The elegance of this solution lies in the fact that no change at all is needed in either the original **Game** or **Room** classes! We can simply add this class to the existing game, and the **goRoom** method will continue to work as it is. Adding the creation of a **TransporterRoom** to the setup of the floor plan is (almost) enough to make it work. Note, too, that the new class does not need a flag to indicate its special nature—its very type and distinctive behavior supply that information.

Because **TransporterRoom** is a subclass of **Room**, it can be used everywhere a **Room** object is expected. Thus, it can be used as a neighboring room for another room or be held in the **Game** object as the current room.

What we have left out, of course, is the implementation of the **findRandomRoom** method. In reality, this is probably better done in a separate class (say **RoomRandomizer**) than in the **TransporterRoom** class itself. We leave this as an exercise for the reader.

Exercise 11.8 Implement a transporter room with inheritance in your version of the *zuul* project.

Exercise 11.9 Discuss how inheritance could be used in the *zuul* project to implement a player and a monster class.

Exercise 11.10 Could (or should) inheritance be used to create an inheritance relationship (super-, sub-, or sibling class) between a character in the game and an item?

11.12 Summary

When we deal with classes with subclasses and polymorphic variables, we have to distinguish between the static and dynamic type of a variable. The static type is the declared type, while the dynamic type is the type of the object currently stored in the variable.

Type checking is done by the compiler using the static type, whereas at runtime method lookup uses the dynamic type. This enables us to create very flexible structures by overriding methods. Even when using a supertype variable to make a method call, overriding enables us to ensure that specialized methods are invoked for every particular subtype. This ensures that objects of different classes can react distinctly to the same method call.

When implementing overriding methods, the **super** keyword can be used to invoke the superclass version of the method. If fields or methods are declared with the **protected** access modifier, subclasses are allowed to access them, but other classes are not.

Terms introduced in this chapter:

static type, dynamic type, overriding, redefinition, method lookup, method dispatch, method polymorphism, protected

Concept summary

- **static type** The static type of a variable **v** is the type as declared in the source code in the variable declaration statement.
- **dynamic type** The dynamic type of a variable **v** is the type of the object that is currently stored in **v**.
- **overriding** A subclass can override a method implementation. To do this, the subclass declares a method with the same signature as the superclass, but with a different method body. The overriding method takes precedence for method calls on subclass objects.
- **method polymorphism** Method calls in Java are polymorphic. The same method call may at different times invoke different methods, depending on the dynamic type of the variable used to make that call.
- **`toString`** Every object in Java has a **`toString`** method that can be used to return a string representation of itself. Typically, to make it useful, a class should override this method.
- **`protected`** Declaring a field or a method **`protected`** allows direct access to it from (direct or indirect) subclasses.

Exercise 11.11 Assume that you see the following lines of code:

```
Device dev = new Printer();
dev.getName();
```

`Printer` is a subclass of **`Device`**. Which of these classes must have a definition of method **`getName`** for this code to compile?

Exercise 11.12 In the same situation as in the previous exercise, if both classes have an implementation of **`getName`**, which one will be executed?

Exercise 11.13 Assume that you write a class **`Student`** that does not have a declared superclass. You do not write a **`toString`** method. Consider the following lines of code:

```
Student st = new Student();
String s = st.toString();
```

Will these lines compile? If so, what exactly will happen when you try to execute?

Exercise 11.14 In the same situation as before (class **`Student`**, no **`toString`** method), will the following lines compile? Why?

```
Student st = new Student();
System.out.println(st);
```

Exercise 11.15 Assume that your class **`Student`** overrides **`toString`** so that it returns the student's name. You now have a list of students. Will the following code compile? If not, why not? If yes, what will it print? Explain in detail what happens.

```
for(Object st : myList) {
    System.out.println(st);
}
```

Exercise 11.16 Write a few lines of code that result in a situation where a variable **`x`** has the static type **`T`** and the dynamic type **`D`**.

This page intentionally left blank