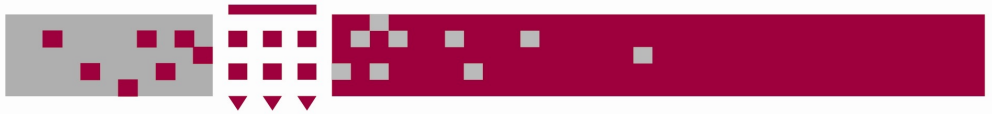
5COSC001W – Object Oriented Programming
Week 4

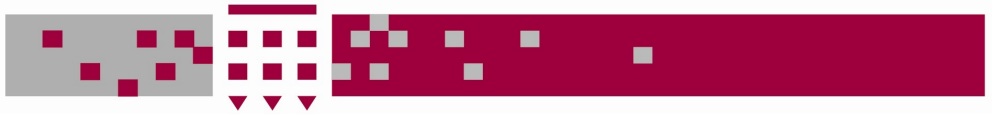
Dr. Barbara Villarini

b.villarini@westminster.ac.uk



What we learnt so far

- A real-life scenario can be modelled using objects.
- All **objects** that share similar characteristics or behaviours, that are of the same kind, belong to the same **class.**
- Object Oriented languages provide the features to implement an object-oriented model.



Object Oriented Principles we saw so far

- **Encapsulation**

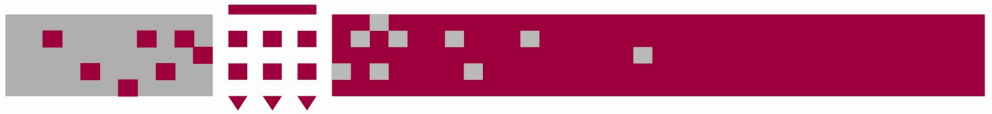
It keeps the data and the code safe from external interference. It is a mechanism for restricting direct access to some of the object's component. Binding the data with the code that manipulates it.

- **Inheritance**

Inheritance allows a class to use the properties and methods of another class. In other words, the derived class inherits the states and behaviors from the base class.

- **Polymorphism**.

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

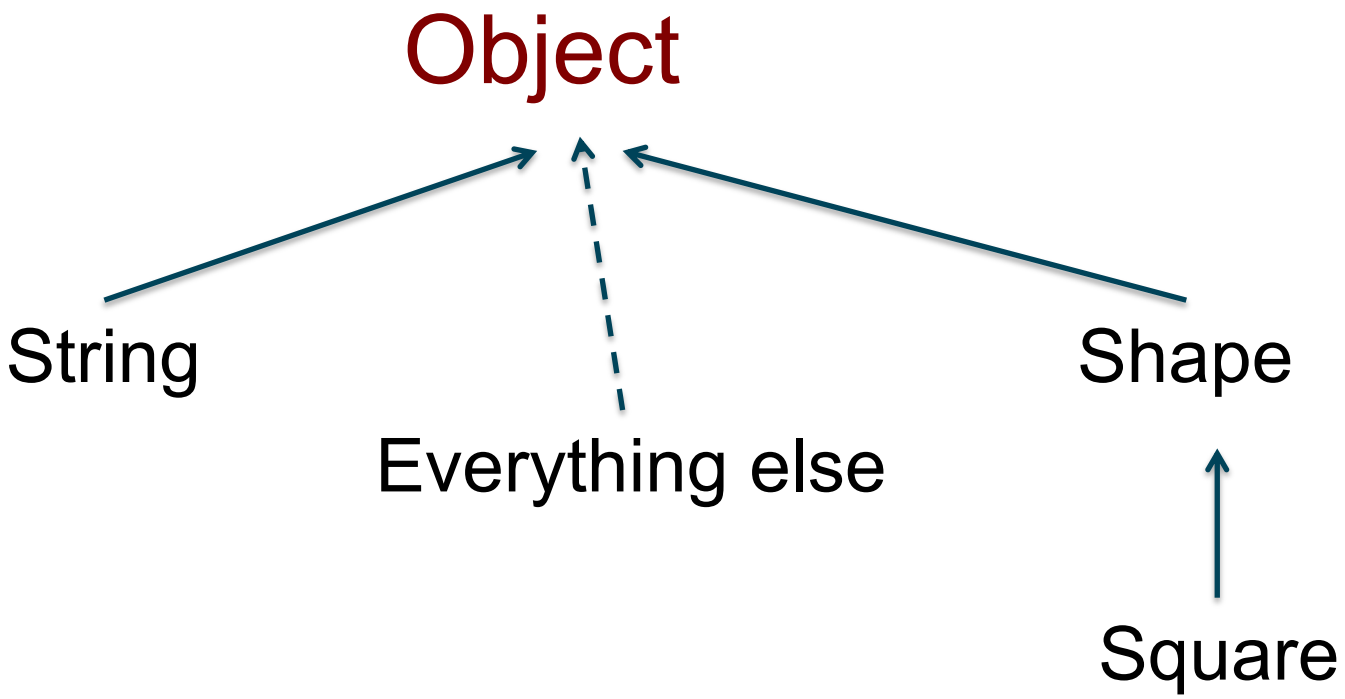


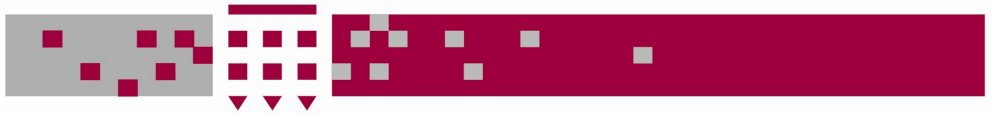
Summary

- Object Class in Java
- Final classes and methods
- Abstract classes
- Interfaces
- Introduction to design patterns

Java Inheritance Tree

- All Java classes you ever use or write yourself are in the *inheritance tree* with class Object at the top:

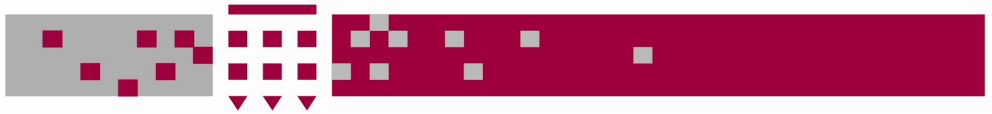




Everything is an Object

```
Object obj = new Square() ;
```

- OK
- But can only call methods declared by class Object.
- Of course, they may be overridden by subclasses.



The Object Class

- Every java class has Object as its superclass and thus inherits the Object methods.
- Object is a non-abstract class (You will see in a few slides the meaning of Abstract class)
- Many Object methods, however, have implementations that aren't particularly useful in general
- In most cases, it is a good idea to override these methods with more useful versions.
- In other cases, it is **required** if you want your objects to correctly work with other class libraries.

Some Object Class methods

- Object methods of interest:
  - equals
  - hashcode
  - toString
- Other object methods
  - getClass
  - wait, notify, notifyAll (relevant for threaded programming)

toString() method

- The Object method

```
String toString();
```

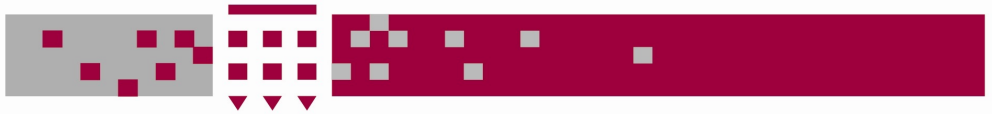
  is intended to return a readable textual representation of the object upon which it is called. This is great for debugging!

- Best way to think of this is using a print statement. If we execute:

```
System.out.println(someObject);
```

`Final` keyword – `Final` Variables

- Public static variables are often used to create symbolic constants.
  - E.g., Math.PI (static variable PI in class Math)
- Such variables are additionally declared final:
  - public *static* final double PI = 3.141;
- The value of a final variable cannot be changed by assignment.



`Final` classes

```
public final class X { }
```

The `final` keyword can be used:

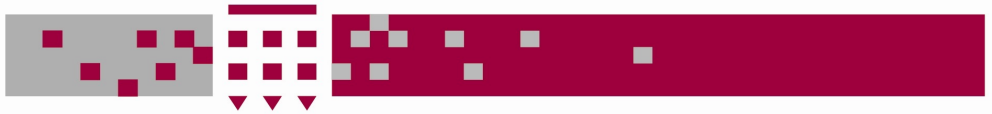
- to prevent inheritance, as final class cannot be extended
- to create an immutable class like the predefined String class. You can not make a class immutable without making it final.

`Final` methods

- Declaring a method “final” stops it being overridden:

```
public final void doSomething() {
      doThis() ;
      ... // Whatever
      doThat() ;
}
```

- doThis and doThat can be overridden but not doSomething.
- A constructor cannot be `final`



Why use final?

- Gives the class programmer control.
- Not all classes or methods are designed to be subclassed or overridden.
- Can explicitly enforce design decisions.

ABSTRACT CLASSES AND INTERFACES

Abstract Method

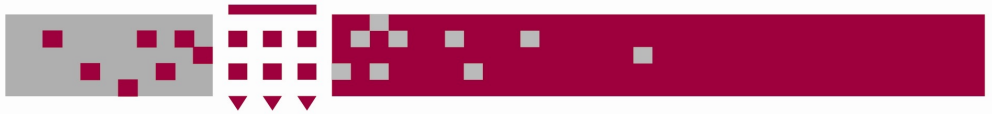
- Methods that are declared, with no implementation
- You can *declare* an object without *defining* it:

```
Person p;
```

- Similarly, you can declare a *method* without defining it:

```
public abstract void draw(int size);
```

  - Notice that the body of the method is missing
  - There is only the **signature** (a method signature is part of the method declaration. It's the combination of the method name and the parameter list)

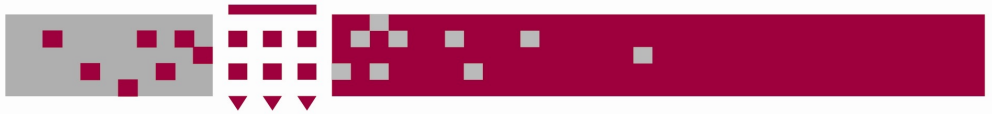


Abstract classes

- **Any class containing an abstract method is an abstract class**
- You must declare the class with the keyword **`abstract`**:

```
abstract class MyClass {...}
```

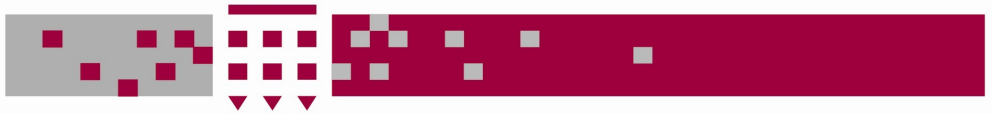
- An abstract class is *incomplete*
  - It has "missing" method bodies
- You cannot instantiate (create a new instance of) an abstract class. But it can provide a constructor!



Abstract classes

- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is "complete" and can be instantiated
  - If the subclass does *not* define all the inherited abstract methods, it must be abstract too

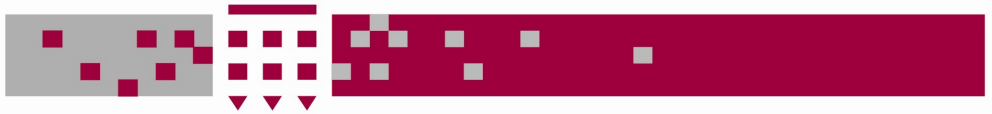
- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated



Example

```
public abstract class Animal {
    abstract String eat();
    abstract void breathe();
}
```

- This class cannot be instantiated
- Any non-abstract subclass of Animal **must** provide the `eat()` and `breathe()` methods



Why abstract classes?

- Suppose you wanted to create a class Shape, with subclasses Square, Rectangle, Triangle, etc.

- You don't want to allow creation of a "Shape"
  - Only *particular* shapes make sense, not *generic* ones
  - If Shape is abstract, you can't create a new Shape
  - You *can* create a new Square, a new Rectangle, etc.

- Abstract classes are good for defining a general category containing specific, "concrete" classes

A problem if we not use Abstract class

```
class Shape { ... }
class Star extends Shape {
    void draw() { ... }
    ...
}
class Square extends Shape {
    void draw() { ... }
    ...
}
Shape someShape = new Star();
someShape.draw();
```

It is not abstract and it does not have draw() method

This is legal, because a Star ***is*** a Shape

This is a syntax error, because someShape might not have a draw() method

Remember: ***A class knows its superclass, but not its subclasses***

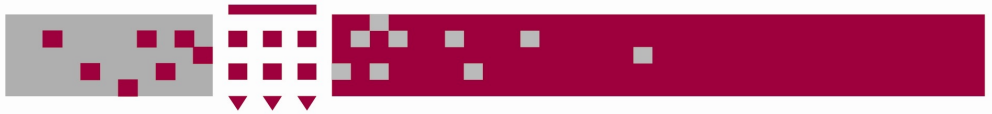
Solution

```
abstract class Shape {
    abstract void draw();
}
class Star extends Shape {
    void draw() { ... }
    ...
}
class Square extends Shape {
    void draw() { ... }
    ...
}
Shape someShape = new Star();
// However, Shape someShape = new Shape(); is no longer legal

someShape.draw();
```

This is legal, because a Star ***is*** a Shape

This is legal, because every actual instance *must* have a draw() method



Interfaces

- An interface declares (describes) methods but does not supply bodies for them

```
interface KeyListener {
        public abstract void keyPressed(KeyEvent e);
        public abstract void keyReleased(KeyEvent e);
        public abstract void keyTyped(KeyEvent e);
}
```

- All the methods are implicitly public and abstract
  - You can add these qualifiers if you like, but why bother?
- You cannot instantiate an interface
  - An interface is like a *very* abstract class—*none* of its methods are defined
- An interface may also contain constants (final variables)

Define interface

- To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Implements Interface

- When you say a class implements an interface, you are promising to *define* all the methods that were *declared* in the interface

- Example:

```
class MyKeyListener implements KeyListener {
      public void keyPressed(KeyEvent e) {...};
      public void keyReleased(KeyEvent e) {...};
      public void keyTyped(KeyEvent e) {...};
}
```

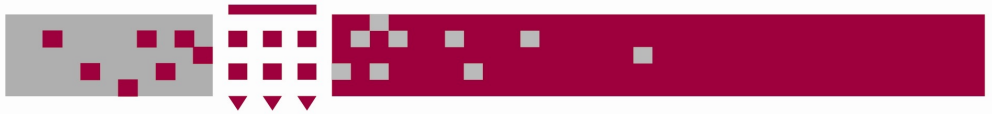
  – The "..." indicates actual code that you must supply

Partially implementing an Interface

- It is possible to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener
{
    public void keyTyped(KeyEvent e) {...};
}
```

- Since this class does not supply all the methods it has promised, it is an abstract class
- You must label it as such with the keyword abstract
- You can even *extend* an interface (to add methods):
  - `interface FunkyKeyListener extends KeyListener { ... }`

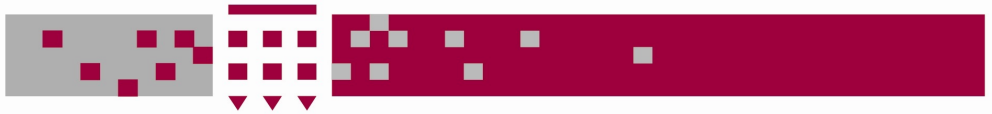


What's the difference between an interface and an abstract class?

- An interface cannot implement any methods, whereas an abstract class can
- A class can implement many interfaces but can have only one superclass (abstract or not)
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

Why are they useful?

- By leaving certain methods undefined, these methods can be implemented by several different classes, each in its own way.



What about inheritance?

- An abstract class can't inherit from more than one other class. (In java multiple inheritance is not allowed! In C++ it is!)

- Interfaces can inherit from other interfaces, and a single interface can inherit from multiple other interfaces

Example

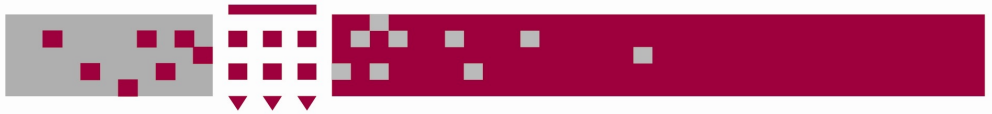
```
interface Singer  {

          void sing();
          void warmUpVoice();
      }

interface Dancer  {

          void dance();
          void stretchLegs();
      }

interface Talented extends Singer, Dancer  {
          // can sing and dance. Wowwee.
      }
```



Where else can interfaces be used?

- You can pass an interface as a parameter
- You can assign a class to an interface variable
- Just like you would do to an abstract class.

Example

Let's assume:

- **`Person`** is an Interface and **`Student`** class implements **`Person`**
- **`Sandwich`** and **`Apple`** are subclasses and extend **`Food`** class

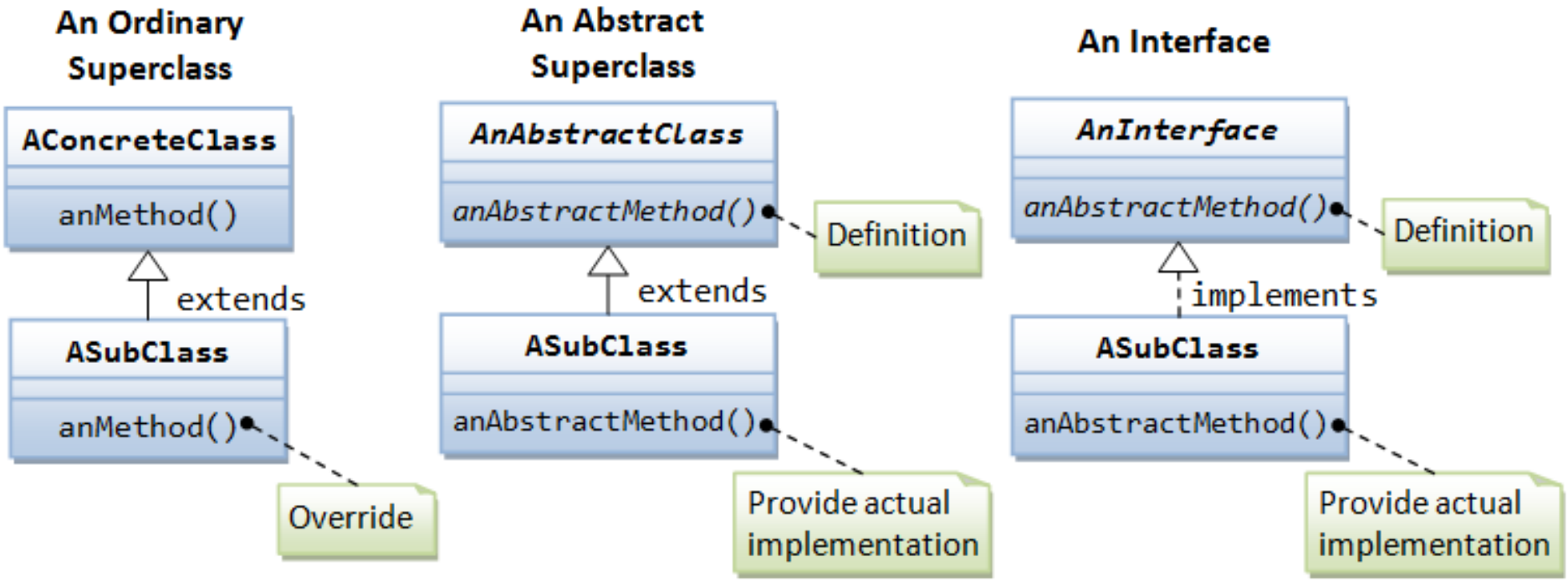
```
Food myLunch = new Sandwich();
Food mySnack = new Apple();
Person steve = new Student();
```

- If **`Person`** has methods **`eat(Food f)`** and **`teach(Person s)`**, the following is possible:

```
Person bob = new Student();
steve.teach(bob);
steve.eat(myLunch);
```

UML Class diagrams

- Abstract classes and interfaces are represented by placing the "<<abstract>>" stereotype and the "<<interface>>" stereotype respectively, above the class name or showing the class name in italics.



What is Design Pattern

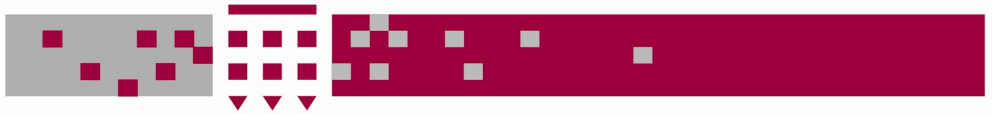
- Christopher Alexander says " *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*"

- A design pattern is a descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

- A pattern is made by four elements:
  - Name
  - Problem
  - Solution
  - Consequences

Design Patter - Name

- Describe a design problems and its solutions in a word or two
- Used to talk about design pattern with our colleagues
- Used in the documentation
- Increase our design vocabulary
- Have to be coherent and evocative

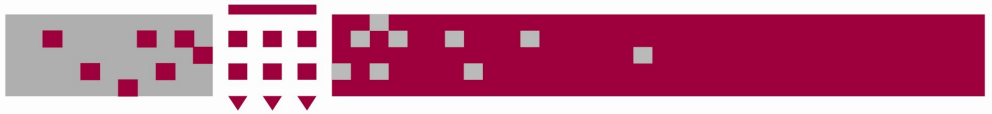


Design Pattern - Problem

- Describes when to apply the patterns
- Explains the problem and its context
- Sometimes include a list of conditions that must be met before it makes sense to apply the pattern
- Have to occurs over and over again in our environment

Design Pattern - Solution

- Describes the elements that make up the design, their relationships, responsibilities and collaborations
- Does not describe a concrete design or implementation
- Has to be well proven in some projects



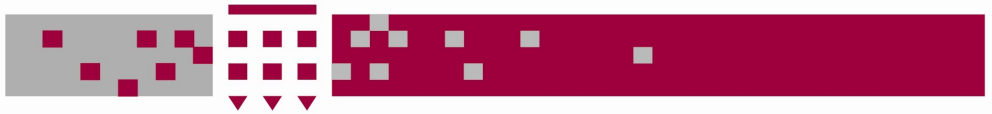
Design Pattern - Consequences

- Results and trade-offs of applying the pattern
- Helpful for describe design decisions, for evaluating design alternatives
- Benefits of applying a pattern
- Impacts on a system's flexibility, extensibility or portability

Classification of Design Pattern

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

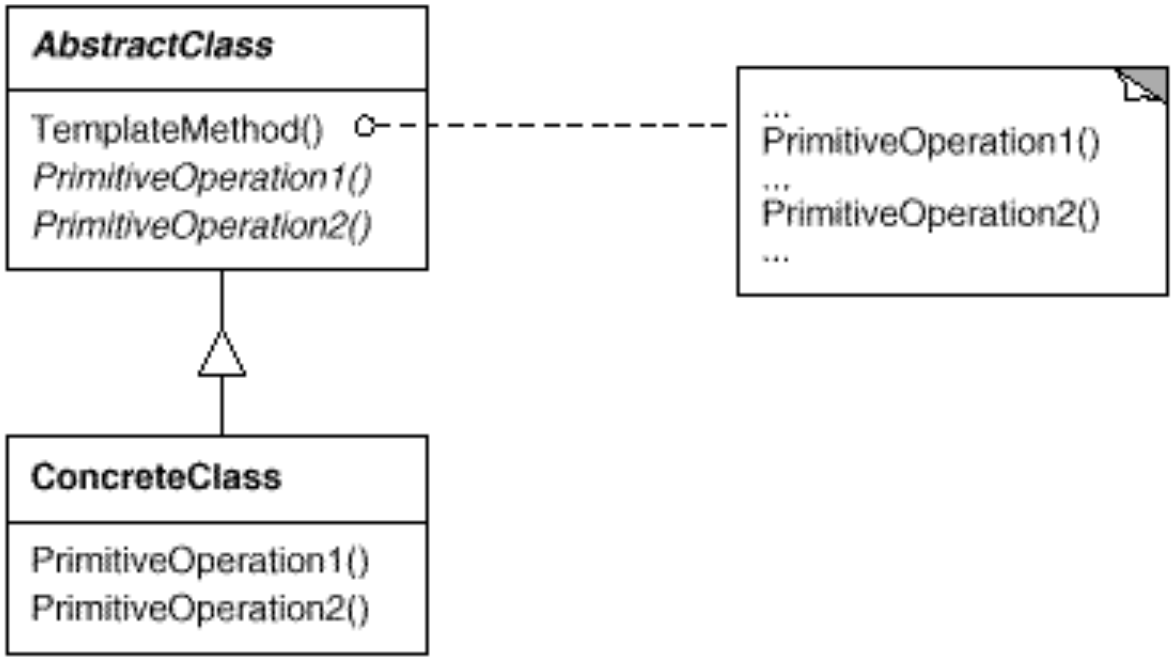
Scope: domain over which a pattern applies
Purpose: reflects what a pattern does



Template Method Pattern

- **Intent/Problem**: Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Solution:** The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all.

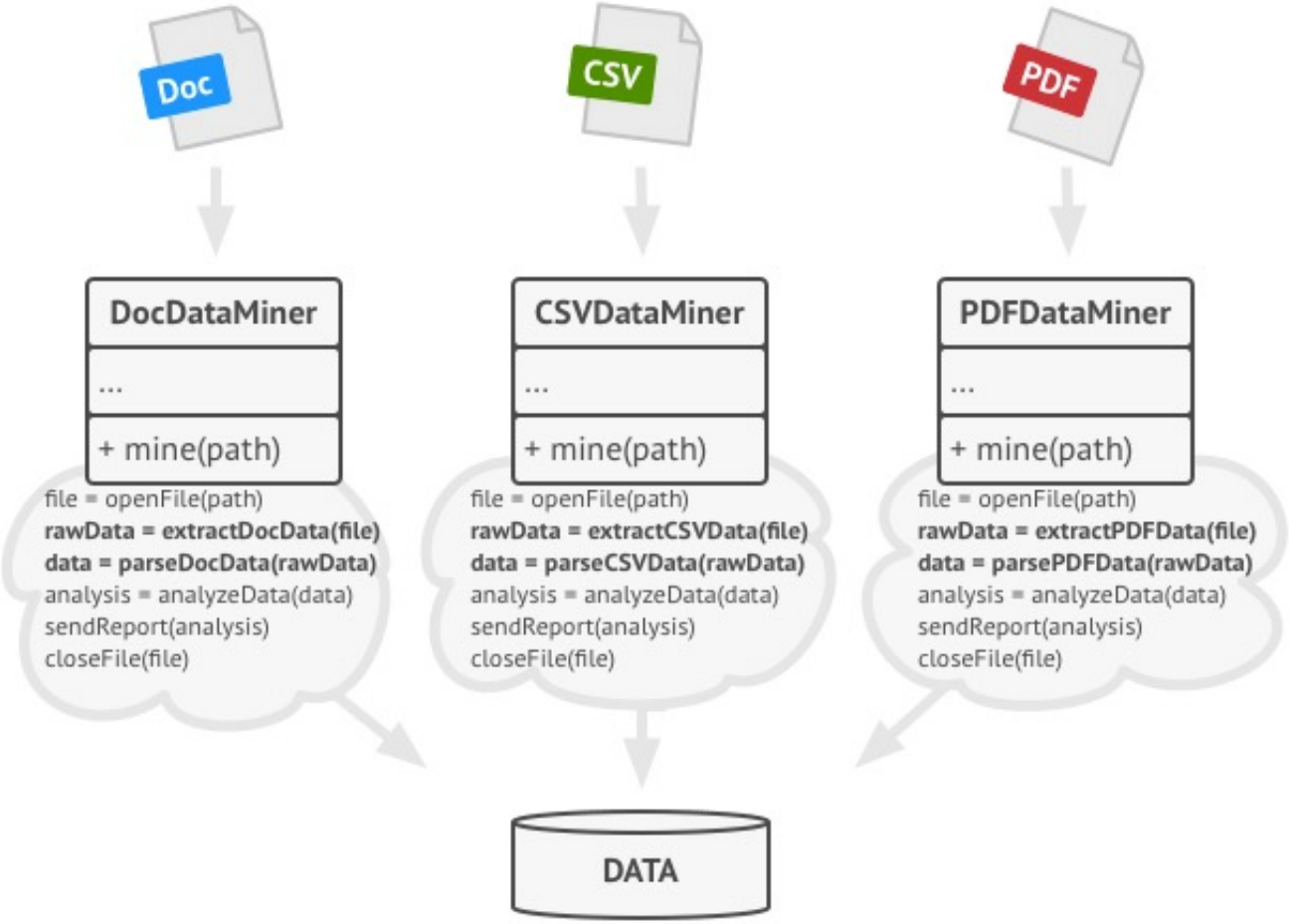
Template Method Pattern



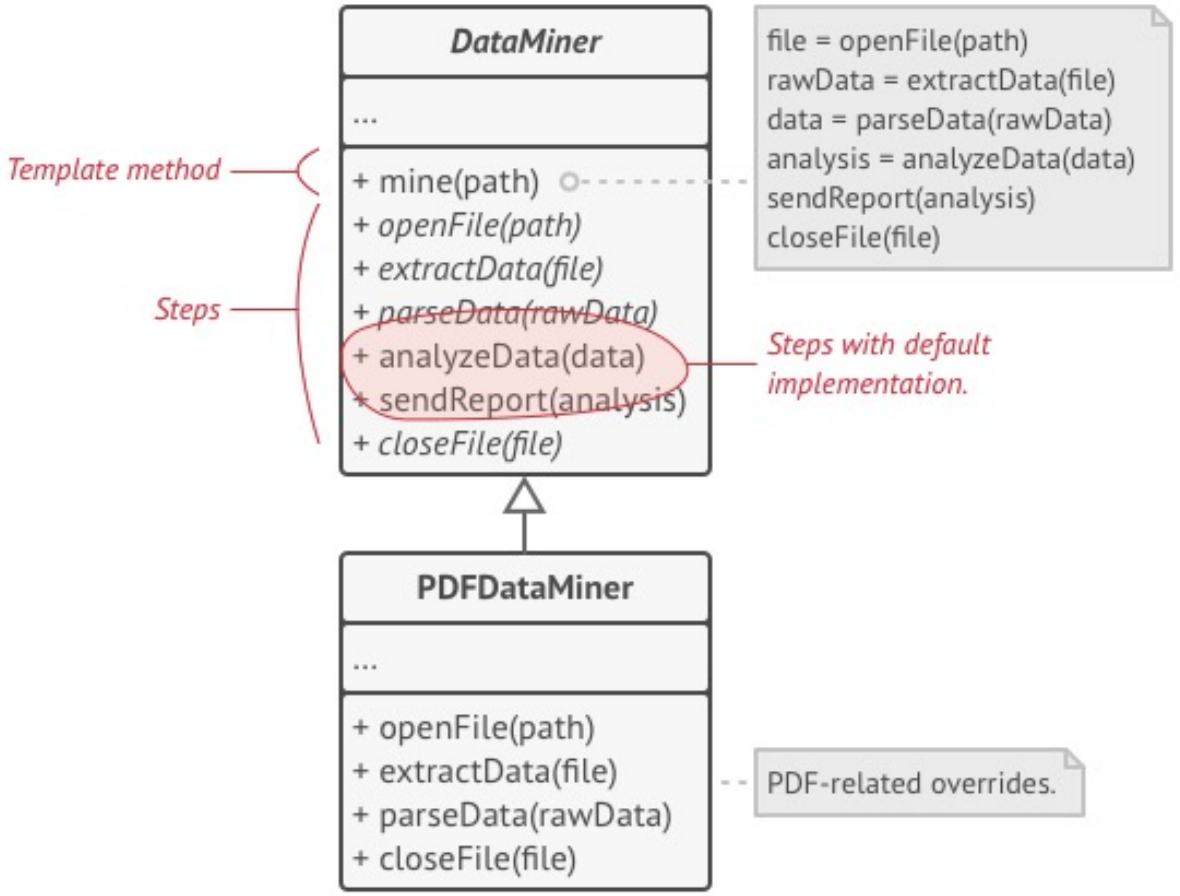
The key point to mention is that template method should be final, so that subclass can not override and change steps of the algorithm, but same time individual step should be abstract, so that child classes can implement them.

Example

Suppose to create a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.



Solution using Template Method Pattern



Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.