Enhanced Class Design -- Introduction

- We now examine several features of class design and organization that can improve reusability and system elegance
- Chapter 9 focuses on:
  - abstract classes
  - formal Java interfaces
  - packages

Abstract Classes

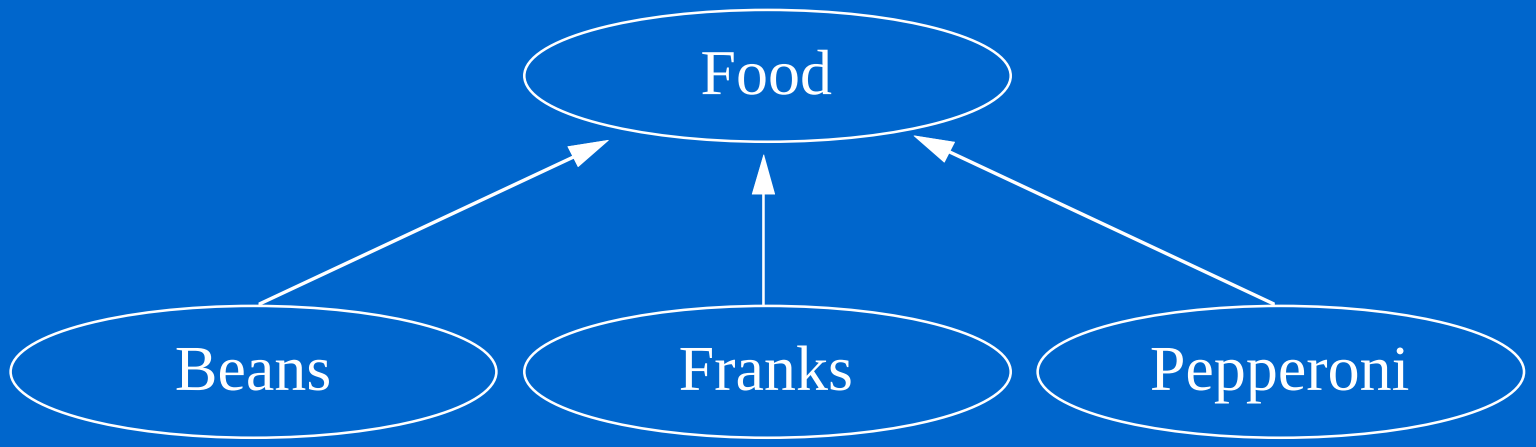
- An *abstract class* cannot be instantiated
- It is used in a class hierarchy to organize common features at appropriate levels
- An *abstract method* has no implementation, just a name and signature
- An abstract class often contains abstract methods
- Any class that contains an abstract method is by definition abstract

Abstract Classes

- The modifier `abstract` is used to define abstract classes and methods
- The children of the abstract class are expected to define implementations for the abstract methods in ways appropriate for them
- If a child class does not define all abstract methods of the parent, then the child is also abstract
- An abstract class is often too generic to be of use by itself

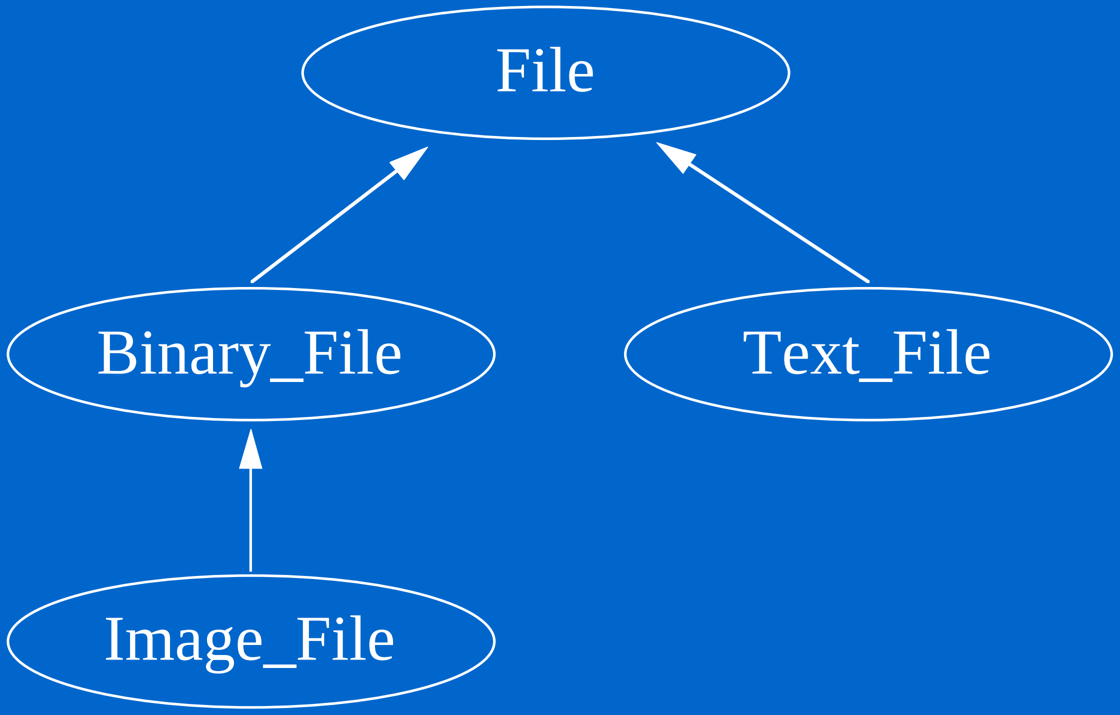
Abstract Classes

- See `Dinner.java`



Abstract Classes

- See `Printer.java`



Abstract Classes

- An abstract method cannot be declared as `final`, because it must be overridden in a child class
- An abstract method cannot be declared as `static`, because it cannot be invoked without an implementation
- Abstract classes are placeholders that help organize information and provide a base for polymorphic references

Interfaces

- We've used the term interface to mean the set of service methods provided by an object
- That is, the set of methods that can be invoked through an object define the way the rest of the system interacts, or interfaces, with that object
- The Java language has an interface construct that formalizes this concept
- A Java *interface* is a collection of constants and abstract methods

Interfaces

- A class that *implements* an interface must provide implementations for all of the methods defined in the interface
- This relationship is specified in the header of the class:

```
class class-name implements interface-name {
}
```

- See `Soap_Box.java`

Interfaces

- An interface can be implemented by multiple classes
- Each implementing class can provide their own unique version of the method definitions
- An interface is not a class, and cannot be used to instantiate an object
- An interface is not part of the class hierarchy
- A class can be derived from a base class <u>and</u> implement one or more interfaces

Interfaces

- Unlike interface methods, interface constants require nothing special of the implementing class
- Constants in an interface can be used in the implementing class as if they were declared locally
- This feature provides a convenient technique for distributing common constant values among multiple classes
- See `File_Protection.java`

Interfaces

- An interface can be derived from another interface, using the `extends` reserved word
- The child interface inherits the constants and abstract methods of the parent
- Note that the interface hierarchy and the class hierarchy are distinct
- A class that implements the child interface must define all methods in both the parent and child

Interfaces

- An interface name can be used as a generic reference type name
- A reference to any object of any class that implements that interface is compatible with that type
- For example, if `Philosopher` is the name of an interface, it can be used as the type of a parameter to a method
- An object of any class that implements `Philosopher` can be passed to that method

Interfaces

- Note the similarities between interfaces and abstract classes
- Both define abstract methods that are given definitions by a particular class
- Both can be used as generic type names for references
- However, a class can implement multiple interfaces, but can only be derived from one class
- See `Printer2.java`

Interfaces

- A class that implements multiple interfaces specifies all of them in its header, separated by commas
- The ability to implement multiple interfaces provides many of the features of *multiple inheritance*, the ability to derive one class from two or more parents
- Java does not support multiple inheritance
- See `Readable_Files.java`

Packages

- A Java *package* is a collection of classes
- The classes in a package may or may not be related by inheritance
- A package is used to group similar and interdependent classes together
- The Java API is composed of multiple packages
- The `import` statement is used to assert that a particular program will use classes from a particular package

Packages

- A programmer can define a package and add classes to it
- The *package statement* is used to specify that all classes defined in a file belong to a particular package
- The syntax of the package statement is:

  `package` *package-name*`;`

- It must be located at the top of a file, and there can be only one package statement per file

Packages

- The classes must be organized in the directory structure such that they can be found when referenced by an import statement
- There is a CLASSPATH environment variable on each computer system that determines where to look for classes when referenced
- See `Simple_IO_Test.java`

Packages

- The import statement specifies particular classes, or an entire package of classes, that can be used in that program
- Import statements are not necessary; a class can always be referenced by its fully qualified name in-line
- See `Simple_IO_Test2.java`
- If two classes from two packages have the same name and are used in the same program, they must be referenced by their fully qualified name

Packages

- As a rule of thumb, if you will use only one class from a package, import that class specifically
- See `Simple_IO_Test3.java`
- If two or more classes will be used, use the * wildcard character in the import statement to provide access to all classes in the package