More About Inheritance

Java version

Objectives

To extend our knowledge about inheritance.

Main concepts discussed in this chapter

- method polymorphism
- overriding
- static and dynamic type
- dynamic method lookup

Resources

Classes needed for this lab (see course website):

- *chapter08.jar* (from previous lab session)
- *chapter09.jar*

To do

The problem: network project's `display` method

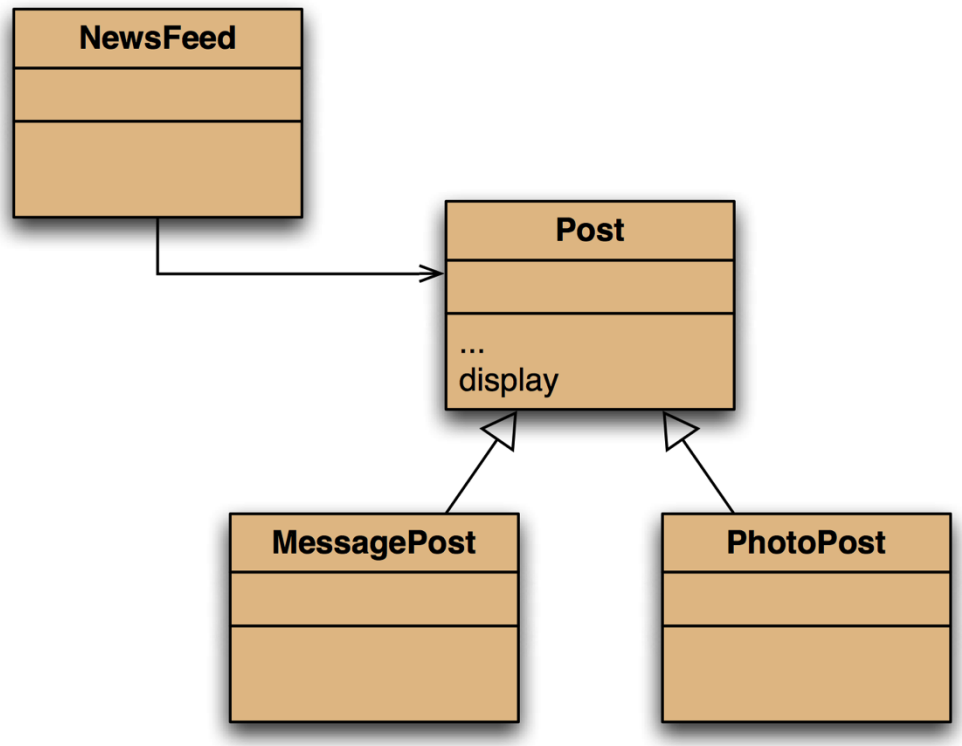
The network project's **`network.v2.Post#display`** method has a problem in that it doesn't show all of a **`Post`**'s data, only that which is common to both **`MessagePost`**s and to **`PhotoPost`**s. For example, given that we have the following posts of each type

```
Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying ...
40 seconds ago - 2 people like this.
  No comments.

Alexander Graham Bell
[experiment.jpg]
I think I might call this thing 'telephone'.
12 minutes ago  -  4 people like this.
  No comments.
```

the inheritance version **network.v2** displays only

```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago  -  4 people like this.
  No comments.
```

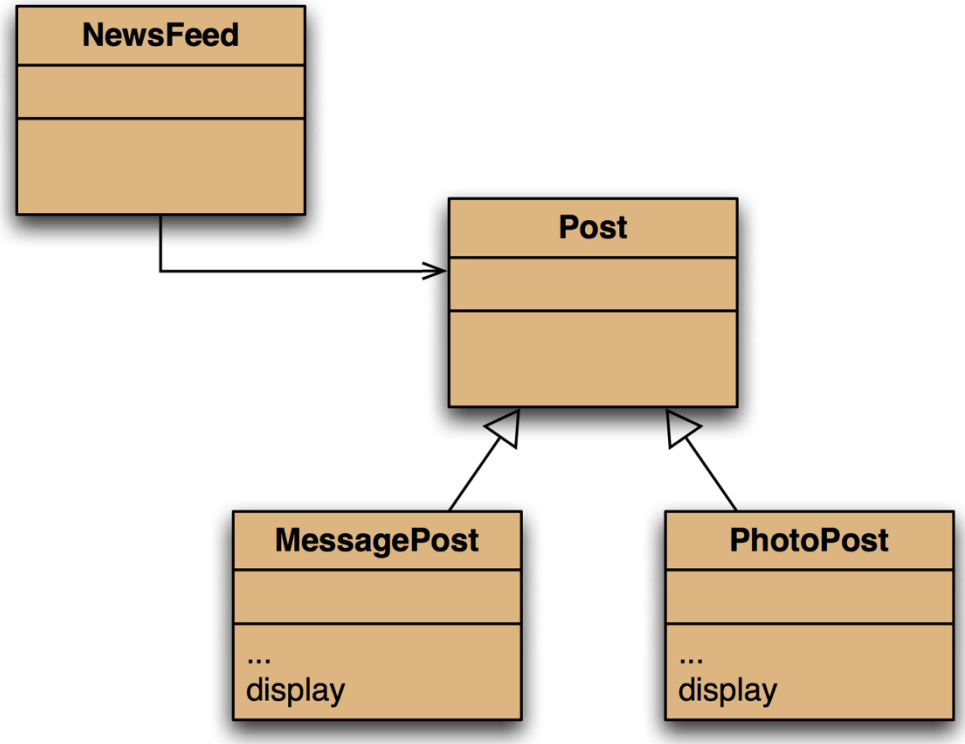
What's going wrong is that **display** is defined in **Post**



and as such can only display information about the fields of **Post** and not any of the fields specific to **MessagePost** or **PhotoPost**. Inheritance only works in one direction, **Post** knows nothing about any of its subclasses.

Static type and dynamic type

As a first attempt at a fix, we'll move the **display** method down into **MessagePost** and **PhotoPost**



Exercise

1. Open the **`network.v2`** version of the *network* project. Remove the **`display`** method from class **`Post`** and move it into the **`MessagePost`** and **`PhotoPost`** classes. Compile. What do you observe?

Now we have *two* unfortunate problems:

- **`MessagePost`** and **`PhotoPost`** no longer have access to the **`private`** fields of **`Post`**;
- **`NewsFeed`** can no longer find a **`display`** method for **`post`**. This is a bit more complicated...

Calling display from NewsFeed

Looking at the **`NewsFeed`** code explains the second problem

```
for (Post post : posts) {
    post.display();
    System.out.println();
}
```

We have just removed **`Post`**'s **`display`** method. To understand in detail why it doesn't work, we need to look more closely at types. Consider the following statement:

```
Car c1 = new Car();
```

We say that the type of **c1** is **Car.** Before we encountered inheritance, there was no need to distinguish whether by “type of **c1**” we meant “the type of the variable **c1**” or “the type of the object stored in **c1.**” It did not matter, because the type of the variable and the type of the objectnwere always the same.

To understand how to fix this problem, we need to introduce *static typing* and *dynamic typing*. Consider again

```
Vehicle v1 = new Car();
```

Vehicle is the *static type* (or declared type) of the variable **v1**. The static type is known at compile time and does not change. The type of the object that the variable refers to when the program executes is the *dynamic type* (or actual type) of the variable. This is not known to the compiler, in fact it's determined only at runtime and it can change, eg,

```
Vehicle v1 = new Car();
v1 = new Bicycle();
```

On the first line, the dynamic type of **v1** is different from its dynamic type on the second line, and this is in principle unknown to the compiler. The network project doesn't compile because the static type of the variable **post** has no **display** method and the compiler has no way of knowing its dynamic type. To get the code to compile, **Post** needs its **display** method.

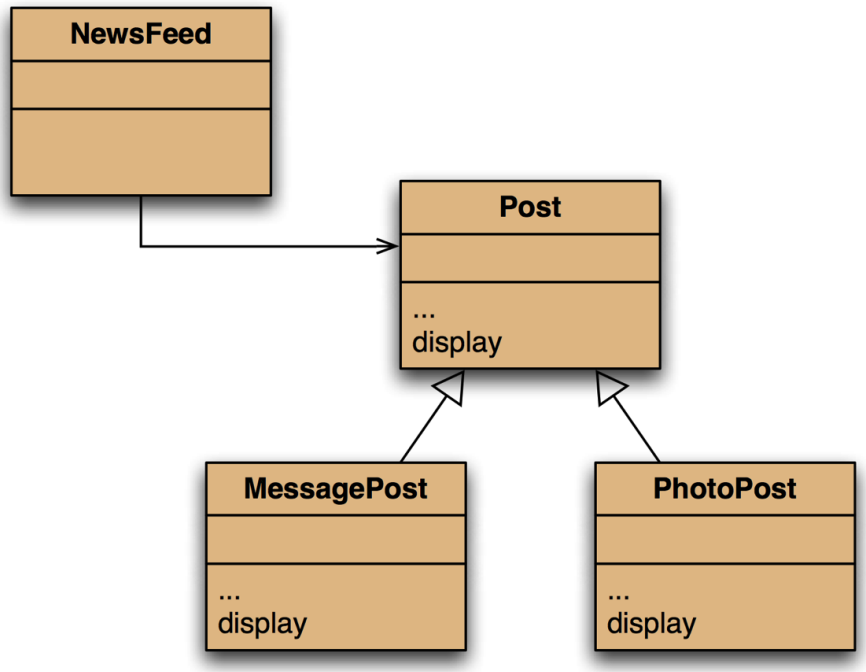
Exercise

2. In your network project, add the **display** method back to the class **Post**. Then modify the **display** methods in **MessagePost** and **PhotoPost** so that they print out their specific fields only.

   You should have the situation corresponding to the figure below, with **display** methods in three classes. This should now compile.
   Before execution, predict which of the **display** methods will get called if you execute the **NewsFeed list** method.
   Try it out. Enter a MessagePost and a PhotoPost into the database and call the **NewsFeed#show** method. Which **display** methods were executed? Was your prediction correct? Try to explain your observations.

Overriding

For the next try at fixing the problem, we'll put the **display** method in *both* superclasses and subclasses




```
class Post {
    ...
    void display() {
        System.out.println(username);
        System.out.print(timeString(timestamp));

        if (likes > 0) {
            System.out.println("  -  " + likes
                   + " people like this.");
        } else {
            System.out.println();
        }

        if (comments.isEmpty()) {
            System.out.println("   No comments.");
        } else {
            System.out.println("   " + comments.size()
                    + " comment(s). Click here to view.");
        }
    }
}


class MessagePost extends Post {
    ...
    void display() {
        System.out.println(message);
```

```
    }
}


class PhotoPost extends Post {
    ...
    void display() {
        System.out.println("  [" + filename + "]");
        System.out.println("  " + caption);
    }
}
```

This now compiles and almost works. The **`display`** method in the subclass, **`MessagePost`** or **`PhotoPost`**, is said to *overrride* the method in the superclass. When a class has two methods with the same signature, its own and the one inherited from its superclass, which one gets invoked?

Dynamic method lookup

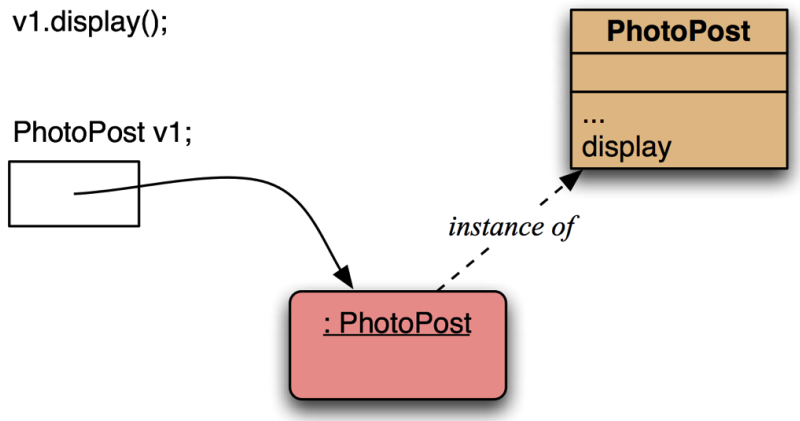
Important statement

> Type checking at compile time uses the static type, but at runtime the methods from the dynamic type are executed.

We look at how methods are invoked in three different situations. We are interested in *method lookup* (also commonly known as *method binding* or *method dispatch*).

In the first case we have something like

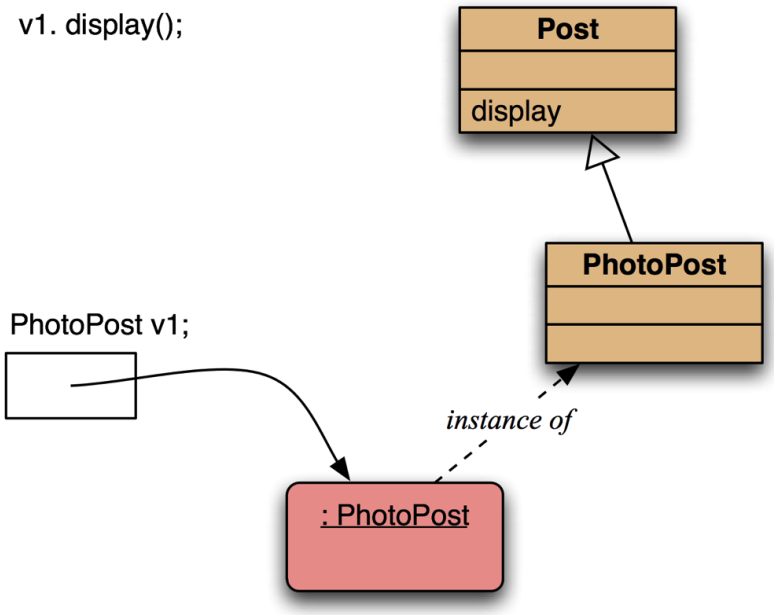
```
PhotoPost v1 = new PhotoPost();
...
vi.display();
```



The sequence of finding the code of the method **`v1.display`** follows four steps:

1. The variable **`v1`** is accessed.
2. The object is found by following the reference in **`v1`**.
3. The class of the object is found.
4. The implementation of the **`display`** method is found and the method is executed.

In the second case, we have inheritance involved and the lookup steps are the same as



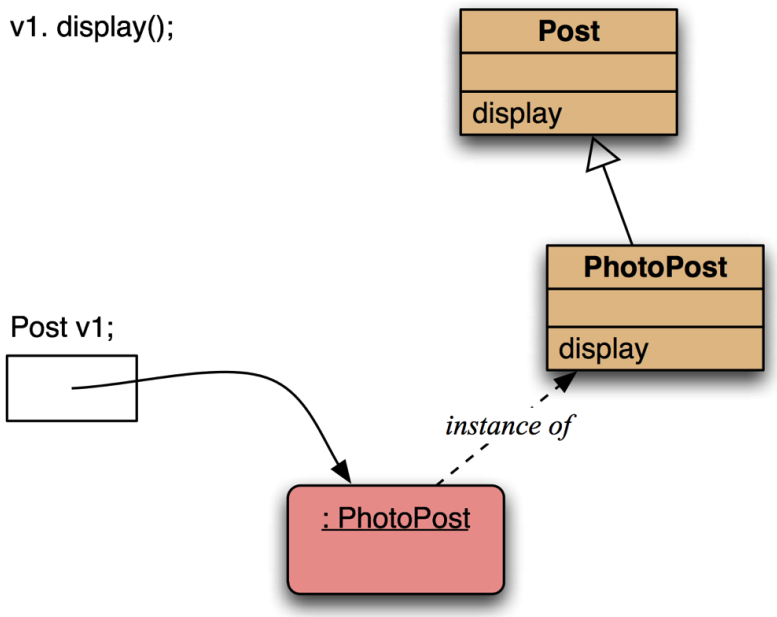
above for steps 1-3. Then

4. No **`display`** method is found in the class **`PhotoPost`**.
5. The superclass is found and searched for a matching method. If none is found, the superclass's superclass is searched, and so on up the inheritance hierarchy until the method is found.
6. The method is executed.

In the third case, the code looks like

```
Post v1 = new PhotoPost();
vi.display();
```

The sequence of finding the code of the method **`v1.display`** is the same as steps 1-4 for the first case.



- There is no special treatment here when the static type and the dynamic type are different.
- The dynamic type determines which method gets executed.
- Overriding methods in sublcasses take precedence over superclass methods.
- When an overriding subclass method is executed, the overridden superclass methods are not automatically executed as well.

Super call in methods

To print all the information about a **`MessagePost`**, we would like **`Post`**'s `display` to print the common information, and then **`MessagePost`**'s **`display`** to print the specific information. And this is easy to do with small addition to **`MessagePost display`**

```
void display() {
    super.display();
    System.out.println(" [" + filename + "]");
    System.out.println(" " + caption);
}
```

Exercise

3. Modify your latest version of the network project to include the **`super`** call in the **`display`** method. Test it. Does it behave as expected? Do you see any problems with this solution?

Method Polymorphism

A polymorphic variable can refer to objects of different types. Method polymorphism means that the same line of code, eg,

```
post.display();
```

can invoke different methods, depending on the dynamic type of the variable **post**.

`Object` methods - toString

Exercise

4. Look up **toString** in the library documentation. What are its parameters? What is its return type?

toString returns a string representation of an object; by default this consists of the type of the object and its memory address. Any class can override the default to provide some more useful information about the object. A final improvement to **Post** would be to have **toString** return the necessary information

Exercise

5. You can easily try this out. Create an object of class **PhotoPost** in your project, and then invoke its **toString** method.

We generally make **toString** more useful by overriding it, eg,

```
class Post {
    ....
    @Override
    public String toString() {
        String text = username + "\n" + timeString(timestamp);
        if(likes > 0) {
            text += " - " + likes + " people like this.\n";
        } else {
            text += "\n";
        }
        if(comments.isEmpty()) {
            return text + " No comments.\n";
        } else {
            return text + " " + comments.size() +
                   " comment(s). Click here to view.\n";
        }
    }

    void display() {
        System.out.println(toString());
    }
}
```

```
class MessagePost extends Post {
    ...
    @Override
    public String toString() {
        return super.toString() + message + "\n";
    }

    public void display() {
        System.out.println(toString());
    }
}
```

Printing to the terminal here is a *side-effect*, and we could get rid of the side-effect by getting rid of the **display** method altogether. The **toString** method returns the same information as a **String** but leaves using the information to the user of a **Post**, **MessagePost**, or **PhotoPost** class. What they do with the information is their own business and not the business of the developers of these three classes.

The tag **@Override** above is optional and is merely intended so that the compiler can check whether the method actually does override another **toString** higher up the inheritance hierarchy. An error is signalled if no superclass has the method.

Note that these two lines of code are equivalent

```
System.out.println(post.toString());
System.out.println(post);
```

Whenever a **String** is needed from an object, its **toString** method is automagically invoked.

Object equality: equals and hashCode

It is often necessary to determine whether two objects are “the same.” The **Object** class defines two methods, **equals** and **hashCode**, that have a close link with determining similarity. We actually have to be careful when using phrases such as “the same”; this is because it can meantwo quite different things when talking about objects. Sometimes we wish to know whether two different variables are referring to the same object,what is called *reference equality*. Reference equality is tested for using the **==** operator. So the following test will return **true** if both **var1** and **var2** are referring to the same object (or are both **null**), and **false** if they are referring to anything else:

```
var1 == var2
```

We also define *content equality*, as distinct from reference equality. A test for content equality asks whether two objects are the same internally—that is, whether the internal states of two objects are the same.

What content equality between two particular objects means is something that is defined by the objects’ class. This is where we make use of the **equals** method that every class

inherits from the **Object** superclass. If we need to define what it means for two objects to be equal accord ing to their internal states, then we must override the **equals** method, which then allows us to write tests such as

```
var1.equals(var2)
```

This is because the **equals** method inherited from the **Object** class actually makes a test for reference equality. It looks something like this:

```
@Override
public boolean equals(Object obj) {
    return this == obj;
}
```

Because the **Object** class has no fields, there is no state to compare, and this method obviously cannot anticipate fields that might be present in subclasses. The way to test for content equality between two objects is to test whether the values of their two sets of fields are equal, eg,

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;  // Reference equality.
    }
    if (!(obj instanceof Student)) {
        return false;  // Not the same type.
    }
    // Gain access to the other student's fields.
    Student other = (Student) obj;
    return name.equals(other.name) &&
            id.equals(other.id) &&
            credits == other.credits;
}
```

The first test is just an efficiency improvement; if the object has been passed a reference to itself to compare against, then we know that content equality must be true. The second test makes sure that we are comparing two students. Having established that we have another student, we use a cast and another variable of the right type so that we can access its details properly. Finally, we make use of the fact that *the private elements of an object are directly accessible to another instance of the same class*.

It will not always be necessary to compare every field in two objects in order to establish that they are equal. For instance, if we know for certain that every **Student** is assigned a unique id, then we need not test the name and credits fields as well. It would then be possible to reduce the final statement above to

```
return id.equals(other.id);
```

Whenever the **equals** method is overridden, the **hashCode** method should also be overridden. The **hashCode** method is used by data structures such as **HashMap** and

HashSet to provide efficient placement and lookup of objects in these collections. Essentially, the **hashCode** method returns an integer value that represents an object. From the default implementation in **Object**, distinct objects have distinct **hashCode** values.

There is an important link between the **equals** and **hashCode** methods in that two objects that are the same as determined by a call to **equals** must return identical values from **hashCode**. An integer value should be computed making use of the values of the fields that are compared by the overridden **equals** method. Here is a hypothetical **hashCode** method that uses the values of an integer field called **count** and a **String** field called **name** to calculate the code:

```
@Override
public int hashCode() {
    int result = 17;  // An arbitrary starting value.
    // Make the computed value depend on the order in which
    // the fields are processed.
    result = 37 * result + count;
    result = 37 * result + name.hashCode();
    return result;
}
```

Protected access

The **private** keyword limits access to within the same class; **public** is accessible from anywhere. The **protected** keyword is in between those two - accessible from within the class, but also from within any subclass

Protected access is usually used only for methods (and constructors), not for instance variables which should remain private.

Exercises

6. The version of **display** shown above produces the output

```
Leonardo da Vinci
40 seconds ago - 2 people like this.
  No comments.
Had a great idea this morning.
But now I forgot what it was. Something to do with
flying...
```

   Reorder the statements in the method of your version of the *network* project so that it prints the details as

```
Had a great idea this morning.
But now I forgot what it was. Something to do with
flying...
Leonardo da Vinci
40 seconds ago - 2 people like this.
  No comments.
```

7. Having to use a superclass call in **display** is somewhat restrictive in the ways in which we can format the output, because it is dependent on the way the superclass formats its fields.

   Make any necessary changes to the **Post** class and to the display method of **MessagePost** so that it produces the output shown below. Any changes you make to the **Post** class should be visible only to its subclasses. Hint: You could add **protected** accessors to do this.

```
Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with
flying...
40 seconds ago - 2 people like this.
  No comments.
```

The instanceof operator

One of the consequences of the introduction of inheritance into the network project has been that the **NewsFeed** class knows only about **Post** objects and cannot distinguish between message posts and photo posts. This has allowed all types of post to be stored in a single list.

However, suppose that we wish to retrieve just the message posts or just the photo posts from the list; how would we do that? Or perhaps we wish to look for a message by a particular author? That is not a problem if the **Post** class defines a **getAuthor** method, but this will find both message and photo posts. Will it matter which type is returned?

There are occasions when we need to rediscover the distinctive dynamic type of an object rather than dealing with a shared supertype. For this, Java provides the **instanceof** operator. The **instanceof** operator tests whether a given object is, directly or indirectly, an instance of a given class. The test

```
obj instanceof MyClass
```

returns **true** if the dynamic type of **obj** is **MyClass** or any subclass of **MyClass.** The left operand is always an object reference, and the right operand is always the name of a class. So

```
post instanceof MessagePost
```

returns **true** only if **post** is a **MessagePost**, as opposed to a **PhotoPost**.

Use of the **instanceof** operator is often followed immediately by a cast of the object reference to the identified type. For instance, here is some code to identify all of the message posts in a list of posts and to store them in a separate list.

```
ArrayList<MessagePost> messages = new ArrayList<>();
for (Post post : posts) {
    if (post instanceof MessagePost) {
```

```
            messages.add((MessagePost) post);
        }
    }
```

It should be clear that the cast here does not alter the **post** object in any way, because we have just established that it already is a **MessagePost** object.

Summary

The static type of a variable is its declared type, its dynamic type is the type of the object currently referred to by the variable. The compiler uses static type checking, whereas runtime lookup is based on the dynamic type. The keyword **super** in an overriding method can be used to invoke the overriden superclass method. The keyword **protected** on fields or methods gives access from subclasses.

Concept summary

static type
: The static type of a variable **v** is the type as declared in the source code in the variable declaration statment.

dynamic type
: The dynamic type of a variable **v** is the type of the object that is currently stored in **v**.

overriding
: A sublclass 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 method protected allows direct access to it from (direct or indirect) subclasses.

Additional exercises

Exercises

11. Assume 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?

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

13. Assume you write a class **Student**, which 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? What *exactly* will happen when you try to execute?

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);
```

15. Assume 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 (Student student : myList) {
    System.out.println(student);
}
```

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**.