Figure 2.2 illustrates the assignment operator for reference variables. By assigning point3 the stored value of point2, we have point3 reference the same object that point2 was referencing. Now, point2==point3 is true because point2 and point3 both store 1024 and thus reference the same object. point1!=point2 is also true because point1 and point2 reference different objects.

The other category of operations deals with the object that is being referenced. There are only three basic actions that can be done:

1. Apply a type conversion (Section 1.4.4).
2. Access an internal field or call a method via the dot operator (.) (Section 2.2.1).
3. Use the instanceof operator to verify that the stored object is of a certain type (Section 3.6.3).

The next section illustrates in more detail the common reference operations.

### 2.2 Basics of Objects and References

In Java, an object is an instance of any of the nonprimitive types. Objects are treated differently from primitive data. Primitive types, as already shown, are handled by value, meaning that the values assumed by the primitive variables are stored in those variables and copied from primitive variable to primitive variable during assignments. As shown in Section 2.1, reference variables store references to objects. The actual object is stored somewhere in memory, and the reference variable stores the object’s memory address. Thus a reference variable simply represents a name for that part of memory. This means that primitive variables
and reference variables behave differently. This section examines these differences in more detail and illustrates the operations that are allowed for reference variables.

### 2.2.1 The Dot Operator (.)

The dot operator (.) is used to select a method that is applied to an object. For instance, suppose we have an object of type Circle that defines an area method. If theCircle references a Circle, then we can compute the area of the referenced Circle (and save it to a variable of type double) by doing this:

```java
double theArea = theCircle.area();
```

It is possible that theCircle stores the null reference. In this case, applying the dot operator will generate a NullPointerException when the program runs. Generally, this will cause abnormal termination.

The dot operator can also be used to access individual components of an object, provided arrangements have been made to allow internal components to be viewable. Chapter 3 discusses how these arrangements are made. It also explains why it is generally preferable to not allow direct access of individual components.

### 2.2.2 Declaration of Objects

We have already seen the syntax for declaring primitive variables. For objects, there is an important difference. When we declare a reference variable, we are simply providing a name that can be used to reference an object that is stored in memory. However, the declaration by itself does not provide that object. For example, suppose there is an object of type Button that we want to add into an existing Panel p, using the method add (all this is provided in the Java library). Consider the statements

```java
Button b; // b may reference a Button object
b.setLabel( "No" ); // Label the button b refers to "No"
p.add( b ); // and add to Panel p
```

All seems well with these statements until we remember that b is the name of some Button object but no Button has been created yet. As a result, after the declaration of b the value stored by the reference variable b is null, meaning b is not yet referring to a valid Button object. Consequently, the second line is illegal because we are attempting to alter an object that does not exist. In this scenario, the compiler will probably detect the error, stating that “b is uninitialized.” In other cases, the compiler will not notice and a run-time error will result in the cryptic NullPointerException error message.