Polymorphism (runtime)

In the last few lectures, we have seen that the declared type of a reference variable does not entirely determine what classes of object the variable can reference at runtime. At runtime, a variable can reference an object of its declared type, or it can also reference an objects whose class is a subclass of the variable’s declared type (if the declared type is a class), or if the declared type is an interface then the variable can reference objects whose class implements the interface.

This property, that the runtime type is more general than the declared type, is called polymorphism. We sometimes refer to the declared type, which is determined at compile time, as the static type, to distinguish it from the dynamic type which is defined at runtime.

Last class we concentrated on the issue of type checking which is done by the compiler (and is typically called static type checking. We looked at when implicit casting was good enough and when we needed to do explicit casting. We also looking at the instanceof operator which is used for dynamic type checking, namely checking the class of an object at runtime. Let’s now assume a program has compiled and focus on the question of which method is invoked. The method used is the one defined by the object that invokes it. This choice is sometimes called dynamic dispatch. There are a few different cases to be aware of here.

The first case is what I discussed above: when an object invokes a method (at runtime, i.e. we are talking about an object), the method is determined by the class that the object belongs to. The object knows what class it belongs to. This information is stored in the object in the class field that references the class descriptor (which is an instance of the Class class – see end of lecture 23, where I have added notes about it). A programmer can access this field with the Class getClass() method, which it inherits from the Object class.

Consider, for example:

```java
boolean b;
Object it;
:
if (b)
    it = new float[23];  // an array of floats
else
    it = new Dog();
System.out.print(it.toString()); (*)
```

At compile time, we cannot say which toString() method should be used, since we don’t know the value of b. Rather, the method must be determined at runtime, when (*) is executed and it references either a float[] or a Dog. In each case, there will be a toString() method used which is appropriate for the object.

A second issue that arises in polymorphism is that subclasses can override methods from the superclass and sometimes it is not so obvious which method gets applied. Let’s just look at a simple example here.

Consider a method threaten() in the Dog() class. This method calls showTeeth(), which is defined in the Dog class only, and bark() which is defined in both the Dog and subclasses of Dog, in particular, Doberman.

---

1 from Latin: poly means “many” and “morph” means forms
Consider what happens when a Doberman object invokes the Dog method threaten(). Which version of bark() is defined for Doberman objects? To reason about this, you must understand that the method bark() in the Dog() class is invoked without specifying which object is invoking it. In the code, however, there is an implicit this reference which the compiler fills in, so the call to bark() within threaten is really this.bark(). Thus, if the caller is a Doberman object, then the bark() that is used is Doberman.bark() rather than Dog.bark().

class Dog{
    String name

    void Dog(){ .. }

    void threaten{
        showTeeth(){ .. }; // this.showTeeth()
        bark(); // this.bark()
    }

    void bark(){
        System.out.println("woof");
    }
}

class Doberman extends Dog{

    Doberman(String name){
        super(name); // alternatively, "this.name = name;"
    }

    void bark(){
        System.out.println("GRRRR! WO WO WO!");
    }

    void barkLikeDog{ // generic Dog impersonator
        super.bark();
    }
}

Note that if you wanted a Doberman object to bark like a Dog, you could do this by explicitly writing super.bark(). The Doberman object would then go to its superclass which is Dog and use the bark() method from the Dog class. See the Doberman.barkLikeDog() above.

Finally, in the lecture slides, I discussed walked through the different steps of the main() method above and showed how the call stack evolves. I suggest you do it for yourself and then verify your answer by checking the slides. Note that the Doberman(String) constructor calls super(String) so don’t forget about that one.
Graphical user interfaces (GUI)

Most applications that you work with provide a graphical interface. These include IDE's such as Eclipse, web browsers such as Firefox, mail programs such as Thunderbird, Word processors, etc. You are so used to interacting with GUIs that you probably rarely think about how they work and how you might you write one.

We will now spend a few lectures looking at some of the object oriented concepts that are involved in creating such interface. The topic is large and we will only cover the basics concepts and techniques. This should be enough to spark your interest and get you going.

Containers and components

The part of the display screen where the input and output for an application occurs is called a frame. A frame is a window which has a minimize, maximize, close icons at the top right corner. A frame often consists a number of regions called panels. We think of frames and panels as containers in that they contain components such as images and text labels that we want to display, buttons that we can click on, and areas where we can enter text, etc. When we say “component”, we mean something that has a position and size in a given frame or panel. A component either displays information or it allows the user to input something, or both.

If you are reading this document with acroread, for example, then the frame is the acroread window. There are several panels. There is the panel containing the document. But there is also a panel with icons at the top which allow you to step through pages of the document, zoom, etc. There may be another panel that has icons that allow you to print the document, add a sticky note, highlight text. And there is the panel which has the File/Edit/View/Window/Help tabs which produce dropdown menus when you click on them.

A container can contain components, but a container itself is also a component – it is special type of component that contains other components. As such, containers and components have a tree structure such that internal nodes are containers and leaves are either components (or empty containers), and the “A contains B” relation is a (parent/A, child/B) edge. This property is not something we will emphasize, but you should be aware of it nonetheless.

When you are creating a GUI in Java, the containers and components are all objects and so they need to be instantiated. You also need to add the components objects to the containers. For example, you can add a component directly to a frame. Or you can add a component to a panel, and then add the panel (now as a component) to the frame.

Let’s now turn to some of the Java classes that are involved. The Java classes that you use to write a graphical user interface are all in `javax.swing.*` and `java.awt.*` where the latter stands for “abstract windowing toolkit”. We will just refer to these GUI tools simply as “swing”. An excellent reference for swing is the Java tutorials [docs.oracle.com/javase/tutorial/ui/index.html](http://docs.oracle.com/javase/tutorial/ui/index.html)

**JFrame**

A JFrame is the main container for an application. It defines the application window that you are used to, i.e. it has a little X in the top right corner that allows you to close it, as well as a max and
min icon, and a title. The frame can be moved around on your screen by dragging with the mouse, and it can also be resized.

The constructor of the JFrame sets the initial size and other properties of the frame. Some of the methods of JFrame are setVisible, setDefaultCloseOperation, setSize, setBounds, pack.

JPanel
Panels are instances of the JPanel class. Panels are used to group together a set of components. We typically extend the JPanel class and define a particular panel that we wish to add to our frame. For example, if we want to make a form for people to fill out, then we might write a MyForm class that extends JPanel. It may seem strange to define a class for this, since often we only have one instance of this extended JPanel class. But that’s how it works, so we have to get used to it.

Note that we can also extend the JFrame class if we wish and by define a subclass that has a particular set of JPanel objects within it. We then instantiate the JPanel objects and add them to the extended frame class. We can also extend both JFrame and JPanel.

JComponent
There are many components classes that we can instantiate and add to frames and panels. Here are some examples:

- ImageIcon - a little graphic/picture e.g. a garbage can to represent “delete”
- JLabel - text or an image
- JButton - allows you to click on it with a mouse
- JCheckBox - allows you to select from several options by clicking with your mouse, which marks the box with an X (or some other symbol)
- JRadioButton - similar to JCheckBox but now you must choose one and only one of a set of buttons (It is called a radio button because this is how old car radios used to work. You could select buttons which were set to tune in particular radio stations.)

Note that JButton, JCheckBox and JRadioButton are often grouped together to make a ButtonGroup.

- JTextField - a region where you can type one line of text
- JPasswordField - like a textfield, but when you type in the text, the field shows only a symbol like * for each character
- JTextArea - a region where you can type multiple lines of text
- JComboBox - a set of items in a dropdown menu from which a single item is selected
LayoutManager

You might assume that when you make a GUI, you are responsible for positioning exactly where the components go in the frame and in each panel. This is not the case, though. One reason is that it is often common for users to resize windows and the programmer doesn’t want to have to specify exactly where each component goes whenever the window is resized.

The positioning of the components is done automatically by an object known as a LayoutManager which has private methods that decide where to position each component, based on the user’s choice of window size. Specifically, each JFrame and JPanel has a private field which references a helper object called its LayoutManager. The layout manager for the container has methods for positioning the components in that container.

LayoutManager is an interface which is implemented by Java classes FlowLayout, GridLayout, BorderLayout. If you want a container to use a “flow layout” manager. Then, within the container class (JFrame or JPanel) you call:

   setLayout( new FlowLayout() )

   The setLayout method is method in the JFrame and JPanel class. When we extend either of these container classes, we put the above method call in the constructor of the extended class. Note that there is an implicit this in the above method call.

Once the LayoutManager) has been assigned to a container, it remains there (unless it is reset, which is not typically done). Also note that, as a programmer, you don’t declare a reference variable for the layout manager of a container. A hidden (private) one is already there. Once you set the layout manager, you don’t need to refer to it again.