**Exercise 9.1**

The project no longer compiles. The print method tries to access the private fields of the Item class, which are not accessible from subclasses. This can be corrected by creating accessor methods in Item and calling these from the subclasses.

If we try to compile after these modifications it still does not work. This time it is in the compilation of the Database class that fails, because it is trying to invoke the print() method on a variable of type Item. But Item has no longer declared a method called print() and hence it fails.

**Exercise 9.2**

The print method in the Item class is never called. If the object is a CD the print method in the CD class is called. If the object is a DVD the print method in the DVD class is called. This is because the dynamic types of the items are used.

**Exercise 9.3**

Yes, it behaves as expected by first calling the print method from the Item class and then calling the print method in the actual class (dynamic type).

One problem is that you can't enforce the call to the super class' print method. This means that if you want to create new subclasses you must remember to call super.print().

Another problem is that you can't modify the order of which the different things are printed out. This is discussed further in exercise 9.8.

**Exercise 9.4**

Add this line to the beginning of the print() method in the DVD class:

```
System.out.print("DVD: ");
```

And this line in the beginning of print() method in the CD class:

```
System.out.print("CD: ");
```

**Exercise 9.5**

The toString() method can be found on the class Object. It has no parameters and the return type is String.

**Exercise 9.7**

The print() method in the CD class:

```
    public void print()
    {
```

```
        System.out.print("CD: " + artist + ": ");
        super.print();
        System.out.println("    tracks: " + numberOfTracks);
    }
```

The print() method in the Item class:

```
    public void print()
    {
        System.out.print(title);
        if(getOwn()) {
            System.out.println("*");
        }
        else {
            System.out.println();
        }
        System.out.println("    " + playingTime + " minutes");
        System.out.println("    " + comment);
    }
```

**Exercise 9.8**

Give all the fields in the Item class protected accessors.

Then modify the method in the CD class to look like this:

```
    public void print()
    {
        System.out.print("CD: " + artist + ": ");
        System.out.print(getTitle());
        if(getOwn()) {
            System.out.println("*");
        }
        else {
            System.out.println();
        }
        System.out.print("    " numberOfTracks + " tracks, ");
        System.out.println("    " + getPlayingTime() + " minutes");
        System.out.println("    " + getComment());
    }
```

**Exercise 9.9**

Download: 09-09-zuul-with-transporter.zip

**Exercise 9.10**

To implement a Monster and a Player class in the Zuul project it would probably make sense to have a common superclass (Character) that contains the common behavior of the two classes.

**Exercise 9.11**

It depends... you could argue for all of the inheritance relations. Which one to choose depends on the current implementation and (if you know) which features you plan to

implement in the near future.

If an Item is a superclass of a Character, it would allow you to treat all Characters as Items. This means that a Character could pick up another Character which might make sense in some scenarios.

If Character is a superclass of Item, you can treat all Items as Characters. If we define Characters as something that can move around, this would allow for items to move around if that is desired.

If the two classes are siblings and have a common superclass (Thing?), you could do a combination of the above solutions. This could allow a Character to pick up Things (which means you can pick up both Items and Characters) and it could also allow for all Things to move around (again, this also applies to Item and Character).

If you don't consider Item and Character to have anything in common you could have no inheritance relations between Item and Character at all.

**Exercise 9.12**

Because type-checking is done on the static type of *dev* (which is Device) the method getName() must be defined in Device.

**Exercise 9.13**

To actually execute a method dynamic method lookup is used. This means that it is the method in the Printer class that will be called, because the dynamic type of *dev* is Printer.

**Exercise 9.14**

Yes, it will compile.

The toString() is implemented in the class Object from the Java library. The Object class is always a superclass of all other classes. Hence, when you execute it is the toString as defined in the class Object that will be called and the return value assigned to the String s.

**Exercise 9.15**

Yes, this will compile.

The System.out.println() method can take an Object as argument and as argued in 9.14 Student is a subclass of Object.

**Exercise 9.16**

Yes, it compiles.

It prints out all the names of the students in the list.

It will run through all the items in the list and call the method System.out.println(st). This method invokes the toString() method of the object - which, because of dynamic method lookup, will call the toString() method in the Student class.

**Exercise 9.17**

D must be a subclass of T

```
T x = new D();
```