```
   int opCmp(Object rhs);
   static Object factory(string classname);
}
```

Let's look closer at the semantics of each of these symbols.

6.8.1 `string toString()`

This returns a textual representation of the object. By default it returns the class name:

```
// File test.d
class Widget {}
unittest {
   assert((new Widget).toString() == "test.Widget");
}
```

Note that the name of the class comes together with the name of the module the class is defined in. By default, the module name is the same as the file name, a default that can be changed with a `module` declaration (§ 11.1.8 on page 348).

6.8.2 `size_t toHash()`

This returns a hash of the object as an unsigned integral value (32 bits on 32-bit machines, 64 bits on 64-bit machines). By default, the hash is computed by using the bitwise representation of the object. The hash value is a concise but inexact digest of the object. One important requirement is *consistency:* If `toHash` is called twice against a reference without an intervening change to the state of the object, it should return the same value. Also, the hashes of two equal objects must be equal, and the hash values of two distinct (non-equal) objects are unlikely to be equal. The next section discusses in detail how object equality is defined.

6.8.3 `bool opEquals(Object rhs)`

This returns `true` if `this` considers that `rhs` is equal to it. This odd formulation is intentional. Experience with Java's similar function `equals` has shown that there are some subtle issues related to defining equality in the presence of inheritance, for which reason D approaches the problem in a relatively elaborate manner.

First off, one notion of equality for objects already exists: when you compare two references to class objects `a1` and `a2` by using the expression `a1 is a2` (§ 2.3.4.3 on page 48), you get `true` if and only if `a1` and `a2` refer to the same object, just as in Figure 6.1 on page 177. This notion of object equality is sensible, but too restrictive to be useful. Often, two actually distinct objects should be considered equal if they hold the same state. In D, logical equality is assessed by using the `==` and `!=` operators. Here's how they work.

Let's say you write ‹*lhs*› == ‹*rhs*› for expressions ‹*lhs*› and ‹*rhs*›. Then, if at least one of ‹*lhs*› and ‹*rhs*› has a user-defined type, the compiler rewrites the comparison as `object.opEquals(‹lhs›, ‹rhs›)`. Similarly, ‹*lhs*› != ‹*rhs*› is rewritten as `!object.opEquals(‹lhs›, ‹rhs›)`. Recall from earlier in this section that `object` is a core module defined by your D implementation and implicitly `import`ed in any module that you build. So the comparisons are rewritten into calls to a free function provided by your implementation and residing in module `object`.

The equality relation between objects is expected to obey certain invariants, and `object.opEquals(‹lhs›, ‹rhs›)` goes a long way toward ensuring correctness. First, `null` references must compare equal. Then, for any three non-`null` references `x`, `y`, and `z`, the following assertions must hold true:

```
// The null reference is singular; no non-null object equals null
assert(x != null);
// Reflexivity
assert(x == x);
// Symmetry
assert((x == y) == (y == x));
// Transitivity
if (x == y && y == z) assert(x == z);
// Relationship with toHash
if (x == y) assert(x.toHash() == y.toHash());
```

A more subtle requirement of `opEquals` is *consistency*: evaluating equality twice against the same references without an intervening mutation to the underlying objects must return the same result.

The typical implementation of `object.opEquals` eliminates a few simple or degenerate cases and then defers to the member version. Here's what `object.opEquals` may look like:

```
// Inside system module object.d
bool opEquals(Object lhs, Object rhs) {
   // If aliased to the same object or both null => equal
   if (lhs is rhs) return true;
   // If either is null => non-equal
   if (lhs is null || rhs is null) return false;
   // If same exact type => one call to method opEquals
   if (typeid(lhs) == typeid(rhs)) return lhs.opEquals(rhs);
   // General case => symmetric calls to method opEquals
   return lhs.opEquals(rhs) && rhs.opEquals(lhs);
}
```

First, if the two references refer to the same object or are both null, the result is trivially `true` (ensuring reflexivity). Then, once it is established that the objects are distinct, if one of them is `null`, the comparison result is `false` (ensuring singularity of `null`). The third test checks whether the two objects have exactly the same type and, if they do, defers to `lhs.opEquals(rhs)`. And a more interesting part is the double evaluation on the last line. Why isn't one call enough?

Recall the initial—and slightly cryptic—description of the `opEquals` method: "returns `true` if `this` considers that `rhs` is equal to it." The definition cares only about `this` but does not gauge any opinion `rhs` may have. To get the complete agreement, a handshake must take place—each of the two objects must respond affirmatively to the question: Do you consider that object your equal? Disagreements about equality may appear to be only an academic problem, but they are quite common in the presence of inheritance, as pointed out by Joshua Bloch in his book *Effective Java* [9] and subsequently by Tal Cohen in an article [17]. Let's restate that argument.

Getting back to an example related to graphical user interfaces, consider that you define a graphical widget that could sit on a window:

```
class Rectangle { ... }
class Window { ... }
class Widget {
   private Window parent;
   private Rectangle position;
   ... // Widget-specific functions
}
```

Then you define a class `TextWidget`, which is a widget that displays some text.

```
class TextWidget : Widget {
   private string text;
   ...
}
```

How do we implement `opEquals` for these two classes? As far as `Widget` is concerned, another `Widget` that has the same state is equal:

```
   // Inside class Widget
   override bool opEquals(Object rhs) {
      // The other must be a Widget
      auto that = cast(Widget) rhs;
      if (!that) return false;
      // Compare all state
      return parent == that.parent
         && position == that.position;
   }
```

The expression `cast(Widget) rhs` attempts to recover the `Widget` from `rhs`. If `rhs` is `null` or `rhs`'s actual, dynamic type is not `Widget` or a subclass thereof, the `cast` expression returns `null`.

The `TextWidget` class has a more discriminating notion of equality because the right-hand side of the comparison must also be a `TextWidget` and carry the same text.

```
// Inside class TextWidget
override bool opEquals(Object rhs) {
   // The other must be a TextWidget
   auto that = cast(TextWidget) rhs;
   if (!that) return false;
   // Compare all relevant state
   return super.opEquals(that) && text == that.text;
}
```

Now consider a `TextWidget tw` superimposed on a `Widget w` with the same position and parent window. As far as `w` is concerned, `tw` is equal to it. But from `tw`'s viewpoint, there is no equality because `w` is not a `TextWidget`. If we accepted that `w == tw` but `tw != w`, that would break reflexivity of the equality operator. To restore reflexivity, let's consider making `TextWiget` less strict: inside `TextWidget.opEquals`, if comparison is against a `Widget` that is not a `TextWidget`, the comparison just agrees to go with `Widget`'s notion of equality. The implementation would look like this:

```
// Alternate TextWidget.opEquals -- BROKEN
override bool opEquals(Object rhs) {
   // The other must be at least a Widget
   auto that = cast(Widget) rhs;
   if (!that) return false;
   // Do they compare equal as Widgets? If not, we're done
   if (!super.opEquals(that)) return false;
   // Is it a TextWidget?
   auto that2 = cast(TextWidget) rhs;
   // If not, we're done comparing with success
   if (!that2) return true;
   // Compare as TextWidgets
   return text == that.text;
}
```

Alas, `TextWidget`'s attempts at being accommodating are ill advised. The problem is that now transitivity of comparison is broken: it is easy to create two `TextWidgets` `tw1` and `tw2` that are different (by containing different texts) but at the same time equal with a simple `Widget` object `w`. That would create a situation where `tw1 == w` and `tw2 == w`, but `tw1 != tw2`.

So in the general case, comparison must be carried out both ways—each side of the comparison must agree on equality. The good news is that the free function `object.opEquals(Object, Object)` avoids the handshake whenever the two involved objects have the same exact type, and even any other call in a few other cases.

6.8.4 `int opCmp(Object rhs)`

This implements a three-way ordering comparison, which is needed for using objects as keys in associative arrays. It returns an unspecified negative number if `this` is less than `rhs`, an unspecified positive number if `rhs` is less than `this`, and `0` if `this` is considered unordered with `rhs`. Similarly to `opEquals`, `opCmp` is seldom called explicitly. Most of the time, you invoke it implicitly by using one of `a < b`, `a <= b`, `a > b`, and `a >= b`.

The rewrite follows a protocol similar to `opEquals`, by using a global `object.opCmp` definition that intermediates communication between the two involved objects. For each of the operators <, <=, >, and >=, the D compiler rewrites the expression `a` ‹*op*› `b` as `object.opCmp(a, b)` ‹*op*› `0`. For example, `a < b` becomes `object.opCmp(a, b) < 0`.

Implementing `opCmp` is optional. The default implementation `Object.opCmp` throws an exception. In case you do implement it, `opCmp` must be a "strict weak order," that is it must satisfy the following invariants for any non-`null` references `x`, `y`, and `z`.

```
// 1. Reflexivity
assert(x.opCmp(x) == 0);
// 2. Transitivity of sign
if (x.opCmp(y) < 0 && y.opCmp(z) < 0) assert(x.opCmp(z) < 0);
// 3. Transitivity of equality with zero
if ((x.opCmp(y) == 0 && y.opCmp(z) == 0) assert(x.opCmp(z) == 0);
```

The rules above may seem a bit odd because they express axioms in terms of the less familiar notion of three-way comparison. If we rewrite them in terms of <, we obtain the familiar properties of strict weak ordering in mathematics:

```
// 1. Irreflexivity of `<'
assert(!(x < x));
// 2. Transitivity of `<'
if (x < y && y < z) assert(x < z);
// 3. Transitivity of `!(x < y) && !(y < x)'
if (!(x < y) && !(y < x) && !(y < z) && !(z < y))
   assert(!(x < z) && !(z < x));
```

The third condition is necessary for making < a strict weak ordering. Without it, < is called a *partial order*. You might get away with a partial order, but only for restricted uses; most interesting algorithms require a strict weak ordering. If you want to define a partial order, you're better off giving up all syntactic sugar and defining your own named functions distinct from `opCmp`.