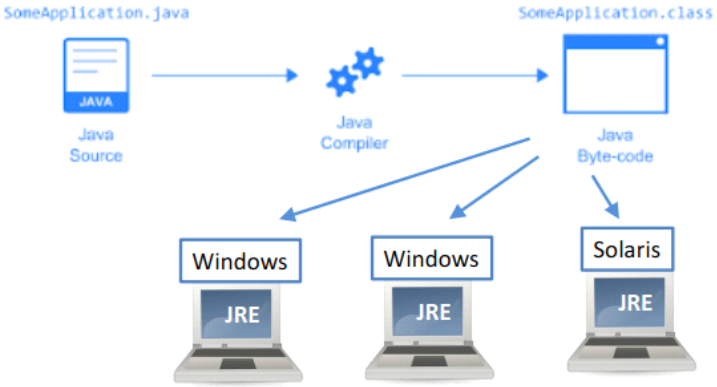
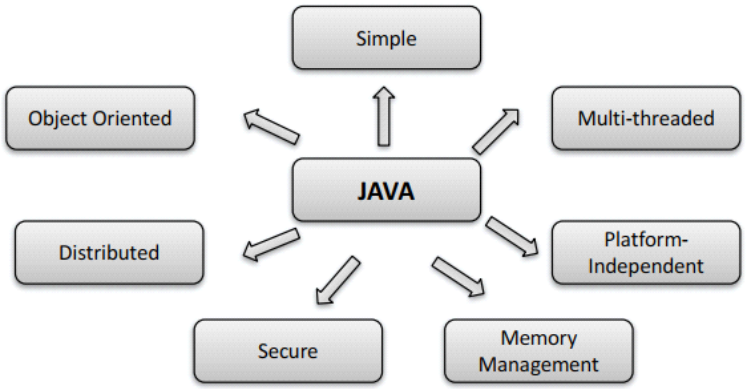
Intro to Java Platform

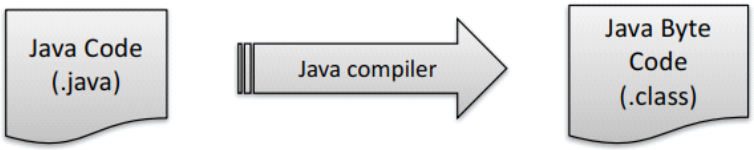
Friday, 31 May 2019 10:44 PM

Java is platform independent meaning it can be executed on different operating systems.





Java code is compiled in to java bytecode (*.class files) by a Java Runtime Environment ***(JRE)*** allowing it to run on multiple operating systems.



Java Language Basics

Java variables are passed by value!

Objects

Objects are real-world entities; they can be tangible and visible or intangible. Objects have *state* (characteristics or *attributes*) and *behaviours* (*methods*). Each object encapsulates some state (the currently assigned values for its attributes)

Objects interact and communicate by sending *messages* to each other. If object *A* wants object *B* to perform one of its methods, it sends a message to *B* requesting that behaviour .

This message is typically made of three parts:

1. The *object* to whom the message is addressed (e.g., John's "Account" object)
2. The *method* you want to invoke on the object(e.g., deposit())
3. Any additional information needed (e.g., amount to be deposited)

Classes

Many objects are of the same *kind* but have different identity. We can *logically group* objects that share some common properties and behaviour into a ***class*** (a blueprint for this logical group of objects).

An object is ***instantiated*** from a class and the object is said to be an ***instance*** of the class .An object instance is a

specific realisation of the class. Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other.

An object has state but a class doesn't. Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other.

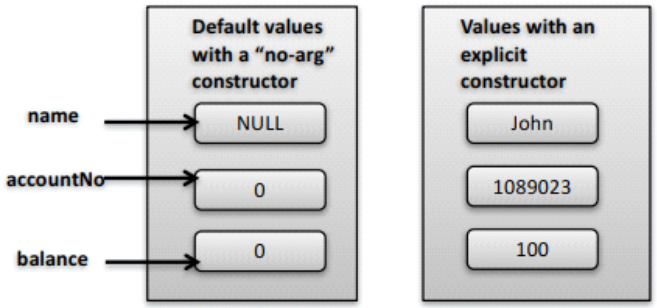
A class is sometimes referred to as an object's type. An object instance is a specific realisation of the class. We create an instance of the Account class like this:

```
public class Account {
    int accountNo;
    int bsb;
    float balance;

    public static void main(String[] args) {
        Account a1 = new Account();
        Account a2 = new Account();
    }
}
```

Constructor & Instance Variables

A ***constructor*** is a special method that creates an object instance and assigns values (initialisation) to the attributes (instance variables). Constructors eliminate default values. When you create a class without a constructor, Java automatically creates a default "no-arg" constructor for you.

```
public class Account {

    // instance variables
    String name;
    int accountNo;
    float balance;

    // constructor
    public Account(String aName, int acctNo, float bal) {
        this.name = aName;
        this.accountNo = acctNo;
        this.balance = bal;
    }

    public static void main(String[] args) {
        Account a1 = new Account("John", 1089023, 250);
    }
}
```



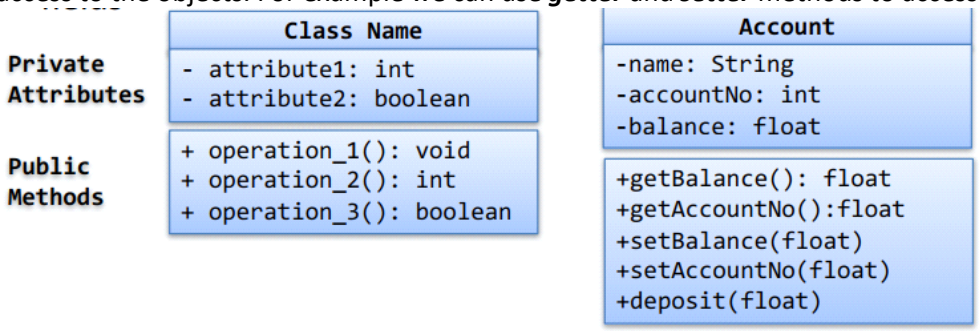
Note: we use **`this.var`** to refer to a class' instance variables

Abstraction

Abstraction helps you focus on the common properties and behaviours of objects. Good abstraction helps us accurately represent the knowledge we gather about the problem domain (discarding anything unimportant or irrelevant.

Encapsulation

Encapsulation is an OO design concept that emphasises hiding the implementation of an object. Essentially it implies the hiding of the object's attributes. An object's attributes represent its individual characteristics or properties, so access to the object's data must be restricted. It is only through ***methods*** do we provide explicit access to the objects. For example we can use ***getter*** and ***setter*** methods to access or modify fields.



Encapsulation is important because:

1. Encapsulation ensures that an object's state is in a consistent state
2. Encapsulation increases usability

    - Keeping the data private and exposing the object only through its interface (public methods) provides a clear view of the role of the object and increases usability
    - Clear contract between the invoker and the provider, where the client agrees to invoke an object's method adhering to the method signature and provider guarantees consistent behaviour of the method invoked (if the client invoked the method correctly)
3. Encapsulation abstracts the implementation, reduces the dependencies so that a change to a class does not cause a rippling effect on the system

Inheritance, Polymorphism and Domain Modelling

Friday, 7 June 2019 9:54 PM

OO in Java

Inheritance

Inheritance models a relationship between classes in which one class represents a more general concept (parent or base class) and another a more specialised class (sub-class). Inheritance models a "***is-a***" type of relationship e.g., a dog is-a type of pet, a manager is-a type of employee.

To implement inheritance, we

1. create a new class (sub-class) , that inherits common properties and behaviour from a base class (parent-class or super-class).
    - We say the child class *inherits/ is-derived from* the parent class
2. the sub-class can *extend* the parent class by defining additional properties and behaviour specific to the inherited group of objects
3. the sub-class can *override* methods in the parent class with their own specialised behaviour

Inheritance in Java:

```
public  abstract class Shape {

    public abstract double area();
    public abstract double circumference();

    protected static int count_shapes = 0;

}
```

Here Shape is the super class or base class and Circle is the sub-class which inherits and extends the behaviour and properties of Shape.

We indicate that Circle inherits Shape but writing 'extends Shape' in the class declaration.

Shape has abstract methods which can be overridden and implemented by Circle.

To call the a parent's method or access a parent's field, use **`super()`**

```
public class GraphicalCircle extends Circle {

    Color  outline, fill;
    public GraphicalCircle(){
        super();
        this.outline = Color.black;
        this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y, int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }
}
```

The invoked method does not have to be defined in the immediate parent class, but could be inherited from some class further up in the inheritance hierarchy.

We have two object instances of Account; a1 and a2.

```
public class Circle  extends Shape {

    protected static final double pi = 3.14159;
    protected  int  x, y;
    protected  int  r;
    protected static int count_circle = 0;

    // Very simple constructor
    public Circle(){
        this.x = 1;
        this.y = 1;
        this.r = 1;

        count_circle++;
        count_shapes++;
    }
    // Another simple constructor
    public Circle(int x, int y, int r){
        this.x = x;
        this.y = y;
        this.r = r;

        count_circle++;
        count_shapes++;
    }

    /**
     * Below, methods that return the circumference
     * area of the circle
     */
    public  double  circumference( ) {
        return 2 * pi * r ;
    }
    public  double  area ( ) {
        return  pi * r * r ;
    }
}
```

Constructors in Java

A sub-class does not inherit constructors. To create an instance of a subclass, there are two options:

1. Use the default "no-arg" constructor
   - This default constructor will make a default call to super(), which is the constructor in the parent class
2. Define a constructor in the sub-class
   - then, a default constructor is no longer provided
   - use super() to invoke the parent's constructor
   - Call to super() must be the first statement of the constructor. If this call is not made, a default call to super() is inserted

Overriding Methods

A sub-class can *override* methods in the parent class with its own specialised behaviour.
class Rectangle overrides method getArea() in parent class Shape to provide its own specific implementation

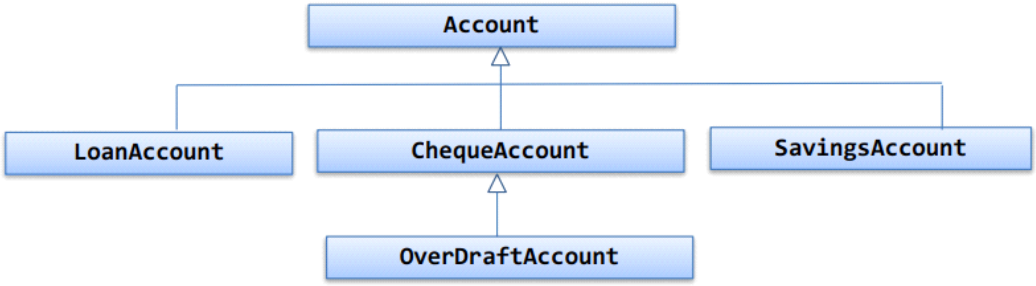
```
public class Shape {

    public String color;

    public Shape(String color) {
        this.color = color;
    }
    /*
     * @return Returns the area of the shape
     */
    public float getArea() {
        return 0;
    }
}
```

```
public class Rectangle extends Shape {

    public int height;
    public int width;

    public Rectangle(String color) {
        super(color);
    }

    @Override
    public float getArea() {
        return this.height * this.width;
    }
}
```

The method name and order of arguments of a child method must match the signature of the corresponding method in parent class.
Overridden methods cannot be less accessible.

Single Inheritance

The Java language allows a class to extend only one other class - ***single inheritance***. Multiple inheritance allows you to inherit from more than one super class. This is supported by languages such as C++, Python



Understanding Object Types

All object references have a type.
An object is just an instance of a class, but it is also an instance of its parent class.
A sub-class ***C*** has all the members of its parent class ***P***

```
Rectangle aRect = new Rectangle();
// But aRect is also an instance of class Shape
```

Hence, wherever an object of class **C**, it can be referenced as an object of class **P**

```
Shape aRect = new Rectangle();
Account myAccount = new SavingsAccount();
```

Java, gives you the ability to refer to an object using its actual form or parent form.

Polymorphism

Polymorphism means "many forms". It is an important OO principle that supports software reuse and maintenance.

The most common used of polymorphism in OOP occurs when a parent class reference is used to refer to a child

class object

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods it can invoke on the object.

Here the *variable* s1 is said to be polymorphic as it can refer to objects of different forms

```
Shape s1 = new Rectangle();
s1 = new Circle();
```

These assignments are legal as Rectangle and Circle are both types of Shape. However, the following does not compile:

```
s1.getHeight();
```

Using the variable s1, you can only access parts of the object that belong to the `class Shape`; the `Rectangle` specific components are hidden;
The Java compiler recognises that `s1` is a `Shape` **NOT** a `Rectangle`

Here, the function `getArea()` defined in class Shape and Rectangle is said to be "polymorphic" as the function can be applied on objects of different classes to achieve the same semantic result e.g.,

```
Shape s1 = Rectangle();
Rectangle r1 = new Rectangle();
```

Calling methods `s1.getArea()` and `r1.getArea()` invokes different behaviour but achieve the semantic result

A ***variable*** is ***polymorphic***, but an *object instance* has only one type (form), defined when it is instantiated. Here, ***dynamic binding*** or Virtual Method Invocation ensures that when a method is invoked, you get the behaviour associated with the object to which the variable refers to at runtime. The behaviour is not determined by the compile time type of the variable

instanceOf operator

As objects can be referenced using their parent classes, it is sometimes necessary to know what is the acutal type of the object at run-time. To find out we can used the `instanceOf` operator

```
public void getCoordinates(Shape s)
  if (s instanceOf Rectangle) {
       // do something
  }
  else if (s instanceOf Circle) {
       // do something
  }
```

Modifier	Same Class	Same Package	Sub Class	Everyone
public	√	√	√	√
protected	√	√	√	
default	√	√		
private	√			

Access Modifiers in Java

Java provides four access modifiers for variables, methods, and classes.

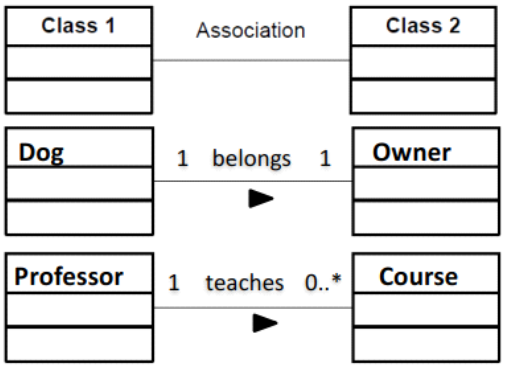
`Public`	Visible to the world
`Private`	Visible only to the class
`Protected`	Visible to the package and all subclasses
No modifier (default)	Visible to the package

Association

Association is a special type of relationship between two classes, that shows that the two classes are:

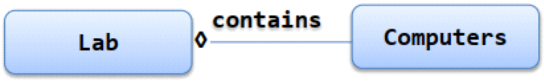
- Linked to each other; e.g. a lecturer *teaches* a course-offering
- Or combined in to some kind of ***"has a"*** relationship, where one class *contains* another class; a course-offering has students

In a UML diagram, the two classes are modelled like this

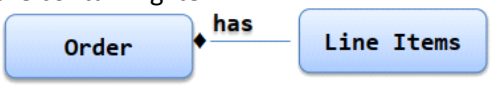


Associations can be further refined as:

- ***Aggregation*** relationship (hollow diamond symbol ◊ in UML diagrams): the contained item is an element of the a collection but ***can exist on its own***.



- ***Composition*** relationship (filled diamond symbol ♦ in UML diagrams): the item contained is an ***integral part*** of the containing item

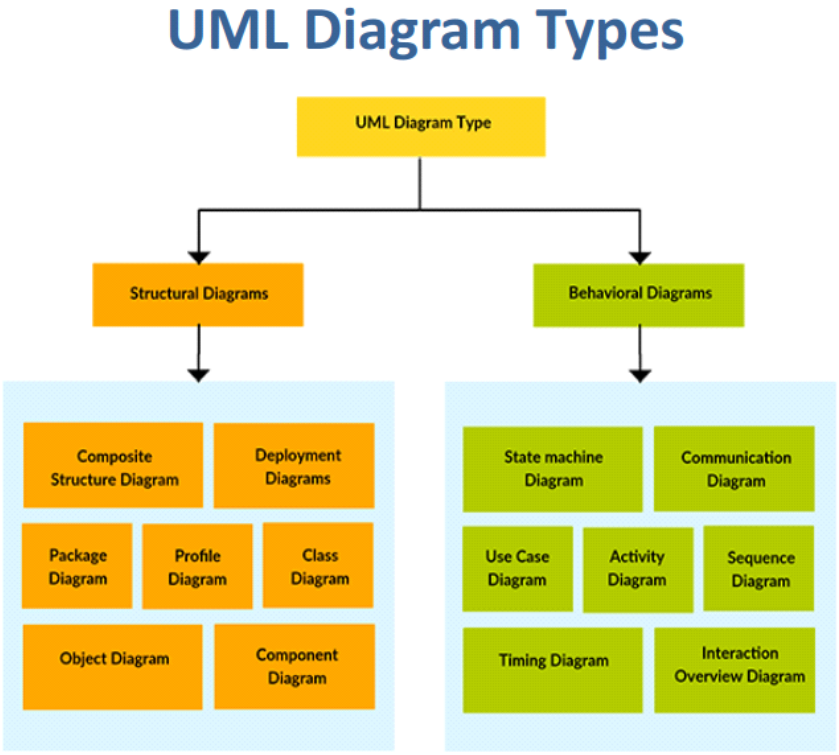


Domain Modelling

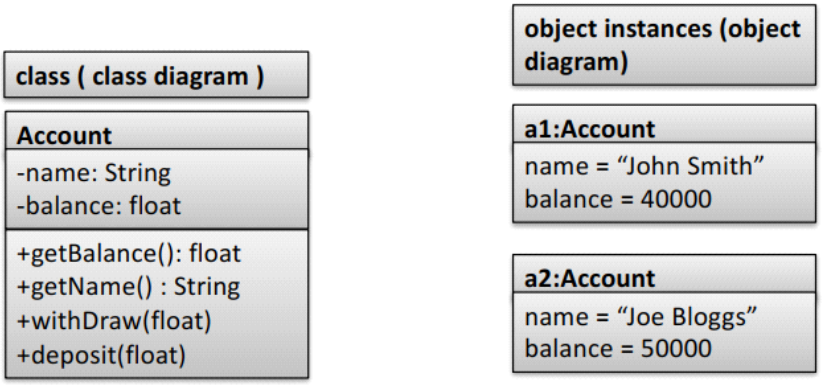
UML stands for Unified Modelling Language (http://www.uml.org/). It is used because programming languages are not abstract enough for OO design. It is an open source, graphical language to model software solutions, application structures, system behaviour and business processes. It is a language independent modelling tool and is sometimes used for auto code generation.

There are several uses for UML

- As a design that communicates aspects of your system
- As a software blue print
- Sometimes, used for auto code-generation



Representing classes in UML



Requirements Analysis vs Domain Modelling

Requirements analysis determines ***external behaviour*** - "What are the features of the system-to-be and who requires these features (actors)".
Domain modelling determines ***internal behaviour*** - "how elements of system-to-be interact to produce the external behaviour".
Requirements analysis and domain modelling are ***mutually dependent*** - domain modelling supports clarification of requirements, whereas requirements help building up the model .

Domain Model

A domain model is also referred to as a conceptual model or domain object model. It provides a visual representation of the problem domain, through decomposing the domain into key concepts or objects in the real-world and identifying the relationships between these objects. Techniques to build a domain model include:

- Noun/Verb Analysis
  - analyse textual description of the domain to identify **noun** phrases
  - Caveats: Textual descriptions in natural languages are ambiguous (different nouns can refer to the same thing and the same noun can mean multiple things
- CRC Card
  - CRC stands for:
    - ***Class*** : Represents a collection of similar objects
    - ***Responsibility*** : Something that the class *knows* or *does*
    - ***Collaborator*** : Another class that a class must interact with to fulfil its responsibilities
  - Written in 4 by 6 index cards, an individual CRC card use to represent a domain object
  - Featured prominently as a design technique in XP programming

Student	
Enrols in a Course-Offering *Knows* Name *Knows* Address *Knows* Phone Number	Course-Offering

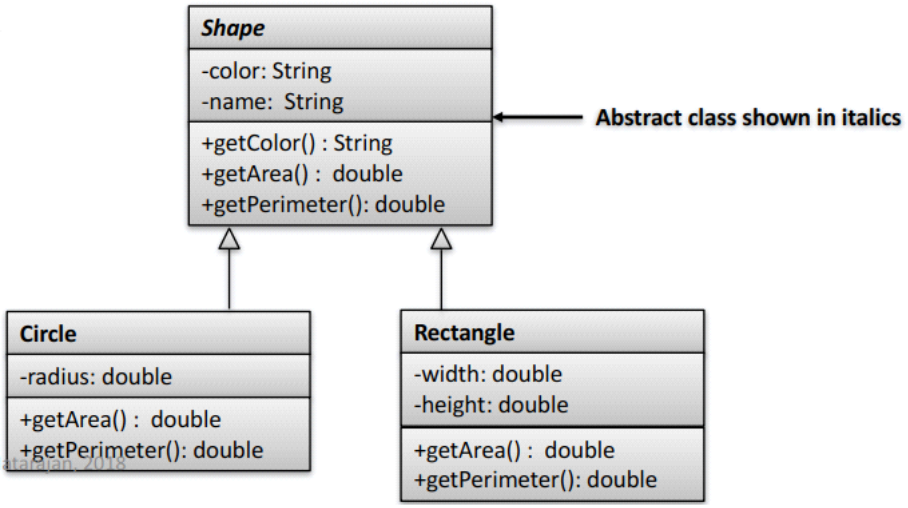
Interfaces, Design Principles and Refactoring I

Friday, 21 June 2019 12:57 PM

Abstract Classes

An ***abstract class*** is a class which cannot be instantiated. It serves as a convenient place-holder for factoring out common behaviour in sub-classes.
An abstract class *may* define ***abstract methods*** (undefined in the abstract class, but defined/implemented in the subclass).

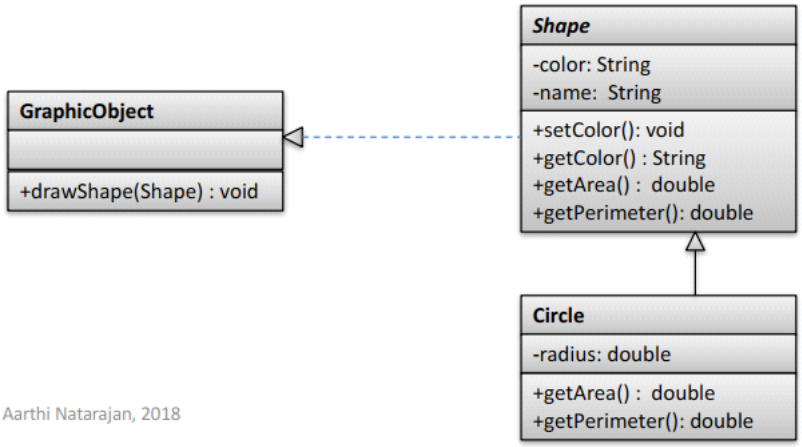
A class with one or more abstract methods must be declared as abstract.



Interfaces

An ***interface*** type defines a collection of abstract methods. i.e. defining methods without a body, so it specifies what need to be done, but not how to do it.
A class implements an interface, by providing an implementation for **ALL** the methods in the interface.

Classes can implement multiple interfaces, but extend only one class.



Aarthi Natarajan, 2018

Polymorphism works with interfaces as well:

```
GraphicObject g1 = new Rectangle();
GraphicObject g2 = new Circle();
```

where `Rectangle` and `Circle` implement the interface `GraphicObject`
Polymorphism guarantees that the right `drawshape()` method is applied to ensure correct result.

Designing Good Software

Building good software is all about

- Making sure your software does what the customer wants it to do. We can model this with use-case diagrams, feature lists, and prioritise them.
- Applying OO design principles to ensure the system is flexible and extensible to accommodate changes in requirements and to strive for maintainable, reusable, extensible design

A change in requirements sometimes reveals problems with your system that you did not even know existed. Remember that change is constant and your system should continually improve when you add these changes or else your software will rot.

Why does software rot?

Because we write bad code. Why?

- Because we don't know how to write better code?
- Requirements change in ways that the original design did not anticipate
- Changes require refactoring and refactoring requires time and we might not have time
- Business pressure causes us to make quick and dirty solutions
- Changes may be made by developers not familiar with the original design philosophy

Design Smells

A ***design smell*** is a symptom of poor design. It is often caused by violation of key design principles. Design smells have structures in software that suggest refactoring.

Design smell symptoms:

- ***Rigidity*** - the tendency of the software being too difficult to change even in simple ways. A single change causes a cascade of changes to other dependent modules
- ***Fragility*** - the tendency of the software to break in many places when a single change is made

Rigidity and fragility complement each other – aim towards minimal impact, when a new feature or change is needed

- ***Immobility*** - the design is hard to reuse. The design has parts that could be useful to other systems, but the effort needed and risk in dis-entangling the system is too high
- ***Viscosity***
    - Software viscosity – changes are easier to implement through 'hacks' over 'design preserving methods'
    - Environment viscosity – development environment is slow and in-efficient
- ***Opacity*** - the tendency of a module to be difficult to understand. Code must be written in a clear and expressive manner
- ***Needless complexity*** - contains constructs that are not currently useful. Developers ahead of requirements
- ***Needless repetition*** - design contains repeated structures that could potentially be unified under a single abstraction. Bugs found in repeated units have to be fixed in every repetition

Characteristics of Good Design

The design quality of software is characterised by

1. Coupling
2. Cohesion

Good software aims to have ***loose coupling*** and ***high cohesion*** among its components so that software entities are:

- Extensible
- Reusable
- Maintainable
- Understandable
- Testable

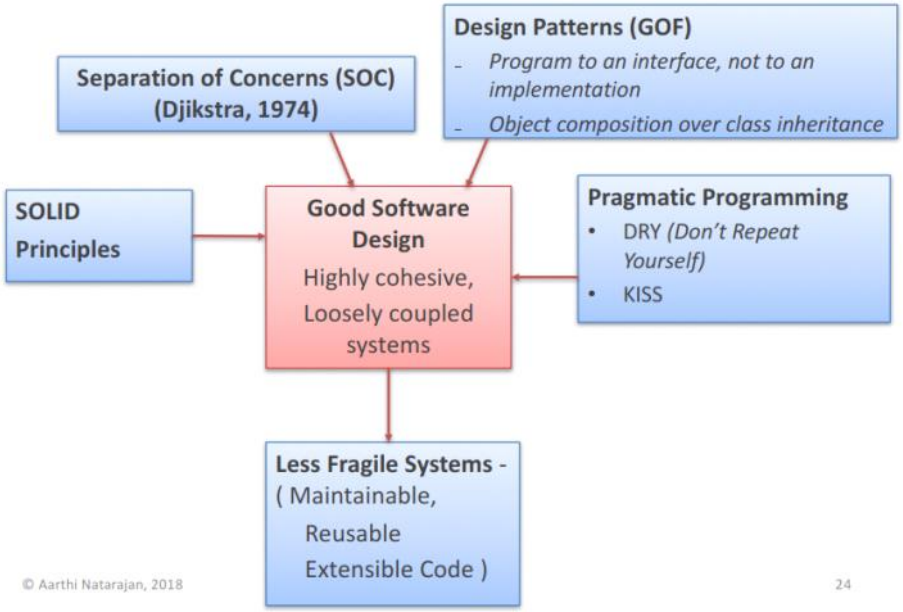
Coupling is the degree of interdependence between components or classes. ***High coupling*** occurs when one component A depends on the internal workings of another component B and is affected by internal changes to component B. High coupling leads to a complex system, with difficulties in maintenance and extension; eventually the software will rot. We aim to write ***loosely coupled*** classes. This allows for components to be used and modified independently of each other. At the same time ***zero coupled*** classes are not usable/useless. We need to strike a balance between the two.

Cohesion is the degree to which all elements of a component or a module work together as a functional unit. ***Highly cohesive*** modules are much easier to maintain, less frequently changed and have higher probability of reusability. Do not put all the responsibilities into a single class to avoid low cohesion.

Cohesion means that the class/component should only have one responsibility. This does not mean you can only have one method per class. It means that all the functions in the module must work together towards the same purpose.

Design Principles

Design principles are a basic tool or technique that can be applied to designing, or writing code to make software more maintainable, flexible and extensible. All design principles have one goal; *making good software*



Design principles help eliminate design smells, but **DO NOT** apply design principle when there are no design smells. Unconditionally conforming to a principle just because it is a principle is a mistake. Over conformance leads to the design smell of needless complexity

Design Principle #1: The Principle of Least Knowledge/Law of Demeter

Classes should know about and interact with as few classes as possible, so we should reduce the interaction between objects to just a few friends. These friends are immediate friends or local object.

This principle helps design loosely coupled systems so that changes to one part of the system does not cascade to other parts of the system. The principle limits interactions through a set of rules.

A method in an object should only invoke methods of:

1. ***The object itself*** - *a method **M** in and object **O** can call on any other method within **O** itself.*
   This rule makes logical sense, a method encapsulated within a class can call any other method that is also encapsulated within the same class

```
public class M {
    public void methodM() {
        this.methodN();
    }

    public void methodN() {
        // do something
    }
}
```

   Here `methodM()` calls `methodN()` as both are methods of the same class
2. ***The object passed in as a parameter to the method*** - *a method **M** in an object **O** can call on any methods of parameters passed to the method **M***
   The parameter is local to the method, hence it can be called as a friend

```
public class O {
    public void M(Friend f) {
        // Invoking a method on a parameter passed to the method is legal
```

```
        f.N();
}

public class Friend {
    public void N() {
        // do something
    }
}
```

3. ***Objects instantiated within the method*** *- A method **M** can call a method **N** of another object, if that object is instantiated within the method **M***
   The object instantiated is considered "local" just as the object passed in as a parameter

```
public class O {
    public void M() {
        Friend f = new Friend();
        // Invoking a method on an object created within the method is legal
        f.N();
    }

public class Friend {
    public void N() {
        // do something
    }
}
```

4. ***Any component object*** *- any method **M** in an object **O** can call on any methods of any type of object that is a direct component of **O***
   This means a method of a class can call methods of classes of its instance variables

```
public class O {
    public Friend instanceVar = new Friend();

    public void M4() {
        // Any method can access the methods of the friend class
        // F through the instance variable "instanceVar"
        instanceVar.N();
    }
public class Friend {
    public void N() {
        // do something
    }
}
```

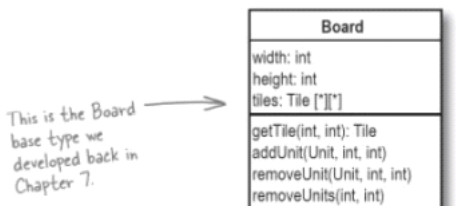
Not of the objects returned by a method.

Do not do something like this: `o.get(name).get(thing).remove(node)`

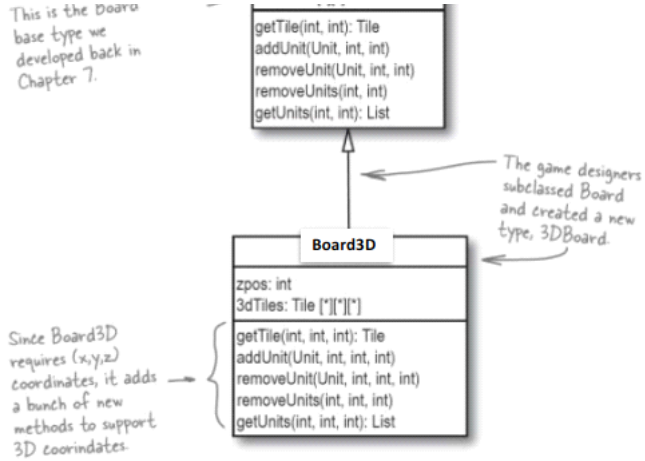
Design Principle #2: LSP (Liskov Substitution Principle)

LSP is about ***well-designed inheritance***.

Barbara Liskov (1988) wrote:
If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.
Bob wrote:
subtypes must be substitutable for their base types



In this example of inheritance, the `Board3D` cannot be treated as the `Board`, which it inherits. None of the



In this example of inheritance, the `Board3D` cannot be treated as the `Board`, which it inherits. None of the methods on `Board` work in a 3D environment.

Inheritance and LSP indicate that any method on Board should be able to be used on Board3D, and that Board3D can stand in for Board without any problems. So the example clearly violates LSP

There are other options besides inheritance:

- ***Delegation*** - delegate the functionality to another class
- ***Composition*** - reuse behaviour using one or more classes with composition

If you favour delegation and composition over inheritance, your software will be more flexible, easier to maintain and extend.

Method Overriding

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method
- The access level cannot be more restrictive or weaker than the overridden method's access level. E.g., if the super class method is declared `public` then the overriding method in the sub class cannot be either `private` or `protected`.
- A method declared `final` cannot be overridden.
- Constructors cannot be overridden.

Can static methods be overridden?
No. Static methods can be defined in the sub-class with the same signature. This however is not overriding, as there is no run-time polymorphism. The method in the derived class hides the method in the base class.

Example:

```
// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // This method hides display() in Base
    public static void display() {
         System.out.println("Static or class method from Derived");
    }

    // This method overrides print() in Base
    public void print() {
```

```
        System.out.println("Non-static or Instance method from Derived");
    }
}

// Driver class
public class Test {
    public static void main(String args[ ])  {
        Base obj1 = new Derived();

        // As per overriding rules this should call to class Derive's static
        // overridden method. Since static method can not be overridden, it

        obj1.display();   // prints "Static or class method from Base

        obj1.print();     // prints "Non-static or Instance method from Derived
    }
}
```

Co-variance of return types in the overridden method

The return type in the overridden method should be the same or the sub-type of the return type defined in the super-class. This means that the return types in the overridden method may be narrower than the parent's return types.

```
public class AnimalShelter {

    public Animal getAnimalForAdoption() {

        return null;
    }

    public void putAnimal(Animal someAnimal){

    }

}
```

```
public class CatShelter extends AnimalShelter {

    /*
     * @see AnimalShelter#getAnimalForAdoption()
     */
    @Override
    public Cat getAnimalForAdoption() {

            //Returning a narrower type than parent
            return new Cat();
    }

}
```

Contra-variance of method arguments in the overridden method

The arguments in an overridden method in a sub-class can be wider than the arguments passed in the parent's method

```
public class CatShelter extends AnimalShelter {


    /*
     * @see AnimalShelter#putAnimal(Animal)
     */
    // Java sees this as an unrelated method.
    // This is not actually overriding parent method
    public void putAnimal(Object someAnimal) {
        // do something
    }
```

Refactoring

Refactoring is the process of *restructuring* (changing the internal structure of software) software to make it *easier to understand* and *cheaper to modify* without changing its external observable behaviour.

Refactoring:

- Improves design of software
- Makes software easier to understand
- Helps you find bugs
- Helps you program faster
- Helps you conform to design principles and avoid design smells

You should refactor when you add a feature to a program and the program's code is not structured in a convenient

way to add the feature. First refactor the program to make it easier to add the feature, then add the actual feature.

You should also refactor when you need to fix a bug, and as you do a code review.

Common Bad Code Smells

- **Duplicated Code**
  - Same code structure in more than one place or
  - Same expression in two sibling classes
- **Long Method**
- **Large Class** (when a class is trying to do too much, it often shows up as too many instance variables)
- **Long Parameter List**
- **Divergent Change** (when one class is commonly changed in different ways for different reasons)
- **Shotgun Surgery** (The opposite of divergent change, when you have to make a lot of little changes to a lot of different classes

Design Principles 1: Strategy Pattern

Wednesday, 26 June 2019 11:49 AM

Remember that knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

Design Principle #3: Encapsulate aspects of your code that varies and separate it from code that stays the same

By separating what changes from what stays the same, the result is fewer unintended consequences from code changes and more flexibility in your software. Another way to think about this principle: *take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.*

Design Principle #4: Program to an interface, not to an implementation

Program to an interface means to *program to a super-type*. I.e. the declared type of the variable should be a super-type (abstract class or interface). For example

```
// programming to an implementation
Dog d = new Dog();
d.bark();

// programming to an interface
Animal a = new Dog();
a.makeSound();
```

What we want is to exploit polymorphism by programming to a super-type so that the actual run-time object isn't locked into the code.

Design Principle #5: Favour Composition over Inheritance

Consider the following example.
Each duck has a fly behaviour and quack behaviour. Instead of inheriting their behaviour, the ducks get their behaviour by being composed with the right behaviour objects and delegate to the behaviour object. This allows you to encapsulate a family of algorithms and enables you to change behaviour at run time.

Design Patterns

A ***design pattern*** is a tried solution to a commonly recurring problem. The original use comes from a set of 250 patterns formulated by Christopher Allexander et al for architectural building design. Every pattern has:

- A short name
- A description of the context
- A description of the problem
- A prescription for a solution

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern

- Represents a template for how to solve a problem
- Captures design expertise and enables this knowledge to be transferred and reused
- Provide shared vocabularies, improve communications and eases implementation
- Is not a finished solution, they give you general solutions to design problems

Using design patterns is essentially and *art & craft*. You have to have a good working knowledge of patterns, understand the problems they can solve, and recognise when a problem is solvable by a pattern.

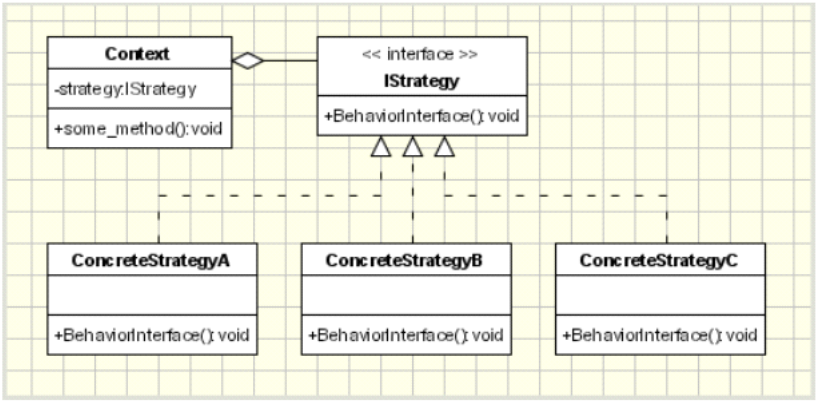
Design patterns are categorised as:

- Behavioural patterns
- structural patterns
- Creational patterns

Design Pattern #1: Strategy Pattern

Strategy pattern allows you to encapsulate a family of algorithms and lets you *change behaviour* at run-time.

It was made from the need to adapt the behaviour of an algorithm at run-time.
The intent of a strategy pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy pattern is a behavioural design pattern that lets the algorithm vary independently from the context class using it.



Strategy pattern is applicable when many related classes differ in their behaviour.
A context class can benefit from different variant of an algorithm.
A class defines many behaviours, and these appear as multiple conditional statements (e.g. if or switch). Instead, we can move each conditional branch into their own concrete strategy class.

Benefits

- Uses composition over inheritance, which allows better decoupling between the behaviour and context class that uses the behaviour.

Drawbacks

- Increases the number of objects
- Client must be aware of different strategies

Examples of strategy patterns

- Sorting a list (quicksort, bubble sort, merge-sort)
    - Encapsulate each sort algorithm into a concrete strategy class
    - Context class decides at run-time which sorting behaviour is needed
- Searching (binary, DFS, BFS, A*)

State Pattern

Thursday, 4 July 2019 12:12 PM

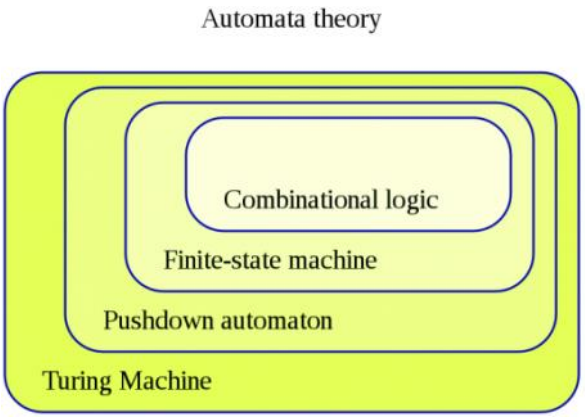
Finite State Machine

A ***finite-state machine (FSM)*** is an abstract machine that can be in exactly **one** of a finite number of ***states*** at any given time. The finite state machine can change from one state to another in response to some external inputs. The change from one state to another is called a ***transition***.

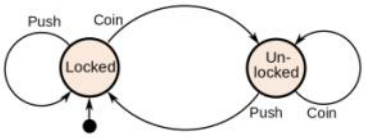
A finite-state machine is **defined** by

- A list of **states**
- The conditions for each **transition**
- Its **initial** state

A finite-state machine is also referred to as ***finite-state automaton***, ***finite automaton***, or ***state machine***.



Example: coin-operated turnstile



Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

State Transition Table: shows for each possible state, the transitions between them (based upon the inputs given to the machine) and the outputs resulting from each input

Simple Examples:

- Vending machines - dispense products when the proper combination of coins is deposited
- Elevators - sequence of stops is determined by the floors requested by riders
- Traffic lights - change sequence when cars are waiting
- Combination locks - require the input of combination numbers in proper order

Terminology

A ***state*** is a description of the status of a system that is waiting to execute a transition.
A ***transition*** is a set of actions to be executed when a condition is fulfilled or when an event is received.

Identical stimuli trigger different actions depending on the current state

For example: audio system
When using an audio system to listen to the radio (the system is in the *radio* state). Receiving a "next" stimulus results in moving to the next station.
When using the system is in the *CD* state, the "next" stimulus results in moving to the next track.

Often the following are also associated with a state:

- An ***entry action*** - the action performed when *entering* the state
- An ***exit action*** - the action performed when *exiting* the state

Representations

The most common representation is…

State transition table

Input \ Current state	State A	State B	State C
Input X	…	…	…
Input Y	…	State C	…
Input Z	…	…	…

See Gumball demo example.

Summary:

- The State Pattern allows an object to have many different behaviours that are based on its internal state
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class
- The context gets its behaviour by delegating to the current state object it is composed with
- Any changes made are localised since each state is encapsulated into a class.
- Has the same class diagram as the Strategy Pattern, but they have different intent
- State Pattern typically configures Context classes with a behaviour or algorithm.
- Allows a Context to change its behaviour as the state of the Context changes
- Transitions can be controlled by the State classes or by the Context classes
- State Pattern typically result in a greater number of classes in your design
- State classes may be shared among Context instances

Observer Pattern

Thursday, 4 July 2019 12:30 PM

The ***observer pattern*** is used to implement distributed ***event handling*** systems in "*event driven*" programming.

In the observer pattern, an object, called the ***subject*** (or ***observable*** or ***publisher***), maintains a list of its dependents, called ***observers*** (or ***subscribers***). The object notified the *observers* automatically of **any** state **changes**, usually by calling one of their methods.

Many programming languages support the observer pattern. Graphical User Interface libraries use the observer pattern extensively.

The observer pattern defines a ***one-to-many*** dependency between objects so that when one object (*subject*) changes state, all of its dependents (*observers*) are notified and updated automatically.

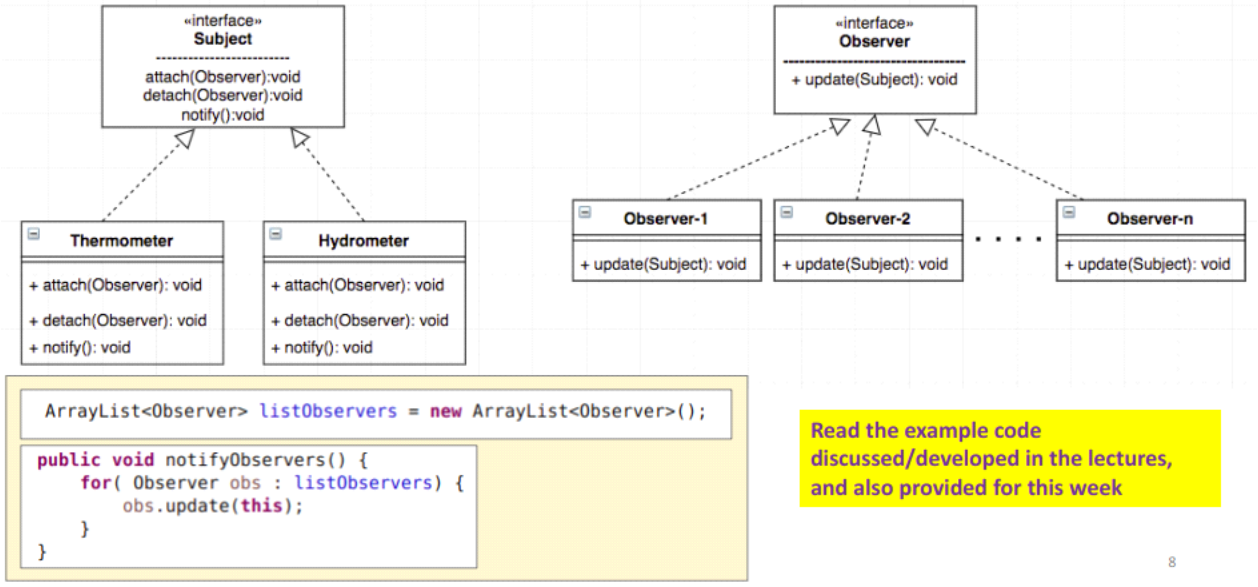
The aim should be to:

- Define a one-to-many dependency between objects **without** making the objects **tightly coupled**
- **Automatically** notify/update an **open-ended** number of *observers* when the *subject* changes state
- Be able to **dynamically** add and remove *observers*

Possible Solution:
Define *Subject* and *Observer* **interfaces**, such that when a subject changes state, all registered observers are notified and updated automatically.

The responsibility of, a ***subject*** is to maintain a list of observers and to notify them of state changes by calling their update() operation. The responsibility of, an ***observers*** is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state when they are notified.

This makes subject and observers **loosely** coupled.
Observers can be **added** and **removed** independently **at run-time**.
This notification-registration interaction is also known as **publish-subscribe**.



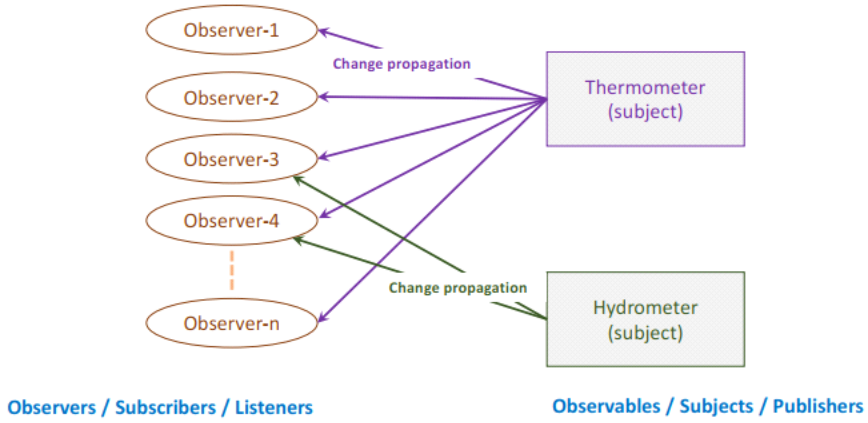
Java Observer and Observable: Deprecated
The following java library classes have been **deprecated** in **Java 9** because the model implemented was quite **limited**.

- java.util.Observer and
- java.util.Observable

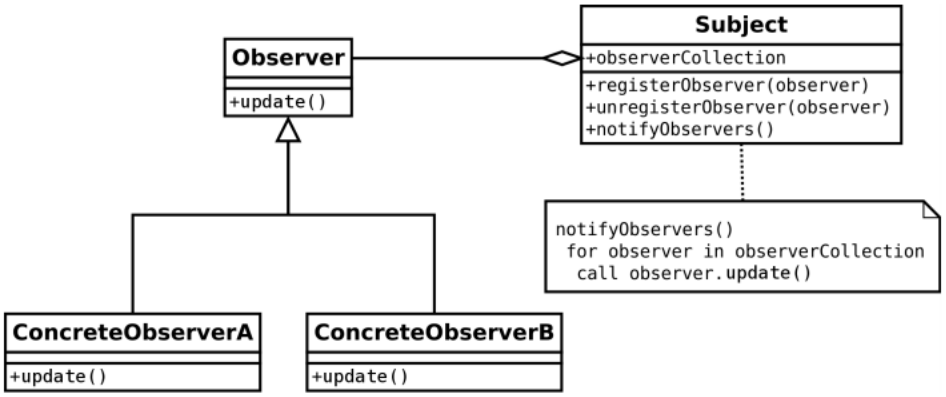
Limitations

- *Observable* is a **class**, not an interface!
- Observable **protects** crucial methods, the setChanged() method is protected.
- we can't call `setChanged()` unless we subclass *Observable*!
- we can't add on the *Observable* behavior to an existing class that already extends another superclass.
- there isn't an *Observable* interface, for a proper custom implementation

Multiple Observers and Subjects



Example UML diagram:



Passing data: Push or Pull

The ***Subject*** needs to pass (change) data while notifying a change to an ***Observer***. Two possible options:

- ***Push data***
  Subject passes the changed data to its observers, for example:
  `update(data1,data2,…);`
  All observers must implement the above method.
- ***Pull data***
  Subject passes reference to itself to its observers, and the observers need to get (pull) the required data from the subject:
  `update(this);`
  Subject needs to provide the required access methods for its observers. For example:
  `public double getTemperature() ;`

Summary:

- The Observer Pattern defines a one-to-many relationship between objects.
- Observables update Observers using a common interface
- Observers are loosely coupled - the Observables know nothing about them other than that they implement the Observer interface
- You can push or pull data from the observable when using the pattern (pull is considered more "correct")
- You don't depend on a specific order of notification for your Observers.
- Java has several implementation of the Observer Pattern (java.util.Observable which has been deprecated, so be careful of issues, or create your own Observable implementation)
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks

Composite Pattern

Thursday, 4 July 2019 12:31 PM

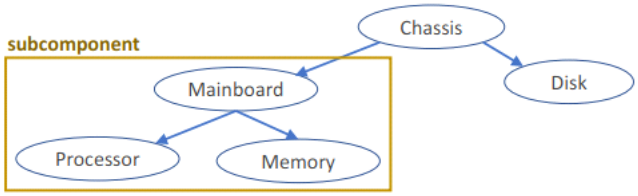
In OO programming, a ***composite*** is an object designed as a composition of one-or-more similar objects (exhibiting similar functionality).

The aim is to be able to manipulate a single instance of the object just as we would manipulate a group of them. For example, calculating the size of the file should be the *same as* a directory.

There is **no discrimination** between a single (leaf) and a composite (group) object. If we discriminate between a single object and a group of objects, our code will become more complex and more prone to error.

More examples:

- Calculating the total price of an individual part or a complete subcomponent (consisting of many parts) without having to treat each part and subcomponent differently



- A text document can be organised as a **part-whole hierarchy** consisting of
  - Characters, pictures, lines, pages (parts)
  - Lines, pages, document (wholes)

  We can display a line, page or the entire document (consisting of many pages) **uniformly** by using the same operation/method.

Possible Solution:
Define a unified ***component interface*** for both ***leaf*** objects and ***composite*** objects.
A *Composite* stores a collection of children components (either *Leaf* and/or *Composite* objects).
Clients can ignore the differences between compositions of objects, this greatly simplifies clients of complex hierarchies and makes them easier to implement, change, test and re-use.

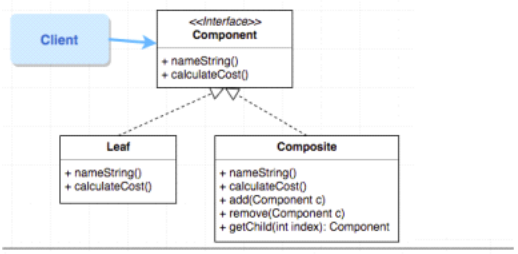
A ***leaf*** object performs operations directly on the object.
A ***composite*** object performs operations on its children, and if required, collects return values and derives the required answers.

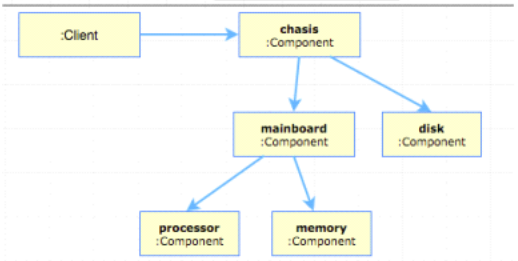
Tree structures are normally used to represent part-whole hierarchies. A multiway tree structure stores a collection of say *Components* at each node (**children** below), to store *Leaf* objects and *Composite* (subtree) objects.

Code Segment from the **`Composite`** class

```
ArrayList<Component>  children = new ArrayList<Component>();

@Override
public double calculateCost() {
    double answer = this.getCost();
    for(Component c : children) {
        answer += c.calculateCost();
    }

    return answer;
}
```

For more, read the example code provided for this week



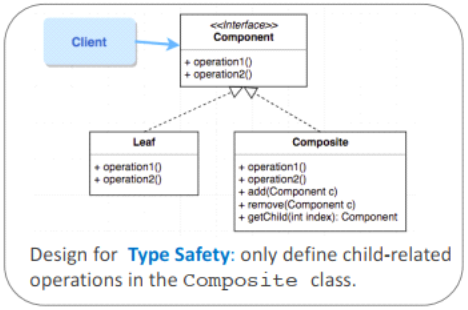


Read the example code discussed/developed in the lectures, and also provided for this week

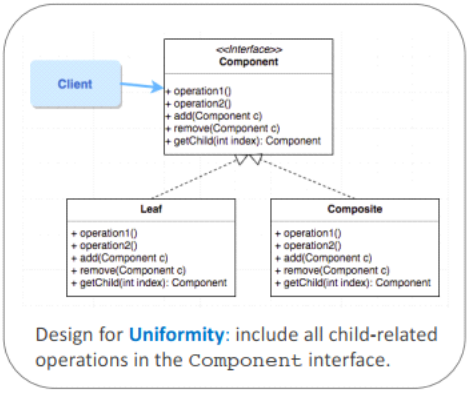
```
Component mainboard = new Composite("Mainboard", 100);
Component processor = new Leaf("Processor", 450);
Component memory    = new Leaf("Memory", 80);
mainboard.add(processor);
mainboard.add(memory);

Component chasis = new Composite("Chasis", 75);
chasis.add(mainboard);

Component disk = new Leaf("Disk", 50);
chasis.add(disk);


System.out.println("[0] "+ processor.nameString());
System.out.println("[0] "+ processor.calculateCost());


System.out.println("[1] "+ mainboard.nameString());
System.out.println("[1] "+ mainboard.calculateCost());


System.out.println("[2] "+ chasis.nameString());
System.out.println("[2] "+ chasis.calculateCost());
```

Implementation Issue: Uniformity vs Type Safety

There are two possible approaches to implement child-related operations (methods like add, remove, getChild etc)



See the next slide for more details.



Design for Uniformity:

- Include all child-related operations in the Component interface, this means the Leaf class needs to implement these methods with *do nothing* or *throw exception*
- A client can treat both *Leaf* and *Composite*

Summary:

- Composite pattern provides a structure to hold both individual objects and composites
- Allows clients to treat composites and individual objects uniformly
- A component is any object in a composite structure - may be a other composites or leaf nodes
- There are many design trade-offs in implementing the Composite pattern. You need to balance transparency and safety with your needs

Iterator Pattern

Saturday, 13 July 2019 10:06 PM

The Iterator design pattern intends to:
Provide a way to access the elements of an aggregate object (a container of data) sequentially without exposing its underlying representation

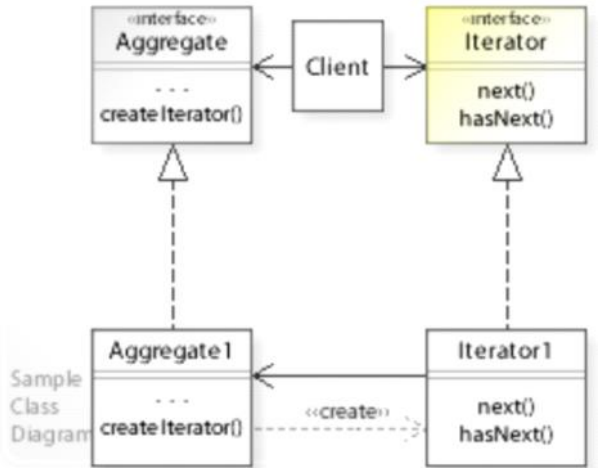
Exposing representation details of an aggregate breaks its encapsulation.

Problem: How can the elements of an aggregate object be accessed and traversed without exposing its underlying representation?
"But you probably don't want to bloat the List [Aggregate] interface with operations for different traversals, even if you could anticipate the ones you will need."

Possible Solution:
Encapsulate the access and traversal of an aggregate in a separate ***Iterator*** object.

Clients request an Iterator object from an aggregate (say by calling createIterator()) and use it to access and traverse the aggregate.
Define an interface for accessing and traversing the elements of an aggregate object (next(), hasNext()).
Define classes (Iterator1,...) that implement the Iterator interface.

An iterator is usually implemented as inner class of an aggregate class. This enables the iterator to access the internal data structures of the aggregate. New access and traversal operations can be added by defining new iterators. For example, traversing back-to-front: previous(), hasPrevious().



An aggregate provides an interface for creating an iterator (createIterator()). Clients can use different Iterator objects to access and traverse an aggregate object in different ways. Multiple traversals can be in progress on the same aggregate object (simultaneous traversals). However, need to consider concurrent usage issues!

The Java Collections Framework provides:

- A general purpose iterator: `next(), hasNext(), remove()`
- An extended listIterator: `next(), hasNext(), previoud(), remove()`,…

Decorator Pattern

Sunday, 21 July 2019 10:40 AM

The Decorator Pattern intends to
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

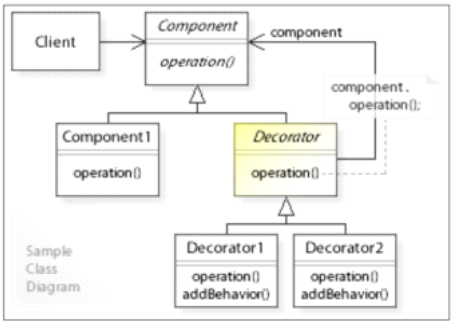
Decorator design patterns allow us to selectively add functionality to an object (not the class) at runtime based on the requirements. The original class is **not** changed, hence satisfying the Open-Closed Principle.

Inheritance extends behaviours as compile time, additional functionality is bound to all the instances of that class for their lifetime.

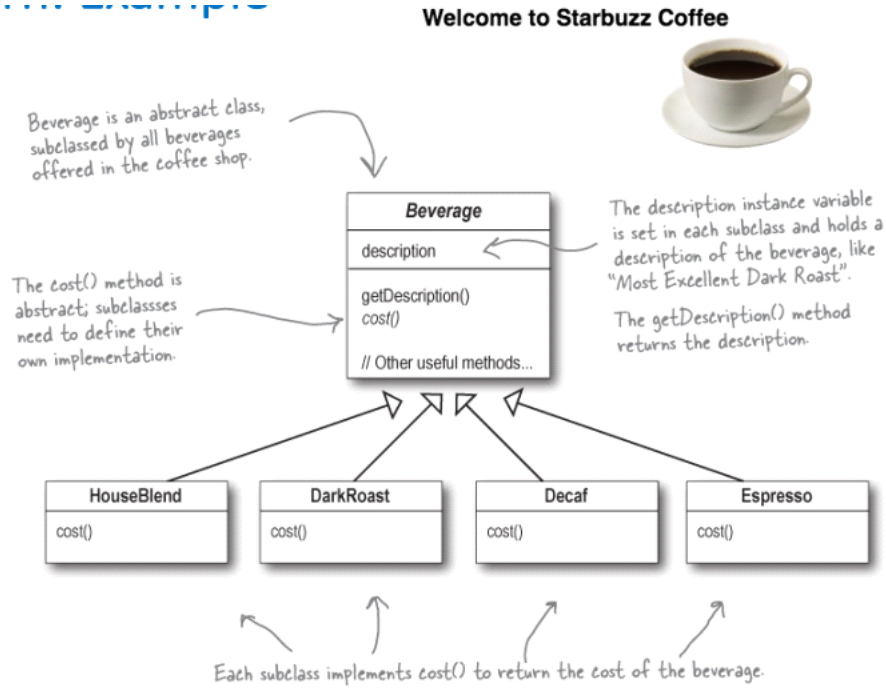
The ***decorator design pattern*** prefers a composition over an inheritance. It is a structural pattern, which provides a wrapper to the existing class. Objects can be decorated multiple times, in different order, due to the recursion involved with this design pattern. You do not need to implement all possible functionality in a single (complex) class

Structure of the Decorator Pattern

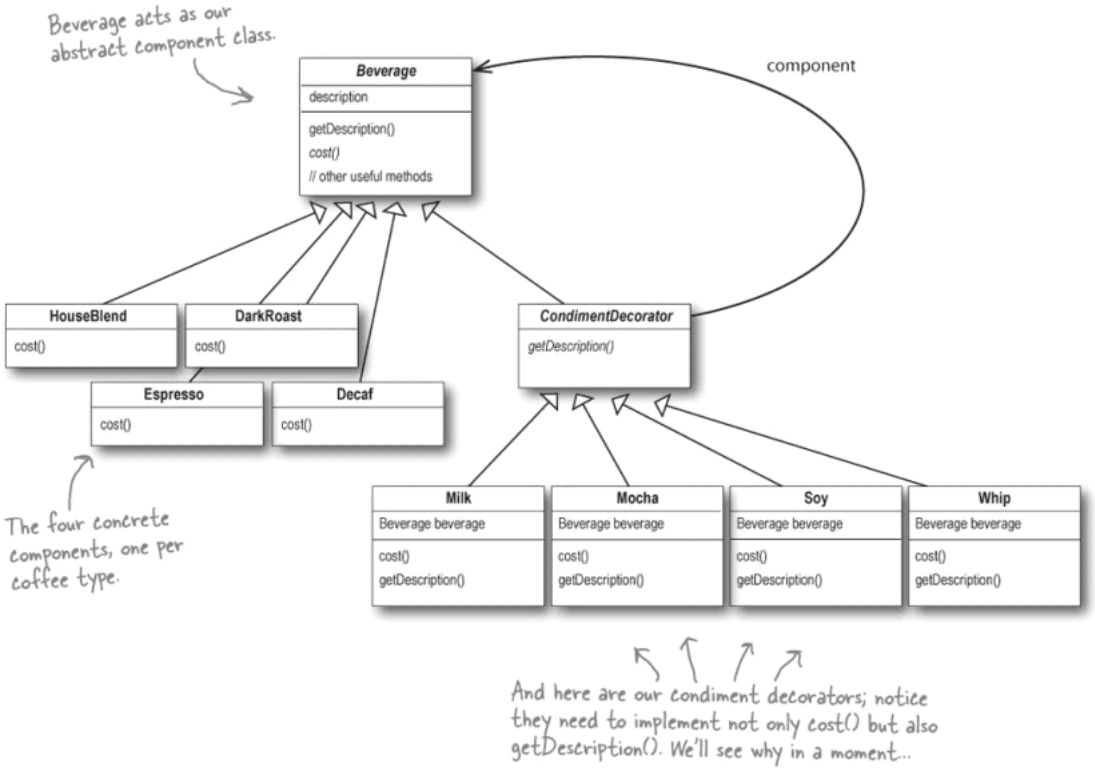
- *Client* : refers to the Component interface.
- *Component*: defines a common interface for *Component1* and *Decorator* objects
- *Component1* : defines objects that get decorated.
- *Decorator*: maintains a reference to a *Component* object, and forwards requests to this component object (*component.operation()*)
- *Decorator1, Decorator2*, ... : Implement additional functionality (*addBehavior()*) to be performed before and/or after forwarding a request.



Given that the decorator has the same supertype as the object it decorates, we can pass around a **decorated** object in place of the original (wrapped) object.
The decorator adds its own behaviour either before and/or after delegating to the object it decorates to do the rest of the job.

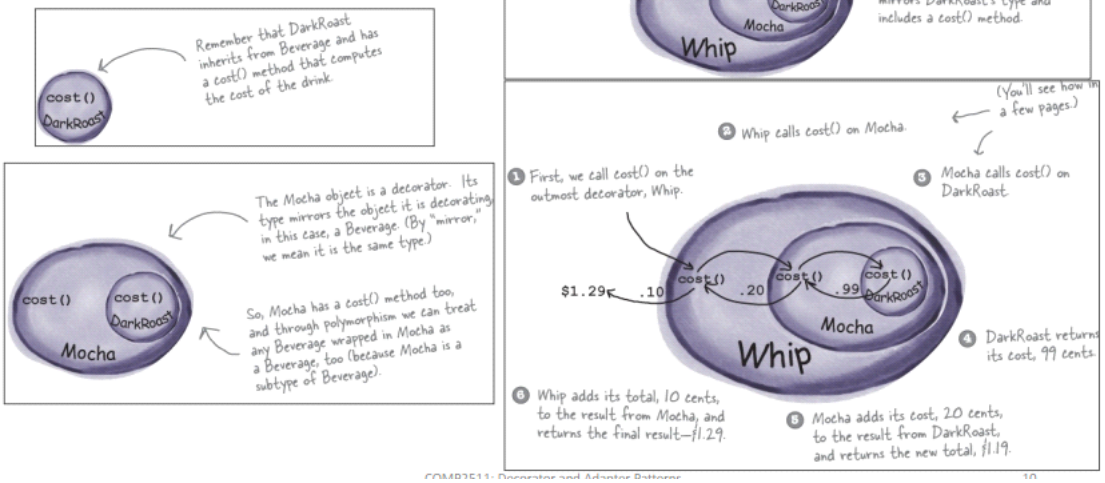


If each cost method computes the cost of the coffee along with the other condiments in the order, but this would make too many combinations. So our solution is to use a decorator:



Decorator Pattern: Example

Constructing a drink order with Decorators



Decorator Pattern: Code

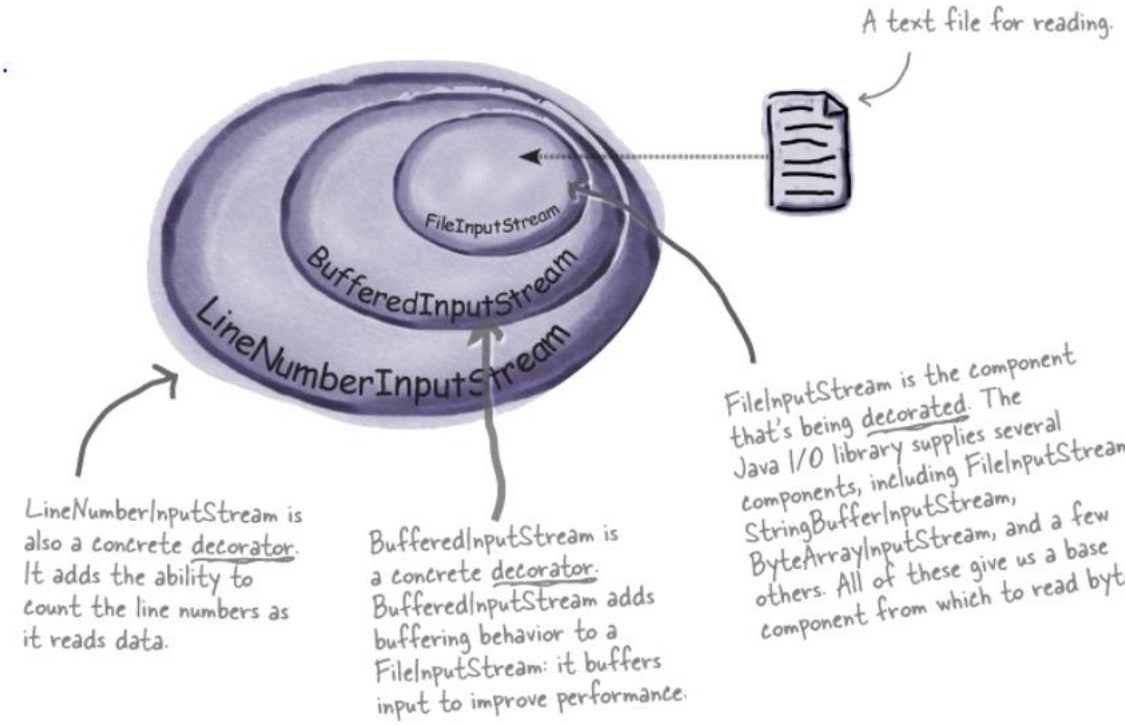
```
Beverage beverage = new Espresso();
System.out.println(beverage.getDescription()
        + " $" + beverage.cost());
System.out.println("-------------------------");
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
        + " $" + beverage2.cost());

System.out.println("------------------------- ");

Beverage beverage3 = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
System.out.println(beverage3.getDescription()
        + " $" + beverage3.cost());
System.out.println("-------------------------
```

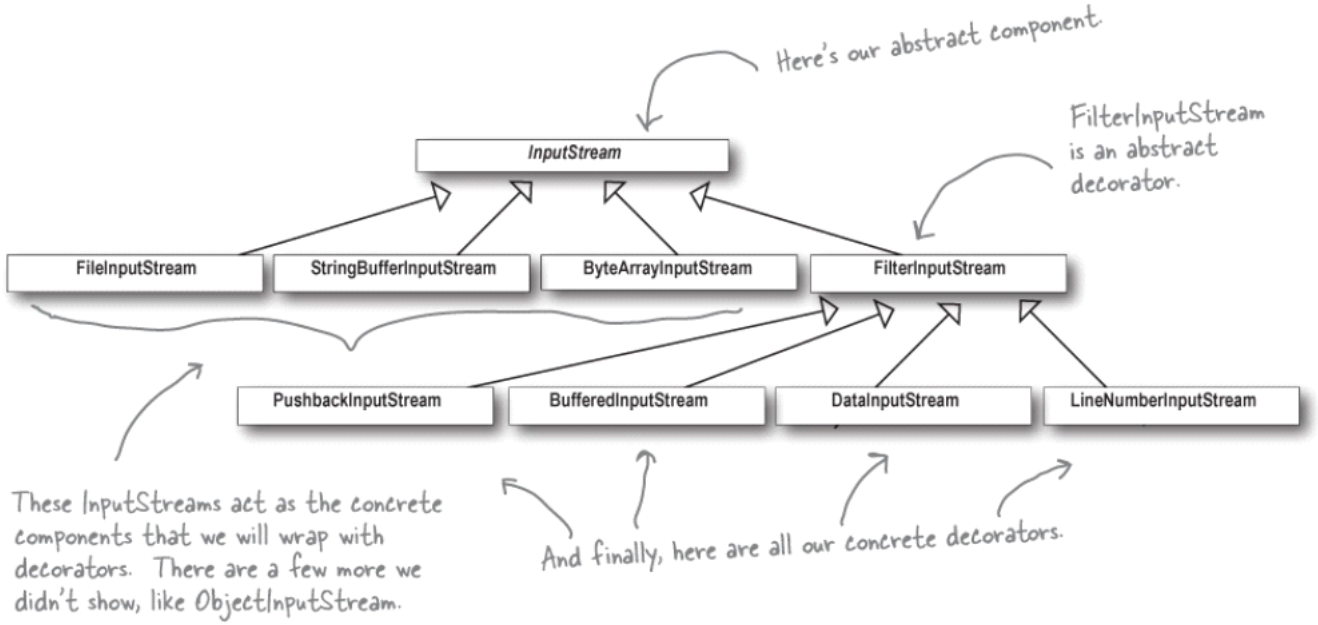
```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Whipe: beverage.cost() is: " + beverage_cost);
    System.out.println("  - adding One Whip cost of 0.10c ");
    System.out.println("  - new cost is: " + (0.10 + beverage_cost ) );

    return 0.10 + beverage_cost ;
}
```

Read the example code discussed/developed in the lectures, and also provided for this week

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Mocha: beverage.cost() is: " + beverage_cost );
    System.out.println("  - adding One Mocha cost of 0.20c ");
    System.out.println("  - new cost is: " + (0.20 + beverage_cost ) );

    return 0.20 + beverage_cost ;
}
```

Example: Java I/O






```
InputStream f1 = new FileInputStream(filename);
InputStream b1 = new BufferedInputStream(f1);
InputStream lCase1 = new LowerCaseInputStream(b1);
InputStream rot13 = new Rot13(b1);

while ((c = rot13.read()) >= 0) {
    System.out.print((char) c);
}
```

If you want to change a *layer* you need to take off all layers after it then put them back on

Adapter Pattern

Sunday, 21 July 2019 10:40 AM

The Adapter Pattern intends to:
Convert the interface of a class into another interface a client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

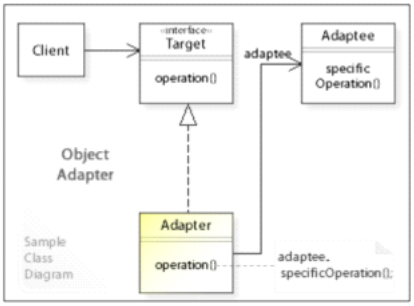
The adapter pattern allow the interface of an existing class to be used as another interface suitable for a client class. The adapter pattern is often used to make existing classes (APIs) work with a client class without modifying their source code. The adapter class maps/joins functionality of two different types/interfaces.

The adapter pattern offers a wrapper around an existing useful class, such that a client class can use functionality of the existing class.

The adapter pattern **does not provide additional functionality**

Structure

The adapter pattern contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object.



Examples:

```
interface LightningPhone {
    void recharge();
    void useLightning();
}

interface MicroUsbPhone {
    void recharge();
    void useMicroUsb();
}
```

```
class Iphone implements LightningPhone {
    private boolean connector;

    @Override
    public void useLightning() {
        connector = true;
        System.out.println("Lightning connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}
```

```
class Android implements MicroUsbPhone {
    private boolean connector;

    @Override
    public void useMicroUsb() {
        connector = true;
        System.out.println("MicroUsb connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect MicroUsb first");
        }
    }
}
```

```java
public class AdapterDemo {
    static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
        phone.useMicroUsb();
        phone.recharge();
    }

    static void rechargeLightningPhone(LightningPhone phone) {
        phone.useLightning();
        phone.recharge();
    }

    public static void main(String[] args) {
        Android android = new Android();
        Iphone iPhone = new Iphone();

        System.out.println("Recharging android with MicroUsb");
        rechargeMicroUsbPhone(android);

        System.out.println("Recharging iPhone with Lightning");
        rechargeLightningPhone(iPhone);

        System.out.println("Recharging iPhone with MicroUsb");
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iPhone));
    }
}
```

```java
class LightningToMicroUsbAdapter implements MicroUsbPhone {
    private final LightningPhone lightningPhone;

    public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}
```

Output

```
Recharging android with MicroUsb
MicroUsb connected
Recharge started
Recharge finished
Recharging iPhone with Lightning
Lightning connected
Recharge started
Recharge finished
Recharging iPhone with MicroUsb
MicroUsb connected
Lightning connected
Recharge started
Recharge finished
```

Template Pattern

Saturday, 27 July 2019 11:24 PM

The Template Pattern intends to:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
*Template Method lets subclasses redefine certain steps of an algorithm **without** changing the algorithm's structure.*

A template pattern defined the skeleton (structure) behaviour by implementing the invariant parts. It calls ***primitive operations***, that could be implemented by subclasses OR has default implementations in an abstract super class. Subclasses can redefine only certain parts of a behaviour **without changing** other parts or the **structure** of the behaviour.

Subclasses do not control the behaviour of a parent class. A parent class calls the operations of a subclass and not the other way around.

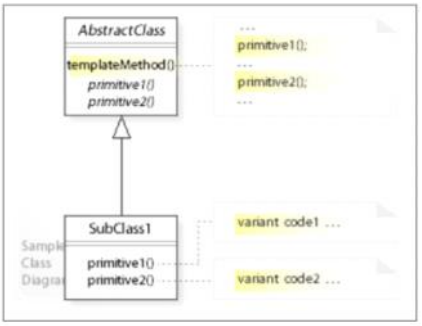
There is an **inversion of control**

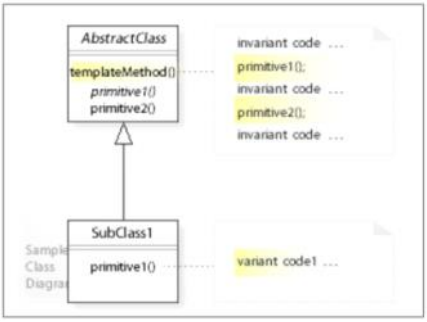
- When using a **library** (reusable classes) since we call the code we want to reuse
- When using a **framework** (like the Template Pattern) since we write subclasses and implement invariant code the framework calls.

The Template Pattern implement the common (invariant) parts of a behaviour once and leave it up to the subclasses to implement the behaviour that can vary. Invariant behaviour is localised as it is in one class.

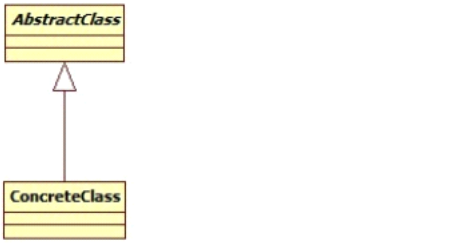
Structure

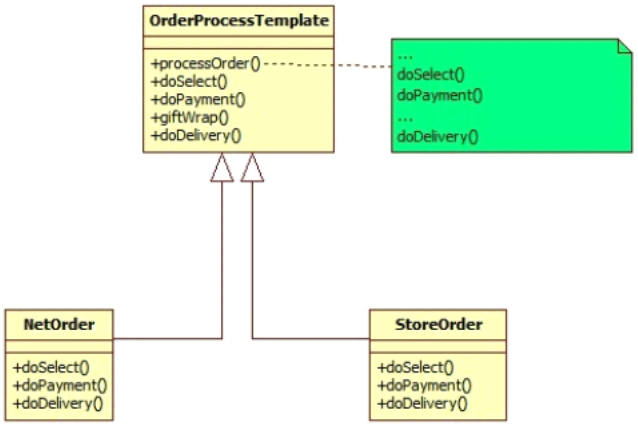
- Abstract class defines a templateMethod() to implement an invariant structure (behaviour)
- templateMethod() calls methods defined in the abstract class (abstract or concrete) - like primitive1, primitive2, etc.
- Default behaviour can be implemented in the abstract class by offering concrete methods
- Importantly, sub classes can implement primitive methods for variant behaviour
- **Primitive operations** are operations that have default implementations or must be implemented by sub classes.
- Final operations are concrete operations that cannot be overridden by sub classes.
- **Hook operations** are concrete operations that do nothing by default and can be redefined by subclasses if necessary. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

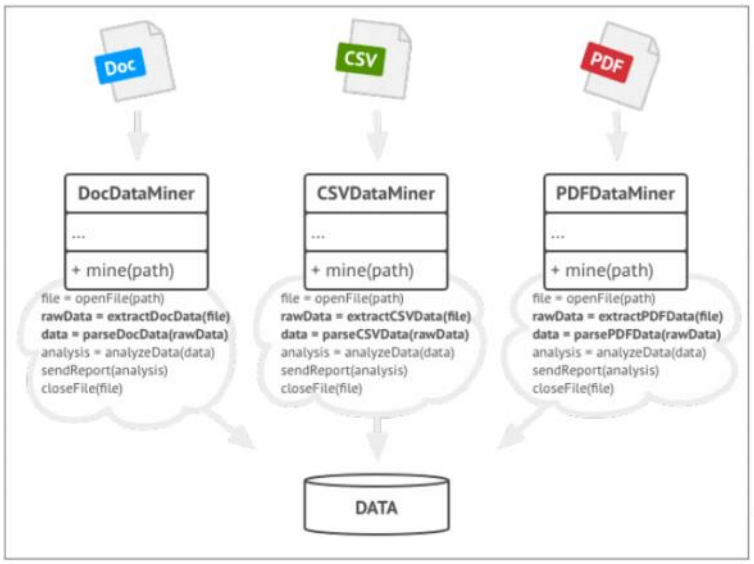


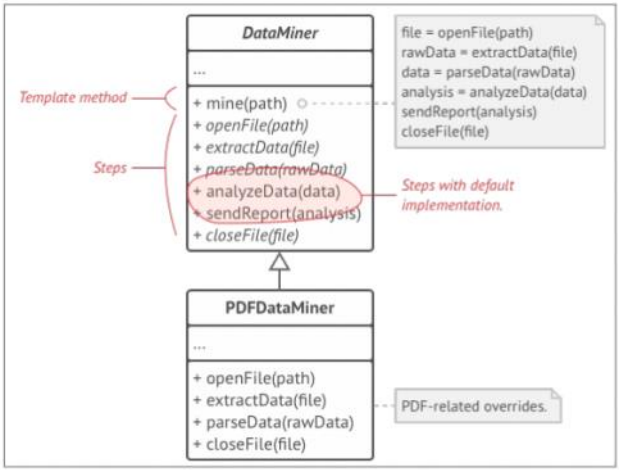


Examples:


AbstractClass
ConcreteClass


OrderProcessTemplate
+processOrder()
+doSelect()
+doPayment()
+giftWrap()
+doDelivery()
...
doSelect()
doPayment()
...
doDelivery()
NetOrder
+doSelect()
+doPayment()
+doDelivery()
StoreOrder
+doSelect()
+doPayment()
+doDelivery()


Doc
CSV
PDF
DocDataMiner
+ mine(path)
CSVDataMiner
+ mine(path)
PDFDataMiner
+ mine(path)
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
DATA


DataMiner
Template method
+ mine(path)
+ openFile(path)
+ extractData(file)
Steps
+ parseData(rawData)
+ analyzeData(data)
+ sendReport(analysis)
+ closeFile(file)
file = openFile(path)
rawData = extractData(file)
data = parseData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
Steps with default implementation.
PDFDataMiner
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ closeFile(file)
PDF-related overrides.

Template Pattern: Example

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup, and add sugar and milk.

From the Head First Design Book

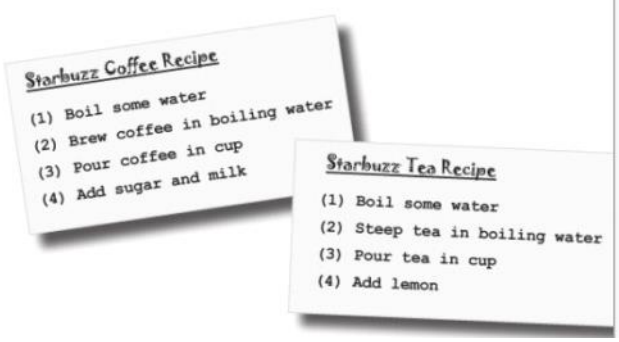

```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

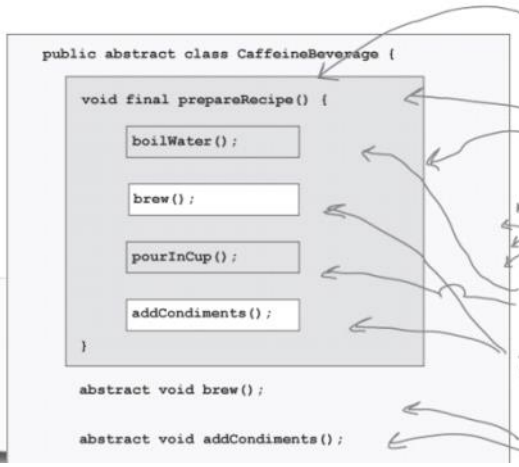
This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

Template Pattern: Example





prepareRecipe() is our template method. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

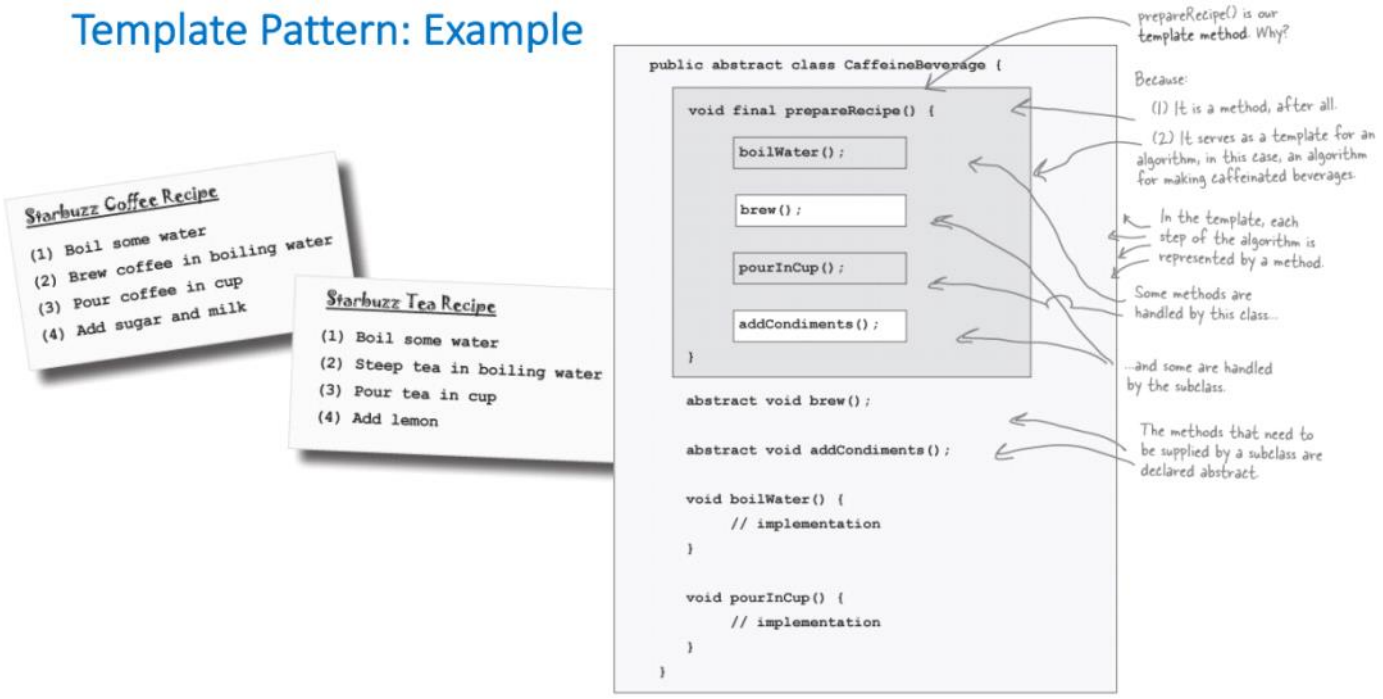
In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

Template Pattern: Example



Template Pattern: Example (hook)

```
public abstract class CaffeineBeverageWithHook {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Template Pattern: Example (hook)

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

ɔm the Head First Design Book

Template vs Strategy Pattern

- Template Method works at the class level, so it's **static**.
- Strategy works on the object level, letting you switch behaviors at **runtime**.
- Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses.
- Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior at runtime.

Visitor Pattern

Tuesday, 20 August 2019 5:30 PM

Visitor Pattern is a behavioural pattern that ***adds new operations/behaviours*** to existing objects without modifying them. The visitor pattern is a way of separating an algorithm for an object structure on which it operates.

It is one way of following the open/closed principle.

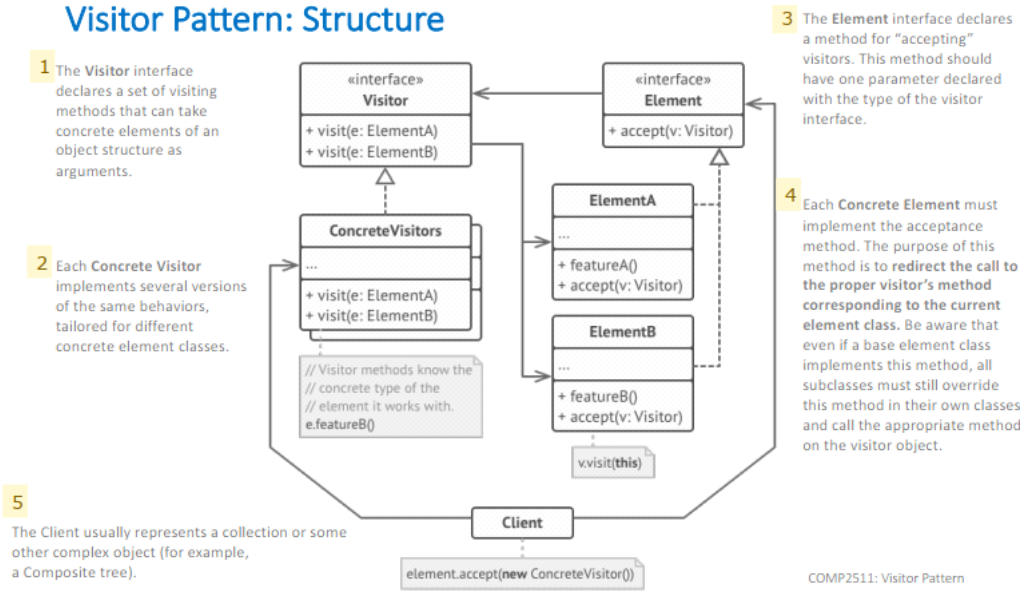
A visitor class implements all of the appropriate specializations of the virtual operation/method. The visitor takes an instance reference as input, and implements the goal (additional behavior) to the instance reference.

Visitor pattern can be added to public APIs, allowing its clients to perform operations on a class without having to modify the source

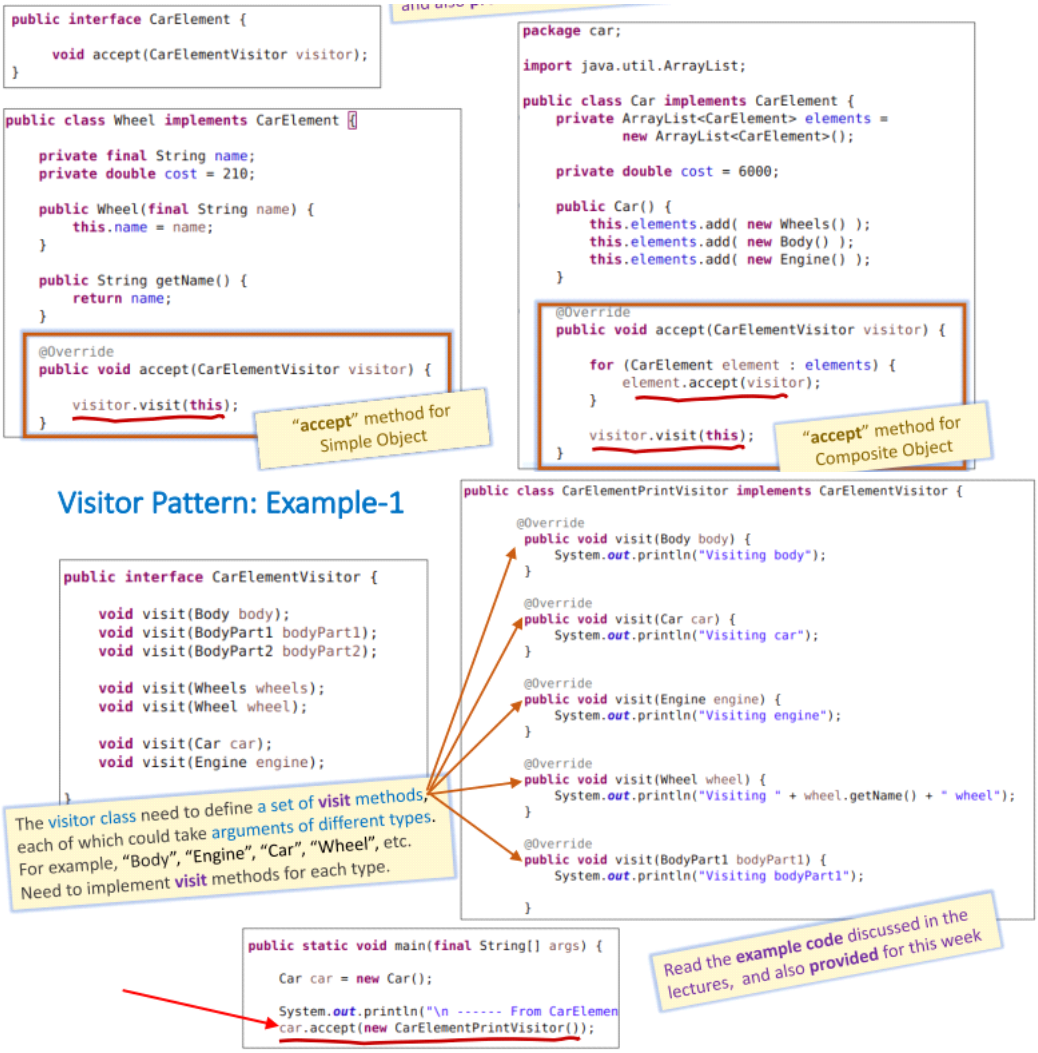
Example:
A geographic information structured as one colossal graph. Each node of the graph may represent a city, an industry, a sightseeing area, etc. Each node type is represented by its own class, while each specific node is an object.
Your task is to export the graph into XML format for each node.

Solution:
Place the new behaviour into a separate class called visitor, instead of trying to integrate it into existing classes. The original object that had to perform the behaviour is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object .
The visitor class needs to define a set of methods for each type of visitor passed to it.
The visitor pattern uses a technique called "**Double Dispatch**" to execute a suitable method on a given object (of different types).
An object "**accept**s" a *visitor* and tells it what visiting method should be executed. One additional method allows us to add further behaviours without further altering the code.

Structure



Example: Car Components



Applicability and Limitations

Moving operations into a visitor is beneficial when

- Many unrelated operations on an object are required
- The classes that make up the object structure are known and not expected to change
- New operations need to be added frequently
- An algorithm involves several classes of the object strucutre, but it is desired to manage it in one single location
- An algorithm needs to work across several independent class hierarchies.

A limitation would be that extensions to the class hierarchy is more difficult as a new class typically requires a new visit method to be added to each visitor

Creational Patterns

Saturday, 3 August 2019 2:49 PM

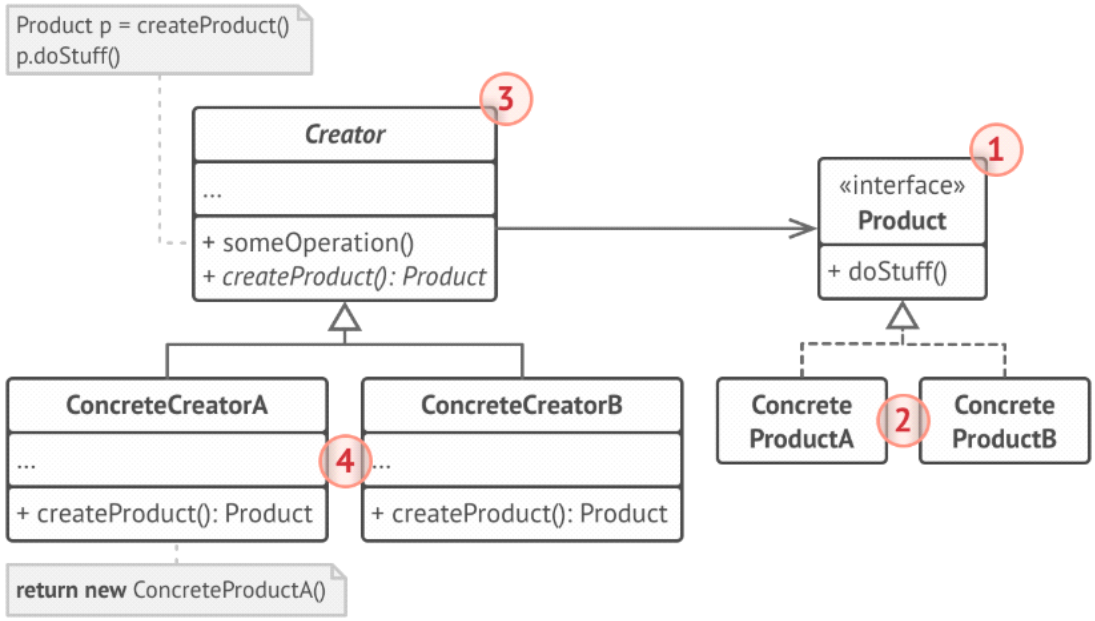
Factory Method

Factory Method is a creational design pattern that **provides an interface for creating objects** in a superclass, but allows superclasses to alter the type of objects that will be created.

The Factory Method pattern suggests that you replace direct object construction calls (using the **new** operator) with calls to a special *factory* method. The objects are still being created via **new** but are just called within the factory method.

The code that uses the factory method (often called the *client* code) doesn't see a difference between the actual products returned by various classes. The client treats all the products as an abstract class. The client knows that all the objects have the same method that can be called, but how it works is not important to the client

Structure



Notes:

- Use the factory method when you don't know beforehand the exact types and dependencies of the objects your code should work with
- Factory method **separates product construction from code that uses the product** making it easier to extend the product construction independently from the rest of the code
- Factory method helps you save system resources by reusing existing objects instead of rebuilding them each time

Pros and Cons

Pros	*Cons*
• Avoid tight coupling between the creator and the concrete products. • ***Single Responsibility Principle*** - can move the product creation code into one place in the program, making the code easier to support. • ***Open/Closed Principle*** - can introduce new types of products into the program without breaking existing client code.	• Complicated code - you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Abstract Factory

Abstract Factory is a creational design pattern that lets you **produce families of related objects** without specifying their concrete classes.

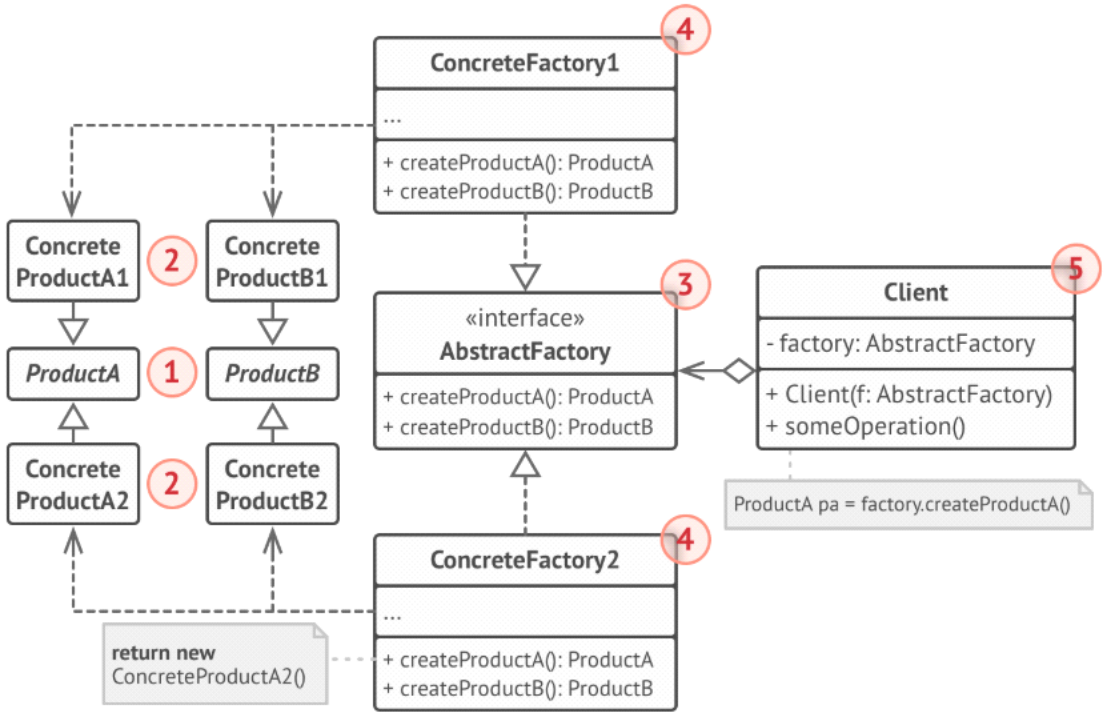
The Abstract Factory patterns suggests you explicitly declare interfaces for each distinct product of the product family (e.g. chair, sofa or coffee table). Then you can make all variants of products follow these interfaces. For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.

Then you declare the ***Abstract Factory*** - an interface with a list of creation methods for all products that are part of the product family (e.g. createChair, createSofa and createCoffeeTable). These methods must return **abstract** product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable

For each variant of a product family, we create a separate factory class based on the AbstractFactory interface. A factory is a class that return products of a particular kind. For example, the ModernFurnitureFactory can only create ModernChair, ModernSofa and ModernCoffeeTable objects.

The client doesn't care about the concrete class of the factory, it just asks for it to do things without caring about how it does things.

Structure



Notes:

- Use when your code needs to work with various families of related products, but don't want it to depend on concrete classes of those products
- Abstract Factory provides you with an interface for creating objects from each class of a the product family

Pros and Cons

Pros	*Cons*
• Always certain that products you're getting from a factory are compatible with each other. • Avoid tight coupling between concrete products and client code. • ***Single Responsibility Principle*** - can extract the product creation code into one place, making the code easier to support. • ***Open/Closed Principle*** - can introduce new variants	• Complicated code - a lot of new interfaces and classes are going to be introduced in this pattern

of products without breaking existing client code.

Builder Pattern

Builder is a creational design pattern that lets you construct complex objects **step by step**. The pattern allows you to produce different types and representations of an object using the same construction code.

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects.

Let our example be a house:
The Builder pattern suggests that you extract the object construction code out of its own class and make it move to separate objects called ***builders***.
The pattern organises object construction into a set of steps (buildWalls, buildDoor etc). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

Some of the construction steps might require different implementations when you need to build various representations of the product. e.g a house made of different materials.
In this case, you can create several different builder classes that implement the same set of building steps but in a different manner. Then you can use these builders in the construction process to produce different kinds of objects

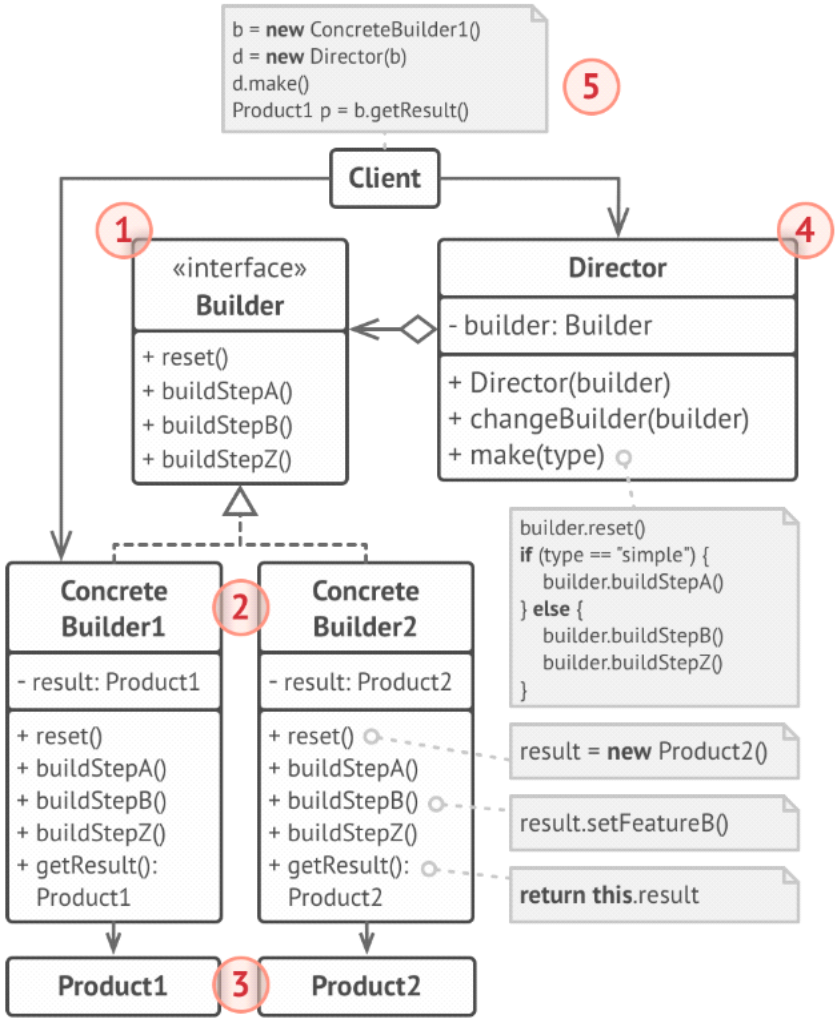
Director

You can extract a series of calls to the builder steps you use to construct a product into a separate class called a ***director***. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

Having a director is not necessary but may be useful when you need to reuse construction routines.

Directors also completely hide the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director and get the end result

Structure



Notes:

- Use the Builder pattern when you want to get rid of a ***telescopic constructor*** (a constructor that has many constructors each taking different amounts of arguments)
- Use the builder pattern when you want your code to be able to create different representations of some product

Pros	*Cons*
• Can construct objects step-by-step, defer construction steps or run steps recursively • Reuse the same construction code when building various representations of products • ***Single Responsibility Principle*** - can isolate complex construction code from business logic of the product	• Increased complexity - requires creating multiple new classes

Singleton Pattern

Singleton is a creational design pattern that lets you ensure that a class has **only one instance**, while providing a global access point to this instance.

The Singleton pattern solves two problems at the same time, while violating the Single Responsibility Principle.

1. **Ensure that a class has just a single instance**: Imagine you created an object but after a while you decide to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

   Note that this behaviour is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.
2. **Provide a global access point to that instance**: lets you access some object from anywhere in the program.

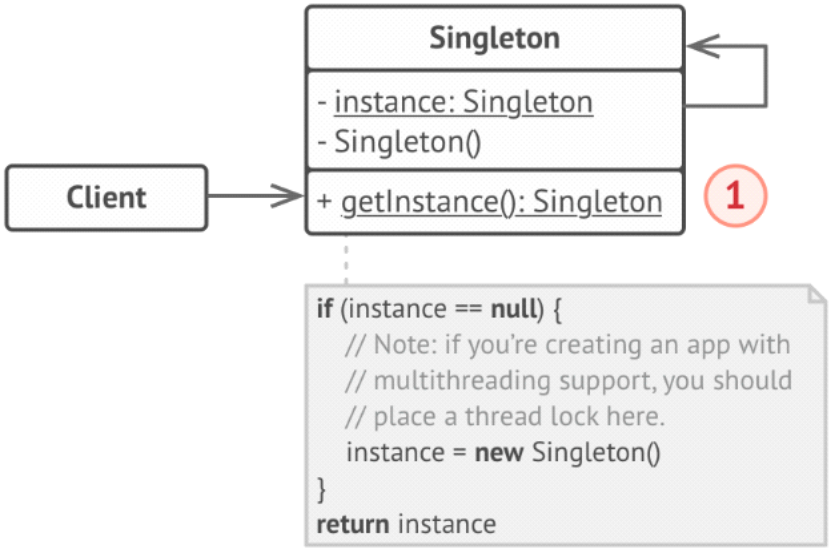
However it also protects that instance from being overwritten by other code.

All implementation of the Singleton have these two steps in common:

1. Make the default constructor private to prevent other objects from using the **new** operator within the Singleton class
2. Create a static creation method that acts as the constructor. This method calls the private constructor to create an object and saves it in a static field. All following calls to this method returns the cached object.

If your code has access to the Singleton class, then it is able to call the Singleton's static method, so whenever the method is called, the same object is always returned.

Structure



Notes:

- Use the Singleton pattern when a class in your program should have just a single instance available to all clients (e.g a single database object shared by different parts of the program)
- Use the Singleton pattern when you need stricter control over global variables

Pros and Cons

Pros	*Cons*
• Certain that a class has a single instance • Gain a global access point to that instance • Singleton object is initialised only when requested for the first time	• Violates Single responsibility Principle - it solves two problems at the same time • Can mask bad design (e.g. when program components know too much about each other) • Requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object multiple times • May be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects.

Design by Contract

Tuesday, 9 July 2019 4:21 PM

Defensive Programming vs Design by Contract

Defensive programming tries to address unforeseen circumstances, in order to ensure the continuing functionality of the software element. For example, it makes the software behave in a predictable manner despite unexpected inputs or user actions.

It is often used where ***high availability,*** safety or security is needed. Defensive programming results in ***redundant checks*** (since both client and supplier may perform such checks) leading to more ***complex software*** for maintenance. If an error occurs, it is difficult to locate the, since there is ***no clear demarcation*** of responsibilities. Defensive programming may safeguard against errors that will never be encountered, thus incurring run-time and maintenance costs.

Design by contract is where, at the time of design, responsibilities are clearly assigned to different software elements. These responsibilities are clearly documented and enforced during development using unit testing and/or language support.

By using clear demarcation of responsibilities, we prevent redundant checks, resulting in simpler code and easier maintenance. However, it required conditions are not satisfied, the program crashes. This may not be suitable for high availability application.

Design by Contract

Bertrand Meyer coined the term for his design of the Eiffel programming language (in 1986).

Design by Contract (DbC) has its roots in work on formal specification, formal verification and Hoare logic. In business, when two parties (supplier and client) *interact* with each other, often they write and sign contracts to clarify the obligations and expectations. For example,

	Obligations	Benefits
Client	*(Must ensure precondition)*	*(May benefit from post-condition)*
	Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	Reach Chicago.
Supplier	*(Must ensure post-condition)*	*(May assume pre-condition)*
	Bring customer to Chicago.	No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

Every software element should define a ***specification*** (or a ***contract***) that governs its interaction with the rest of the software components. A ***contract*** should address the following three questions:

- ***Pre-condition*** - What is expected? If the precondition is true, it can avoid handling cases outside of the precondition. For example, expected argument value (mark>=0) and (marks<=100).
- ***Post-condition*** - What is guaranteed? Return value(s) is guaranteed, provided the precondition is true. For example: correct return value representing a grade.
- ***Invariant*** - What remains unchanged? Some values must satisfy constraints, before and after the execution (say of the method). For example: a value of mark remains between zero and 100.

A contract should be ***declarative*** and must not include implementation details. It should also be as precise, formal and verifiable as possible.

Benefits of Design by Contract

- Do not need to do error checking for conditions that not satisfy the preconditions!
- Prevents redundant validation tasks.
- Given the preconditions are satisfied, clients can expect the specified post-conditions.
- Responsibilities are clearly assigned, this helps in locating errors and resulting in easier code maintenance.
- Helps in cleaner and faster development.

Implementation Issues

Some programming languages (like Eiffel) offer native support for DbC. Java does not have native support for DbC, although there are various libraries to support DbC. In the absence of a native language support, unit testing is used to test the contracts (preconditions, post-conditions and invariants).
Often preconditions, post-conditions and invariants are included in the documentation.

Example of Design by Contract in Java: State them in javadoc comments and provide unit test (to be covered)

```
/**
@param value to calculate square root
@returns sqrt - square root of the value
@pre  value >= 0
@post value = sqrt * sqrt
*/
public double squareRoot ( double value );
```

```
/**
 * @invarient age >= 0
 */

public class Student {
```

```
/**
@param amount to be deposited into the account
@pre  amount > 0
@post  balance = old balance + amount
*/
public void deposit( double amount);
```

Pre-Conditions

A pre-condition is a condition or predicate that must always be true just **prior** to the execution of some section of code. If a precondition is violated, the effect of the section of code becomes undefined and thus may or may not carry out its intended work.

Security problems can arise due to incorrect pre-conditions. Often, preconditions are included in the documentation of the affected section of code. Preconditions are sometimes tested using guards or assertions within the code itself, and some languages have specific syntactic constructions for testing .

In Design by Contract, a software element can assume that preconditions are satisfied, resulting in removal of redundant error checking code.

```
/**
 * @pre (mark >=0) and (mark<=100)
 * @param mark
 */
public void printGradeDbC(double mark) {

    if(mark < 50 ) {
        System.out.println("Fail");
    }
    else {
        System.out.println("Pass");
    }
}
```

Incorrect behaviour if *mark* is outside the expected range

```
/**
 * Get Student at i'th position
 * @pre i < number_of_students
 * @param i - student's position
 * @return student at i'th position
 */
public Student getStudentDbC(int i) {

    return students.get(i);

}
```

Throws runtime exception if (*i >= number_of_students*)

Design by Contract
No additional error checking for pre-conditions

```
/**
 * @pre (mark >=0) and (mark<=100)
 * @param mark
 */
public void printGradeDefensive(double mark) {
    if( (mark < 0) || (mark > 100) ){
        System.out.println("Error");
    }

    if(mark < 50 ) {
        System.out.println("Fail");
    }
    else {
        System.out.println("Pass");
    }
}
```

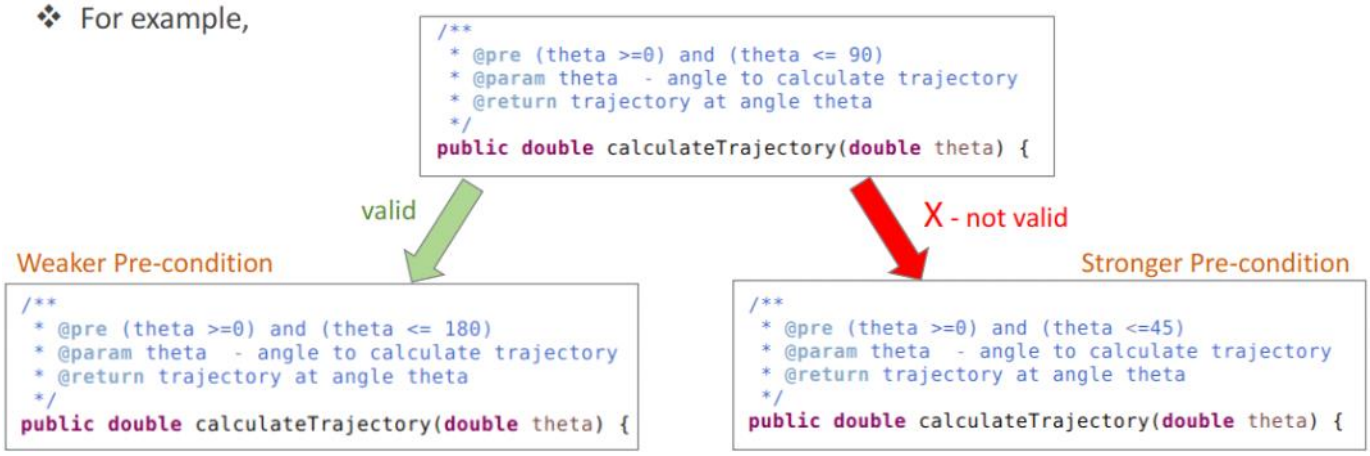
Defensive Programming:
Additional error checking for pre-conditions

Pre-Conditions in Inheritance

An implementation or redefinition (method overriding) of an inherited method must comply with the inherited contract for the method. Pre-conditions may be ***weakened*** (relaxed) in a subclass, but it must comply with the

inherited contract.
An implementation or redefinition may ***lessen*** the obligation of the client, but it cannot increase it

❖ For example,



Post-Conditions

A post-condition is a condition or predicate that must *always be true* just **after** the execution of some section of code.
The post-condition for any routine is a declaration of the properties which are guaranteed upon completion of the routine's execution. Often, preconditions are included in the documentation of the affected section of code. Post-conditions are sometimes tested using guards or assertions within the code itself, and some languages have specific syntactic constructions for testing .

In Design by Contract, the properties declared by the post-condition(s) are assured, provided the software element is called in a state in which its pre-condition(s) were true.

```
/**
@param value to calculate square root
@returns sqrt - square root of the value
@pre  value >= 0
@post value = sqrt * sqrt
*/
public double squareRoot ( double value );
```

Post-Conditions in Inheritance

An implementation or redefinition (method overriding) of an inherited method must comply with the inherited contract for the method.

Post-conditions may be ***strengthened*** (more restricted) in a subclass, but it must comply with the inherited contract. An implementation or redefinition (overridden method) may increase the benefits it provides to the client, but not decrease it.
For example, the original contract requires returning a set. The redefinition (overridden method) returns sorted set, offering *more* benefit to a client.

Invariants

The class invariant constrains the state (i.e. values of certain variables) stored in the object.

Class invariants are established during construction and constantly maintained between calls to public methods. Methods of the class must make sure that the class invariants are satisfied/preserved. Code within a method may break invariants as long as the invariants are restored before a public method ends.

Class invariants help programmers to rely on a valid state, avoiding risk of inaccurate / invalid data. It also helps in locating errors during testing.

Class Invariants in Inheritance

Class invariants are inherited, that means, "*the invariants of all the parents of a class apply to the class itself.*" !
A subclass can access implementation data of the parents, however must always satisfy the invariants of all the parents – preventing invalid states!

Subclass	Pre-condition	Post-Condition	Invariant
	Weaken	Strengthen	The same

Generic Types and Collections

Thursday, 11 July 2019 9:53 AM

Resources:
https://docs.oracle.com/javase/tutorial/java/generics/index.html
https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Generics in Java

Generics enable types (classes and interfaces) to be parameters when defining:

- classes,
- interfaces and
- methods.

Benefits

- ***Removes casting*** and offers stronger type checks at compile time.
- Allows implementations of generic algorithms, that work on collections of different types, can be customized, and are type safe.
- Adds stability to your code by making more of your bugs detectable at compile time.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Without Generics

```
List<String> listG = new ArrayList<String>();
listG.add("hello");
String sg = listG.get(0);    // no cast
```

With Generics

A generic type is a generic class or interface that is parameterized over types.
A generic class is defined with the following format:

```
class name< T1, T2, ..., Tn > { /* ... */ }
```

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

For example,

```
Box<Integer> integerBox = new Box<Integer>();
```

OR

```
Box<Integer> integerBox = new Box<>();
```

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Multiple Type Parameter

A generic class can have multiple type parameters.
For example, the generic OrderedPair class, which implements the generic Pair interface

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Usage examples:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");
... ...
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");
... ...
OrderedPair<String, Box<Integer>> p=new OrderedPair<>("primes", new Box<Integer>
(...));
```

Generic Methods

Generic methods are methods that introduce their own type parameters.

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown above.
Generally, this can be left out and the compiler will **infer** the **type** that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

Bounded Type Parameters

There may be times when you want to **restrict the types** that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

```
public <U extends Number> void inspect(U u){
    System.out.println("U: " + u.getClass().getName());
}
```

```
public class NaturalNumber<T extends Integer> {
```

Multiple Bounds

A type parameter can have multiple bounds:

```
< T extends B1 & B2 & B3 >
```

A type variable with multiple bounds is a subtype of all the types listed in the bound.
Note that B1, B2, B3, etc. in the above refer to interfaces or a class. There can be at most one class (single inheritance), and the rest (or all) will be interfaces.
If one of the bounds is a class, it must be specified first.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler erro X - invalid
            ++count;
    return count;
}
```

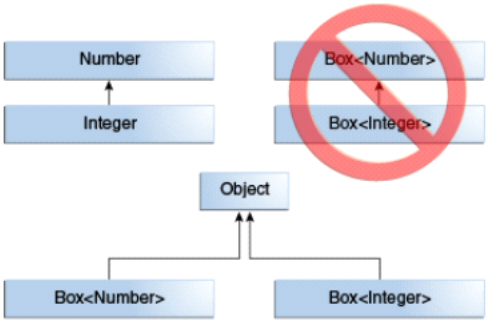
```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem)
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)        Valid
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes

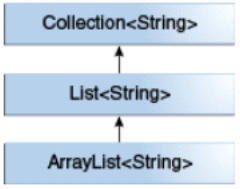
Consider the following method:

`public void boxTest( Box<Number> n ) { /* ... */ }`



You are not allowed to pass in Box<Integer> and Box<Double> because they are not subtypes of Box<Number>.
This is a **common misunderstanding** when it comes to programming with generics.

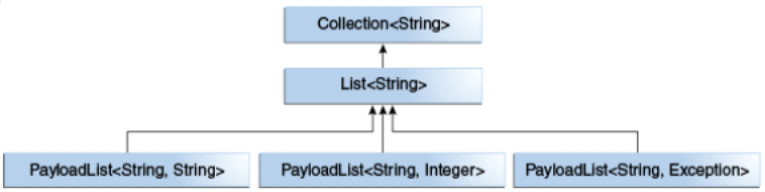
You can subtype a generic class or interface by extending or implementing it.
The relationship between the type parameters of one class or interface and the type parameters of another are determined by the **extends** and **implements** clauses.



ArrayList<E> implements List<E>, and List<E> extends Collection<E>.
So ArrayList<String> is a subtype of List<String>, which is a subtype of Collection<String>.
As long as you do not vary the type argument, the subtyping relationship is preserved between the types.

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

```
PayloadList<String,String>
PayloadList<String,Integer>
PayloadList<String,Exception>
```



COMP2511: Generics, Collections, Iterator

Wildcards

Upper Bounded

In generic code, the question mark (**?**), called the ***wildcard***, represents an unknown type.

The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type. The ***upper bounded wildcard***, **< ? extends** Foo >, where Foo is any type, matches Foo and any subtype of Foo. You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

Unbounded

The **unbounded wildcard** type is specified using the wildcard character (?), for example, List< ? >. This is called a list of unknown type.

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

It prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

To write a generic printList method, use List<?>

Lower Bounded

An upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the **extends** keyword.

A ***lower bounded wildcard*** is expressed using the wildcard character ('?'), following by the **super** keyword, followed by its lower bound: < ? **super** A >.

To write the method that works on lists of Integer and the super types of Integer, such as Integer, Number, and Object, you would specify List<? **Super** Integer>.

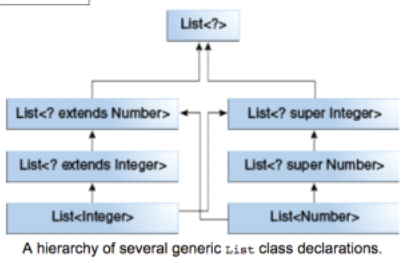
The term List<Integer> is more restrictive than List<? **super** Integer>.

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

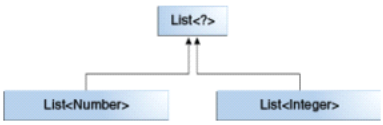
Wildcards and Subtyping

Although Integer is a subtype of Number, List<Integer> is **not** a subtype of List<Number> and, these two types are not related.

The **common parent** of List<Number> and List<Integer> is **List<?>.**



A hierarchy of several generic `List` class declarations.



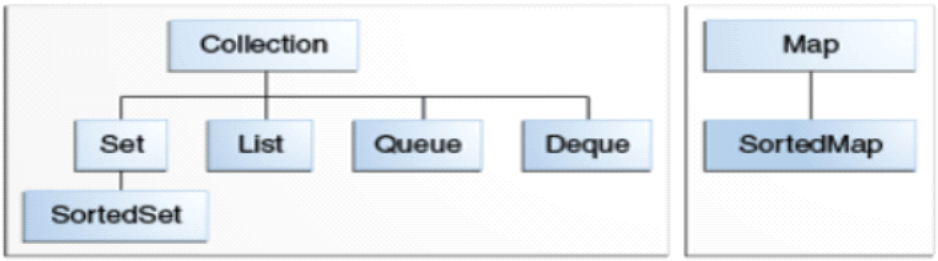
Collections in Java

A collections framework is a unified architecture for representing and manipulating collections. A collection is simply an object that groups multiple elements into a single unit.

All collections frameworks contain the following:

- ***Interfaces***: allows collections to be manipulated independently of the details of their representation.
- ***Implementations***: concrete implementations of the collection interfaces.
- ***Algorithms***: the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
  The algorithms are said to be ***polymorphic***: that is, the same method can be used on many different implementations of the appropriate collection interface.

The core collection interfaces encapsulate different types of collections. The interfaces allow collections to be manipulated independently of the details of their representation.



The core collection interfaces.

A ***Collection*** represents a group of objects known as its elements.

The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a

Collection argument.
The Collection interface contains methods that perform basic operations, such as

- `int size(),`
- `boolean isEmpty(),`
- `boolean contains(Object element),`
- `boolean add(E element),`
- `boolean remove(Object element),`
- `Iterator<E> iterator(),`
- `…many more`

The general purpose implementations are summarized in the following table:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Implemented Classes in the Java Collection, Read their APIs.

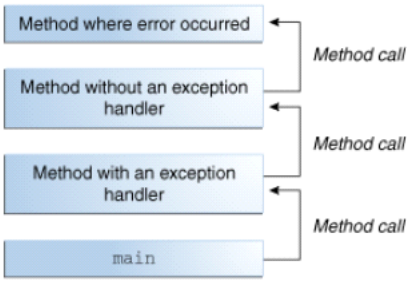
Exceptions

Thursday, 11 July 2019 9:54 AM

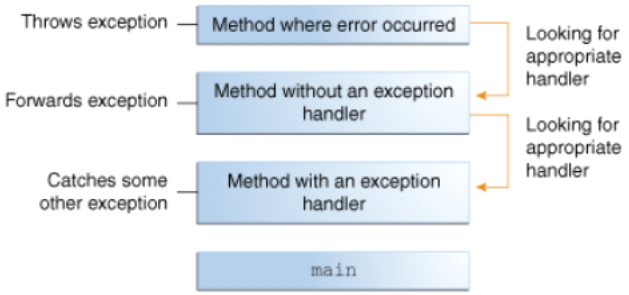
Resources:
https://www.vogella.com/tutorials/JUnit/article.html
https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

An ***exception*** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. When error occurs, an ***exception*** object is created and given to the runtime system, this is called ***throwing*** an exception.

The runtime system searches the call stack for a method that contains a block of code that can ***handle*** the exception. The exception handler chosen is said to ***catch*** the exception.



The call stack.



Searching the call stack for the exception handler.

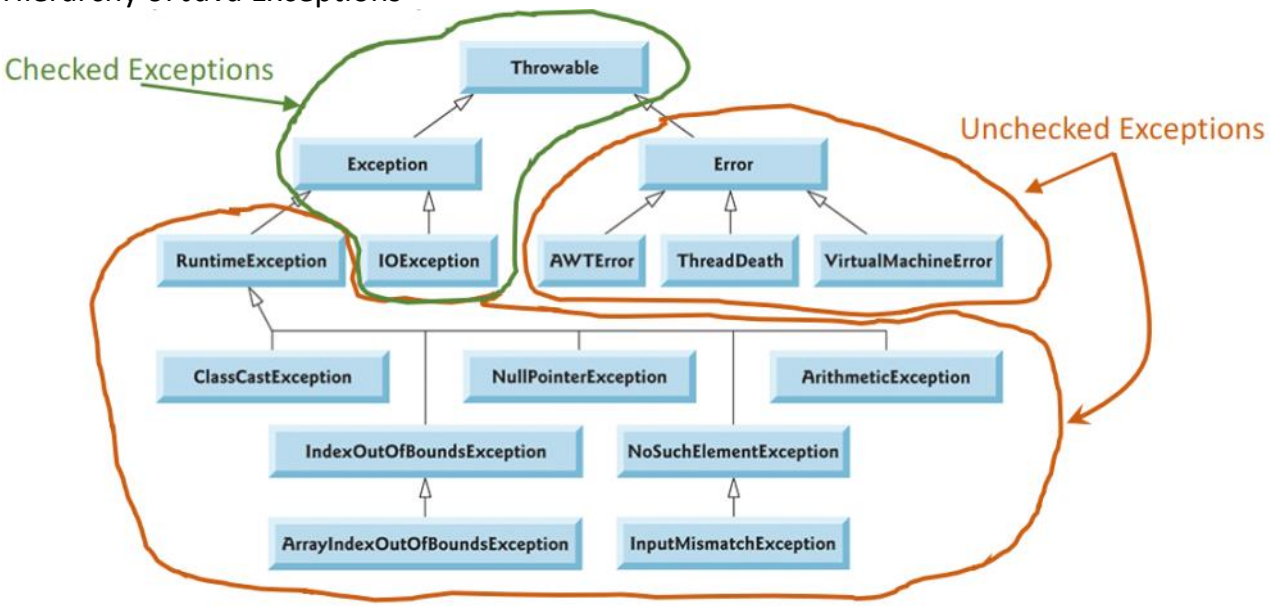
There are three kinds of exceptions:

1. Checked exception (IOException, SQLException, etc)
2. Error (VirtualMachineError, OutOfMemoryError, etc)
3. *Runtime exception* (ArrayIndexOutOfBoundsExceptions, ArithmeticException, etc)

Checked vs. Unchecked Exceptions

An exception's type determines whether it's checked or unchecked.
All classes that are subclasses of ***RuntimeException*** (typically caused by defects in your program's code) or ***Error*** (typically 'system' issues) are ***unchecked*** exceptions.
All classes that inherit from class *Exception* but not directly or indirectly from class ***RuntimeException*** are considered to be **checked** exceptions.

Hierarchy of Java Exceptions



From the book "Java How to Program, Early Objects", 11th Edition, by Paul J. Deitel; Harvey Deitel

Example of a try-catch statement

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

try

catch

finally

User Defined Exceptions

We can also create ***user defined*** exceptions. All exceptions must be a child of `Throwable`

A checked exception needs to extend the `Exception` class, but not directly or indirectly from class `RuntimeException`. It will be enforced by the Handle or Declare Rule. An unchecked exception (like a runtime exception) need to extend the `RuntimeException` class.

```
public class MyException1 extends Exception {

    private static final long serialVersionUID = 1L;

    private String message;

    public MyException1(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

}
```

```
int val = 5 + 7;
if(val > 7) {
    throw new MyException1("asdasd");
}
```

Exceptions in Inheritance

If a subclass method overrides a superclass method, a subclass's *throws* clause can contain a subset of a superclass's *throws* clause. It must not throw more exceptions!

Exceptions are part of an API documentation and contract.

Assertions in Java

An assertion is a statement in the Java that enables you to test your assumptions about your program. Assertions are useful for checking:

- Preconditions, Post-conditions, and Class Invariants (DbC!)
- Internal Invariants and Control-Flow Invariants

You should not use assertions:

- for argument checking in public methods.
- to do any work that your application requires for correct operation.
- Evaluating assertions should not result in side effects.

The following document shows how to use **assertions in Java** :

https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

Important: for backward compatibility, by **default,** Java **disables** assertion validation feature.

It needs to be explicitly **enabled** using the following command line argument:

- `-enableassertions` or `-ea`

```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param  interval refresh interval in milliseconds.
*/
```

```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param  interval refresh interval in milliseconds.
*/
 private void setRefreshInterval(int interval) {
  // Confirm adherence to precondition in nonpublic method
  assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

  ... // Set the refresh interval
 }
```

Summary

- Consider your exception-handling and error-recovery strategy in the design process.
- Sometimes you can prevent an exception by validating data first.
- If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.
- If a subclass method overrides a superclass method, a subclass's *throws* clause can contain a subset of a superclass's *throws* clause. It must not throw more exceptions!
- Programmers should handle checked exceptions.
- If unchecked exceptions are expected, you must handle them gracefully.
- Only the first matching catch is executed, so select your catching class(es) carefully.
- Exceptions are part of an API documentation and contract.
- Assertions can be used to check preconditions, post-conditions and invariants.

Refactoring and Code Smells

Saturday, 27 July 2019 11:25 PM

Refactoring

Code refactoring is the process of restructuring existing computer code without changing its external behaviour. Refactoring is different to adding features and debugging

Originally Martin Fowler and Kent Beck defined refactoring as,
"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour… It is a disciplined way to clean up code that minimizes the chances of introducing bugs."

Refactoring improves code readability, reduces complexity, improves maintenance and extensibility.
If done well, helps to identify *hidden* or *dormant* bugs or vulnerabilities, by simplifying code logic.
If done poorly, may change external behaviour, and/or introduce new bugs!

Refactoring is usually noticed by a code smell (a case of bad design/coding practices). A ***code smell*** is a hint that something may or may not be wrong with your code. Identifying code smells allows us to re-check the implementation details and consider ***better*** alternatives.

Automatic unit tests should be set up before refactoring to ensure routines still behave as expected.

Refactoring is an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation.

Software Maintenance

Software systems evolve over time to meet new requirements and features. Software maintenance involves:

- Fixing bugs
- Improving performance
- Improving design
- Adding features

A majority of software maintenance is for the last three points.

It is harder to maintain code than it is to write from scratch. Most of the development time is spent on maintenance. Good design, coding and planning can reduce maintenance pain and time. Avoid code smells to reduce maintenance and pain time.

Low-Level Refactoring

Low-level refactoring includes:

- Names:
  - Renaming methods, variables etc
  - Naming extracting *magic* constants
- Procedures
  - Extracting code into a method
  - Extracting common functionality (including duplicate code) into a class/method/ etc
  - Changing method signatures
- Reordering
  - Splitting one method into several to improve cohesion and readability (by reducing its size)
  - Putting statements that semantically belong together near each other

IDEs support low-level refactoring by providing the following features:

- Renaming
  - Variable, method, class.
- Extraction:
  - Method, constant

    - Repetitive code snippets
    - Interface from a type
- Inlining: method, etc.
- Change method signature.
- Warnings about inconsistent code

Higher-Level Refactoring

Higher-level refactoring involves:

- Refactoring to ***design patterns***
- Changing language idioms (safety, brevity)
- Performance optimisation
- Generally high-level refactoring is much more important but not very well-supported by tools

Code Smells

- Duplicated code
- Poor abstraction (change one place → must change others)
- Large loop, method, class, parameter list; deeply nested loop
- Class has too little cohesion
- Modules have too much coupling
- Class has poor encapsulation
- A subclass doesn't use majority of inherited functionalities
- A "data class" has little functionality
- Dead code
- Design is unnecessarily general
- Design is too specific

Code and Design Smells

Monday, 29 July 2019 8:11 PM

Smells are design aspects that violate fundamental design principles and impact software quality.

They occur at different levels of granularity

- ***Code smells*** - structures in implementation of code such as large methods, classes with multiple responsibilities, complex conditional statements that lead to poor code
- ***Design smells*** - design aspects at a higher level of abstraction (class level abstractions) such as classes with multiple responsibilities, refused bequest

Regardless of granularity, smells in general indicate violation of software design principles, and eventually lead to code that is rigid, fragile and require *refactoring*.

There are different types of smells

- ***Bloaters*** - code, methods and classes that have grown in size, that they are hard to work with (e.g. long methods, large classes, long parameter list, data clumps)
- ***OO Abusers*** - result from incorrect or incomplete application of OO principles (e.g switch statements, refused bequest)
- ***Change preventers*** - code changes that are difficult (rigid code). This includes divergent change and shot gun surgery
- ***Dispensables*** - code that is pointless and unnecessary (e.g. comments, data class, lazy class, duplicate code)
    - ***Couplers*** - excessive coupling between classes

Smells and their Refactoring Techniques

Long Methods

You can reduce the length of a method body via Extract Method. This makes it

- More readable, and results in less code duplication
- Isolates independent parts of code, - errors are less likely

If local variables and parameters interfere with extracting a method, use

- Replace Temp With Query
- Introduce Parameter Object
- Preserve Whole Object

If the above doesn't work, try moving the entire method to a separate object via Replace Method with Method Object

Replace Method with Method Object

Conditional operators and loops are a good clue that code can be moved to a separate method

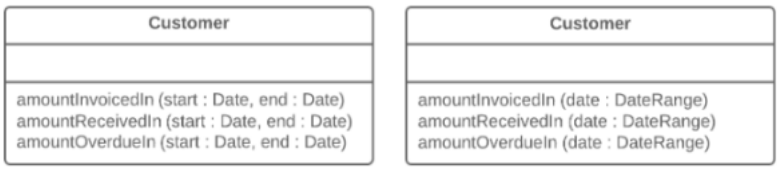
Example of **Extract Method**:

```java
public void debit(float amount) {
    // deducts amount by balance
    balance -= amount;

    // records transaction
    transactions.add(new Transaction(amount, true));

    // record last transaction date
    lastTransactionDate = LocalDate.now().toString();
}
```

```java
public void debit(float amount) {
    deductBalance(amount);
    recordTransaction(amount, true);
    recordLastTransaction();
}

private void deductBalance(float amount) {
    balance -= amount;
}
private void recordTransaction(float amount, boolean isDebit) {
    transactions.add(new Transaction(amount,isDebit));
}
private void recordLastTransaction() {
    lastTransactionDate = LocalDate.now().toString();
}
```

Example of **Introduce Parameter Object**:
This is for methods containing repeating groups of parameters, causing code duplication. Introducing a parameter object eliminates a smell such as a long parameter list, data clumps, primitive obsession, long method.

Consolidate these parameters into a separate class. This also helps to move the methods for handling this data.
Beware, if only data is moved to a new class and associated behaviours are not moved, this begins to smell of a **Data class**



Example of **Replace Temp with Query**:
Often we place the result of an expression in a local variable for later use in the code. With **Replace Temp With Query**, we:

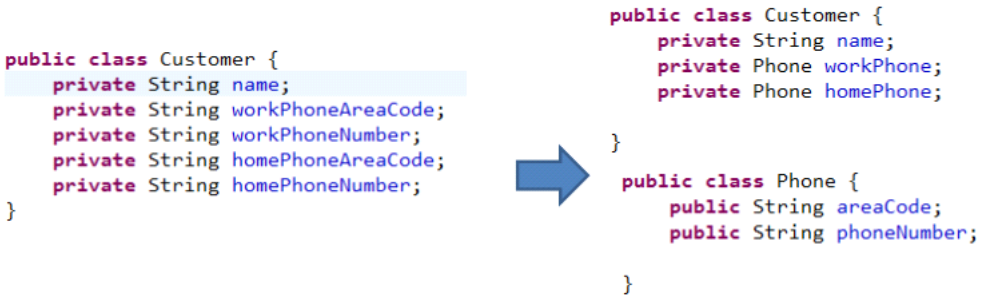
- Move the entire expression into a separate method and return the result from it
- Query the method instead of using a variable
- Reuse the new method in other methods

This will help us eliminate smells such as long methods and duplicate code.

```java
double calculateTotal() {
 double basePrice = quantity * itemPrice;
if (basePrice > 1000) {
  return basePrice * 0.95;
}
else {
 return basePrice * 0.98;
}
}
```

```java
double calculateTotal() {
 if (basePrice() > 1000) {
     return basePrice() * 0.95; }
else {
   return basePrice() * 0.98; } }
double basePrice() {
  return quantity * itemPrice;
}
}
```

Example of **Extract Class**:
Having all the phone details in class Customer is not a good OO design and also breaks SRP . We can refactor it into two separate classes, each with its appropriate responsibility



Large Class

This smell is similar to large method and usually violates SRP. It may have a large number of instance variables and several methods. There is typically a lack of cohesion and potential for duplicate code smells.

To resolve this, we can bundle a group of variables via **Extract Class** or **Extract Sub-Class**

Long Parameter List

Problem: Calling a query method and passing its results as the parameter of another method, while that method could have called the query directly. This results in too many parameters to remember, bad readability, usability and maintenance.

```
public String getSummary() {
    return buildCustomerSummary(getFirstName(), getLastName(), getTitle(),
            address.getCity(), address.getPostCode());
}

private String buildCustomerSummary(String firstName, String lastName,
        String title, String city, String postCode) {
    return title + " " + firstName + " " + lastName + "," + city + "," + "postcode";
}
```

Solution: Try placing a query call inside the method body via **replace**

```
// 1. Apply replace parameter with method call
// 2. Apply change method signature to remove the first three parameters
// 3. Preserve whole object - passing in the entire object instead of object data
private String buildCustomerSummary(Address address) {
    return getTitle() + " " + getFirstName() + " " + getLastName() + "," + address.getCity() + ","
           + address.getPostCode();
}
```

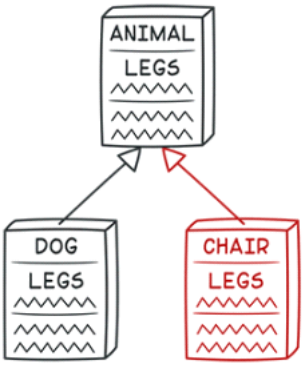
Data Clumps

Problem: Different parts of the code contain identical groups of variables. e.g. fields in many classes, parameters in many method signatures. This can lead to the code smell Long Parameter List

Solution: Move the behaviour to the data class via **Move Method.**
If repeating data comprises the fields of a class, use **Extract Class** to move the fields to their own class.
If the same data clumps are passed in the parameters of methods, use **Introduce Parameter Object** to set them off as a class.
If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields **Preserve Whole Object** will help with this

Refused Bequest

Problem: A subclass uses only some of the methods and properties inherited from its parents. The unneeded methods may simply go unused or be redefined and give off exceptions. This is often caused by creating inheritance between classes only by the desire to reuse the code in a super-class.



Solution: If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favour of **Replace Inheritance with Delegation**.

If inheritance is appropriate, but the superclass contains fields and methods not applicable to all classes, then consider the following options.

- Create a new subclass
- Apply **Push Down Field** to move field relevant only to the subclass from the superclass
- Apply **Push Down Method** to move behaviour from superclass to subclass, as behaviour makes sense only to the subclass

Often you may apply an **Extract Sub-Class Class** to combine the above steps

Example:

class Camel does not use field model. It should be pushed down to class Car

```
public abstract class Transport {
  // Push Down Field
  private String model;
  // Push Down Method
  public String getModel() throws Exception
  {
        return model;
  }
  ...
}
public class Car extends Transport { ... }
public class Camel extends Transport {
  ...
  public String getModel() {
        throw new NotSupportedException();
  }
}
public abstract class Transport {
  ...
}
```



```
// Use Push Down Field to move field and
// Push Down Method to move behaviour
// only relevant to sub class
// from super class to sub class

public abstract class Transport {
        ...
}
public class Car extends Transport
{
  private String model;
  public String getModel()
  {
        return model;
  }
  ...
}
public class Camel extends Transport
{
  ...
}
```

Duplicate Code

If the same code is found in ***two or more methods in the same class***, use **Extract Method** and place calls for the new method in both places

If the same code is found in ***two subclasses of the same level***,

- use **Extract Method** for both classes, followed by **Pull Up Field** for the fields used in the method that you are pulling up.
- If the duplicate code is inside a constructor, use **Pull Up Constructor Body**
- If the duplicate code is similar but not completely identical, use **Form Template Method**
- If two methods do the same thing but use different algorithms, select the best algorithm and apply **Substitute Algorithm**

If duplicate code is found in ***two different classes***:

- If the classes are not part of a hierarchy, use **Extract SuperClass** in order to create a single superclass for these classes that maintains all the previous functionality

Feature Envy

Problem: A method that is more interested in a class other than itself invokes several methods of another object to calculate some value. This create unnecessary coupling between the classes.

Solution: a goal of OO design is to put the methods with its associated data. So the method must be moved to the relevant class via **Move Method**
If only part of a method accesses the data of another object, use **Extract Method** followed by **Move Method** to move the part in question.
If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data

Divergent Change, Shot Gun Surgery

Divergent Change is when one class is changed in different ways for different reasons.
Solution: Any change to handle a variation should change a single class, and all the typing in the new class should express the variation.
To clean this up you identify everything that changes for a particular cause and use **Extract Class** to put them all together.

Shot Gun Surgery is a small change in the code that forces lots of little changes to different classes.
Solution: use **Move Method** or **Move Field** to put all the changes into a single class. Often you can use **Inline Class** to bring a whole bunch of behaviour together

In divergent change, one class suffers many kinds of changes, while in shotgun surgery, one change alters many classes.

Data Classes

Problem: Classes that just have attributes with setters and getters and no behaviour.
One of the goals of OO design is to put behaviour where the data is

```
public class CustomerSummaryView {
    private Customer customer;
    public CustomerSummaryView(Customer customer) {
        this.customer = customer;
    }
    public String getCustomerSummary() {
        Address addr = customer.getAddress();
        return customer.getTitle() + " " + customer.getFirstName() + " "
        + customer.getLastName() + "," + addr.getCity() + "," + addr.getPostCode();
    }
}
```

Solution: Move the behaviour to the data class via **Move Method**

```
// 1. Apply move method inside method getCustomerSummary() in address class
//    to move method to class Customer
// 2. Extract the address summary
// 3. Move the address summary to the class Address

public String getCustomerSummary() {
    return getTitle() + " " + getFirstName() + " "
    + getLastName() + "," + address.getAddressSummary();
}
```

Lazy Classes

Problem: Classes that aren't doing much to justify their existence. Subclass without any overridden methods or additional fields can be lazy classes as well.

```java
public class PostCode {
    private String postcode;

    public PostCode(String postcode) {
        this.postcode = postcode;
    }

    public String getPostcode() {
        return postcode;
    }

    public String getPostcodeArea(){
        return postcode.split("")[0];
    }
}
```

```java
public class Address {

    private final String number;
    private final String street;
    private final String city;
    private final String country;
    private final PostCode postcode;

    public Address(String no, String st, String city,
                   PostCode pCode, String country) {
        this.number = no;
        this.street = st;
        this.city = city;
        this.postcode = pCode;
        this.country = country;
    }
```

Solution: Move the data (postcode) from lazy class PostCode to the class Address. Then delete the lazy class

Switch Statements

Problem: Switch statements are bad from an OO design point of view.
Solution: Replace switch statements with a polymorphic solution based on **Strategy Pattern** applying a series of refactoring techniques (**Extract Method, Move Method, Extract Interface etc**).

Refactoring Techniques to be familiar with:

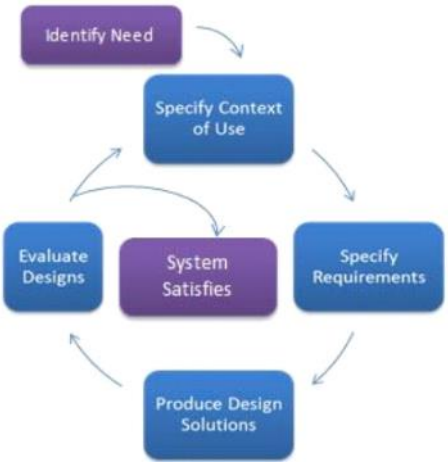
- Move Field/Method
- Extract Class/Inline Class
- Extract Method
- Inline Method/Temporary Variable
- Replace Temp with Query
- Replace Method with Method Object
- Rename Method
- Substitute Algorithm
- Introduce Parameter Object
- Preserve Whole Object
- Extract Sub Class/Super Class/Interface
- Extract Method
- Pull Up Field/Method/Constructor Body
- Form Template Method
- Replace Inheritance with Delegation
- Replace Conditional with Polymorphism

Resource:
https://refactoring.guru/refactoring/smells
https://www.refactoring.com/catalog/

User Centred Design and MVC

Saturday, 27 July 2019 11:24 PM

User Centred Design



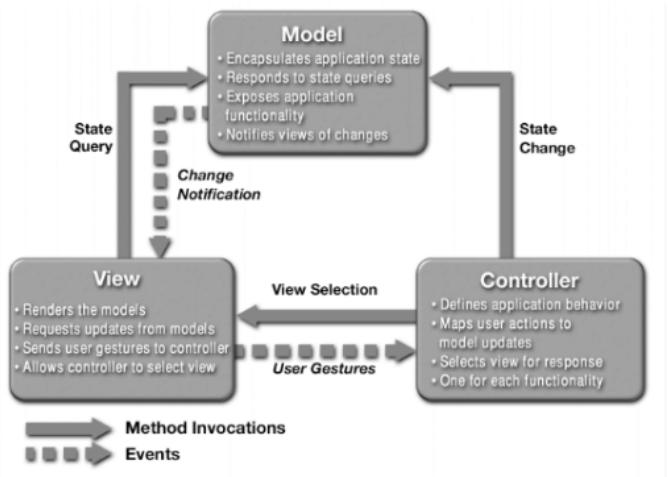
Specify the context of use: Identify the people who will use the product (Persona), what they will use it for, and under what conditions they will use it. (Scenario)
Specify requirements : Identify any business requirements or user goals that must be met for the product to be successful. (Use-Cases)
Create design solutions: This part of the process may be done in stages, building from a rough concept to a complete design.
Evaluate designs: Evaluation - ideally through usability testing with actual users - is as integral as quality testing is to good software development

User Centred Design (UCD) can be applied to the design of anything that has a user—from mobile phones to kitchens. The era of **feature-centric development** is coming to an end. Consumers are beginning to realize that more features do not always mean a better product. Quality of experience is far more likely to be a product differentiator than product features. UCD provides a way to engineer these quality experiences.

Usability Heuristics

1. Visibility of system status
2. Match between system and real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalist design
9. Help users recognize, diagnose, recover from errors
10. Help and documentation

MVC Architecture



View

This is the ***presentation layer*** which provides interaction that the user sees (e.g. a webpage). View component takes inputs from the user and sends actions to the ***controller*** for manipulating data. View is responsible for displaying the results obtained by the controller from the model component in a way that user wants them to see or a pre-determined format. The format in which the data can be visible to users can be of any 'type' such as HTML or XML depending upon the presentation tier.
It is responsibility of the controller to choose a view to display data to the user.

For example we have an array [14, 26, 31] and we can have different views for the same model

Model

The model holds all the data and state of the system. It responds to instructions to change of state (from the controller). It also responds to requests for information about its state (usually from the view). It sends notifications of state changes to the *observer* (view). The model does NOT depend on the controller or the view.

Controller

The controller is the glue between the user and processing (*model*) and formatting (*view*) logic. It accepts the user request or inputs to the application, parses them and decides which type or mode or view should be invoked.

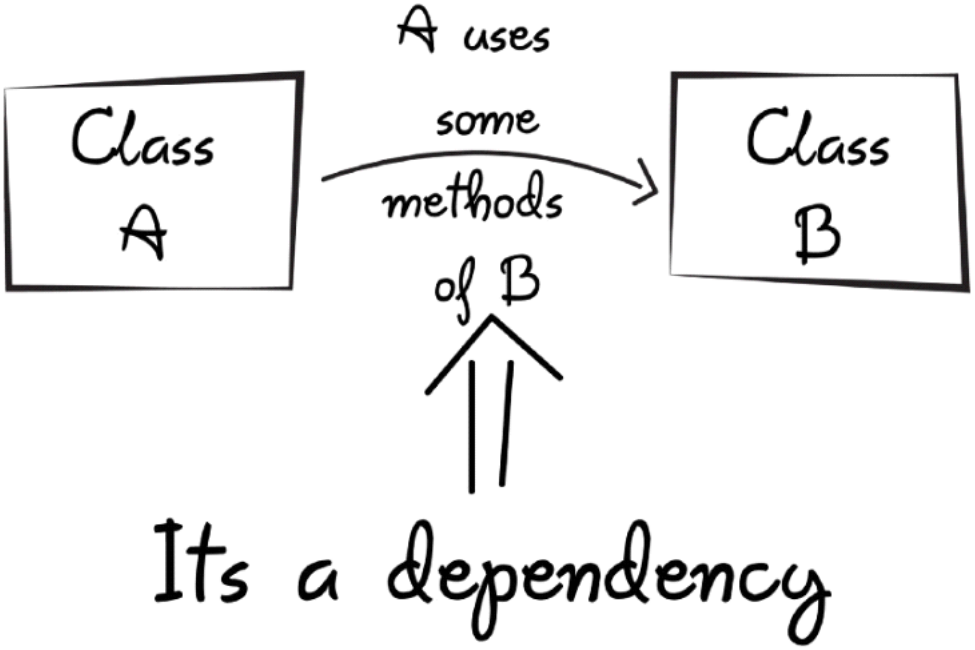
Benefits	Weakness
• Abstraction of application architecture into three high-level components (Model, View, and Controller). Model has no knowledge of the View that is provided to the user. • Abstraction makes model and view components reusable without modification • Supports multiple views of the same data on different platforms at the same time • Enhances testability	• Complexity • Cost of frequent updates - an active model that undergoes frequent changes could flood the views with update requests

Dependency Injection

Saturday, 3 August 2019 2:52 PM

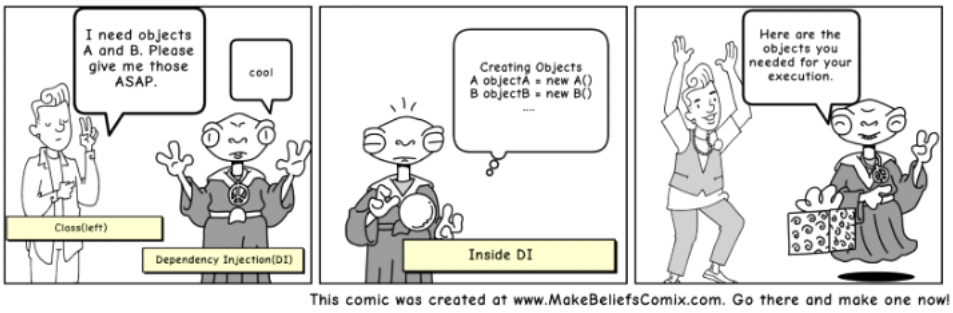
In software engineering, **dependency injection** is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service).

When class A has some functionality of class B then it is said that class A has a dependency of class B.



In Java, before we can use methods of other classes, we first need to create the object of that class (i.e. class A needs to create an instance of class B).

So, transferring the task of creating the object to someone else and directly using the dependency is called dependency injection.



Types of dependency injection:

1. ***Constructor injection*** - the dependencies are provided through a class constructor
2. ***Setter injection*** - the client exposes a setter method that the injector uses to inject the dependency
3. ***Interface injection*** - the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency

So it is the dependency injection's responsibility to:

1. Create the objects
2. Know which classes require those objects
3. Provide them all those object

JavaFX

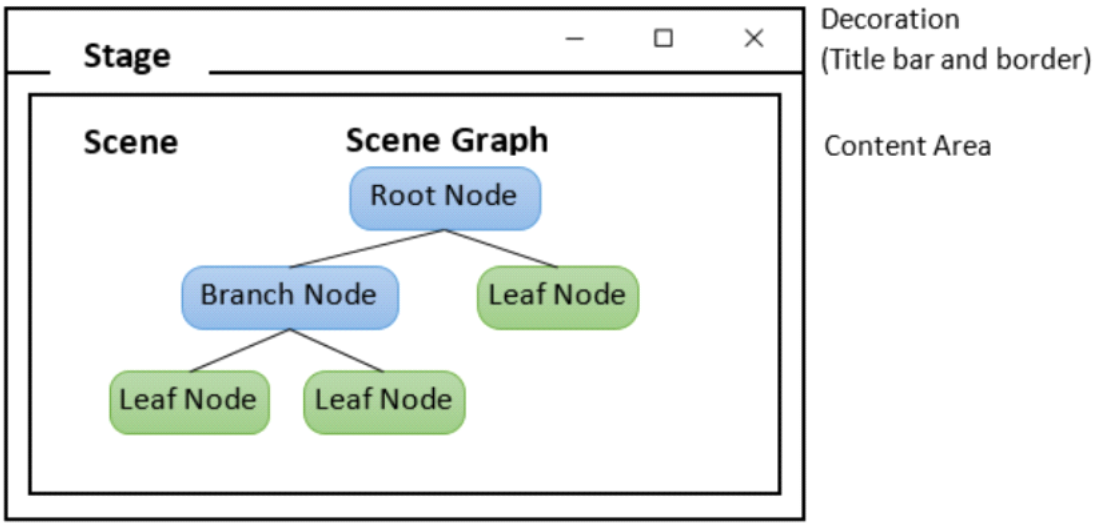
Monday, 22 July 2019 2:01 PM

Understanding the JavaFX Architecture

JavaFX uses the metaphor of a theatre to model graphical user interfaces.
A **stage** (defined by the `javafx.stage.Stage` class) represents the top-level container (window). The individual controls (buttons, textboxes, labels, etc) are contained in a **scene** (defined by the `javafx.scene.Scene` class).

An application can have more than one scene, but only one of the scenes can be displayed on the stage at any given time.

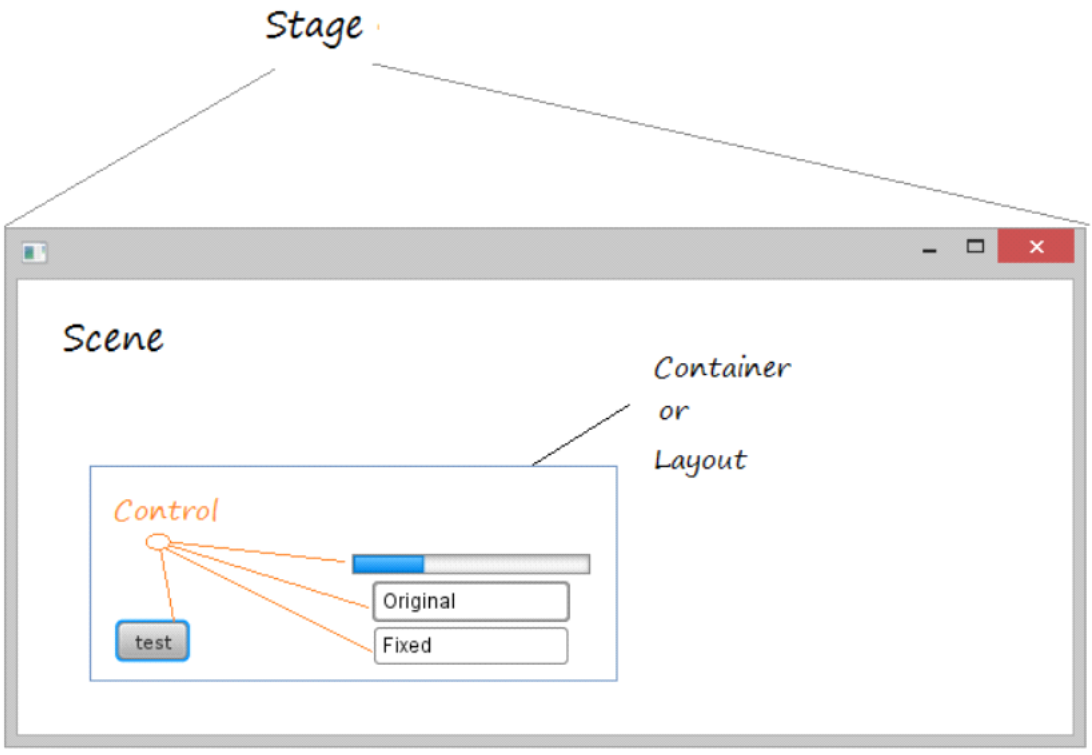
The contents of a scene is represented by a hierarchical scene graph of nodes that represents all of the visual elements of the application's user interface. A single element in a scene graph is called a node (defined by `javafx.scene.Node`). With the exception of the root node of a scene graph, each node has a single parent and zero or more children. Hence, each node is either a "leaf", with no children, or a "branch" with multiple children. Leaf nodes are typically basic geometric shapes, like rectangles, or text nodes. Branch nodes (defined by `javafx.scene.Parent`) allow nodes to be composed to form more complex UI elements.



Panes (also known as **Layout Containers**) are a specific type of branch in the graph. Panes can be used to allow for flexible and dynamic arrangements of the UI controls within a scene graph of a JavaFX application. There are many JavaFX predefined panes. Some of these are briefly described below. You can read more about panes here.

- The ***BorderPane*** lays out its content nodes in the top, bottom, right, left, or center region.
- The ***HBox*** arranges its content nodes horizontally in a single row.
- The ***VBox*** class arranges its content nodes vertically in a single column.
- The ***StackPane*** class places its content nodes in a back-to-front single stack.
- The ***GridPane*** class enables the nodes to be created in a flexible grid of rows and columns in which to lay out content nodes.

The picture below shows the relationship between **Stage**, **Scene**, **Container** and **Controls**.



Creating a Graphical User Interface (GUI) using JavaFX based on MVC

A typical JavaFX application consists of a graphical user interface built using the components described above, that interacts with the back-end application logic. The JavaFX platform provides **FXML**, a scriptable, XML based markup language for constructing scene graphs. FXML can be a more convenient alternative to constructing such graphs in procedural code and is ideally suited to defining the user interface of a JavaFX application, since the hierarchical structure of an XML document closely parallels the structure of the JavaFX scene graph. The benefits of using FXML is that the *application interface design is separated from the application logic*, thereby making the code easier to maintain.

Use of FXML enables us to structure a JavaFX application based on the MVC (Model-View-Controller) architecture which typically comprises of three key components as described below:
(MVC will be covered more generally in later lectures)

- A **view** consisting of a FXML file which defines the user-interface components represented by the hierarchical scene graph of nodes
- A **Controller** (written as a Java class) that receives the events initiated by clicking on the different UI controls (e.g., button) on the GUI and directs these events to the Model
- A **Model** (also a java class) which contains the state of the application, and methods to change that state.

The MVC architecture provides for a software architecture where the model, view and controller components are decoupled, making the application more easier to understand and maintain. More details about FXML and the benefits of its use can be found in this link.

JavaFX provides all the major UI controls that are required to develop a full-featured application. However, the look and feel of your UI can be customised using Cascading Style Sheets (CSS) which separate appearance and style from implementation. Customised styling can be applied to UI of a JavaFX application without changing any of that application's source code. JavaFX CSS styles can also be easily assigned to the scene at runtime, allowing an application's appearance to dynamically change.

JavaFX SceneBuilder

JavaFX Scene Builder, a design tool for the JavaFX platform, allows simple drag-and-drop positioning of the graphical user interface (GUI) components onto a JavaFX scene. JavaFX SceneBuilder is a WYSIWYG (What You See Is What You Get) editor for FXML files and as you build your GUI components, the Scene Builder generates FXML markup code.

When dealing with multiple screens, for every screen, we create one Screen class and an associated Controller class for that screen.