Recall that a **variable** is a named place to store a value, and that every variable has a **type**, or range of values that it may hold.

Some types are called **primitive** types. These types are built in to the Java language and encompass ranges of whole numbers (int, long, short, and byte), numbers with fractional parts (float and double), individual characters (char), and truth values (boolean). A variable with a **primitive type** contains a value within the range of that primitive type.

Some other types are called **reference** types. Reference types include class types, interface types (discussed in chapter 11), and array types (discussed in chapter 8). Reference types are so called because a variable with a reference type will contain a reference to (or address of) an object. A variable with a **reference type** will contain a reference to a particular kind of object or array. Note that such a variable will not contain an actual object or array.

Think about why this difference is important. Why do reference variables hold references to objects instead of actual objects?

---

### The dialogue at right refers to the following program fragment:

```java
int a = 5;
int b = a;
/* P1 */

Integer w = new Integer(a);
Integer x = w;
/* P2 */

Clicker c = new Clicker();
Clicker d = new Clicker();
Clicker e = c;
/* P3 */
```

---

| **What is the result of the statement** `a = a + 1;`? | It increments the value stored in `a` by 1. |
| **What would happen if I added the above statement to the program at the point marked P1?** | `a` would contain the value 5 before that statement executed and 6 afterwards. Note that the value of `b` would not change. |
| **What kind of variable is `w`?** | `w` is a **reference variable**. Do you know how to tell? |
| **Well, I know it is of type Integer, and that’s not a primitive type.** | Yes, and the value that `w` gets is constructed with operator `new`, which always returns a reference. Is there still another way to tell? |
| **Yes! Since Integer is capitalized, I can be pretty sure that it is a class name, as long as the programmer is following the Java naming conventions and not trying to trick me.** | That’s right! Of course, you know that `Integer` is a type because of where it appears in the line of code. You know that it isn’t a primitive type, since there are only a few of those to remember; and you know it's not an array, since we haven't learned about them yet. Therefore, even if the programmer is being deliberately tricky, we can be fairly certain that `Integer` is a reference type, and that a variable of type `Integer` holds a reference to an object. |
| **What is the value of `w`?** | `w` contains a reference to an `Integer` object. |
| **No, seriously, what is the value of `w`?** | Seriously, `w` contains a reference to an `Integer` object. The particular `Integer` object that `w` contains a reference to “wraps” (or represents) the value 5. |
| **Yikes! How can we notate that?** | Java programmers will often use **memory diagrams** to talk about objects, references, and variables. |
| **Can you show me an example?** | Sure. This diagram shows the value of `w` after it's initialized: |

```
  w  Integer ...
```

| **I assume that the diagram means that `w` points to an instance of class `Integer`. What’s up with the ellipsis? Are you hiding details from me?** | You assume correctly! The ellipsis is just there to save space. Usually when we make a memory diagram, we'll show the instance fields of each individual object. In this case, we don't know what the instance fields of `Integer` are, but we have an idea that the `Integer` pointed to by `w` might "wrap" the `int` value 5. |
| **What is the result of the statement** `Integer x = w;`? | It declares a new variable of type `Integer` called `x`. Furthermore, it initializes `x` to contain the same value as that stored in `w`. |
Here is the source code for the instantiable Clicker class:

```java
public class Clicker {
    private int ct;

    public Clicker() {
        ct = 0;
    }

    public void click() {
        ct = ct + 1;
    }

    public void reset() {
        ct = 0;
    }

    public int inspect() {
        return ct;
    }
}
```

We'll use the following symbols in our memory diagrams:

- **o**: reference variable, pointing to object (not shown)
- **x**: primitive-type variable with integer value
- **Clicker**: object with class name and field values

---

What does it mean for "x to contain the same value as that stored in w?"

In the case of reference variables, it means that x and w refer to the same object. (If x and w were primitive-type values, each would have its own copy of some value.)

A ha! This is sort of like how I can refer to the same real-world place by saying "1210 W. Dayton Street" and "the Computer Sciences building at UW-Madison." How would we notate this relationship in a memory diagram?

You're right. The memory diagram would look like this:

```
\[
\text{\textbf{x} \rightarrow \text{Integer}}
\]
```

What is a Clicker object for?

It models an incrementing counter, like one that an usher at a concert venue or a cinema might use.

What happens to a Clicker object when you send it a click() message?

Its count, as stored in its ct field, is incremented by one. You can inspect the count of a Clicker object by sending it an inspect() message.

Let's say that I insert the following lines of code into the program at the point labeled P3:

```java
System.out.println(c.inspect());
System.out.println(d.inspect());
System.out.println(e.inspect());
c.click();
System.out.println(d.inspect());
System.out.println(e.inspect());
```

What output would the program produce?

Well, at point P3, memory looks like so, with c and e pointing to the same Clicker:

```
\[
\begin{align*}
\text{c} & \quad \text{Clicker} & \quad \text{ct} = 0 \\
\text{d} & \quad \text{Clicker} & \quad \text{ct} = 0 \\
\text{e} & \quad \text{Clicker} & \quad \text{ct} = 0 \\
\end{align*}
\]
```

Right, so it's not surprising that changing the ct field of the Clicker pointed to by c (by sending the click() message) would change the ct field of the Clicker pointed to by e (that we access by sending an inspect() message). Since we're referring to the same object by two different names, there's some potential for confusion!

I'll have to be very careful when I have multiple references to the same object.

I couldn't have said it better myself.

By the way, "instantiable" isn't in any dictionary I can find, and it makes my spell-checker complain. Could it possibly be a real word?

Your textbook certainly thinks so.