of every object that contains it. Assuming that class-tags and integers occupy one word each, the instance variables `x` and `y` have offsets 1 and 2, respectively, not only in `Point` objects but also in `Circle` and `Box` objects (and indeed in objects of any subclass of `Point`). A variable such as `p` can point to any `Point`, `Circle`, or `Box` object, but the addressing formula for `p.x` (or `p.y`) does not depend on the class of object that `p` currently points to.
- Likewise, every instance method has a fixed offset relative to the base of every class-object that 'contains' it. Assuming that code pointers occupy one word each, the instance methods `move`, `area`, and `dist` have offsets 1, 2, and 3, respectively, not only in the `Point` class-object but also in the `Circle` and `Box` class-objects (and indeed in the class-object of any subclass of `Point`). Thus a method call such as '`p.area()`' can be implemented efficiently using the known offset of `area`. (Relative to the cost of an ordinary procedure call, an instance-method call carries an overhead of two indirections. This is the price we must pay for dynamic method selection.)
- Method inheritance and overriding work as required. All objects of the same class have class-tags that point to the same class-object. But each subclass has a distinct class-object, in which some methods are the same as those of the superclass-object (the inherited methods), some methods are different (the overridden methods), and some methods are extra.

So far we have neglected class variables and methods. A ***class variable*** is a variable associated with a class, but not a component of a particular object. A ***class method*** is an operation associated with a class, but not attached to a particular object.

Class variables are global in terms of their lifetime, so static storage allocation is sufficient. Class methods are like ordinary procedures, so they also need no special treatment. It may be convenient to make class variables and class methods part of the corresponding class-object.

6.8 Case study: the abstract machine TAM

TAM (the Triangle Abstract Machine) was designed specifically to support the implementation of high-level programming languages, and in particular the run-time organization techniques described in Sections 6.1 through 6.6. Thus:

- The low-address end of the data store is reserved as a stack. This is used both for stack storage allocation and for expression evaluation. Operations are provided for pushing and popping values at the stack top.
- The high-address end of the data store is reserved as a heap. Operations are provided for allocating and deallocating heap variables.
- The call and return instructions handle frames automatically. The call instruction pushes a new frame, with all its link data. The return instruction pops a frame, and also replaces the routine's arguments by its result (if any).
- There are no general-purpose registers that could be used for storing data. All registers are dedicated to specific purposes: registers SB and ST delimit the stack; registers HB and HT delimit the heap; register LB points to the topmost frame on the stack; and so on. Updating of registers is always implicit: LB is updated by call and return instructions; ST is updated by load, store, and many other instructions; and so on.

In these respects TAM is quite similar to some other real and abstract stack machines (such as the JVM, see Section 2.4). But a detailed look at TAM reveals a number of interesting design features, some of which are discussed below. A complete description of TAM may be found in Appendix C.

TAM is implemented by an interpreter. This interpreter is described in Section 8.3, and is available from our Web site.

Addressing and registers

Most instructions have address operands. An address operand is always of the form '$d[r]$', where r names a register that points to the base of a store segment or frame, and d is a displacement:

address denoted by $d[r]$ = d + register r