13. If \( a \) is a variable of type `BankAccount` that holds a non-null reference, what do you know about the object to which \( a \) refers?

14. If \( a \) refers to a checking account, what is the effect of calling \( a.transfer(1000, \ a) \)?

**Special Topic 10.1**

**Abstract Classes**

When you extend an existing class, you have the choice whether or not to override the methods of the superclass. Sometimes, it is desirable to force programmers to override a method. That happens when there is no good default for the superclass, and only the subclass programmer can know how to implement the method properly.

Here is an example. Suppose the First National Bank of Java decides that every account type must have some monthly fees. Therefore, a `deductFees` method should be added to the `BankAccount` class:

```java
public class BankAccount {
    public void deductFees() { . . . }
    . . .
}
```

But what should this method do? Of course, we could have the method do nothing. But then a programmer implementing a new subclass might simply forget to implement the `deductFees` method, and the new account would inherit the do-nothing method of the superclass. There is a better way—declare the `deductFees` method as an **abstract method**:

```java
public abstract void deductFees();
```

An abstract method has no implementation. This forces the implementors of subclasses to specify concrete implementations of this method. (Of course, some subclasses might decide to implement a do-nothing method, but then that is their choice—not a silently inherited default.)

You cannot construct objects of classes with abstract methods. For example, once the `BankAccount` class has an abstract method, the compiler will flag an attempt to create a `new BankAccount()` as an error. Of course, if the `CheckingAccount` subclass overrides the `deductFees` method and supplies an implementation, then you can create `CheckingAccount` objects.

A class for which you cannot create objects is called an **abstract class**. A class for which you can create objects is sometimes called a **concrete class**. In Java, you must declare all abstract classes with the reserved word `abstract`:

```java
public abstract class BankAccount {
    public abstract void deductFees();
    . . .
}
```

A class that declares an abstract method, or that inherits an abstract method without overriding it, **must** be declared as abstract. You can also declare classes with no abstract methods as abstract. Doing so prevents programmers from creating instances of that class but allows them to create their own subclasses.

Note that you cannot construct an **object** of an abstract class, but you can still have a **variable** whose type is an abstract class. Of course, the actual object to which it refers must be an instance of a concrete subclass:

```java
BankAccount anAccount; // OK
anAccount = new BankAccount(); // Error—`BankAccount` is abstract
```
Chapter 10  Inheritance

```java
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

The reason for using abstract classes is to force programmers to create subclasses. By specifying certain methods as abstract, you avoid the trouble of coming up with useless default methods that others might inherit by accident.

Abstract classes differ from interfaces in an important way—they can have instance variables, and they can have concrete methods and constructors.

---

### Special Topic 10.2

#### Final Methods and Classes

In Special Topic 10.1 on page 437 you saw how you can force other programmers to create subclasses of abstract classes and override abstract methods. Occasionally, you may want to do the opposite and prevent other programmers from creating subclasses or from overriding certain methods. In these situations, you use the `final` reserved word. For example, the `String` class in the standard Java library has been declared as:

```java
public final class String { ... }
```

That means that nobody can extend the `String` class.

The `String` class is meant to be immutable—string objects can't be modified by any of their methods. Since the Java language does not enforce this, the class designers did. Nobody can create subclasses of `String`; therefore, you know that all `String` references can be copied without the risk of mutation.

You can also declare individual methods as `final`:

```java
public class SecureAccount extends BankAccount {
    ...  
    public final boolean checkPassword(String password) {
        ...  
    }
}
```

This way, nobody can override the `checkPassword` method with another method that simply returns `true`.

---

### Common Error 10.5

#### Overriding Methods to Be Less Accessible

If a superclass declares a method to be publicly accessible, you cannot override it to be more private. For example,

```java
public class BankAccount {
    public void withdraw(double amount) { ... }
}
```