Chapter 7

Modeling Real Objects

`CircleCalculator` (4.1) objects allow us to easily compute the circumference and area of a circle if we know its radius. We really need to create only one `CircleCalculator` object, because if want to compute the circumference of a different circle, we only need to pass a different radius value to the `circumference()` method. Should we create more than one circle calculator object there would be no way to distinguish among them, except by their unique address identifiers; that is, given `CircleCalculator` (4.1):

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> cc1 = new CircleCalculator();
> cc2 = new CircleCalculator();
> cc1
CircleCalculator@2f4538
> cc2
CircleCalculator@a0ebc2
```

Otherwise, `cc1` and `cc2` behave identically. Similarly, the `convert()` method for `TimeConverter` objects computes the same result given the same input regardless of which `TimeConverter` object is used. We say that `CircleCalculator` objects and `TimeConverter` objects are *stateless*. A primitive-type variable like an integer has a state; its value defines its state. Given the following integer variables:

```
int x = 0, y = 4, z = 8;
```

each variable is in a different state, since each has a different value. The concept of state can be more complex than a simple value difference. We can define state however we choose; for example:

- `x`, `y`, and `z` are in different states, since they all have values different from each other.
- `y` and `z` are both in a nonzero state, while `x` is not.
- `x`, `y`, and `z` are all in the same state—nonnegative.

In this chapter we examine objects with state. Objects with state possess qualities that allow them to be distinguished from one another and behave differently over time.

7.1 Objects with State

`Circle1` (7.1) is a slight textual variation of `CircleCalculator` (4.1) that has a profound impact how clients use circle objects. (We name it `Circle1` since this is the first version of the circle class.)

```
public class Circle1 {
    private final double PI = 3.14159;
    private double radius;
    public Circle1(double r) {
        radius = r;
    }
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 7.1: `Circle1`—a slightly different circle class

In `Circle1` (7.1) we add a new instance variable, `radius`, and new kind of element called a *constructor*:

- A new instance variable is introduced:

  ```
  private double radius;
  ```

  `radius` is really a variable because it lacks the `final` qualifier. Since it is private, however, clients cannot change its value because they cannot even see it.

- The method-looking construct named `Circle1` is called a *constructor*:

  ```
  public Circle1(double r) {
      radius = r;
  }
  ```

  A constructor definition is similar to a method definition except:

  – it has no return type specified,

  – its name is the same as the class in which it is defined,

  – it can be invoked only indirectly through the `new` operator.

  Constructor code is executed when the `new` operator is used to create an instance of the class. Constructors allow objects to be properly initialized. Clients can pass to the constructor information that ensures the object begins its life in a well-defined state. In this case, the radius of the circle is set. Since `radius` is private, a client would have no other way to create a circle object and make sure that its radius was a certain value.

- Both of the original methods (`circumference()` and `area()`) now have empty parameter lists. Clients do not provide any information when they call these methods.

After saving and compiling `Circle1`, try out this session:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> Circle1 c1 = new Circle1(10.5),
          c2 = new Circle1(2.0);
> c1.circumference()
65.97339
> c2.circumference()
12.56636
```

Let us carefully examine this session:

- First we create two `Circle1` objects:

  ```
  > Circle1 c1 = new Circle1(10.5),
            c2 = new Circle1(2.0);
  ```

  The radius of each circle is specified within the parentheses. This `new` expression calls the constructor, and the argument provided by the client is passed to the constructor.

- We see that `c1` and `c2` have different circumferences.

  ```
  > c1.circumference()
  65.97339
  > c2.circumference()
  12.56636
  ```

  We do not pass any information into the calls to `circumference()` because we do not need to. Each object maintains its own radius. It would be an error if we tried to pass a parameter to `circumference()` since `Circle1`'s `circumference()` method is not defined to accept any parameters. If you continue the interaction, you can demonstrate this:

  Interactions

  ```
  > c1.circumference(2.5)
  Error: No 'circumference' method in 'Circle1' with arguments:
  (double)
  ```

`Circle1` objects are *stateful*. There is a distinction between the objects referenced by `c1` and `c2`. Conceptually, one object is bigger than the other since `c1` has a larger radius than `c2`.[1] This is not the case with `CircleCalculator` objects. `CircleCalculator` objects are *stateless* because their `circumference()` methods always compute the same values given the same arguments. All `CircleCalculator` objects behave the same given the same inputs to their methods. Their variation in behavior was due to variation in clients' state, because the client could choose to pass different values as parameters.

A class may define more than one constructor. `Circle2` (7.2) includes two constructors and a new method:

[1] This size distinction is strictly conceptual, however. Both `c1` and `c2` occupy the same amount of computer memory—they each hold two integer values, their own copies of `PI` and `radius`, and all integers require four bytes of memory.


```
public class Circle2 {
    private final double PI = 3.14159;
    private double radius;
    public Circle2(double r) {
        radius = r;
    }
    //  No argument to the constructor makes a unit circle
    public Circle2() {
        radius = 1.0;
    }
    public double getRadius() {
        return radius;
    }
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 7.2: `Circle2`—adds a new constructor to the `Circle1` class

In `Circle2` (7.2):

- The first constructor initializes the instance variable `radius` to the value passed in by the client at the time of creation.
- The second constructor makes a unit circle (radius = 1).
- The new method `getRadius()` returns the value of the instance variable `radius`. It is important to note that this does not allow clients to tamper with `radius`; the method returns only a copy of `radius`'s value. This means clients can see the value of `radius` but cannot touch it. We say that `radius` is a *read-only* attribute.

We can test the constructors in the Interactions pane:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> Circle2 circle = new Circle2(10.5);
> circle.getRadius()
10.5
> Circle2 circle2 = new Circle2();
> circle2.getRadius()
1.0
```

The constructors all have the same name (the same name as the class). They must be differentiated by different parameter lists, however. Here, one constructor accepts a single `double` value, while the other accepts no parameters. If no constructors were defined, the radius would default to zero, since numeric instance variables are automatically initialized to zero unless constructors direct otherwise.

When a class has more than one constructor, we say its constructor is *overloaded*. Java permits methods to be overloaded as well, but we have little need to do so and will not consider overloaded methods here. `Circle3` (7.3) gives a slightly different version of `Circle2`.

```
public class Circle3 {
    private final double PI = 3.14159;
    private double radius;
    public Circle3(double r) {
        radius = r;
    }
    public Circle3() {
        this(1.0);  //  Defer work to the other constructor
    }
    public double getRadius() {
        return radius;
    }
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 7.3: `Circle3`—one constructor calls another

One of the constructors in `Circle3` (7.3) contains the statement

```
this(1.0);
```

Here the reserved word `this` is used like a method call. It can be used in this way only within a constructor body. It allows one constructor to call another constructor within the same class. Here the parameterless constructor provides a default value for the "main" constructor that is responsible for initializing `radius`. We say the parameterless constructor defers the work to the other constructor, passing it the default argument. In this case, for `Circle2` both constructors initialize `radius` using an assignment statement. In `Circle3` only the first constructor uses assignment directly; the second constructor supplies a default value of 1.0 and invokes the services of the first constructor.

Constructors are optional. `CircleCalculator` (4.1) had no constructor. The compiler provides a *default constructor* for a class if the programmer omits a constructor definition for that class. The default constructor simply initializes all otherwise uninitialized numeric instance variables to zero. If a programmer defines at least one constructor for a class, the compiler will not create a default constructor for that class.

Constructors can do more than just initialize variables. They can do anything that methods can do.

7.2 Traffic Light Example

A traffic light is a real-world object that has a readily apparent state. A simple traffic light is shown in Figure 7.1. It can be red, yellow, green, or off (no lamps lit). Typically only one lamp is illuminated at one time. We wish to

Figure 7.1: A simple traffic light to be modeled in software

model such a traffic light in software. Our immediate purpose is to give us more experience defining classes and using objects, but such a software model could be put to good use in a program that a civil engineer might use to visualize a traffic flow simulation for an intersection or region of a city.

So far we have dealt with items of a mathematical nature. We have have used numbers in computations, and the state of one of our circle objects can be represented by a number—its radius. Our traffic light model's behavior seems to have nothing to do with numbers or mathematics. It changes colors, and colors are not numbers. We do, however, have all the tools at our disposal to be able to model such an object.

First, we must describe exactly what one of our traffic light model objects is expected to do. We keep it simple here, but we will soon see that this simplicity will not necessarily hamper us when we want to make a more elaborate traffic light object. Our traffic light model must be able to:

- assume at the time of its creation a legitimate color, one of red, yellow, or green,
- change its color from its current color to the color that should follow its current color in the normal order of a traffic light's cycle, and
- indicate its current color.

For now, our tests will be text based, not graphical. A red traffic light will be printed as

```
[(R) ( ) ( )]
```

A yellow light is rendered as

```
[( ) (Y) ( )]
```

and a green light is

```
[( ) ( ) (G)]
```

(We consider a different text representation in § 10.2.)

While numbers may seem irrelevant to traffic light objects, in fact the clever use of an integer value can be used to represent a traffic light's color. Suppose we store an integer value in each traffic light object, and that integer represents the light's current color. We can encode the colors as follows:

- 1 represents red
- 2 represents yellow
- 3 represents green

Since remembering that 2 means “yellow” can be troublesome for most programmers, we define constants that allow symbolic names for colors to be used instead of the literal numbers. `RYGTrafficLight` (7.4) shows our first attempt at defining our traffic light objects:

```
//  The "RYG" prefix stands for the Red, Yellow, and Green colors that
//  this light can assume.
public class RYGTrafficLight {
    //  Colors are encoded as integers
    private final int RED    = 1;
    private final int YELLOW = 2;
    private final int GREEN  = 3;

    //  The current color of the traffic light
    private int color;

    //  The constructor ensures that the light has a valid color
    //  (starts out red)
    public RYGTrafficLight() {
        color = RED;
    }

    //  Changes the light's color to the next legal color in its normal cycle.
    //  The light's previous color is returned.
    //  The pattern is:
    //      red --> green --> yellow
    //       ^                  |
    //       |                  |
    //       +------------------+
    public void change() {
        if (color == RED) {
            color = GREEN;  //  Green follows red
        } else if (color == YELLOW) {
            color = RED;    //  Red follows yellow
        } else if (color == GREEN) {
            color = YELLOW; //  Yellow follows green
        }
    }

    //  Render the individual lamps
    public String drawLamps() {
        if (color == RED) {
            return "(R) ( ) ( )";
        } else if (color == GREEN) {
            return "( ) ( ) (G)";
        } else if (color == YELLOW) {
```



```
            return "( ) (Y) ( )";
        } else {
            return " **Error** ";  //  Defensive programming
        }
    }
    //  Render the light in a crude visual way
    public String draw() {
        return "[" + drawLamps() + "]";
    }
}
```

Listing 7.4: `RYGTrafficLight`—simulates a simple traffic light

Some remarks about `RYGTrafficLight` (7.4):

- Comments are supplied throughout to explain better the purpose of the code.
- Color constants are provided to make the code easier to read.

  ```
  private final int RED    = 1;
  private final int YELLOW = 2;
  private final int GREEN  = 3;
  ```

  These constants are private because clients do not need to access them.
- The only way a client can influence a light's color is through `change()`. Note that the client cannot change a light's color to some arbitrary value like 34. The `change()` method ensures that light changes its color in the proper sequence for lights of this kind.
- The `draw()` and `drawLamps()` methods render the traffic light using text characters. The drawing changes depending on the light's current color. If because of a programming error within the `RYGTrafficLight` class the `color` variable is set outside of its legal range 0...3, the value returned is `"[ **Error** ]"`. This is a form of *defensive programming*. While we expect and hope that everything works as it should, we make provision for a `RYGTrafficLight` object that is found in an illegal state. Should this string ever appear during our testing we will know our code contains a logic error and needs to be fixed. The `draw()` method renders the "frame" and uses the `drawLamps()` method to draw the individual lamps.

The following interactive sequence creates a traffic light object and exercises it through two complete cycles:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> t = new RYGTrafficLight();
> t.draw()
"[(R) ( ) ( )]"
> t.change(); t.draw()
"[( ) ( ) (G)]"
> t.change(); t.draw()
"[( ) (Y) ( )]"
> t.change(); t.draw()
```



```
"[(R) ( ) ( )]"
> t.change(); t.draw()
"[( ) ( ) (G)]"
> t.change(); t.draw()
"[( ) (Y) ( )]"
> t.change(); t.draw()
"[(R) ( ) ( )]"
```

Now we are in better position to see the value of the `public` and `private` access specifiers. Suppose the `color` instance variable were declared `public`. Clients then could create a traffic light object and set its color to, say, 34. The value 34 is not interpreted as one of the colors red, yellow, or green. By making `color` private, the `RYGTrafficLight` class developer can protect her traffic light objects from being abused either accidentially or maliciously. Clients cannot set a traffic light's color outside of its valid range. In fact, the client would need access to the source code for `RYGTrafficLight` (7.4) to even know that an integer was being used to control the light's color. (An alternate implementation might use three Boolean instance variables named `red`, `yellow`, and `green`, and a red light would have `red` = `true`, `yellow` = `false`, and `green` = `false`.)

7.3 `RationalNumber` Example

In mathematics, a *rational number* is defined as the ratio of two integers, where the second integer must be nonzero. Commonly called a *fraction*, a rational number's two integer components are called *numerator* and *denominator*. Rational numbers possess certain properties; for example, two fractions can have different numerators and denominators but still be considered equal (for example, $\frac{1}{2} = \frac{2}{4}$). Rational numbers may be added, multiplied, and reduced to simpler form. `RationalNumber` (7.5) is our first attempt at a rational number type:

```
public class RationalNumber {
    private int numerator;
    private int denominator;

    //  num is the numerator of the new fraction
    //  den is the denominator of the new fraction
    public RationalNumber(int num, int den) {
        if (den != 0) {  //  Legal fraction
            numerator = num;
            denominator = den;
        } else {  //  Undefined fraction changed to zero
            numerator = 0;
            denominator = 1;
        }
    }

    //  Provide a human-readable string for the fraction, like "1/2"
    public String show() {
        return numerator + "/" + denominator;
    }

    //  Compute the greatest common divisor for two integers m and n
```



```
    //  Uses Euclid's algorithm, circa 300 B.C.
    //  This method is meant for internal use only; hence, it is
    //  private.
    private int gcd(int m, int n) {
        if (n == 0) {
            return m;
        } else {
            return gcd(n, m % n);
        }
    }

    //  Returns a new RationalNumber consisting of the current
    //  fraction reduced to lowest terms
    public RationalNumber reduce() {
        int factor = gcd(numerator, denominator);
        return new RationalNumber(numerator/factor,
                                  denominator/factor);
    }

    //  Returns a new RationalNumber consisting of the sum of this
    //  fraction and another fraction (other).  The result is reduced
    //  to lowest terms.
    public RationalNumber add(RationalNumber other) {
        int num1 = numerator * other.denominator,
            num2 = other.numerator * denominator;
        RationalNumber result = new RationalNumber(num1 + num2,
                                  denominator * other.denominator);
        return result.reduce();
    }
}
```

Listing 7.5: `RationalNumber`—defines a rational number type

In `RationalNumber` (7.5):

- Two instance variables:

  ```
  private int numerator;
  private int denominator;
  ```

  define the state of a rational number.

- The constructor ensures that the denominator of the new fraction will not be zero.

  ```
  public RationalNumber(int num, int den) {
      if (den != 0) {  . . .
  ```

  Since the instance variables `numerator` and `denominator` are private and none of the methods change these variables, it is impossible to have an undefined rational number object.

- The `show()` method:

```
public String show() { . . .
```

allows a rational number to be displayed in a human-readable form, as in "1/2."

- The `gcd()` method:

```
private int gcd(int m, int n) { . . .
```

computes the greatest common divisor (also called greatest common factor) or two integers. This method was devised by the ancient Greek mathematician Euclid around 300 B.C. This method is interesting because it calls itself. A method that calls itself is called a *recursive* method. All proper recursive methods must involve a conditional (such as an `if`) statement. The conditional execution ensures that the method will not call itself under certain conditions. Each call to itself should bring the method closer to the point of not calling itself; thus, eventually the recursion will stop. In an improperly written recursive method, each recursive call does not bring the recursion closer to its end, and infinite recursion results. Java's runtime environment, catches such infinite recursion and generates a runtime error.

- The `reduce()` method:

```
public RationalNumber reduce() {
    int factor = gcd(numerator, denominator);
    return new RationalNumber(numerator/factor,
                              denominator/factor);
}
```

returns a new rational number. It uses the `gcd()` method to find the greatest common divisor of the numerator and denominator. Notice that a new rational number object is created and returned by the method.

- The `add()` method:

```
public RationalNumber add(RationalNumber other) {
```

accepts a reference to a rational number as a parameter and returns a new rational number as a result. It uses the results of the following algebraic simplification:

$$\frac{a}{b}+\frac{c}{d} = \left(\frac{d}{d}\right)\cdot\frac{a}{b}+\frac{c}{d}\cdot\left(\frac{b}{b}\right) = \frac{ad}{bd}+\frac{bc}{bd} = \frac{ad+bc}{bd}$$

The expression

```
new RationalNumber(num1 + num2, denominator * other.denominator)
```

creates a new `RationalNumber` object. We assign its value to a local `RationalNumber` variable named `result` and then return the reduced value of `result`. The call to `reduce()` creates a new rational number object from the rational number just created; thus, `add()` creates two rational number objects each time it is called, but it returns only one of them (the reduced form).

7.4 Object Aliasing

We have seen that Java provides some built-in primitive types such as `int`, `double`, and `boolean`. We also have seen how to define our own programmer-defined types with classes. When we declare a variable of a class type, we call the variable a *reference variable*.

Consider a very simple class definition, `SimpleObject` (7.6):

```
public class SimpleObject {
    public int value;
}
```

Listing 7.6: `SimpleObject`—a class representing very simple objects

The variable `value` is public, so it is fair game for any client to examine and modify. (`SimpleObject` objects would not be very useful within a real program.)

Object references and primitive type variables are fundamentally different. We do not use the `new` operator with primitive types, but we must create objects using `new`. The interpreter illustrates this difference:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> SimpleObject obj;
> obj.value = 2;
NullPointerException:
  at sun.reflect.UnsafeFieldAccessorImpl.ensureObj(Unsafe ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.getInt(Un ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.get(Unsaf ...
  at java.lang.reflect.Field.get(Field.java:357)
> obj.value
NullPointerException:
  at sun.reflect.UnsafeFieldAccessorImpl.ensureObj(Unsafe ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.getInt(Un ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.get(Unsaf ...
  at java.lang.reflect.Field.get(Field.java:357)
> int number;
> number = 2;
> number
2
```

Simply declaring `obj` is not enough to actually use it, but simply declaring `number` is sufficient to allow it to be used in any legal way `int`s can be used.

For another example of the differences between primitive types and objects, consider the following interactive sequence:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int i1 = 1, i2 = 2;
```



```
> i1
1
> i2
2
> i1 = i2;
> i1
2
> i2
2
> i1 = 0;
> i1
0
> i2
2
> SimpleObject o1 = new SimpleObject(),
               o2 = new SimpleObject();
> o1.value = 1;
> o1.value
1
> o2.value = 2;
> o2.value
2
> o1 = o2;
> o1.value
2
> o2.value
2
> o1.value = 0;
> o1.value
0
> o2.value
0
```

When one integer variable is assigned to another, as in

```
i1 = i2;
```

the value of the right-hand variable is assigned to the left-hand variable. `i1` takes on the value of `i2`. Reassigning `i1` does not effect `i2`; it merely gives `i1` a new value. `i1` and `i2` are distinct memory locations, so changing the value of one (that is, storing a new value in its memory location) does not change the value of the other. Figure 7.2 illustrates primitive type assignment.

A reference, on the other hand, is different. An object reference essentially stores the memory address of an object. A reference *is* a variable and is therefore stored in memory, but it holds a memory address, not a "normal" numeric data value.[2] In the statement

```
o1 = new SimpleObject();
```

[2] In fact, a reference *is* a number since each memory location has a unique numeric address. A reference is interpreted as an address, however, and the Java language does not permit arithmetic operations to be performed on references.

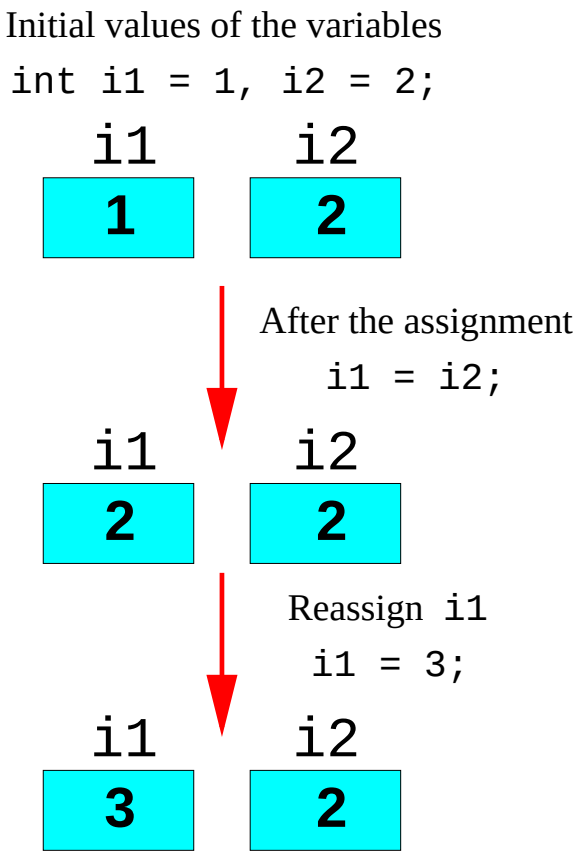

Figure 7.2: Assignment of Primitive Types

the `new` operator allocates memory for a new `SimpleObject` object and returns the address where the object was allocated. This address is then stored in `o1`. Interacting with the object, as in

```
o1.value = 2;
```

will affect the object at the address stored in `o1`. The statement

```
i1 = i2;
```

causes `i1` and `i2` to hold identical integer values. The statement

```
o1 = o2;
```

causes `t1` and `t2` to hold identical object *addresses*. If an object is modified via `o1`, then the object referenced via `o2` is also modified because `o1` and `o2` refer to exactly the same object! When two references refer to the same object we say that the references are *aliases* of each other. It is helpful to visualize object references as variables that point to the object they reference. Figure 7.3 shows object references as pointers (arrows).

Pictorially, `o1` is an alias of `o2` because both `o1` and `o2` point to the same object.

7.5 Summary

- Real-world objects have state; Java objects model state via instance variables.

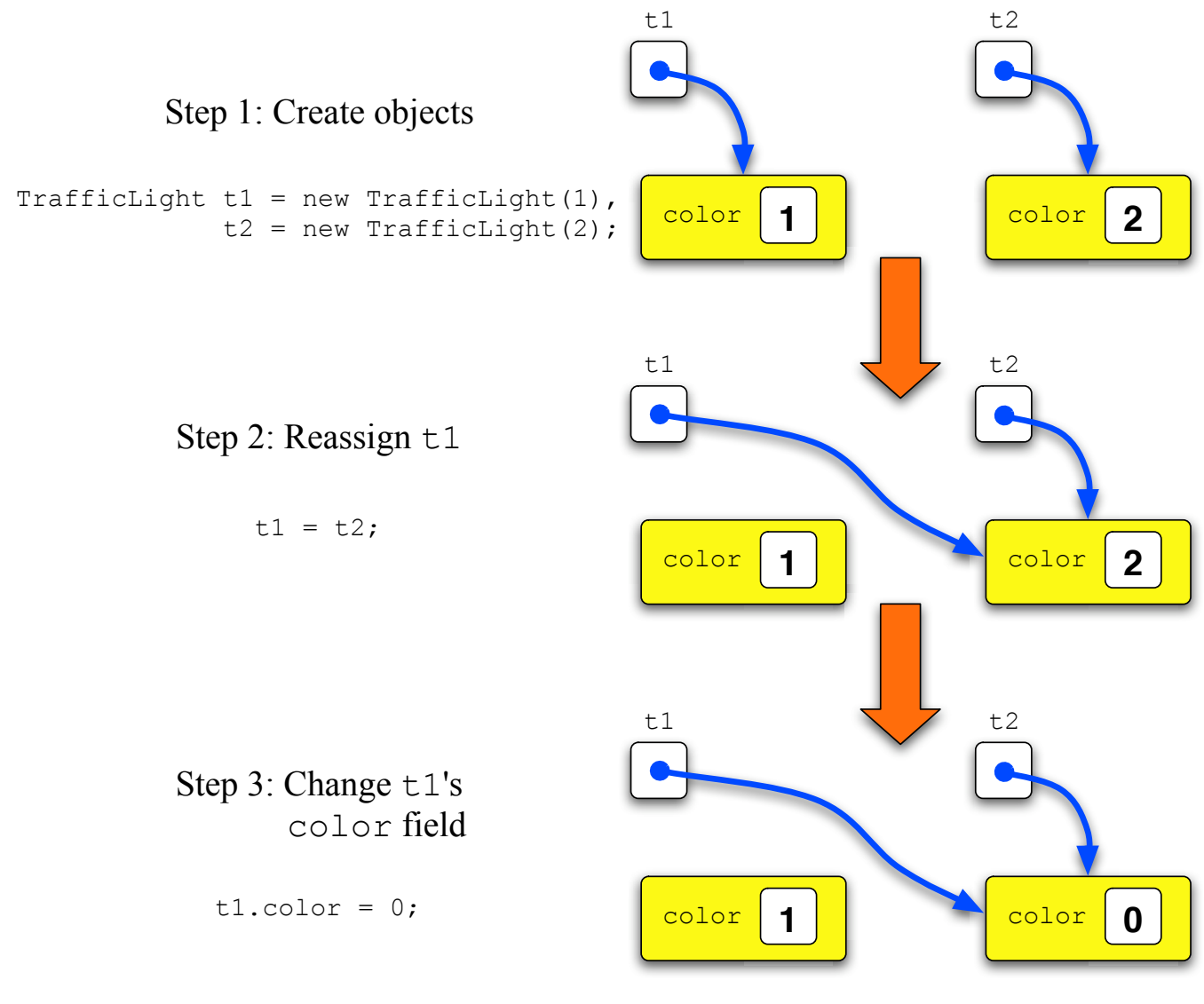

Figure 7.3: Assignment of Reference Types

- A class's constructor is called when an instance (object) is created via `new`.
- A constructor is similar to a method (but it technically is not a method), but it specifies no return type and is "called" indirectly only when `new` is used.
- A recursive method calls itself in its definition.
- When two object references refer to the same object we say one reference aliases the other reference.

7.6 Exercises

1. How does an object maintain its state?
2. In what three ways are a constructor different from a method?
3. What does it mean for a constructor to be overloaded?
4. How can one constructor directly call another constructor within the same class?

5. What does the compiler do if you do not provide a constructor for a class?

6. Is it legal for a method to specify an object reference as a parameter?

7. Is it legal for a method to return an object reference as a result?

8. What does it mean for a method to be recursive?

9. When speaking on objects, what is an alias?

10. Implement the alternate version of `RYGTrafficLight` mentioned in 7.2 that uses the three Boolean variables `red`, `yellow`, and `green` instead of the single integer `color` variable. Test it out to make sure it works.

11. Create a class named `Point` that is used to create (x,y) point objects. The coordinates should be `double` values. Provide a method that computes the distance between two points.

