return new Location(xMid, yMid);

Either way—with or without the local variable—is fine.
Here's an example to show how the static midpoint method is used. The method creates two locations and then computes their midpoint:

Location low = new Location(0, 0);
Location high = new Location(1000, 5280);
Location medium = Location midpoint(low, high);

In this example, the answer from the midpoint method is stored in a variable called medium. After the three statements, we have three locations:

Programming Tip

How to Choose the Names of Methods

Accessor methods: The name of a boolean accessor method will usually begin with "is" followed by an adjective (such as "isOn"). Methods that convert to another kind of data start with "to" (such as "toString"). Other accessor methods start with "get" or some other verb followed by a noun that describes the return value (such as "getFlow").

Modification methods: A modification method can be named by a descriptive verb (such as "shift") or by a short verb phrase (such as "shutdown").

Static methods that return a value: Try to use a noun that describes the return object (such as "distance" or "midpoint").

Rules like these make it easier to determine the purpose of a method.

Java's Object Type

One of the Location methods is an accessor method called equals with this heading:

public boolean equals(Object obj)

An accessor method with this name has a special meaning in Java. Before we discuss that meaning, you need to know a bit about the parameter type Object. In Java, Object is a kind of "super data type" that encompasses all data except the eight primitive types. So a primitive variable (byte, short, int, long, char, float, double, or boolean) is not an Object, but everything else is. A String is an Object, a Location is an Object, and even an array is an Object.

Using and Implementing an Equals Method

As your programming progresses, you'll learn a lot about Java's Object type, but to start, you need just a few common patterns that use Object. For example, many classes implement an equals method with the heading we have seen. An equals method has one argument: an Object called obj. The method should return true if obj has the same value as the object that activated the method. Otherwise, the method returns false. Here is an example to show how the equals method works for the Location class:

Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = new Location(10, 0); // Declare s at coordinates (10,0)

After these two declarations, we have two separate locations:

In this example, p and s refer to two separate objects with different values (their y coordinates are different), so both p.equals(s) and s.equals(p) are false.

Here's a slightly different example:

Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = new Location(10, 0); // Declare s at coordinates (10,0)
s.shift(0, 2); // Move s to (10,2)
We have the same two declarations, but afterward we shift the \( y \) coordinate of \( s \) so that the two separate locations have identical values, like this:

![Diagram of Location objects](image)

Now \( p \) and \( s \) refer to identical locations, so both \( p.equals(s) \) and \( s.equals(p) \) are true. However, the test \( (p == s) \) is still false. Remember that \( (p == s) \) returns true only if \( p \) and \( s \) refer to the exact same location (as opposed to two separate locations that happen to contain identical values).

The argument to the \( equals \) method can be any object, not just a location. For example, we can try to compare a location with a string, like this:

```
Location p = new Location(10, 2);
System.out.println(p.equals("10, 2")); // Prints false.
```

This example prints false; a Location object is not equal to the string "10, 2" even if they are similar. You can also test to see whether a location is equal to null:

```
Location p = new Location(10, 2);
System.out.println(p.equals(null)); // Prints false.
```

The location is not null, so the result of \( p.equals(null) \) is false. Be careful with the last example: The argument to \( p.equals \) may be null and the answer will be \( false \). However, when \( p \) itself is null, it is a programming error to activate any method of \( p \). Trying to activate \( p.equals \) when \( p \) is null results in a \( NullPointerException \) (see page 55).

Now you know how to use an \( equals \) method. How do you write an \( equals \) method so that it returns true when its argument has the same value as the object that activates the method? A typical implementation follows an outline that is used for the \( equals \) method of the Location class, as shown here:

```java
public boolean equals(Object obj)
{
    if (obj is actually a Location)
    {
        // Figure out whether the location that obj refers to has the same value as the location that activated this method. Return true if they are the same; otherwise, return false.
    }
    else
    {
        return false;
    }
}
```

The method \( equals \) creates a Location object that can be compared to \( obj \) and returns \( true \) if they are equal, or \( false \) otherwise.
The method starts by determining whether obj actually refers to a Location object. In pseudocode, we wrote this as “obj is actually a Location.” In Java, this is accomplished with the test (obj instanceof Location). This test uses the keyword instanceof, which is a boolean operator. On the left of the operator is a variable, such as obj. On the right of the operator is a class name, such as Location.

The test returns true if it is valid to convert the object (obj) to the given data type (Location). In our example, suppose that obj does not refer to a valid Location. It might be some other type of object, or perhaps it is simply null. In either case, we go to the else-statement and return false.

On the other hand, suppose that (obj instanceof Location) is true, so the code enters the first part of the if-statement. Then obj does refer to a Location object. We need to determine whether the x and y coordinates of obj are the same as the location that activated the method. Unfortunately, we can’t just look at obj.x and obj.y because the compiler thinks of obj as a bare object with no x and y instance variables. The solution is an expression (Location) obj. This expression is called a typecast, as if we were pouring obj into a casting mold that creates a Location object. The expression can be used to initialize a Location reference variable, like this:

```
Location candidate = (Location) obj;
```

The typecast, on the right side of the declaration, consists of the new data type (Location) in parentheses, followed by the reference variable that is being cast. After this declaration, candidate is a reference variable that refers to the same object that obj refers to. However, the compiler does know that candidate refers to a Location object, so we can look at candidate.x and candidate.y to see if they are the same as the x and y coordinates of the object that activated the equals method. The complete implementation looks like this:

```
public boolean equals(Object obj) {
    if (obj instanceof Location) {
        Location candidate = (Location) obj;
        return (candidate.x == x) && (candidate.y == y);
    } else {
        return false;
    }
}
```

The implementation has the return statement:

```
return (candidate.x == x) && (candidate.y == y);
```

The boolean expression in this return statement is true if candidate.x and candidate.y are the same as the instance variables x and y. As with any method, these instance variables come from the object that activated the method. For future reference, the details of using a typecast are given in Figure 2.7.
**CLASS CAST EXCEPTION**

Suppose you have a variable such as `obj`, which is an `Object`. You can try a typecast to use the object as if it were another type. For example, we used the typecast `Location candidate = (Location) obj`.

What happens if `obj` doesn't actually refer to a `Location` object? The result is a runtime exception called `ClassCastException`. To avoid this, you must ensure that a typecast is valid before trying to execute the cast. For example, the `instanceof` operator can validate the actual type of an object before a typecast.

Every Class Has an Equals Method

You may write a class without an equals method, but Java automatically provides an equals method anyway. The equals method that Java provides is actually taken from the `Object` class, and it works exactly like the `==` operator. In other words, it returns true only when the two objects are the exact same object—but it returns false for two separate objects that happen to have the same values for their instance variables.

**FIGURE 2.7** Typecasts

A Simple Pattern for Typecasting an Object

A common situation in Java programming is a variable or other expression that is an `Object`, but the program needs to treat the `Object` as a specific data type such as `Location`. The problem is that when a variable is declared as an `Object`, that variable cannot immediately be used as if it were a `Location` (or some other type). For example, consider the parameter `obj` in the equals method of the `Location` class:

```java
public boolean equals(Object obj)
```

Within the implementation of the equals method, we need to treat `obj` as a `Location` rather than a mere `Object`. The solution has two parts: (1) Check that `obj` does indeed refer to a valid `Location`, and (2) Declare a new variable of type `Location` and initialize this new variable to refer to the same object that `obj` refers to, like this:

```java
public boolean equals(Object obj) {
    if (obj instanceof Location) {
        Location candidate = (Location) obj;
        ...
    }
    // The expression (Location) obj, used in the declaration of candidate, is a typecast to tell the compiler that obj may be used as a Location.
    return true;
}
```

As you can see, this pattern can be applied to any two data types that are related in this way to affect type safety.
Using and Implementing a Clone Method

Another feature of our Location class is a method with this heading:

```java
public Location clone()
```

The purpose of a clone method is to create a copy of an object. The copy is separate from the original so that subsequent changes to the copy won’t alter the original, nor will subsequent changes to the original alter the copy. Here’s an example of using the clone method for the Location class:

```java
Location p = new Location(10, 2); // Declare p at (10,2)
Location s = p.clone(); // Initialize as a copy of p
```

The expression `p.clone()` activates the clone method for `p`. The method creates and returns an exact copy of `p`, which we use to initialize the new location `s`. After these two declarations, we have two separate locations, as shown in this picture:

![Diagram showing two Location objects](image)

As you can see, `s` and `p` have the same values for their instance variables, but the two objects are separate. Changes to `p` will not affect `s`, nor will changes to `s` affect `p`.

**Older Java Code Requires a Typecast for Clones**

Prior to Java 5.0, the data type of the return value of the `clone` method was always an `Object` and not a specific type such as `Location`. Because of this requirement, the `clone` return value could not be used directly in older versions of Java. For example, we could not write a declaration:

```java
Location s = p.clone();
```

Instead, we must apply a typecast to the `clone` return value, converting it to a `Location` before we use it to initialize the new variable `s`, like this:

```java
Location s = (Location) p.clone(); // Typecast required for older Java compilers
```
Cloning is considerably different than using an assignment statement. For example, consider this code that does not make a clone:

```java
Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = p;                     // Declare s and make it refer
                                      // to the same object that p
                                      // refers to
```

After these two declarations, we have just one location, and both variables refer to this location:

![Diagram showing Location p and Location s referencing the same object]

This is the situation with an ordinary assignment. Subsequent changes to the object that p refers to will affect the object that s refers to because there is only one object.

You now know how to use a clone method. How do you implement such a method? You should follow a three-step pattern:

1. **Modify the Class Head.** You must add the words “implements Cloneable” in the class head, as shown here for the Location class:

   ```java
   public class Location implements Cloneable
   ```

   The modification informs the Java compiler that you plan to implement certain features that are specified elsewhere in a format called an interface. The full meaning of interfaces will be discussed in Chapter 5. At the moment, it is enough to know that implements Cloneable is necessary when you implement a clone method.

2. **Use super.clone to Make a Copy.** The implementation of a clone method should begin by making a copy of the object that activated the method. The best way to make the copy is to follow this pattern from the Location class:
public Location clone() {
    // Clone a Location object.
    Location answer;
    try {
        answer = (Location) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException
                       ("This class does not implement Cloneable.");
    }

    In an actual implementation, you would use the name of your own class (rather than Location), but otherwise you should follow this pattern exactly.

    It's useful to know what's happening in this pattern. The pattern starts by declaring a local Location variable called answer. We then have this block:

    try {
        answer = (Location) super.clone();
    }

    This is an example of a try block. If you plan extensive use of Java exceptions, you should read all about try blocks in Appendix C. But for your first try block, all you need to know is that the code in the try block is executed, and the try block will be able to handle some of the possible exceptions that may arise in the code. In this example, the try block has just one assignment statement: answer = (Location) super.clone(). The right side of the assignment activates a method called super.clone(). This is actually the clone method from Java's Object type. It checks that the Location class specifies that it "implements Cloneable" and then correctly makes a copy of the location, assigning the result to the local variable answer.

    After the try block is a sequence of one or more catch blocks. Each catch block can catch and handle an exception that may arise in the try block. Our example has one catch block:

    catch (CloneNotSupportedException e) {
        throw new RuntimeException
               ("This class does not implement Cloneable.");
    }

    This catch block will handle a CloneNotSupportedException. This exception is thrown by the clone method from Java's Object type when a programmer tries to call super.clone(), without including the implements Cloneable
clause as part of the class definition. The best solution is to throw a new 
RuntimeException, which is the general exception used to indicate a program-
ner error. In Chapter 13 (page 694), we’ll see that the try/catch block isn’t 
always needed, but until we deal with the Chapter 13 topics (extended classes) 
in detail, we’ll always use the format just shown.

After the try and catch blocks, the local variable answer refers to an exact 
copy of the location that activated the clone method, and we can move to the 
third part of the clone implementation.

3. Make Necessary Modifications and Return. The answer is present, and it 
refers to an exact copy of the object that activated the clone method. Sometimes 
further modifications must be made to the copy before returning. You’ll see the 
reasons for such modifications in Chapter 3. However, the Location clone needs 
no modifications, so the end of the clone method consists of just the return 
statement: return answer.

The complete clone implementation for the Location class looks like this, 
including an indication of the likely cause of the CloneNotSupportedException:

```java
public Location clone() 
{    // Clone a Location object.
    Location answer;

    try
    {
        answer = (Location) super.clone();
    }
    catch (CloneNotSupportedException e)
    {    // This exception should not occur. But if it does, it would indicate a
        // programming error that made super.clone unavailable. The
        // most common cause would be forgetting the
        // "implements Cloneable" clause at the start of the class.
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }

    return answer;
}
```

The method returns the local variable, answer, which is a Location object. 
This is allowed, even though the return type of the clone method is Object. A 
Java Object can be anything except the eight primitive types. It might be better 
if the actual return type of the clone method was Location rather than Object. 
Using Location for the return type would be more accurate and would make 
the clone method easier to use (without having to put a typecast with every 
usage). Unfortunately, the improvement is not allowed: The return type of the 
clone method must be Object.
Always Use super.clone for Your Clone Methods

Perhaps you thought of a simpler way to create a clone. Instead of using super.clone and the try/catch blocks, could you write this code:

    Location answer = new Location(x, y);
    return answer;

You could combine these into one statement: return new Location(x, y). This creates and returns a new location, using the instance variables x and y to initialize the new location. These instance variables come from the location that activated the clone method, so answer will indeed be a copy of that location. This is a nice direct approach, but the direct approach will encounter problems when we start building new classes that are based on existing classes [See Chapter 13]. Therefore, it is better to use the pattern with super.clone and a try/catch block.

Programming Tip

When to Throw a Runtime Exception

A RuntimeException is thrown to indicate a programming error. For example, the clone method from Java's Object type is not supposed to be called by an object unless that object's class has implemented the Cloneable interface. If we detect that the exception has been thrown by the Object clone method, then the programmer probably forgot to include the "implements Cloneable" clause.

When you throw a RuntimeException, include a message with your best guess about the programming error.

A Demonstration Program for the Location Class

As one last example, let's look at a program that creates two locations called still and mobile. Both are initially placed at \( x = -2 \) and \( y = -1.5 \), as shown in Figure 2.8(a). To be more precise, the still location is placed at this spot, and then mobile is initialized as a clone of the still location. Because the mobile location is a clone, later changes to one location will not affect the other.

The program prints some information about both locations, and then the mobile location undergoes two 90° rotations, as shown in Figure 2.8(b). The information about the locations is then printed a second time.
Widening Conversions

Here's some code to show how an assignment can be made to an `Object` variable. The code declares a `String` with the value "Objection overruled!". A second `Object` variable is then declared and made to refer to the same string.

```java
String s = new String("Objection overruled!");
Object obj;

obj = s;
```

At this point, there is only one string—"Objection overruled!"—with both `s` and `obj` referring to this one string, as shown here:

```
String s          Object obj
   A String object "Objection overruled!"
```

Assigning a specific kind of object to an `Object` variable is an example of a widening conversion. The term "widening" means that the `obj` variable has a "wide" ability to refer to different things; in fact, `obj` is very "wide" because it could refer to any kind of object. On the other hand, the variable `s` is relatively narrow with regard to the things that it can refer to—`s` can refer to only a `String`.

### Widening Conversions with Reference Variables

Suppose `x` and `y` are reference variables. An assignment `x = y` is a **widening conversion** if the data type of `x` is capable of referring to a wider variety of things than the type of `y`.

**Example:**

```java
String s = new String("Objection overruled!");
Object obj;
obj = s;
```

Java permits all widening conversions.
Narrowing Conversions

After the widening conversion `obj = s`, our program can continue to do other things, perhaps even making `s` refer to a new string, as shown here:

```java
String s = new String("Objection overruled!");
Object obj;

obj = s;
s = new String("Make it so.");
```

At this point, `s` refers to a new string, and `obj` still refers to the original string, as shown here:

At a later point in the program, we can make `s` refer to the original string once again with an assignment statement, but the assignment needs more than merely `s = obj`. In fact, the Java compiler forbids the assignment `s = obj`. As far as the compiler knows, `obj` could refer to anything—it does not have to refer to a `String`, so the compiler forbids `s = obj`. The way around the restriction is to use the typecast expression shown here:

```java
s = (String) obj;
```

The expression `(String) obj` tells the compiler that the reference variable `obj` is really referring to a `String` object. This is a typecast, as discussed in Chapter 2. With the typecast, the compiler allows the assignment statement, though you must still be certain that `obj` actually does refer to a `String` object. Otherwise, when the program runs, there will be a `ClassCastException` (see “Pitfall: Class Cast Exception” on page 82).

The complete assignment `s = (String) obj` is an example of a narrowing conversion. The term “narrowing” means that the left side of the assignment has a smaller ability to refer to different things than the right side.
Narrowing Conversion with Reference Variables

Suppose x and y are reference variables, and x has a smaller ability to refer to things than y. A narrowing conversion using a typecast can be made to assign x from y.

Example:
```java
String s = new String("Objection overruled!");
Object obj;
obj = s;
...
s = (String) obj;
```

By using a typecast, Java permits all narrowing conversions, though a ClassCastException may be thrown at runtime if the object does not satisfy the typecast.

Narrowing conversions also occur when a method returns an Object, and the program assigns that Object to a variable of a particular type. For example, the IntLinkedBag class from Chapter 4 has a clone method:

```java
clone
public IntLinkedBag clone()
Generate a copy of this bag.

Returns:
The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to an IntLinkedBag before it is used.
```

In the implementation of this method, we have the assignment:

```java
IntLinkedBag answer;
animal = (IntLinkedBag) super.clone();
```

The super.clone method from Java's Object class returns an Object. Of course, in this situation, we know that the Object is really an IntLinkedBag, but the compiler doesn't know that. Therefore, to use the answer from the super.clone method, we must insert the narrowing typecast, (IntLinkedBag). This tells the compiler that the programmer knows that the actual value is an IntLinkedBag, so it is safe to assign it to the answer variable.

Methods That Returns an Object

If a method returns an Object, then a narrowing conversion is usually needed to actually use the return value.
Using ArrayBag as the Type of a Parameter or Return Value

The bag has several methods which have bags as parameters or as the return type. For example, our IntArrayBag has an addAll method:

```java
public void addAll(IntArrayBag addend)...
```

When a program activates b1.addAll(b2), all the integers from b2 are put in the bag b1.

For the generic bag, the way to implement such a parameter is to specify its data type as `ArrayBag<E>` (rather than just `ArrayBag`):

```java
public void addAll(ArrayBag<E> addend)...
```

This will allow us to activate `b1.addAll(b2)` for two bags, `b1` and `b2`, but only if the type of elements in `b1` is the same as the type of elements in `b2`.

For our array bag, all the uses of the `ArrayBag` data type within its own implementation will be written as `ArrayBag<E>`.

Counting the Occurrences of an Object

Counting the occurrences of an object in a bag requires some thought. Here’s an example to get you thinking. Suppose you construct a bag of `Location` objects and create two identical `Location` objects with coordinates of `x=2, y=4` (using the `Location` class from Section 2.4). The locations are then added to the bag:

```java
ArrayBag<Location> spots = new ArrayBag<Location>();
Location p1 = new Location(2, 4); // x=2 and y=4
Location p2 = new Location(2, 4); // Also at x=2 and y=4

spots.add(p1);
spots.add(p2);
```

The two locations are identical but separate objects, as shown in this drawing:

```
Location p1

A Location object
x 2 y 4

Location p2

A Location object
x 2 y 4
```

Keep in mind that the boolean expression `(p1 == p2)` is false because `==` returns true only if `p1` and `p2` refer to the exact same object (as opposed to two separate objects that happen to contain identical values). On the other hand, the `Location` class has an `equals` method with this specification:
equals (from the Location class)

public boolean equals(Object obj)
Compare this Location to another object for equality.

Parameters:
obj – an object with which this Location is compared

Returns:
A return value of true indicates that obj refers to a Location object with
the same value as this Location. Otherwise, the return value is false.

Both p1.equals(p2) and p2.equals(p1) are true.

So here’s the question to get you thinking: With both locations in the bag, what is the value of spots.countOccurrences(p1)? In other words, how many times does the target p1 appear in the bag? The answer depends on exactly how we implement countOccurrences, with these two possibilities:

1. countOccurrences could step through the elements of the bag, using the
   “==” operator to look for the target. In this case, we find one occurrence
   of the target p1, and spots.countOccurrences(p1) is 1.

2. countOccurrences could step through the elements of the bag, using
   equals() to look for the target. In this case, both locations are equal to
   the target, and spots.countOccurrences(p1) is 2.

Every class has an equals method. For example, the equals method of the
String class returns true if the two strings have the same sequence of characters. The equals method of the Integer wrapper class returns true if the two Integer objects hold the same int value. Because the equals method is always available, we’ll use the second approach toward counting occurrences—and spots.countOccurrences(p1) is 2. Our bag documentation will make it clear that we count the occurrences of a non-null element by using its equals method.

Generic Collections Should Use the equals Method

When a generic collection tests for the presence of a non-null element, you should generally use the equals method rather than the "==" operator.

The Collection Is Really a Collection of References to Objects

There is another aspect of generic collections that you must understand. Although we use phrases such as “bag of objects,” what we really have is a collection of references to objects. In other words, the bag does not contain separate copies of each object. Instead, the bag contains only references to each object that is added to the bag.