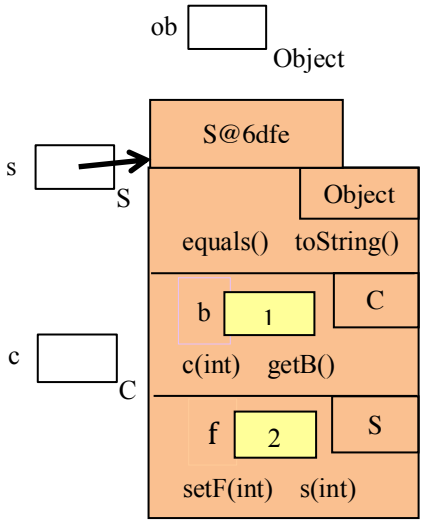
Casting about

Consider the object of class S on the right. Class S was declared as a subclass of class C. Variables ob, c, and s were declared like this:

```
Object ob;
C c;
S s= new S(…);
```



A cast of a pointer to an object to another class has the form

(<class-name>) <pointer-to-object>

The first statement below casts pointer s to Object and stores it in ob. The second statement casts pointer ob to C and stores it in variable c.

```
ob=  (Object) s;
c=   (C) ob;
```

These casts do *not* change the object or the pointer. They simply provide a different view of the object. They do not take any time —except to make sure that the cast can be done.

The pointer in ob views the object as if it were of class Object. The pointer in c views the object as if it were of class C. Read the dictionary entry for the *compile-time reference-rule* for information about the effects of the different views of an object.

A cast (class) is a *unary prefix operator*, like negation –. If there is a sequence of them, they are carried out from right to left. For example, consider the expression below. It casts c to class type S, then to C, then to Object, and then back to to C, so the result of evaluating the expression is a pointer with class-type C.

```
(C) (Object) (C) (S) c
```

Rules governing casts

Here are rules governing class casts.

1. **Object-casting rule:** A pointer to an object can be cast to any class for which it has a partition, and to none other.
2. Upward casts (from a subclass to some superclass) will be inserted automatically when necessary. Downward casts must be given explicitly.

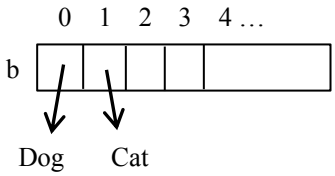
Here are examples, given the three variables shown above.

- The pointer in s can be cast to S, C, and Object and to nothing else.
- One can write a sequence of casts like this: (S) (C) (C) (Object) s. They are performed right to left.
- ob= s; is equivalent to ob= (Object) s; . The upward cast is inserted automatically.
- The cast (JFrame) s is syntactically incorrect because it is known at compile-time that no partition of an S object has name JFrame.
- The cast (S) ob is OK if ob contains a pointer to an S object but is not ok if it contains a pointer to an object of class Object. In the latter case, at runtime, a ClassCastException will be thrown.

Please note that we were careful to say that the "pointer to the object" is cast, not the object. However, we generally are not so precise and simply write that the object is cast.

Reasons for having casts

Consider a class Animal with subclasses Dog, Cat, and Cow. We draw an Animal array b to the right. Element b[0] is a Dog and b[1] is a Cat.



Each element b[i] of the array has type Animal. Suppose the assignment

```
b[2]= new Cow(…);
```

is executed. An upward cast is automatically inserted because the type of b[2] is Animal, so the assignment is equivalent to the following one:

```
b[2]= (Animal) (new Cow(…));
```

The following example illustrates a cast upward to store an argument value in a parameter. It happens frequently. Suppose class Animal has fields for the birthdate of an animal. In class Animal, we might have this static method:

```
/** Return the age in years of Animal an. */
public static int age(Animal an) { … }
```

In the second statement below, Argument d is automatically cast to class Animal because the pointer has to be stored in parameter an, which has type Animal.

```
Dog d= new Dog(…);
int age= Animal.age(d);
```

Testing whether a downward cast will work: operator instanceof

As you will see from studying the *compile-time reference-rule*, the call ob.c(int) is syntactically incorrect and will not compile. In order to use method c, ob has to be cast down to class C:

```
(C ob).c(5);        //Cast ob to C and then call method c.
```

But this will throw an exception if ob cannot be cast to C, which will happen if ob points to an object of class Object, which has no C partition.

Use the instanceof operator to tell whether the cast is OK:

```
if (ob instanceof C) {
   C v= (C) ob;
   v.c(5);
   …
}
```

The instanceof operator has the syntax:

<pointer-to-object> instanceof <class-name>

Its evaluation yields true if and only if the object has a partition named <class-name> and false otherwise.

Checking the specific class with getClass

While instanceof checks whether a value is of class C *or any subclass of C*, sometimes you need to check whether it is *just* of class C and not a subclass. The method getClass can do this:

```
if (ob.getClass() == C.class) {
   …
}
```

This comparison is true only if ob's bottom partition in its diagram is named C, i.e., if it was created using new C(…).

Be frugal with the use of instanceof and getClass

Neither instanceof and getClass should have to be used often. If they are being used a lot, then probably good use is not being made of the OO features of Java, and a restructuring of the program is in order.

However, getClass *has* to be used when overriding function equals (which is declared in class Object). Turn to the JavaHyperText entry for equals(…) for an explanation.