About this example

This example shows how to create an alternative to an inheritance-based solution for a filebrowswer, which should be able to list files and a text-based thumbnail.

An inheritance-based approach might turn into an exercise of writing numerous very similar classes, representing differnt types of files with only a thumbnail that differs:

```
public class FBFile {
  // variables for name and thumbnail
  // constructor

  public String thumbnail() {
    return "[file]";
  }
}

public class TextFile extends FBFile {
  // constructor...

  @Override
  public String thumbnail() {
    return "[txt]";
  }
}

public class Mp3FBFile extends FBFile {
  // constructor...

  @Override
  public String thumbnail() {
    return "[mp3]";
  }
}

// etc with a hundred other file types...
```

Also, the listing of files performed by some method in the file browser, must check what type of file it should represent as an FBFile, probably using a long if-statement which must be maintained and altered as we discover new file types:

```
  for (File file : files) {
    if (file.isFile()) {
      if (file.getPath().endsWith(".java")) {
        list.add(new FBJavaFile(file.getPath()));
      } else if (file.getPath().endsWith(".mp3")) {
```

```
        list.add(new FBMp3(file.getPath()));
      } else if (file.getPath().endsWith(".ogg")) {
        list.add(new FBOgg(file.getPath()));
      } else {
        list.add(new FBFile(file.getPath()));
      }
    } else if (file.isDirectory()) {
      list.add(new FBDir(file.getPath()));
    }
  }
```

This clearly doesn't scale very well...

The approach used by this example, is to use a single class FBFile which uses composition to represent a file for the filebrowser.

If we look at the characteristics of the FBFile objects, they clearly all have a name() method for accessing the filename, and a thumbnail() method for getting the thumbnail. Different types of files have different thumbnails.

So, let's encapsulate what varies, the thumbnail production and the actual file the FBFile represents!

We can use a dedicated thumbnail generator as an instance variable of the FBFile class, meaning that every FBFile object has a generator for its thumbnail:

```
public class FBFile {
  private java.io.File file; // the actual file on the file system
  private ThumbnailGenerator thumbGen; // knows how to create the thumb!

  public FBFile(File file, ThumbnailGenerator thumbGen) {
    this.file = file;
    this.thumbGen = thumbGen;
  }
  public String name() { return file.getName(); }

  public String thumbnail() {
    return thumbGen.thumbnail();
  }

  // toString etc...
}
```

Now, we have encapsulated the generation of the thumbnail String into a dedicated object, that all FBFile objects have as an instance variable. This means that we can use only one class to represent any type of file, as long as the code which creates the FBFile objects, can supply an appropriate ThumbnailGenerator to the constructor of FBFile.

We no longer need to write a new class which extends FBFile for every type of

file, just to override the thumbnail() method. Instead, we create each FBFile object so that it has its own name and its own ThumbnailGenerator.

How do we represent the ThumbnailGenerator?

We'll use a simple functional interface for the ThumbnailGenerator (a functional interface is an interface with only one abstract method).

The ThumbnailGenerator interface

The interface thus becomes:

```
@FunctionalInterface
public interface ThumbnailGenerator {
  public String thumbnail();
}
```

That was pretty simple, wasn't it? Don't worry about the @FunctionalInterface annotation, it's just there to help us follow the rule that functional interfaces can only declare one single abstract method, and the compiler helps us to enforce that rule. It's not important for this example, so you can pretend that it's not there. Or think of it as similar to the @Override annotation we use to get the compiler to check that we properly do overriding.

Creation of an FBFile

To create an FBFile, we need to, like we said, provide a file and a ThumbnailGenerator to the constructor. We could write a class for each kind of ThumbnailGenerator, but that would defeat the purpose of avoiding the class explosion with hundreds of classes covering all kinds of file types.

Instead, we can use an anonymous inner class representing a thumbnail generator or a so called lambda expression. The technique with anonymous inner classes would look like this:

```
// We have a java.io.File representing some file on the file system:
File f = ...
// Create the thumbnail generator for some file extension:
String[] parts = f.getName().split("\\.");
// powderfinger.mp3 would become ["powderfinger", "mp3"]

String suffix = parts[parts.length -1]; // "mp3"

// Anonymous inner class implementing the ThumbnailGenerator interface:
```

```
ThumbnailGenerator mp3Thumb = new ThumbnailGenerator() {
   @Override
   public String thumbnail() {
     return suffix;
   }
};

//Create the FBFile:
FBFile fb = new FBFile(f, mp3Thumb);
```

And the lambda version would look like this:

```
// We have a java.io.File representing some file on the file system:
File f = ...
// Create the thumbnail generator for some file extension:
String[] parts = f.getName().split("\\.");
// powderfinger.mp3 would become ["powderfinger", "mp3"]

String suffix = parts[parts.length -1]; // "mp3"

ThumbnailGenerator mp3Thumb = () -> suffix;

FBFile fb = new FBFile(f, mp3Thumb);
```

That way, we would dynamically discover the file type (only caring about the file suffix, a behavior sadly often implemented by file browsers ;-)) and create files which have a different thumbnail generated depending on their file suffix (which we pretend always reflects the file type).

Of course, we should have a special case for directories and files without a suffix. See the `org.progund.fb.util.FBList` class source code to see one possible implementation of a method which creates FBFile objects from all files found in some actual directory on your computer.

Javadoc for the example

Please run the `gendoc.sh` script to generate javadoc for the whole example. You probably need to change the permission of the script and run it:

```
$ chmod u+x gendoc.sh
$ ./gendoc.sh
```

Then, open docs/index.html using your favorite browser to read the documentation.

Taking things one step further

For those who want to learn a little more about interfaces, we've also included an example on how to decorate an existing class with new behavior. This is done using a mix of inheritance and composition.

Let's say there's an interface `org.progund.mediaplayer.Playable` which only declares one abstract method `public void play();` .

Now, we could re-use our FBFile class for a mediaplayer project. It would be nice if we had a subtype of FBFile for files which are media files (and Playable).

Such a class could be declared like this:

```
public class FBMediaFile extends FBFile implements Playable {
  // variables, constructor, methods for being an FBFile...

  public void play() {
    // code for simulating playing the media file
  }
}
```

A technique for decorating a class, such as FBFile, with new capabilities, such as being Playable, is to wrap an instance of the class to decorate inside a new class by allowing an e.g. FBFile to be passed to the constructor of the extending class, e.g. FBMediaFile .

The class now becomes:

```
public class FBMediaFile extends FBFile implements Playable {
  private FBFile fb;

  public FBMediaFile(FBFile fb) {
    // file and thumbGen are protected in
    // FBMediaFile, so we can use those names here!
    super(fb.file, fb.thumbGen);
  }

  @Override
  public String name() {
    return fb.name(); // This an FBFile can do!
  }

  @Override
  public String thumbnail() {
    return fb.thumbnail(); // This too, an FBFile can do!
  }

  // perhaps a toString() also, but we'll leave that for now
```

```
  // we must implement the play() method since we
  // say that our class implements Playable
  @Override
  public void play() {
    // code for simulating playing the media file
  }
}
```

Now we have a class which is both an FBFile and a Playable!

This allows us to get a list of Playable references and call the play() method on all of them. The objects referred to by the list can very well be FBMediaFile objects.

Now, the observant student asks, "couldn't we simply extend FBFile and implement Playable, the old fascion way, without passing and FBFile to the constructor of FBMediaFile?"

Yes, we could. But what if there are many decorator classes, and we want to pick a few of them and decorate an FBFile? Let's say that there's a Displayable interface (for images and video), and a Streamable interface and some more decorator interfaces. Instead of creating classes which implmentents all of the combinations of the various interfaces, we can create one standard FBFile and then instantiate a whole chain of those decorator constructors as we see fit.

We could do something like:

```
FBFile fb = ... // we have a normal undecorated FBFile

// Let's decorate fb with capabilities of doing play(), display(), stream()
FBMediaFile fbm = new FBMediaFile(new FBDisplayable(new FBStreamable(fb)));
```

We think you need to read this text a few times and try to understand it, but also read the source code and try to understand that too, in order to fully grasp this example. But we think it shows some common and nice uses of interfaces, for those students who wish to learn a little more about Java.

The decorator technique used for FBMediaFile is very common. See for instance the java.io package and the streams classes found there.

One concrete example is the `java.io.PrintStream` class, which you use every day via `System.out` and the various `println()` methods.

The `PrintStream` class was created using the decorator pattern. You create a `PrintStream` object, for instance by passing some kind of `java.io.OutputStream` reference to the constructor of `PrintStream`. Now you create an object which is a normal `OutputStream` (using the reference you passed to the constructor), but which is also decorated with a lot of convenient methods, namely the various overloaded `println()` methods!

So you perhaps started with some kind of `OutputStream` but discovered that it would be very nice if you could also have all the `println()` methods for printing e.g. an int, a boolean, a String, some Object of unspecified type etc.

So, what you do, is discover the `PrintStream` class, and see that you can actually "wrap" your `OutputStream` inside a new `PrintStream`!

This makes your old `OutputStream` (which honestly doesn't have a lot of fancy methods for printing) "decorated" with all the new methods (println etc) of `PrintStream`, while keeping its type `OutputStream` because `PrintStream` extends `OutputStream`.

We did something similar with `FBMediaFile`. That class extends FBFile so an instance of FBMediaFile can be used anywhere an FBFile can be used - since an FBMediaFile *is an FBFile* via inheritance. At the same time, an FBMediaFile is a Playable object, since the class implements the Playable interface and its play() method.

The decorator pattern of course gets more useful when you have several decorator classes which can be chained together.

Links:

- Wikipedia on Decorator
- Old article on Oracle on Decorator
- OO Design - Decorator
- TutorialsPoint - Decorator Pattern
- DZone - Decorator Pattern Tutorial with Java Examples

Compiling and running the example

To compile and run, you can use the `build_and_run.sh` script like this:

```
$ chmod u+x build_and_run.sh
$ ./build_and_run.sh            #will list the files in current dir
Listing all files in .
[pdf] readme.pdf
[sh] gendoc.sh
[dir] somedir
[sh] clean.sh
[dir] org
[sh] build_and_run.sh
[dir] docs
[md~] README.md~
[md] README.md

Listing all Media files in .
```

```
Listing all Text files (.txt | .java) in . using a custom file filter:

Pretending to be a mediaplayer and playing all playables obtained from FBList.playables():


$ ./build_and_run.sh somedir/   #will list the files in somedir/
Listing all files in somedir/
[wmv] media.wmv
[txt] textfile3.txt
[java] SomeJavaClass1.java
[wma] media.wma
[wav] media.wav
[ogg] media.ogg
[mkv] media.mkv
[txt] textfile1.txt
[dir] directory1
[dir] directory3
[txt] textfile2.txt
[mp3] media.mp3
[java] SomeJavaClass2.java
[dir] directory2
[txt] textfile4.txt
[java] SomeJavaClass4.java
[avi] media.avi
[java] SomeJavaClass3.java

Listing all Media files in somedir/
[wmv] media.wmv
[wma] media.wma
[wav] media.wav
[ogg] media.ogg
[mkv] media.mkv
[mp3] media.mp3
[avi] media.avi

Listing all Text files (.txt | .java) in somedir/ using a custom file filter:
[txt] textfile3.txt
[java] SomeJavaClass1.java
[txt] textfile1.txt
[txt] textfile2.txt
[java] SomeJavaClass2.java
[txt] textfile4.txt
[java] SomeJavaClass4.java
[java] SomeJavaClass3.java
```

```
Pretending to be a mediaplayer and playing all playables obtained from FBList.playables():
Playing: [wmv] media.wmv
Playing: [wma] media.wma
Playing: [wav] media.wav
Playing: [ogg] media.ogg
Playing: [mkv] media.mkv
Playing: [mp3] media.mp3
Playing: [avi] media.avi
```