9.11 Another example of inheritance with overriding

To discuss another example of a similar use of inheritance, we go back to a project from Chapter 7: the *zuul* project. In the *world-of-zuul* game, we used a set of `Room` objects to create a scene for a simple game. One of the exercises toward the end of the chapter suggested that you implement a transporter room (a room that beams you to a random location in the game if you try to enter or leave it). We revisit this exercise here, since its solution can greatly benefit from inheritance. If you don't remember this project well, have a quick read through Chapter 7 again, or look at your own *zuul* project.

There is no single solution to this task. Many different solutions are possible and can be made to work. Some are better than others, though. They may be more elegant, easier to read, easier to maintain and to extend.

Assume we want to implement this task so that the player is automatically transported to a random room when she tries to leave the magic transporter room. The most straight-forward solution that comes to mind first for many people is to deal with this in the `Game` class, which implements the player's commands. One of the commands is 'go', which is implemented in the `goRoom` method. In this method, we used the following statement as the central section of code:

```
nextRoom = currentRoom.getExit(direction);
```

This statement retrieves from the current room the neighboring room in the direction we want to go. To add our magic transportation, we could modify this similar to the following:

```
if(currentRoom.getName().equals("Transporter room")) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

The idea here is simple: we just check whether we are in the transporter room. If we are, then we find the next room by getting a random room (of course, we have to implement the `getRandomRoom` method somehow), otherwise we just do the same as before.

While this solution works, it has several drawbacks. The first is that it is a bad idea to use text strings, such as the room's name, to identify the room. Imagine that someone wanted to translate your game into another language – say, to German. They might change the names of the rooms – 'Transporter room' becomes 'Transporterraum' – and suddenly the game does not work any more! This is a clear case of a maintainability problem.

The second solution, which is slightly better, would be to use an instance variable instead of the room's name to identify the transporter room. Similar to this:

```
if(currentRoom == transporterRoom) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

This time, we assume that we have an instance variable `transporterRoom` of class `Room`, where we store the reference to our transporter room.[4] Now the check is independent of the room's name. That is a bit better.

There is still a case for further improvement, though. We can understand the shortcomings of this solution when we think about another maintenance change. Imagine we want to add two more transporter rooms so that our game has three different transporter locations.

A very nice aspect of our existing design was that we could set up the floor plan in a single spot, and the rest of the game was completely independent of it. We could easily change the layout of the rooms, and everything would still work – high score for maintainability! With our current solution, though, this is broken. If we add two new transporter rooms, we have to add two more instance variables or an array (to store references to those rooms), and we have to modify our `goRoom` method to add a check for those rooms. In terms of easy changeability, we have gone backwards.

The question, therefore, is: Can we find a solution that does not require a change to the command implementation each time we add a new transporter room? Here is our next idea.

We can add a method `isTransporterRoom` in the `Room` class. This way, the `Game` object does not need to remember all the transporter rooms – the rooms themselves do. When rooms are created, they could receive a boolean flag indicating whether this room is a transporter room. The `goRoom` method then could use the following code segment:

```
if(currentRoom.isTransporterRoom()) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

Now we can add as many transporter rooms as we like – there is no need for any more changes to the `Game` class. However, the `Room` class has an extra field whose value is really needed only because of the nature of one or two of the instances. Special-case code such as this is a typical indicator of a weakness in class design. This approach also does not scale well should we decide to introduce further sorts of special room, each requiring its own flag field and accessor method.[5]

With inheritance we can do better and implement a solution that is even more flexible than this one.

We can implement a class `TransporterRoom` as a subclass of class `Room`. In this new class we override the `getExit` method and change its implementation so that it returns a random room:

```
public class TransporterRoom extends Room
{

    /**
     * Return a random room, independent of the direction
     * parameter.
     * @param direction Ignored.
     * @return A random room.
     */
```

[4] Make sure that you understand why a test for reference equality is the most appropriate here.

[5] We might also think of using `instanceof`, but the point here is that none of these ideas is the best.

```
    public Room getExit(String direction)
    {
        return findRandomRoom();
    }

    /*
     * Choose a random room.
     * @return A random room.
     */
    private Room findRandomRoom()
    {
        ... // implementation omitted
    }
}
```

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. Simply 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 open as an exercise for the reader.

Exercise 9.9 Implement a transporter room with inheritance in your version of the *zuul* project.

Exercise 9.10 Discuss how inheritance could be used in the *zuul* project to implement a player and a monster class.

Exercise 9.11 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?

9.12 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` key word 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 it. 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.