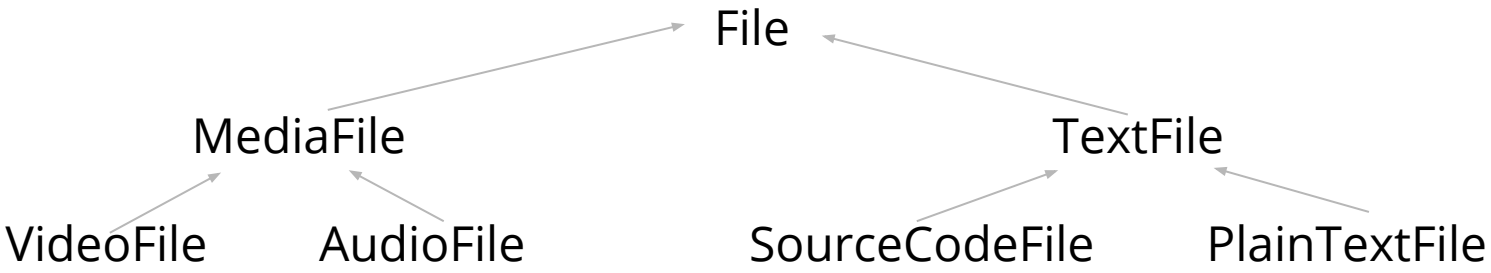
# Inheriting your own classes

Watch out for pitfalls

# You could use inheritance with your own classes

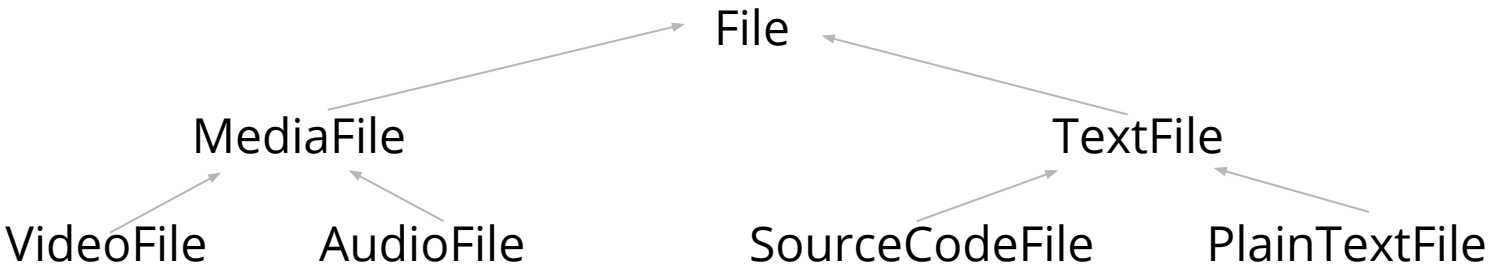
We could write a hierarchy for our file manager program classes like this:




(Arrows go from subclass to superclass)

# Some classes are still abstractions - abstract

Would we ever instantiate an object which truly is a File or a MediaFile?




We could use abstract classes here, using the keyword Abstract

# An abstract class

```
public abstract class File{
  // instance methods common to all types of files...
  // name,thumbnail image?, creation date, mod.date...

  // methods common to all files - but no implementation!
  public abstract void thumbnail(); // note the semicolon!
  public abstract String name();
  // more methods...
}
```

# The point of abstract classes is to define an api

With abstract classes we can define an API common to all subclasses.

Abstract methods don’t have any method body blocks with code (no implementation).

You could think of abstract methods as a promise: “all non-abstract subclasses will have these methods for real”.

# What if the instance variables are private?

We can’t access private variables directly from a subclass. But we can inherit public methods which do something with the private variables.

We actually inherit the private variables but need to use methods which are non-private to use them.

An alternative approach is to use the keyword protected which means that the variables (or methods) which are protected, can be accessed directly in the code of a subclass.

We (the authors) are not big fans of protected, however. We’ll pass that in this course!

# What separates a concrete class from an abstract?

Concrete subclasses implement all inherited abstract methods.

If we don’t implement all inherited abstract methods, then our subclass must be abstract too!

You cannot instantiate an abstract class - the compiler doesn’t allow it:

```
TestAbstractFile.java:3: error: File is abstract; cannot be instantiated
    File af = new File("myfile.bin");
                       ^
1 error
```

# It doesn’t hurt to name the class AbstractBlaBla

If our superclass File were to be abstract, we could actually name it AbstractFile to make this fact intuitive to all users of our classes.

# Avoid class explosions

It has been shown that the possibility to use inheritance using extends sometimes tempts some developers to make endless amounts of classes.

Imagine a program which stores information about animals in a zoo. If we're not careful, some eager inheritance fan developer could go crazy in designing the class hierarchy...

# Class explosion...

OrganicOrganism <-  Animal <- Mammal <- Primate <- Ape <- Gorilla <- SomeSpecificGorilla....

Writing a great large amount of hierarchical classes only makes the system hard to overview and understand. And it takes a lot of time to write all those classes.

How to know where to stop? It probably takes years to get a feel for *The right level of classes* ™

# Alternatives to inheritance

You could use composition instead of inheritance. Composition is what we use for normal classes, we compose them of instance variables and methods.

A File could be concrete and have attributes telling us what the type of the file is, how to generate the thumbnail etc.

The instance variables can be references to objects of other classes which are simply components of our class:

private FileType fileType;

private ThumbnailGenerator thumbNailGenerator;

# You could also use Interfaces as inheritance

Interfaces, we like. We like them so much, in fact, that we have dedicated the next chapter of the course to them.

# Can we prevent our classes from being extended?

Of course we can! (And sometimes we should!)

We can use the keyword `final` in the class declaration to say, “this class cannot be extended”.

This can be a security measure to prevent people from making unsafe versions of our class as subclasses and pass them as arguments to methods expecting our safe class.

It should also be done when you create immutable classes (like String), to make sure that the class stays immutable. Otherwise a subclass can break immutability.