Contents

“It's hard to beat a person who never gives up.”..Babe Ruth

Interview Preparation – Questionnaire ***- Ashay Raut***

Let me tell you something you already know. The world ain’t all sunshine and rainbows. It’s a very mean and nasty place and I don’t care how tough you are it will beat you to your knees and keep you there permanently if you let it. You, me, or nobody is gonna hit as hard as life. But it ain’t about how hard ya hit. It’s about how hard you can get it and keep moving forward. How much you can take and keep moving forward. That’s how winning is done! Now if you know what you’re worth then go out and get what you’re worth. But ya gotta be willing to take the hits, and not pointing fingers saying you ain’t where you wanna be because of him, or her, or anybody! Cowards do that and that ain’t you! You’re better than that!
Sylvestor Stalone, Rocky

1. Websites:

WWW.CAREERCUP.COM

http://www.geeksforgeeks.org/java/

WWW.LEETCODE.COM

http://www.programmerinterview.com

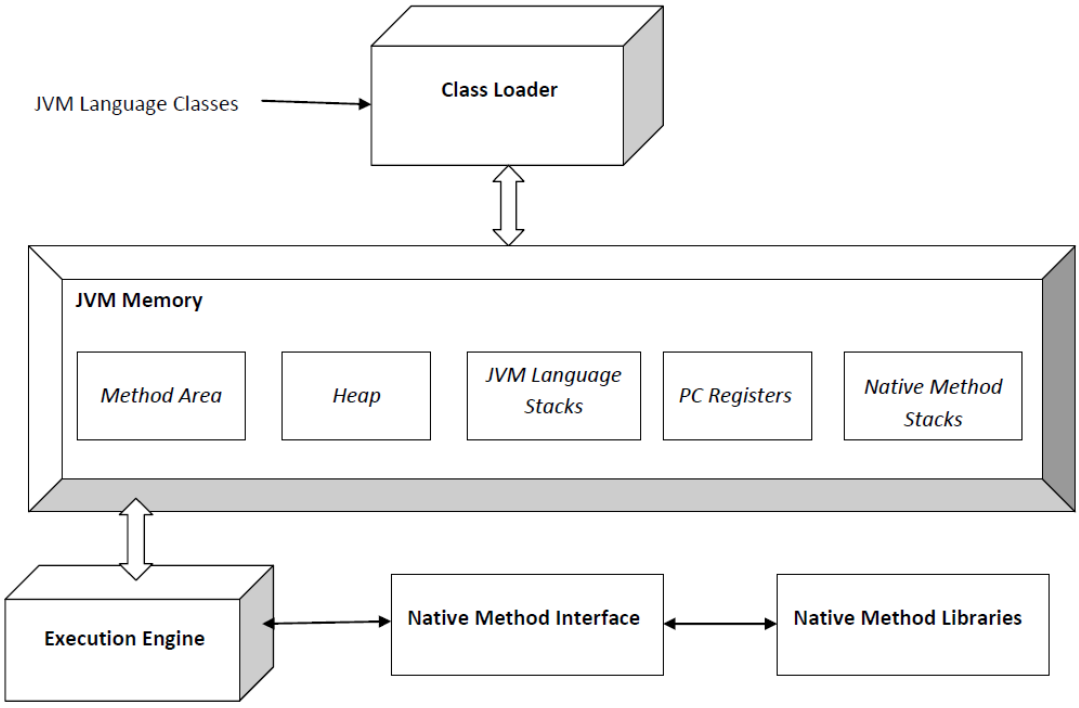
www.codercareer.com

2. JVM

JVM: Abstract Machine which executes bytecodes.

JRE: It implements JVM for specific OS and provides other class libraries to execute java specific programs.

JDK: Provides development kit and system/Java libraries



Class Loader Sub System:

Loading, Linking, Initialization

Loading:

The Class loader reads the ***.class*** file, **generate the corresponding binary data and save it in method area**. For each *.class* file, JVM stores following information in method area.

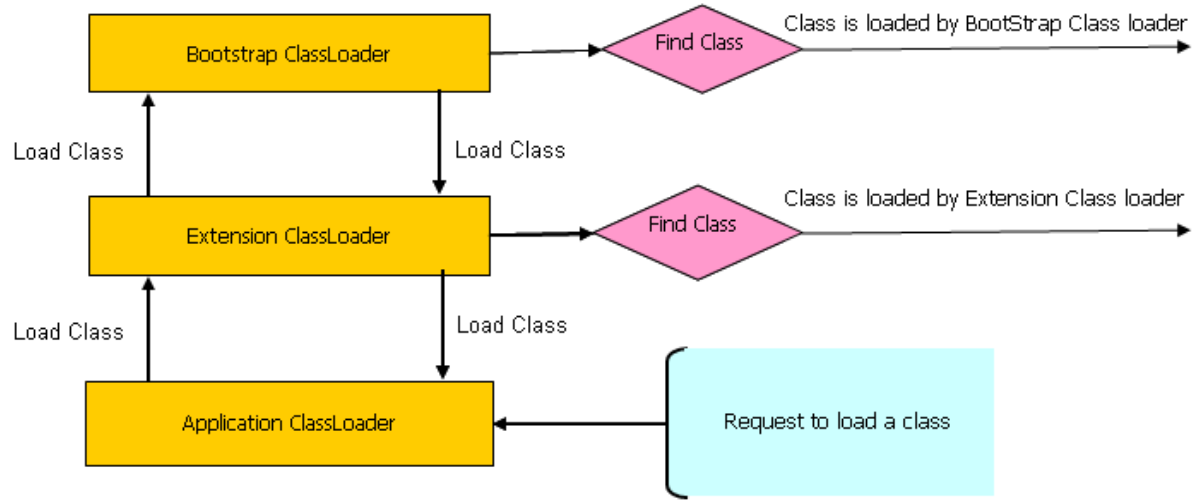
- Fully qualified name of the loaded class and its immediate parent class.
- Whether *.class* file is related to Class or Interface or Enum
- Modifier, Variables and Method information etc

Linking : Performs **verification, preparation, and (optionally) resolution**.

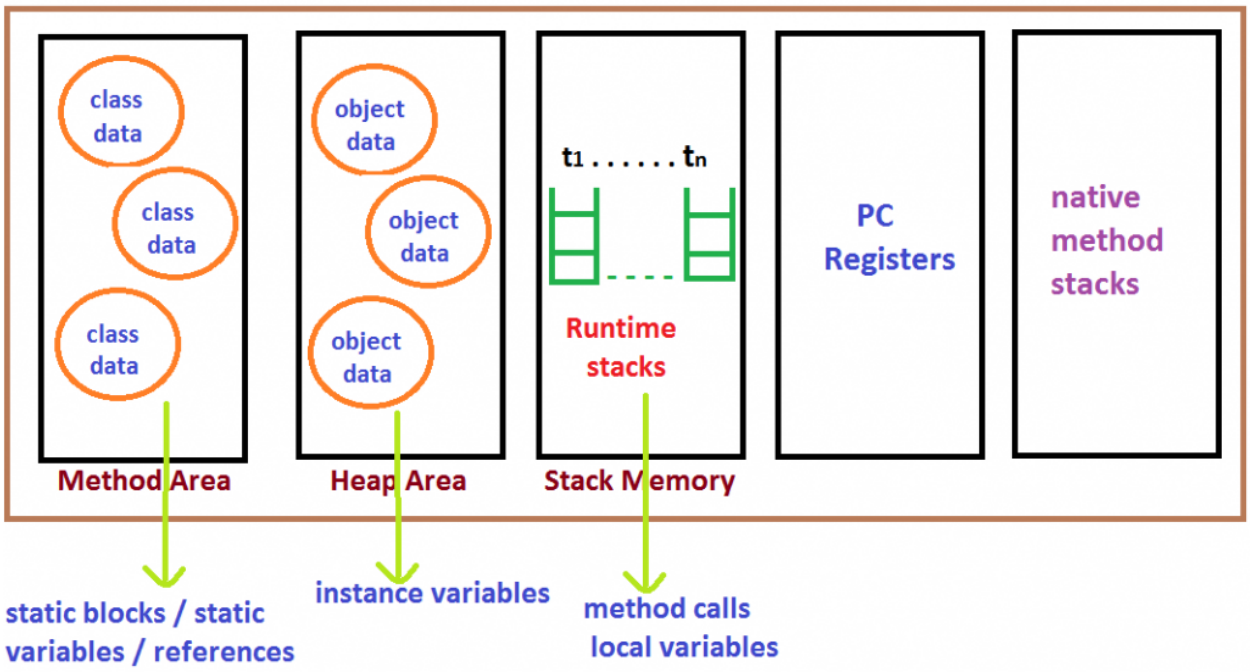
1. ***Verification*** : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
2. ***Preparation*** : JVM allocates memory for class variables and initializing the memory to default values.
3. ***Resolution*** : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

Initialization : In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.
In general there are three class loaders:

1. ***Bootstrap class loader*****:** Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in *JAVA_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
2. ***Extension class loader*****:** It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA_HOME/jre/lib/ext*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
3. ***System/Application class loader*****:** It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$ExtClassLoader* class.



JVM Memory:



Method area: In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only **one method area per JVM**, and it is a shared resource.

Heap area: Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.
Stack area: For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.

PC Registers: Store address of current execution instruction of a thread. Obviously, each thread has
separate PC Registers.

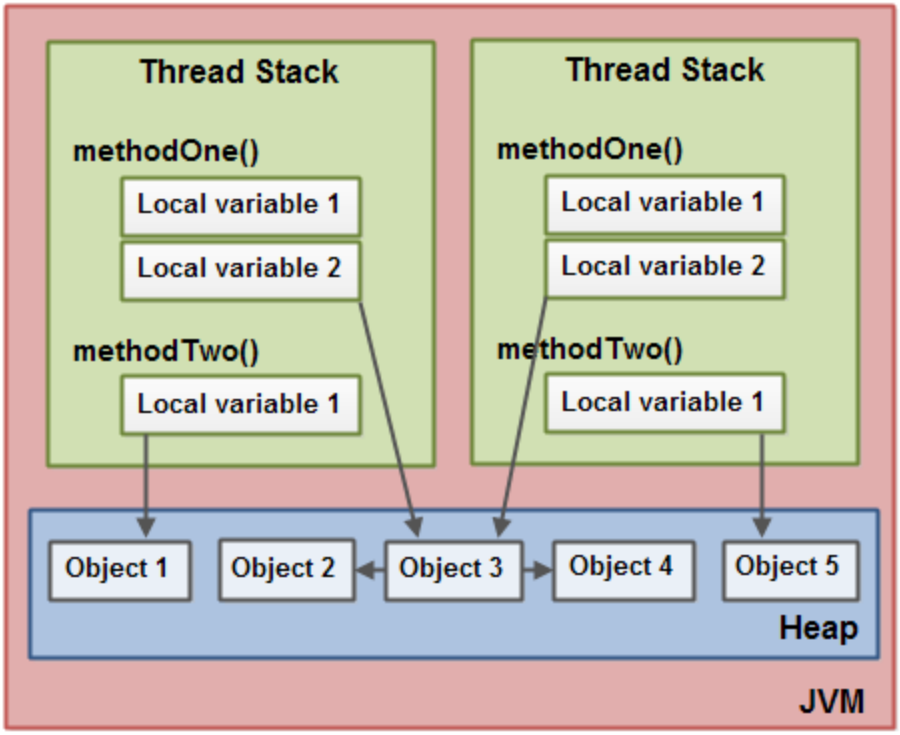
Native method stacks: For every thread, separate native stack is created. It stores native method information.

Execution Engine:

Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

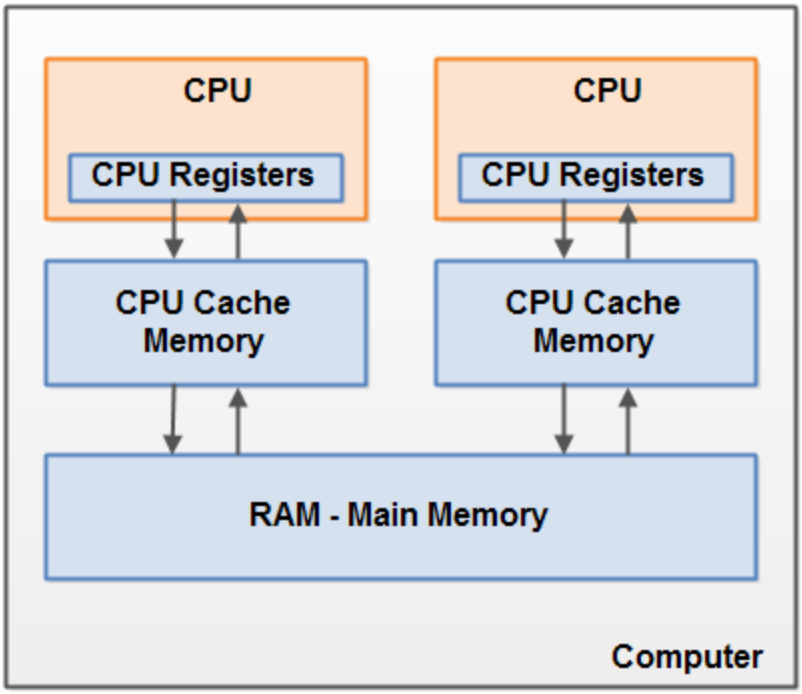
- ***Interpreter***: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- ***Just-In-Time Compiler(JIT)***: It is used to increase efficiency of interpreter. It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
- ***Garbage Collector***: It destroy un-referenced objects. For more on Garbage Collector,refer Garbage Collector.

Thread Stack

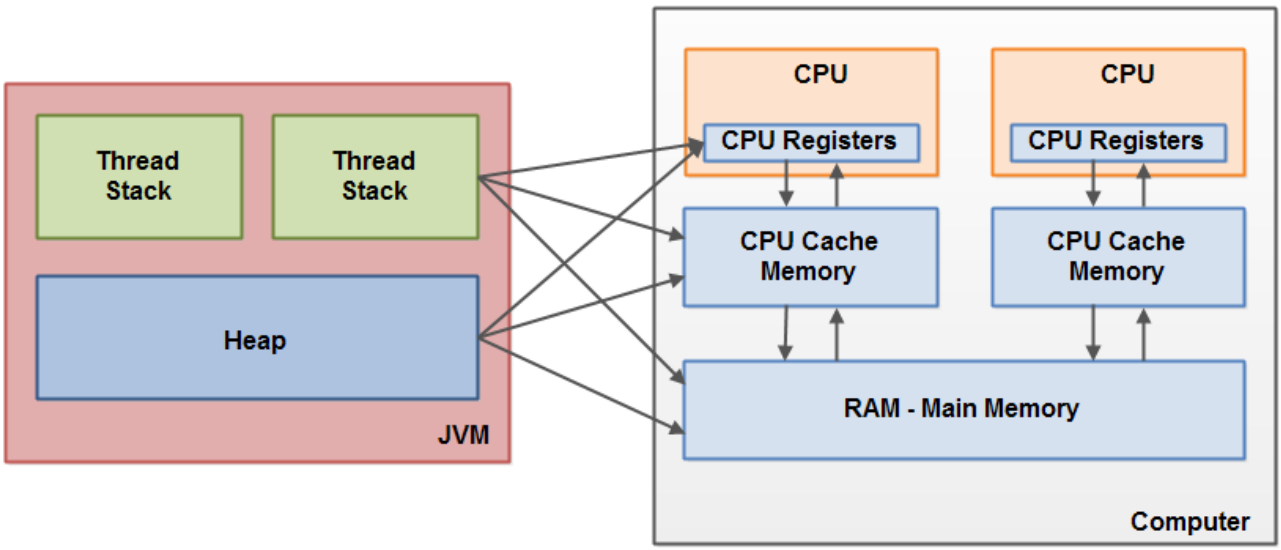


A modern computer often has 2 or more CPUs in it. Some of these CPUs may have multiple cores too

Each CPU contains a set of registers which are essentially in-CPU memory. The CPU can perform operations much faster on these registers than it can perform on variables in main memory. That is because the CPU can access these registers much faster than it can access main memory.



As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. This is illustrated in this diagram:



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

1. Visibility of thread updates (writes) to shared variables.
2. Race conditions when reading, checking and writing shared variables.

Stack dump

Kill -3 on linux

Jstat -l <pid>

Stack Heap:

Jmap -dump:format=b

java.lang.OutOfMemoryError: Metaspace
this error is Java 8 equivalent of Permgen error because the Permanent generation has been removed in java 8 and a new Metaspace is introduced.

Read more: http://javarevisited.blogspot.com/2017/05/solution-of-javalangoutofmemoryerror-gc-overhead-limit-exceeded.html#ixzz4loT5ckJ2

3. 32 bits vs 64 bits JVM

64-bit vs. 32-bit really boils down to the size of object *references*, not the size of numbers.

In 32-bit mode, references are four bytes, allowing the JVM to uniquely address 2^32 bytes of memory. This is the reason 32-bit JVMs are limited to a maximum heap size of 4GB (in reality, the limit is smaller due to other JVM and OS overhead, and differs depending on the OS).

In 64-bit mode, references are (surprise) eight bytes, allowing the JVM to uniquely address 2^64 bytes of memory, which should be enough for anybody. JVM heap sizes (specified with `-Xmx`) in 64-bit mode can be huge.

But 64-bit mode comes with a cost: references are double the size, increasing memory consumption. This is why Oracle introduced "Compressed oops". With compressed oops enabled (which I believe is now the default), object references are shrunk to four bytes, with the caveat that the heap is limited to four billion objects (and 32GB Xmx). Compressed oops are not free: there is a small computational cost to achieve this big reduction in memory consumption.

Problem 1: 30-50% of more heap is required on 64-bit. Why so? Mainly because of the memory layout in 64-bit architecture. First of all – object headers are 12 bytes on 64-bit JVM. Secondly, object references can be either 4 bytes or 8 bytes, depending on JVM flags and the size of the heap. This definitely adds some overhead compared to the 8 bytes on headers on 32-bit and 4 bytes on references. You can also dig into one of our earlier posts for more information about calculating the memory consumption of an object.

Problem 2: Longer garbage collection pauses. Building up more heap means there is more work to be done by GC while cleaning it up from unused objects. What it means in real life is that you have to be extra cautious when building heaps larger than 12-16GB. Without fine tuning and measuring you can easily introduce full GC pauses spanning several minutes. In applications where latency is not crucial and you can optimize for throughput only this might be OK, but on most cases this might become a showstopper.

4. Multithreading

Process & Thread:

Wait: Object wait methods has three variances, one which waits indefinitely for any other thread to call notify or notifyAll method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.

notify: notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.

notifyAll: notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

InterruptedException:

The Object class in java contains three final methods that allows threads to communicate about the lock status of a resource.

The current thread which invokes these methods on any object should have the object **monitor** else it throws **java.lang.IllegalMonitorStateException** exception.

ExecutorService service=Executors.newSingleThreadExecutor(1); //thread pool of just 1

ExecutorService service=Executors.newFixedThreadpool(1); //fixed # of threads

ExecutorService service=Executors. newCachedThreadPool (); //cached thread pool

Future f = executorService.submit(Callable<>); //instead of Runnable, it's callable and instead of run, it's call()

AtomicInteger:

Combination of volatile variable and **CompareAndSwap** (optimistic locking).

It compares last value as the expected memory in the memory and if it matches then updates it.

It is **nonblocking concurrent algorithm** where it does require explicit synchronized lock. It works best with low to medium thread contention.

```
public final int getAndIncrement() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
 }
```

CountDownLatch.await() will wait until count reaches zero. It can't be reused once it's reaches to zero.
CyclicBarrier: CyclicBarrier by calling reset()

ExecutorService extends Executor (Parent Interface) **Executor**
defines execute() method which accepts an object of the Runnable interface, while submit() method can accept objects of both Runnable and Callable interfaces.
Executor's execute() method doesn't return any result, its return type is void.
ExecutorService -> submit() method returns the result of computation via a Future object
Future.get() can result at later time, blocking call.

ExecutorService also provides methods to control the thread pool e.g. terminate the thread pool by calling the shutDown() method.

Executors class provides factory methods to create different kinds of thread pools e.g. newSingleThreadExecutor() creates a thread pool of just one thread, newFixedThreadPool(int numOfThreads) creates a thread pool of fixed number of threads and newCachedThreadPool() creates new threads when needed but reuse the existing threads if they are available

Executor	ExecutorService
Executor is the core interface to interact with thread pool in Java e.g. submitting task for parallel execution.	ExecutorService is an extension of Executor interface which provides facility for asynchronous execution and shutdown of pool.
Provides execute() method for submitting task.	Provides submit() method for submitting tasks.
The execute() method returns nothing.	The submit() method return Future object, which can be polled for completion of task later.
You cannot cancel the task once completed.	You can cancel the task by using Future.cancel() method.
Doesn't provide any method for shutdown().	Provides methods for shutdown() of pool.

Producer Consumer Problem

Wait, notify, notifyall methods are on object to manage communication between objects. They are not on thread as it can't be on individual thread instance. It requires lock before using them for mutually exclusive lock then operation carry out.

```
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to solve Producer Consumer problem using wait and notify
 * method in Java. Producer Consumer is also a popular concurrency design pattern.
 *
 * @author Javin Paul
 */
public class ProducerConsumerSolution {

    public static void main(String args[]) {
        Vector sharedQueue = new Vector();
        int size = 4;
        Thread prodThread = new Thread(new Producer(sharedQueue,
size), "Producer");
        Thread consThread = new Thread(new Consumer(sharedQueue,
size), "Consumer");
        prodThread.start();
        consThread.start();
    }
}
```

```
class Producer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Producer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println("Produced: " + i);
            try {
                produce(i);
            } catch (InterruptedException ex) {
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
            }

        }
    }

    private void produce(int i) throws InterruptedException {

        //wait if queue is full
        while (sharedQueue.size() == SIZE) {
            synchronized (sharedQueue) {
                System.out.println("Queue is full " + Thread.currentThread().getName()
                                    + " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();
            }
        }

        //producing element and notify consumers
        synchronized (sharedQueue) {
            sharedQueue.add(i);
            sharedQueue.notifyAll();
        }
    }
}
```

```
class Consumer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Consumer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while (true) {
            try {
                System.out.println("Consumed: " + consume());
                Thread.sleep(50);
            } catch (InterruptedException ex) {
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null,
ex);
            }

        }
    }

    private int consume() throws InterruptedException {
        //wait if queue is empty
        while (sharedQueue.isEmpty()) {
            synchronized (sharedQueue) {
                System.out.println("Queue is empty " + Thread.currentThread().getName()
                                + " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();
            }
        }

        //Otherwise consume element and notify waiting producer
        synchronized (sharedQueue) {
            sharedQueue.notifyAll();
            return (Integer) sharedQueue.remove(0);
        }
    }
}
```

Output:

```
Produced: 0
Queue is empty Consumer is waiting , size: 0
```

```
Produced: 1
Consumed: 0
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Queue is full Producer is waiting , size: 4
Consumed: 1
Produced: 6
Queue is full Producer is waiting , size: 4
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
Queue is empty Consumer is waiting , size: 0
```

5. Core Java

Primitives

byte, **short**, **int** and **long** data types are used for storing whole numbers.
float and **double** are used for fractional numbers. **char** is used for storing characters(letters). **boolean** data type is used for variables that holds either true or false.

byte: Default size of 1 byte. Can hold smallest whole number -128 to 128

short: Default size is 2 byte2. -32,768 to 327677

int: Default size: 4 bytes. Number can be of: -2,147,483,648 to 2,147,483,647

float: Sufficient for holding 6 to 7 decimal digits
size: 4 bytes

double: Default size: 8 bytes. Sufficient for holding 15 decimal digits

long: Default size: 8 bytes. it has wider range than int data type, ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

boolean: holds either true of false.

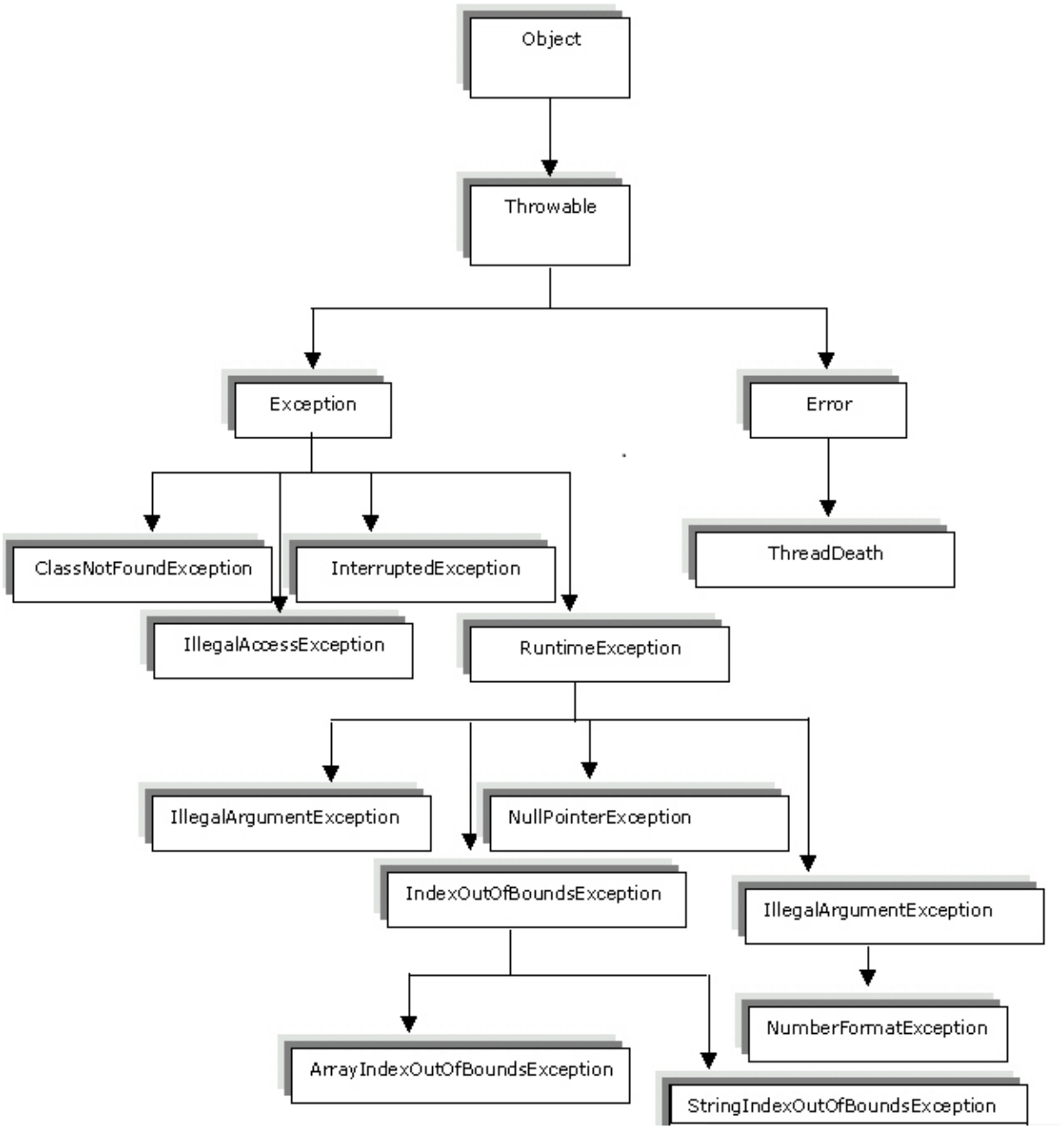
char: holds characters. size: 2 bytes

Exceptions:

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program, processing gets terminated and doesn't continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled. We will cover the handling part later in this same tutorial.

Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions)

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example, memory error, hardware error, JVM error etc.



Serialization

Serialization is a mechanism to convert an object into stream of bytes so that it can be written into a file, transported through a network or stored into database. De-serialization is just a vice versa.

```
public static void main(String args[])
```

```
 {
   Student obj = new Student(101, 25, "Chaitanya", "Agra", 6);
   try{
     FileOutputStream fos = new FileOutputStream("Student.ser");
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     oos.writeObject(obj);
     oos.close();
     fos.close();
     System.out.println("Serialzation Done!!");
  }catch(IOException ioe){
     System.out.println(ioe);
   }
 }
```

De-Serialization:

```
try{
     FileInputStream fis = new FileInputStream("Student.ser");
     ObjectInputStream ois = new ObjectInputStream(fis);
     o = (Student)ois.readObject();
     ois.close();
     fis.close();
   }
   catch(IOException ioe)
   {
      ioe.printStackTrace();
      return;
   }catch(ClassNotFoundException cnfe)
    {
      System.out.println("Student Class is not found.");
      cnfe.printStackTrace();
      return;
    }
```

Generics

T is a general Type which can be used for any Objet Type.

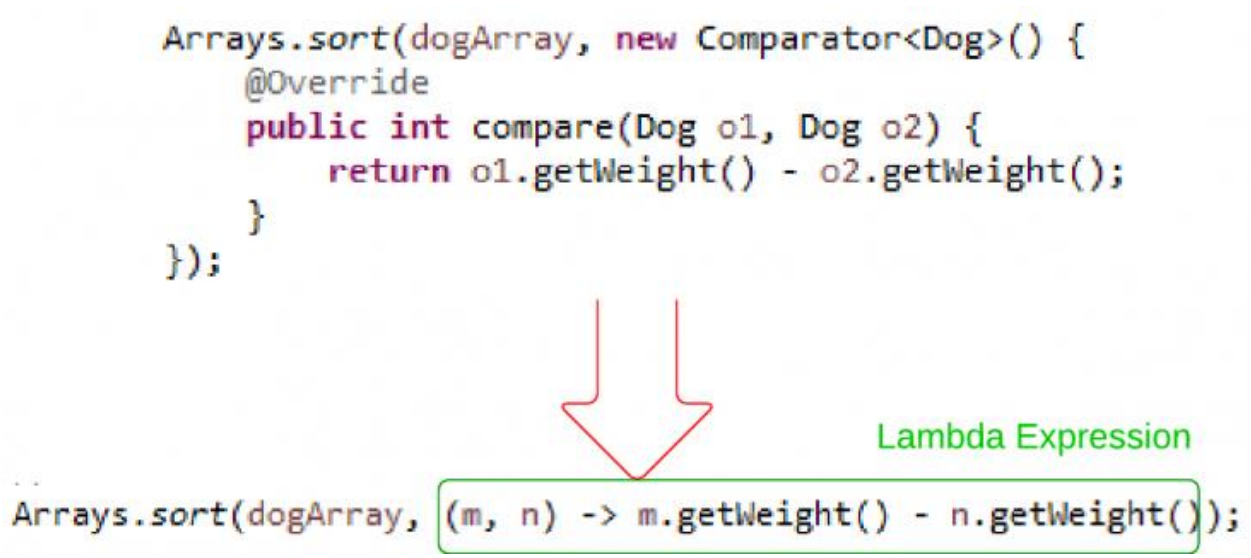
```
GenericClass<T>  means GenericClass<Integer>  and GenericClass<String>
List<E> myList;
```

E is called a type variable, namely a variable that will be replaced by a type.

1. A generic type that uses a type variable E allows you to pass E when declaring or instantiating the generic type.
2. If E is a class, you may also pass a subclass of E.
3. If E is an interface, you may also pass a class that implements E.
4. By convention, you use a single uppercase letter for type variable names.

Lambda

A lambda expression is a block of code that can be passed around to execute.



Above Lambda is inferred to Integer.compare();

```
//with type declaration
MathOperation addition = (int a, int b) -> a + b;


private int operate(int a, int b, MathOperation mathOperation){
  return mathOperation.operation(a, b);
}


System.out.println("10 + 5 = " + tester.operate(10, 5, addition));
```

Method references help to point to methods by their names. A method reference is described using **::** (double colon) symbol. A method reference can be used to point the following types of methods −

- Static methods
- Instance methods
- Constructors using new operator (TreeSet::new)

Stream

Stream represents a sequence of objects from a source, which supports aggregate operations.

Sequence of elements supporting sequential and parallel aggregate operations.

//sorted function

```
public static void main(String[] args) {
                    // create an array of dogs
                    Dog d1 = new Dog("Max", 2, 50);
                    Dog d2 = new Dog("Rocky", 1, 30);
                    Dog d3 = new Dog("Bear", 3, 40);
                    Dog[] dogArray = { d1, d2, d3 };

                    // use stream to sort
                    Stream<Dog> dogStream = Stream.of(dogArray);
                    Stream<Dog> sortedDogStream = dogStream.sorted((Dog m, Dog n) -
> Integer.compare(m.getHeight(), n.getHeight()));

                    sortedDogStream.forEach(d -> System.out.print(d));
          }
```

//filter function

```
//filter function
Stream<String> stream = list.stream().filter(p -> p.length() >
String[] arr = stream.toArray(String[]::new);
```

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
```

```
List<String> filtered = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.toList());
```

//For each:

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

//map

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
//get list of unique squares
List<Integer> squaresList = numbers.stream().map( i ->
i*i).distinct().collect(Collectors.toList());
```

Limit – limit size of the stream

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

sorted – sort the stream. Below shows how to print 10 random numbers in a sorted order.

```
random.ints().limit(10).sorted().forEach(System.out::println);
```

ParallelStream:

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
//get count of empty string
int count = strings.parallelStream().filter(string -> string.isEmpty()).count();
```

Collectors – combine the result of processing on the elements of a stream.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);
String mergedString = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.joining(", "));
```

```
System.out.println("Merged String: " + mergedString);
```

Statistics:

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());
System.out.println("Lowest number in List : " + stats.getMin());
System.out.println("Sum of all numbers : " + stats.getSum());
System.out.println("Average of all numbers : " + stats.getAverage());
```

Common API

java.lang.String

common Constructors:

String(byte[] b)

String(String s)

String(char[] b)

String(StringBuffer sf)

String(StringBuilder sb)

Common Methods:

Char charAt(int index)

int compareTo(String anotherStirng)

int compareToIgnoreCase(String anotherString)

boolean contains(CharSequence s)

String[] split(expression)

java.util.StringTokenizer

```
StringTokenizer st = new StringTokenizer(str);

while (st.hasMoreElements()) {

                        System.out.println(st.nextElement());

                }
```

```
String Str = new String("Welcome-to-Tutorialspoint.com");
    System.out.println("Return Value :" );
    for (String retval: Str.split("-")) {
      System.out.println(retval);
    }
```

```
String message = String.join("-", "Java", "is", "cool");
    // message returned is: "Java-is-cool"
```

If we again write **str = new String(“very”)**, then it will create a new object with value “very”, rather than pointing to the available objects in heap area with same value.But if we write **str = “very”**,then it will point to String constant object with value “very”,present in String pooled area.

1. equals() method: It compares **values** of string for equality. Return type is boolean. In almost all the situation you can use useObjects.equals().
2. == operator: It compares **references not values**. Return type is boolean. == is used in rare situations where you know you’re dealing with interned strings.
3. compareTo() method: It compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.For example,if str1 and str2 are two string variables then if
str1==str2: return0
str1>str2: returnpositive
str1 < str2: return a negative value
Note: The positive and negative values returned by **compareTo** method is the difference of first unmatched character in the two strings

```
String str5 = new String("abc");
String str6 = new String("abc");
System.out.println("str5 == str6 => are NOT equals");
```

```
String str5_i = new String("abc").intern();
String str6_i = new String("abc").intern();
System.out.println("str5_i == str5_i => are equals");

String str7 = "abc";
String str8 = "abc";
```

```
if(str7 == str8)
System.out.println("str7 == str8 => are equals");
```

```
char[] sArray = s1.toCharArray();
```

```
s.charAt(i-1)
```

```
StringBuffer s=new StringBuffer("GeeksforGeeks");
s.delete(0,5);
System.out.println(s); //returns forGeeks
s.deleteCharAt(7);
System.out.println(s); //returns forGeek
```

Java.lang.Character

Maximum Occurance of characters:

```
for (int i=1; i<=n; i++)
            {
               for (int j=1; j<=n; j++)
               {
                  // If characters match and indexes are not same
                  if (strArray[i-1] == strArray[j-1] && i!=j)
                     dp[i][j] =  1 + dp[i-1][j-1];
                  // If characters do not match
                  else
                     dp[i][j] = Math.max(dp[i][j-1], dp[i-1][j]);
               }
               System.out.println("i ="+  i );
               printArray(dp);
```

Palindrome

String is equal in characters – either way you read it

```
boolean isPalindrome(String str) {
   int n = str.length();
   for( int i = 0; i < n/2; i++ )
      if (str.charAt(i) != str.charAt(n-i-1)) return false;
   return true;
}


private boolean isPalindrome(String s) {
   int length = s.length();

   if (length < 2) // If the string only has 1 char or is empty
      return true;
   else {
      // Check opposite ends of the string for equality
      if (s.charAt(0) != s.charAt(length - 1))
         return false;
      // Function call for string with the two ends snipped off
      else
         return isPalindrome(s.substring(1, length - 1));
   }
}
```

java.lang.String

common Constructors:

String(byte[] b)

String(String s)

String(char[] b)

String(StringBuffer sf)

String(StringBuilder sb)

Common Methods:

Char charAt(int index)

int compareTo(String anotherStirng)

int compareToIgnoreCase(String anotherString)

boolean contains(CharSequence s)

java.lang.Integer

parseInt(String s)

java.util.StringTokenizer

java.util.Arrays

asList(T... a)

```
StringTokenizer st = new StringTokenizer(str);

while (st.hasMoreElements()) {

                    System.out.println(st.nextElement());
```

Longest # for subsequence – dynamic programming with multi dimestion

6. Data Structures
Array

1) What's an array?

- An arrangement of items at equally spaced addresses in computer memory. Elements are inserted and retrieved by accessing an index. Insert and lookup time by index is O(1) aka "Big O of 1"

2) What can you tell me about the memory allocation of an array?

- It is a continuous block of memory that is statically allocated with a specific size.

3) If I have an array of primitive integers of capacity five, how many bytes of memory did I just allocate on the heap?

- 20 bytes (4bytes per primitive integer x 5 elements)
    - Make sure the candidate clarifies how they arrived at their answer.
    - The candidate may mistake the size of a primitive integer with 1 byte or 2 bytes or 8 bytes. I give them half credit if they still multiply that number by 5.
    - The candidate may add some bytes to the 20 bytes because they are accounting for object allocation or the array size. Just let them know that they don't have to worry about that for the question.

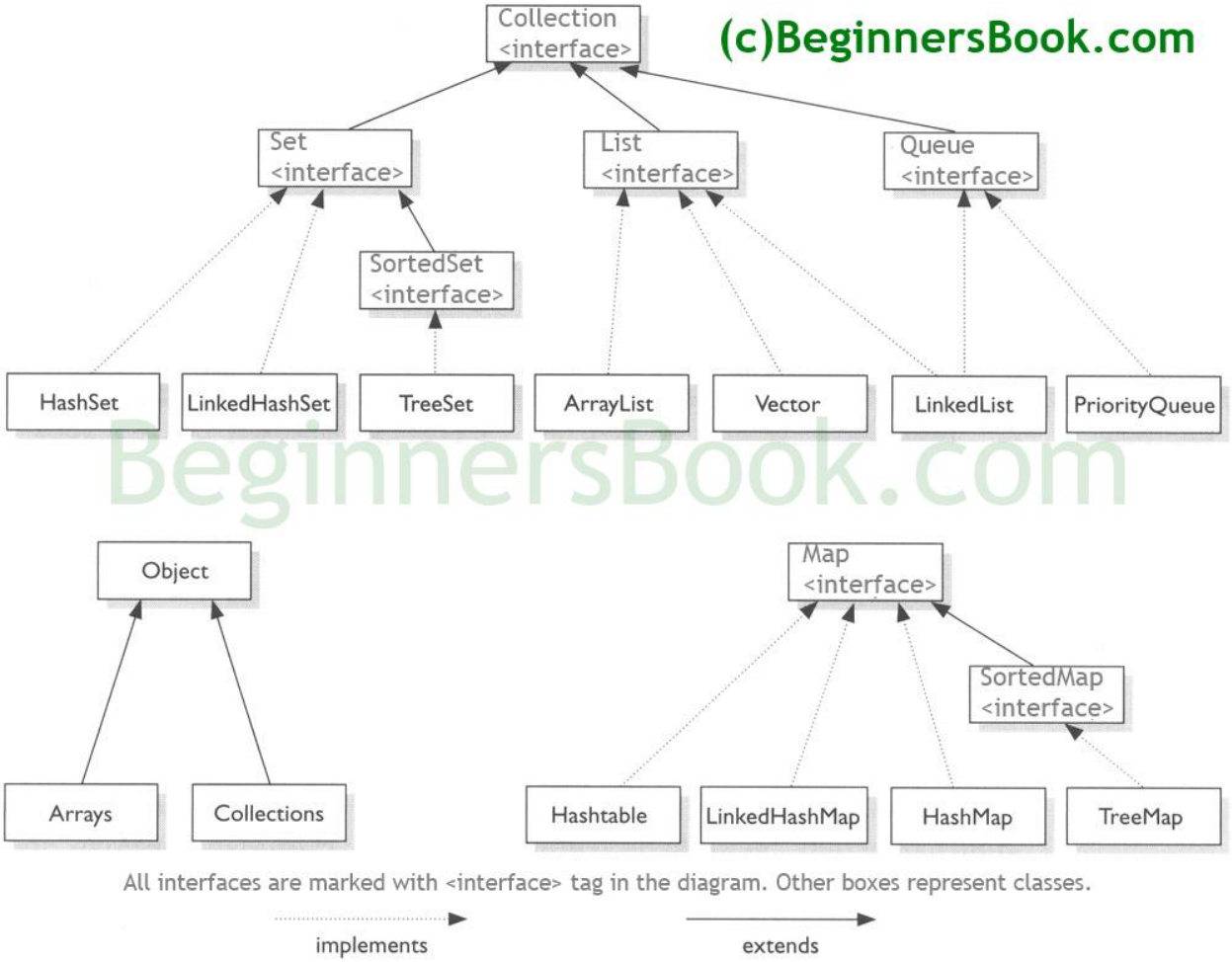
4) If I want to retrieve index number two from this array, what are the operations in memory that are done?

- Take the head pointer and add SIZE_OF_INT x 2, and read the next SIZE_OF_INT bytes.
    - Ideally, SIZE_OF_INT should be 4 bytes, but if they did not answer the previous question properly, they should still get half credit for this question for this question if they plugged in whatever value they thought is the SIZE_OF_INT
    - The candidate will most likely mistake "index number two" for the second element, which is not correct. They should get half credit if they make this mistake.

5) If I have an array that holds five strings, how many bytes of memory did I just allocate on the heap?

- 20 bytes on 32bit architecture or 40 bytes on 64bit architecture (SIZE_OF_REFERENCE x 5 elements)

- The candidate may ask how long each string is. You should tell them, no strings have been allocated.
- If the candidate assumes 32bit architecture, ask them if it will be the same for 64bit architecture.



List: Ordered collection. May contain duplicates. Inserted and accessed by their position using index.

ArrayLIst, LinkedList, Vector

Set: does not contain duplicates. HashSet, TreeSet and LInkedHashSet

HashSet, which stores its elements in a **hash table**, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.

TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.

LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

Map: key value pairs. Cannot contain duplicates.HashMap, TreeMap, and LinkedHashMap.
HashMap: it makes no guarantees concerning the order of iteration
TreeMap: It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap.
LinkedHashMap: It orders its elements based on the order in which they were inserted into the set (insertion-order).

What are the changes in Collections framework in Java 8?

There are several changes in Collections Framework in Java 8 mostly influenced by the inclusion of Lambda expression in Java 8 -

- **Stream API** - Stream can be obtained for Collection via the stream() and parallelStream() methods;

As exp if you have a list of cities in a List and you want to remove the duplicate cities from that list it can be done as

```
cityList = cityList.stream().distinct().collect(Collectors.toList());
```

- **ForEach loop** which can be used with Collection. **Spliterators** which are helpful in parallel processing where several threads can iterate/process part of the collection.
- New methods are added to Collections like replaceAll, getOrDefault, putIfAbsent in Map.
- HashMap, LinkedHashMap and ConcurrentHashMap.implementation is changed to reduce hash collisions. Instead of linked list a balanced tree is used to store the entries after a certain threshold is reached.

- **HashMap changes in Java 8**
- Though HashMap implementation provides constant time performance O(1) for get() and put() method but that is in the ideal case when the Hash function distributes the objects evenly among the buckets.
- **But the performance may worsen in the case hashCode() used is not proper and there are lots of hash collisions. As we know now that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst case scenario the complexity becomes O(n).**
- To address this issue in Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger

than a certain threshold, the hash will change from using a linked list to a balanced tree, this will improve the worst case performance from O(n) to O(log n).

-

HashMap

- Put is called with two parameters, a key and a value.
- A hash is taken from the key by calling its hashCode method.
- Convert the value returned by the hashCode (a primitive integer) and mod it by the size of the underlying array.
- Take the index returned by the previous operation and jump to that index in the array.
- If a value does not exist, add a Map.Entry object containing the key and value and add it to this location.
- If a value does exist, scan each Map.Entry to see if the new key equals any key that is currently in the chain.
- If it matches any key, replace the value in that Map.Entry, else add the new Map.Entry to the end of the chain.
    - The candidate might summarize the last three steps and call it “chaining” which is fine.
    - If the candidate refers to the underlying array as “a bucket”, ask him to define a bucket by asking him what kind of data structure is a bucket. If he doesn’t say array, he is incorrect – no partial credit.

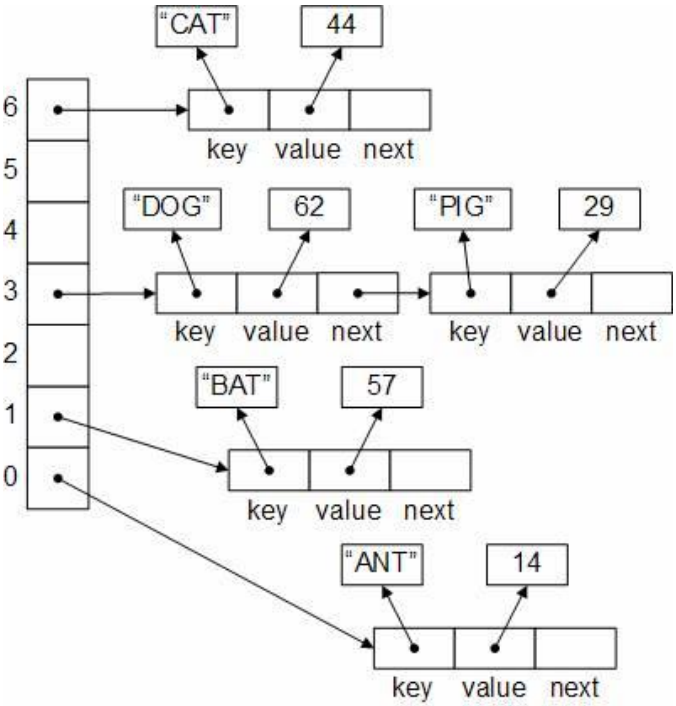
- **HashMap** works on the principal of hashing.
- **Map.Entry interface** - This interface gives a map entry (key-value pair). HashMap in Java stores both key and value object, in bucket, as an object of Entry class which implements this nested interface Map.Entry.
- **hashCode()** -HashMap provides put(key, value) for **storing** and get(key) method for **retrieving** Values from HashMap. When put() method is used to store (Key, Value) pair, HashMap implementation **calls hashcode** on Key object to calculate a hash that is used to find a bucket where Entry object will be stored. When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.
- **equals()** - equals() method is used to **compare objects for equality**. In case of HashMap key object is used for comparison, also using equals() method Map knows how to handle **hashing collision** (hashing collision means more than one key having the same hash value, thus assigned to the same bucket. In that case objects are stored in a linked list, refer figure for more clarity. Where hashCode method helps in finding the bucket where that key is stored, equals

method helps in finding the right key as there may be more than one key-value pair stored in a single bucket.

** Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0, table[1] as bucket1 and so on.

And the general contract of hashCode is -

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- **If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.**
- It is not required that **if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results**. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.



LinkedHashMap

LinkedHashMap is also one of the implementation of the Map interface, apart from implementing Map interface LinkedHashMap also extends the HashMap class. So just like HashMap, **LinkedHashMap also allows one null key and multiple null values**.

How it differs from other implementations of the Map interface like HashMap and TreeMap is that *LinkedHashMap maintains the insertion order of the elements which means if we iterate a LinkedHashMap we'll get the keys in the order in which they were inserted in the Map*. Note that insertion order is not affected if a key is re-inserted into the map (*if a same key in inserted more than once the last time it was inserted will be taken into account*).

```
Map<String, String> cityTemperatureMap = new LinkedHashMap<String, String>();

cityTemperatureMap.put("Delhi", "24");
        cityTemperatureMap.put("Mumbai", "32");
        cityTemperatureMap.put(null, "26");


// iterating the map
        for(Map.Entry<String, String> me : cityTemperatureMap.entrySet()){
            System.out.println(me.getKey() + " " + me.getValue());
        }
```

Points to note here

- It can be seen that the insertion order is maintained. While iterating the LinkedHashMap elements are displayed in the order they were inserted.
- Even though two null keys were inserted only one is stored (only one null key is allowed in map).
- Two values were inserted with the same key "Chennai" which results in the overwriting of the old value with the new value (as the calculated hash will be the same for the keys). Thus the last one is displayed while iterating the values.

LinkedHashMap can be used to Least Recently Used cache –

Where access order is passed in the constructor.

```
Map<String, String> cityMap = new LinkedHashMap<String, String>(16, 0.75f, true);
```

Here it can be seen that **access-order** parameter is passed as true, that means ordering mode is Access order, if passed as **false** that would mean ordering mode is insertion order.

LinkedHashMap is not thread-safe, is fail fast (if structurally modified at any time after the iterator is created then the iterator will throw **ConcurrentModificationExcpetion**.

TreeMap

TreeMap is also one of the implementation of the Map interface like HashMap and LinkedHashMap.

How it differs from other implementations of the Map interface like HashMap and LinkedHashMap is that **objects in TreeMap are stored in sorted order**. The elements are ordered using their natural ordering or a comparator can be provided at map creation time to provide custom ordering

```
Map<String, String> cityTemperatureMap = new TreeMap<String, String>();


cityTemperatureMap.put("Kolkata", "28");
        cityTemperatureMap.put("Chennai", "36");


        // iterating the map
        for(Map.Entry<String, String> me : cityTemperatureMap.entrySet()){
            System.out.println(me.getKey() + " " + me.getValue());
        }
```

Though HashMap and LinkedHashMap allow one null as key, **TreeMap doesn't allow null as key**. Any attempt to add null in a TreeMap will result in a **NullPointerException**.

TreeMap is not thread-safe, is fail fast (if structurally modified at any time after the iterator is created then the iterator will throw **ConcurrentModificationExcpetion**.

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));



//in case of using custom sorter;
Map<String, String> cityTemperatureMap = new TreeMap<String, String>(new TreeComparator());
```

```
//Comparator class
class TreeComparator implements Comparator<String>{
 @Override
 public int compare(String str1, String str2) {
    return str2.compareTo(str1);
 }
```

ArrayList vs LinkedList

ArrayList is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array.

LinkedList is implemented as a double linked list. Its performance on add and remove is better than Arraylist, but worse on get and set methods.

ArrayList is a better choice if your program is thread-safe. Vector and ArrayList require more space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50% of its size each time. LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc.

Singly list has limitations which are overcome by Doubly list:

Singlylist: insertion at the front in O(1), insertion at other position O(n), need reference to previous node to insert next node, can only traverse in forward direction.

Circular list- in doubly linked list – point last node's next to first, and first node's previous to last.

Can always traverse back and forth, but drawback is infinite loop.

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

ArrayList has:

O(n) time complexity for arbitrary indices of add/remove,

but O(1) for the operation at the end of the list.

LinkedList has:

O(n) time complexity for arbitrary indices of add/remove, but O(1) for operations at end/beginning of the List.

How does ArrayList grow dynamically

When we add an element to an ArrayList it first verifies whether it has that much capacity in the array to store new element or not, in case there is not then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that much capacity (Actually uses Arrays.copyOf which returns the original array increased to the new length).

What happens when element is removed

When elements are removed from an ArrayList using either remove(int i) (i.e using index) or remove(Object o), in the underlying array the gap created by the removal of an element has to be filled. That is done by Shifting any subsequent elements to the left (subtracts one from their indices). **System.arrayCopy** method is used for that.

Singly Linked List

```
class SinglyLinkedList {

	private Node head;

	/*
	 * Add new Node
	 */
	public void addNode(int d){
		Node newNode = new Node(d);
		if(head == null){
			head = newNode;
			return;
		}
		Node current = head;
		while(current.next != null){
			current = current.next;
		}
		current.next=newNode;
	}

	public int legnthOfList(){
		int count = 0;
		if(head == null)  //assume it's head node or empty list
			return count;
		else if (head.next == null)
			return 1;
		else
		{
			count =1;
			Node current = head;
			while(current.next != null){
				count +=1;
				current = current.next;
			}
		}
		return count;
	}

	public int legnthOfListRecusively(){
		return LengthRecursively(head);
	}
	public int LengthRecursively(Node node){
		int count = 0;
		if(node == null)  //assume it's head node or empty list
```

```
                return count;
        else if (node.next == null)
                return 1;
        else
                count = 1 + LengthRecursively(node.next);
        return count;
}
/**
 * Print singly link list in the natual order
 */
public void printLinkedList(){
        if(head == null)
                return;
        Node current = head;
        while(current != null){
                System.out.println("Data: " + current.data);
                current = current.next;
        }
}

public void reverseAndPrintLinkedList(){
        Node newHead = reverseLinkedList();
        head=newHead;
        printLinkedList();
}

public Node reverseLinkedList()
{
        Node currentNode = head;
        // For first node, previousNode will be null
        Node previousNode=null;
        Node nextNode;
        while(currentNode!=null){
                nextNode=currentNode.next;
                //reversing the link
                currentNode.next=previousNode;
                //moving currentNode and previousNode by 1 node
                previousNode=currentNode;
                currentNode=nextNode;
        }
        return previousNode;
 }

public void middleNodeinList(){
        Node middle = findMiddleNode();
        System.out.println("Middle Node is: " + middle.data);
```

```
}

public Node findMiddleNode(){
 Node slowPointer, fastPointer;
 slowPointer = fastPointer = head;
          while(fastPointer !=null) {
                   fastPointer = fastPointer.next;
                   if(fastPointer != null && fastPointer.next != null) {
                            slowPointer = slowPointer.next;
                            fastPointer = fastPointer.next;
                   }
          }
          return slowPointer;
}

public Node nthFromLastNode(int n)
 {
  Node firstPtr=head;
  Node secondPtr=head;

  for (int i = 0; i < n; i++) {
   firstPtr=firstPtr.next;

  }

  while(firstPtr!=null)
  {
   firstPtr=firstPtr.next;
   secondPtr=secondPtr.next;
  }

  return secondPtr;
 }

// Reverse linked list in pair
public void reverseLinkedListInPairs() {
          Node currentNode = head;
          /* Traverse only till there are atleast 2 nodes left */
    while (currentNode != null && currentNode.next != null) {
          /* Swap the data */
        int k = currentNode.data;
        currentNode.data = currentNode.next.data;
        currentNode.next.data = k;
        currentNode = currentNode.next.next;
    }
```

```
        }
}
```

Doubly Linked List

```
public void addFirst(int d){
		DNode node = new DNode(d);
		if(head == null){
			head = node;
			head.next=null;
			head.prev=null;
		}
		else{
			node.next = head;
			head.prev = node;
			head=node;
		}
	}

	/**
	 * Iterate though current.next != null then just add newNode as curr.next, set prev link.
	 * @param d
	 */
	public void addLast(int d){
		DNode node = new DNode(d);
		if(head == null){
			head = node;
			head.next=null;
			head.prev=null;
		}
		else{
			DNode current = head;
			while(current.next != null){
				current = current.next;
			}
			current.next = node;
			node.prev = current;
		}
	}

	  public void reverse() {
	DNode temp = null;
	DNode current = head;
	/* swap next and prev for all nodes of
	 doubly linked list */
	while (current != null) {
	    temp = current.prev;
	    current.prev = current.next;
```

```
            current.next = temp;
            current = current.prev;
        }
        /* Before changing head, check for the cases like empty
         list and list with only one node */
        if (temp != null) {
            head = temp.prev;
        }
    }
```

Remove Duplicates From linked list:

Keep 2 pointers—one for outer loop and 2nd for inner loop (p2 = p1.next)

```
/* Function to remove duplicates from an
       unsorted linked list */
    void remove_duplicates() {
        Node ptr1 = null, ptr2 = null, dup = null;
        ptr1 = head;

        /* Pick elements one by one */
        while (ptr1 != null && ptr1.next != null) {
            ptr2 = ptr1;

            /* Compare the picked element with rest
                of the elements */
            while (ptr2.next != null) {

                /* If duplicate then delete it */
                if (ptr1.data == ptr2.next.data) {

                    /* sequence of steps is important here */
                    dup = ptr2.next;
                    ptr2.next = ptr2.next.next;
                    System.gc();
                } else /* This is tricky */ {
                    ptr2 = ptr2.next;
                }
            }
            ptr1 = ptr1.next;
        }
    }
```

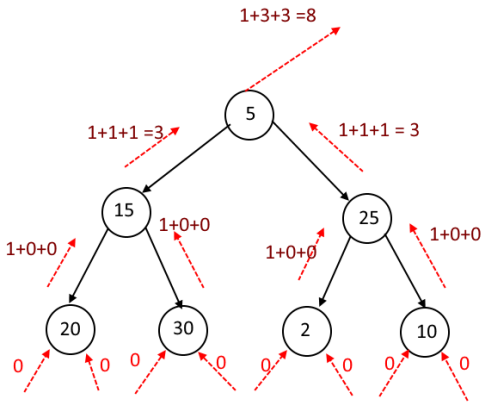
Tree

```
public int heightOfBinaryTree(Node node)
{
   if (node == null)
   {
      return 0;
   }
   else
   {
      return 1 +
      Math.max(heightOfBinaryTree(node.left),
         heightOfBinaryTree(node.right));
   }
}
```

Given a (Input) Binary tree, Find the size of the tree.

- Very Simple solution
- Start from the root.
- Size = 1 (for the root) + Size Of left Sub-Tree + Size Of right Sub-Tree
- solve the left sub-tree and right sub-tree recursively.

Time Complexity : O(n)



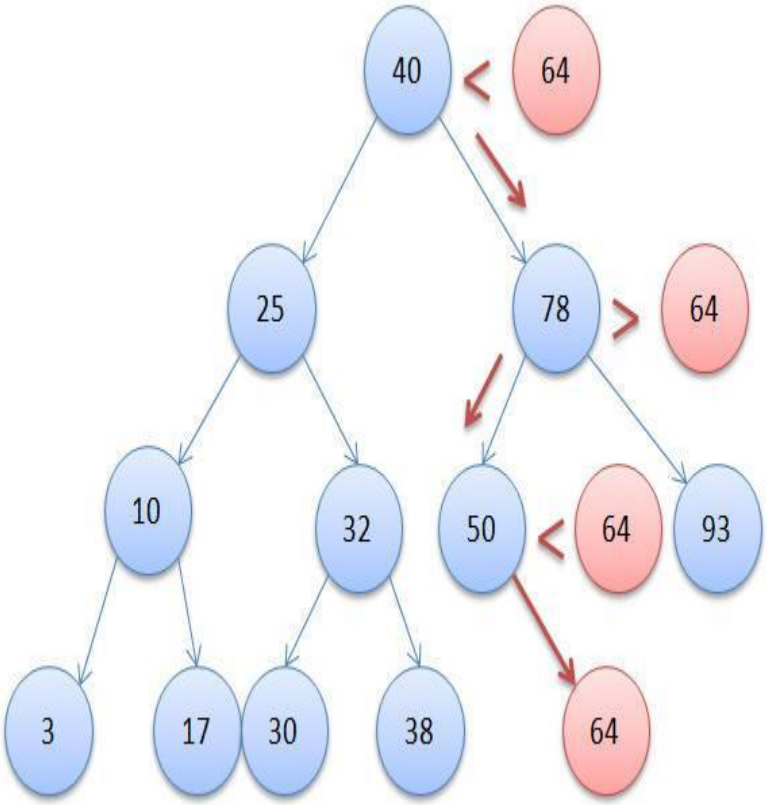
```
public
static void
main(String
args[]){
                Node root = new Node(5);
                root.left = new Node(15);
                root.right = new Node(25);
                root.left.left = new Node(20);
                root.left.right = new Node(30);
                root.right.left = new Node(2);
                root.right.right = new Node(10);

                SizeofTree t = new SizeofTree();
                System.out.println("Size of the Tree is : " + t.getSize(root));
        }
```

```
public int
getSize(Node
root){
                if(root==null){
                        return 0;
                }
                return 1 + getSize(root.left) + getSize(root.right);
        }
```

Binary Search Tree

The basic idea is that at each node we compare with the value being inserted.

If the value is lesser then we traverse through the left sub tree and if the value is greater we traverse through the right subtree




```
public class Node<T> {
  public int value;
  public Node left;
  public Node right;

  public Node(int value) {
    this.value = value;
  }

                        }
```

```
public class BinarySearchTree {
  public Node root;

  public void insert(int value){
    Node node = new Node<>(value);

    if ( root == null ) {
      root = node;
      return;
      }

      insertRec(root, node);

  }

  private void insertRec(Node latestRoot, Node node){

    if ( latestRoot.value > node.value){

      if ( latestRoot.left == null ){
        latestRoot.left = node;
        return;
      }
      else{
        insertRec(latestRoot.left, node);
      }
    }
    else{
      if (latestRoot.right == null){
        latestRoot.right = node;
        return;
      }
```

```
      else{
        insertRec(latestRoot.right, node);
      }
    }
  }
}
```

```
/**
 * Returns the minimum value in the Binary Search Tree.
 */
public int findMinimum(){
  if ( root == null ){
    return 0;
  }
  Node currNode = root;
  while(currNode.left != null){
    currNode = currNode.left;
  }
  return currNode.value;
}

/**
 * Returns the maximum value in the Binary Search Tree
 */
public int findMaximum(){
  if ( root == null){
    return 0;
  }

  Node currNode = root;
  while(currNode.right != null){
    currNode = currNode.right;
  }
  return currNode.value;
}
```

Heap – (min Heap, max Heap)
Graph Data Structure
Trie

Collections – Common API

Arrays:

```
// Sorting int Array in descending order
Arrays.sort(intArray, Collections.reverseOrder());
```

7. Sorting Algorithm

Algorithm	Data structure	Time complexity:Best	RunTime
Merge sort	Array	O(n log(n))	
Heap sort	Array	O(n log(n))	
Smooth sort	Array	O(n)	
Bubble sort	Array	O(n)	O(n2)
Insertion Sort	Array	O(n2)	

Bubble Sort I Runtime: 0(n2) average and worst case. Memory: 0(1) .
In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater
than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly"bubble" up to the beginning of the list.

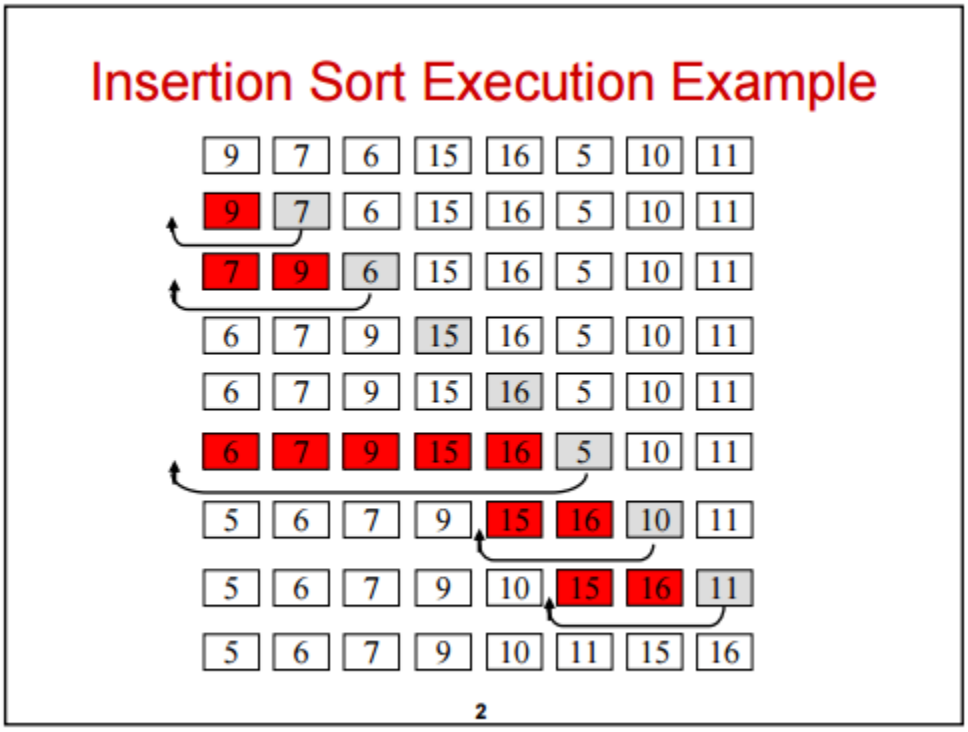
Selection Sort I Runtime: 0(**n2) average and worst case. Memory:** 0(1) .
Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan
and move it to the front (swapping it with the front element). Then, find the second smallest and move it,
again, doing a linear scan. Continue doing this until all the elements are in place.

Merge Sort I Runtime: 0 (n log (n)) average and worst case. Memory: Depends.
Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single element arrays. It is the "merge" part that does all the heavy lifting.

Insertion Sort

Insertion Sort – sort an array like a playing cards in hand. – pick a one and iterate through each to find the correct slot.

Algorithm
// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]
Time Complexity: O(n*n)
Auxiliary Space: O(1)



// Java program for implementation of Insertion Sort
class InsertionSort

```
{
    /*Function to sort array using insertion sort*/
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i=1; i<n; ++i)
        {
            int key = arr[i];
            int j = i-1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j>=0 && arr[j] > key)
            {
                arr[j+1] = arr[j];
                j = j-1;
            }
            arr[j+1] = key;
        }
    }

    /* A utility function to print array of size n*/
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6};

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);

        printArray(arr);
    }
} /* This code is contributed by Rajat Mishra. */
```

Quick Sort:

Quick Sort: Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

Quick Sort I **Runtime:** O(n log(n)) **average,** O(n2) **worst case. Memory:** 0(log(n)). In quick sort we pick a random element and partition the array, such that all numbers that are less than the
partitioning element come before all elements that are greater than it. The partitioning can be performed
efficiently through a series of swaps (see below).

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become
sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the
median), our sorting could be very slow. This is the reason for the 0(n2) worst case runtime.

```
void quickSort(int[] arr, int left, int right) {
int index = partition(arr, left, right);
if (left< index - 1) { // Sort left half
quickSort(arr, left, index - 1);
}

if (index< right) { // Sort right half
 quickSort(arr, index, right);
 }
 }

 int partition(int[] arr, int left, int right) {
 int pivot = arr[(left + right) / 2]; // Pick pivot point
 while (left<= right) {
 // Find element on left that should be on right
while (arr[left] < pivot) left++;

// Find element on right that should be on left
 while (arr[right] > pivot) right--;

// Swap elements, and move left and right indices
 if (left<= right) {
 swap(arr, left, right); // swaps elements
 left++;
 right--;
 }
 }
```

```
 return left;
 }
```

Merge Sort:

Merge Sort: divide and Conquer – algorithm to divide an array into 2 halves and continue dividing until element is 1. Then merge the result.

Time : O(n log n)

Space: O(n)

MergeSort(arr[], l, r)
If r > l
1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

The merge method operates by copying all the elements from the target array segment into a helper array,
keeping track of where the start of the left and right halves should be (he lperleft and he lperRight).
We then iterate through helper, copying the smaller element from each half into the array. At the end, we
copy any remaining elements into the target array.

```
void mergesort(int[] array) {
        int[] helper = new int[array.length];
        mergesort(array, helper, 0, array.length - 1);
 }

void mergesort(int[] array, int[] helper, int low, int high) {
        if (low< high) {
        int middle = (low + high)/ 2;
}

        mergesort(array, helper, low, middle); // Sort left half
        mergesort(array, helper, middle+l, high); // Sort right half
```

```
        merge(array, helper, low, middle, high); // Merge them
}


void merge(int[] array, int[] helper, int low, int middle, int high) {
 /* Copy both halves into a helper array*/
 for (int i= low; i <= high; i++) {
 helper[i] = array[i];
 }

int helperleft = low;
int helperRight =middle + l;
int current = low;

/* Iterate through helper array. Compare the left and right half, copying back
 * the smaller element from the two halves into the original array. */

 while (helperLeft <= middle && helperRight <= high) {
 if (helper[helperleft] <= helper[helperRight]) {
 array[current] = helper[helperleft];
 helperleft++;
 } else {//If right element is smaller than left element
 array[current] = helper[helperRight];
 helperRight++;
 }
 current++;
 }
/* Copy the rest of the left side of the array into the target array*/
 int remaining= middle - helperleft;
for (int i= 0; i <= remaining; i++) {
 array[current + i] = helper[helperleft + i];
 }
}
```

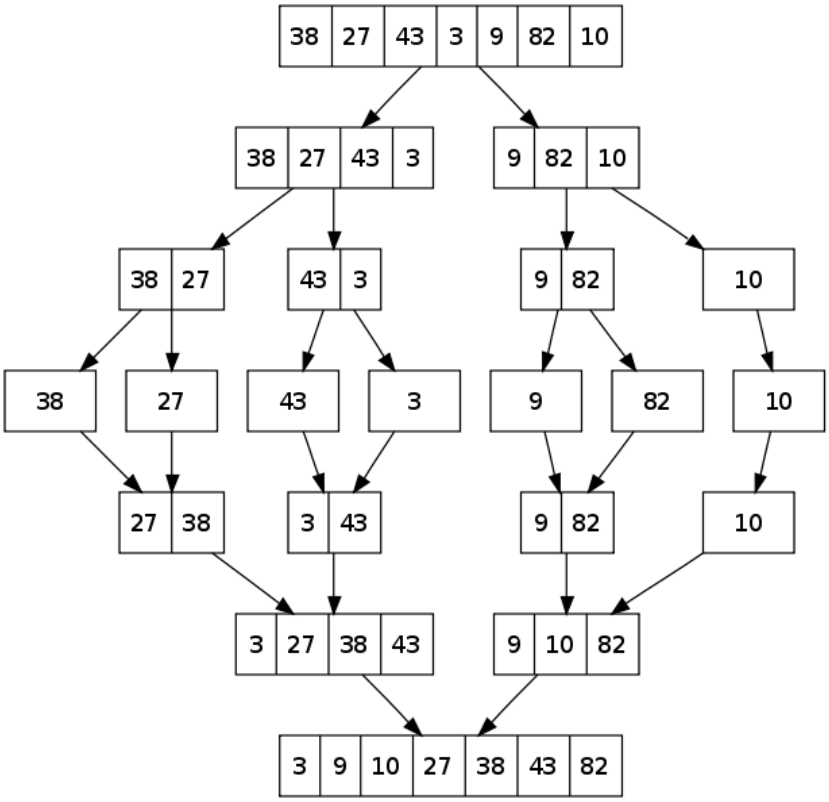
Best for Linked List merge as we don't have to add extra space due to random memory for list vs. array;s continuous memory location.

```
/* Java program for Merge Sort */
class MergeSort
{
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
```

```
int n1 = m - l + 1;
int n2 = r - m;

/* Create temp arrays */
int L[] = new int [n1];
int R[] = new int [n2];

/*Copy data to temp arrays*/
for (int i=0; i<n1; ++i)
    L[i] = arr[l + i];
for (int j=0; j<n2; ++j)
    R[j] = arr[m + 1+ j];


/* Merge the temp arrays */

// Initial indexes of first and second subarrays
int i = 0, j = 0;

// Initial index of merged subarry array
int k = l;
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}

/* Copy remaining elements of L[] if any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

/* Copy remaining elements of R[] if any */
while (j < n2)
```

```
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr , m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

// Driver method
public static void main(String args[])
{
    int arr[] = {12, 11, 13, 5, 6, 7};

    System.out.println("Given Array");
    printArray(arr);

    MergeSort ob = new MergeSort();
    ob.sort(arr, 0, arr.length-1);

    System.out.println("\nSorted array");
```

```
        printArray(arr);
    }
}
```

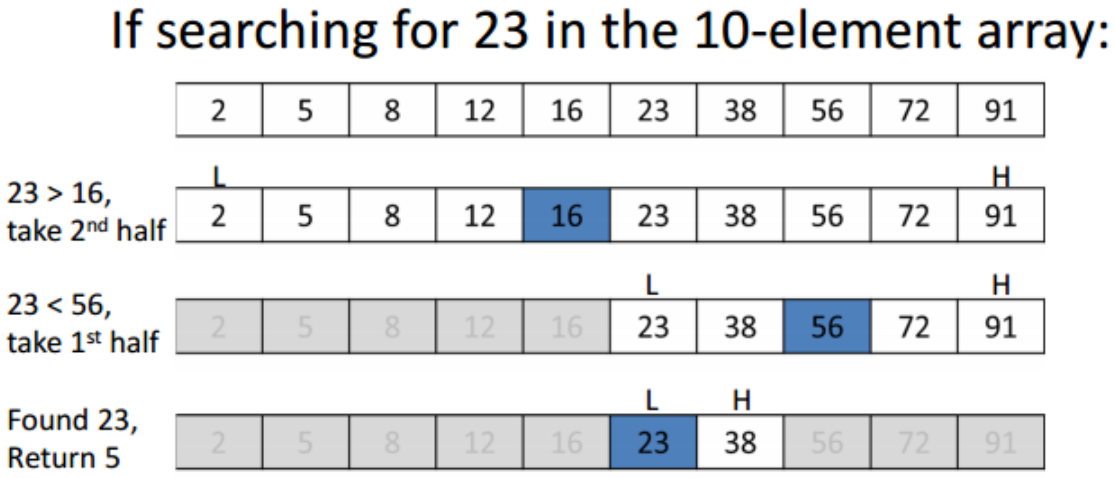


Binary Search

Binary Search = repeatedly dividing the search interval in half.

Time Complexity of a Linear Search is O(n).

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Logn).




```
// Java implementation of recursive Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[l..r], else
    // return -1
    int binarySearch(int arr[], int l, int r, int x)
    {
        if (r>=l)
        {
            int mid = l + (r - l)/2;

            // If the element is present at the middle itself
            if (arr[mid] == x)
               return mid;

            // If element is smaller than mid, then it can only
            // be present in left subarray
            if (arr[mid] > x)
               return binarySearch(arr, l, mid-1, x);

            // Else the element can only be present in right
```

```
            // subarray
            return binarySearch(arr, mid+1, r, x);
        }

        // We reach here when element is not present in array
        return -1;
    }

    // Driver method to test above
    public static void main(String args[])
    {
        BinarySearch ob = new BinarySearch();
        int arr[] = {2,3,4,10,40};
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr,0,n-1,x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "+result);
    }
}
```

8. Recursion

Recursion is a basic programming technique you can use in Java, in which a method calls itself to solve some problem

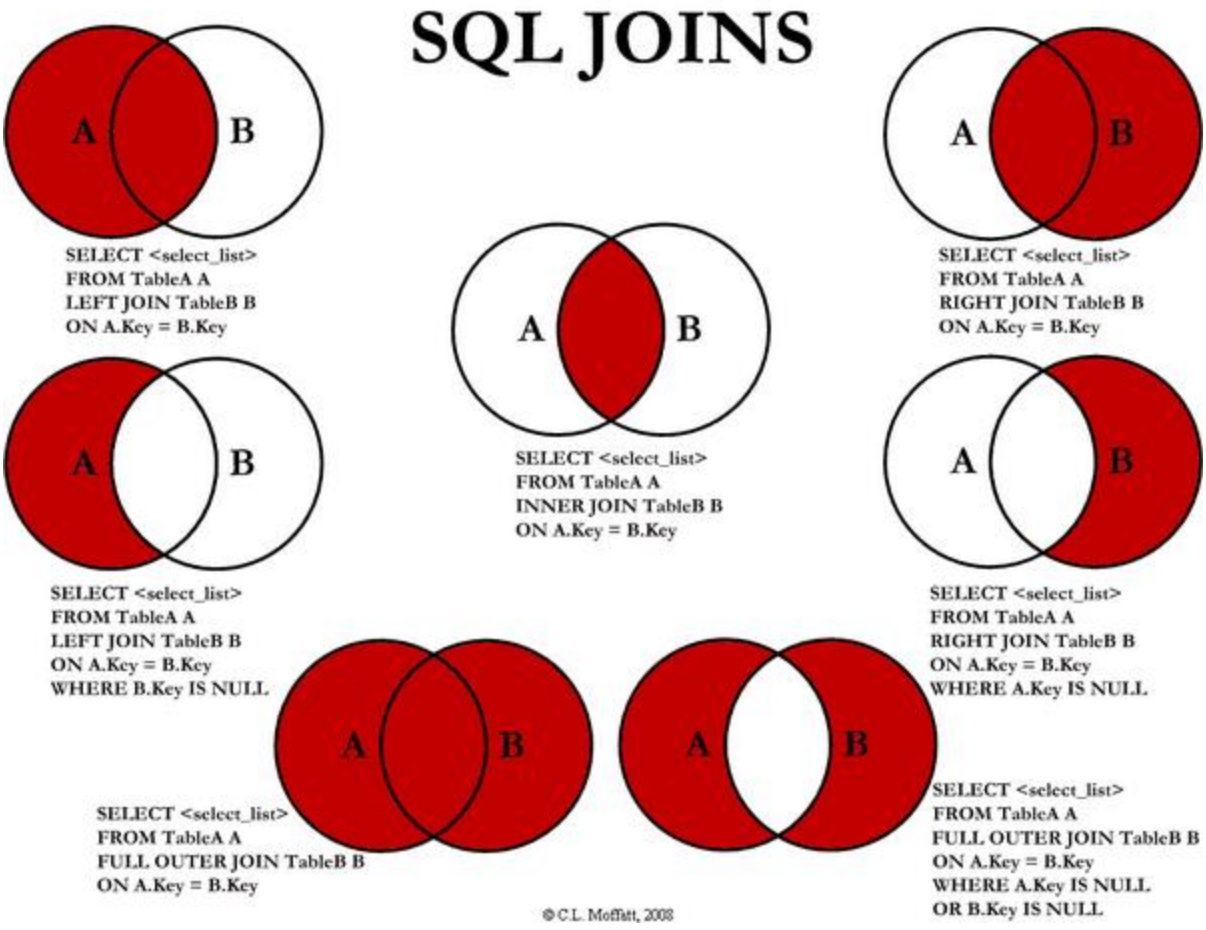
```
private static long factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

9. Database

A left outer join retains all of the rows of the left table, regardless of whether there is a row that matches on the right table. (employee left join location -> gives all from employee)

A right outer join is pretty much the same thing as a left outer join, except that the rows that are retained are from the right table

The difference between an inner join and an outer join is that an inner join will return ***only*** the rows that actually match based on the join predicate



Primary Key vs Unique (or candidate) key:

Primary key Is unique to identify a row in a table. It can be only 1 key.

Unique can be multiple but there can't be clustered index on them and only non-clustered can exist

ACID:

- **Atomicity:** A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency:** This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

- **Isolation:** There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.
- **Durability:** Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

Truncate – delete all rows of table, without specifying WHERE clause. removes data by deallocating space used by a table which removes a lot of overhead in terms of logging and locking and that's why to truncate is faster than delete. Can't Rollback data.

DELETE: takes where clause, operation can be rolled back, also fires triggers. DISADVANTAGE is logging and locking, and so speed.

1. truncate is fast, delete is slow.
2. truncate doesn't do logging delete logs on per row basis.
3. rollback is possible with delete not with truncate until specifically supported by the vendor.
4. truncate doesn't fire trigger, delete does.
5. Don't delete, truncate it when it comes to purge tables.
6. truncate reset identity column in table if any, delete doesn't.
7. truncate is DDL while delete is DML (use this when you are writing exam)
8. truncate doesn't support where clause, delete does.

Database Transactions: Connection includes five values to represent isolation using JDBC.

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	Not supported	*Not applicable*	*Not applicable*	*Not applicable*
TRANSACTION_READ_COMMITTED	Supported	Prevented	Allowed	Allowed

TRANSACTION_READ_UNCOMMITTED	**Supported**	**Allowed**	**Allowed**	**Allowed**
TRANSACTION_REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
TRANSACTION_SERIALIZABLE	**Supported**	**Prevented**	**Prevented**	**Prevented**

A *non-repeatable read* occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.

A *phantom read* occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.

Spring JDBC

Create a DataSource, and then create springjdbctemplate by passing datasource in the constructor.

Then run sql to insert/update/query on the jdbctemplate.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

  <!-- Initialization for data source -->
  <bean id="dataSource"
     class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
     <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
     <property name="username" value="root"/>
     <property name="password" value="password"/>
  </bean>

  <!-- Definition for studentJDBCTemplate bean -->
  <bean id="studentJDBCTemplate"
     class="com.tutorialspoint.StudentJDBCTemplate">
     <property name="dataSource"  ref="dataSource" />
  </bean>

</beans>
```

JDBC:

- Register the driver class
- Creating a connection
- Creating Statement
- Executing queries
- Closing a connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcConnection {

    public static void main(String a[]){

        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection( url: "jdbc:oracle:thin:@<hostname>:<port num>:<DB name>"
                            , user: "user", password: "password");
            Statement stmt = con.createStatement();
            System.out.println("Created DB Connection....");
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

SQL Exception: is a checked exception.

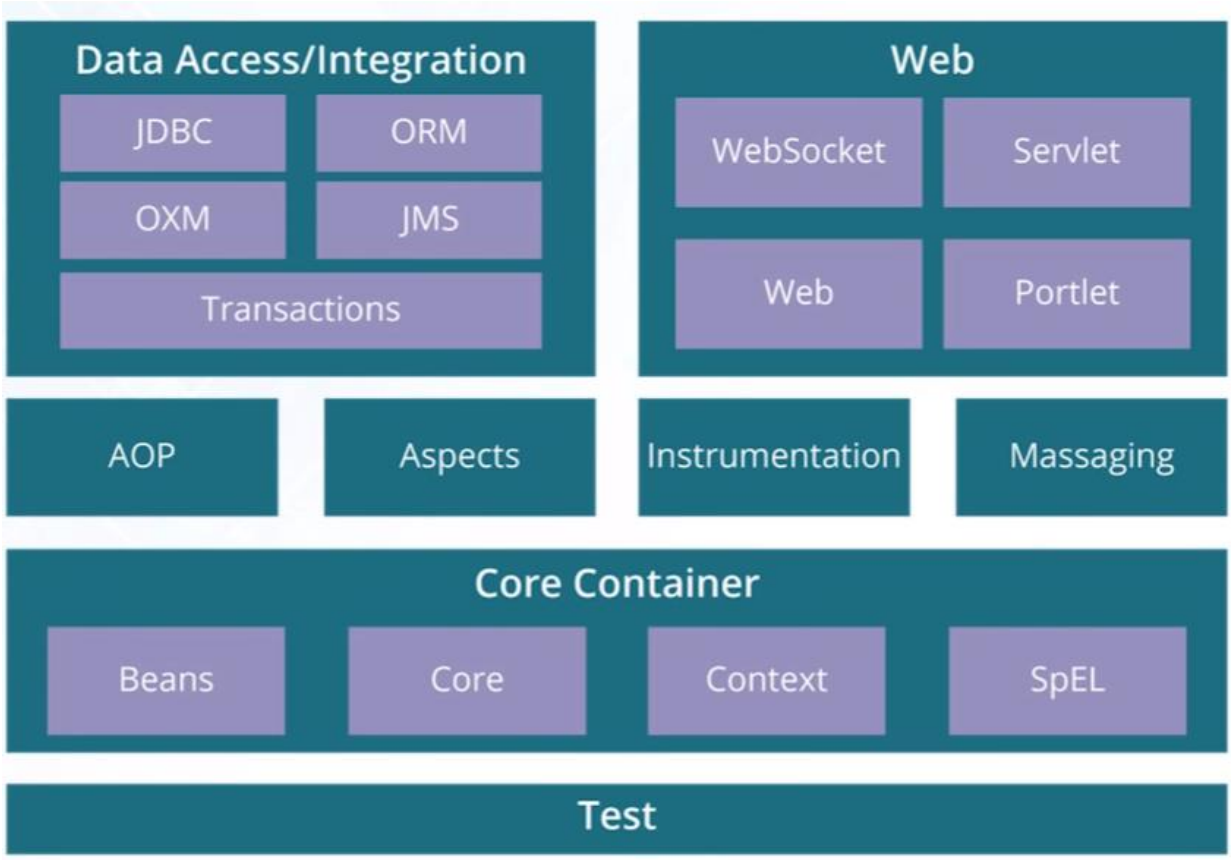
Question: given an Employee table with EmpId, Salary. Find 2nd row with maximum salary.

Find managers, subordinate that report to manager. Find count of reporting employee for a given manager.

10. Spring

Spring Framework: Light weight framework, loose coupling, integrated framework to address complexity of the enterprise application development. Lightweight, IOC, AOP, container, MVC, transaction management.

IOC, Dependency Injection: higher level components DO Not depend on smaller components. Smaller/lower level components are injects/created to the higher level as dependency. IOC container creates those objects (wire/configure via xml file) and handed over. Container manages complete Lifecyle from construction till destruction.



Bean Factory	Application Context
1. Uses Lazy initialization	1. Uses Eager/Aggressive initialization
2. Explicitly provided a resource object using the syntax	2. Creates and manages resources objects on its own
3. Doesn't supports internationalization	3. Supports internationalization
4. Annotation based dependency Injection is not supported	4. Annotation based dependency Injection is supported

@Component:

@Controller

@Repository: classes to represent Persistence Layer

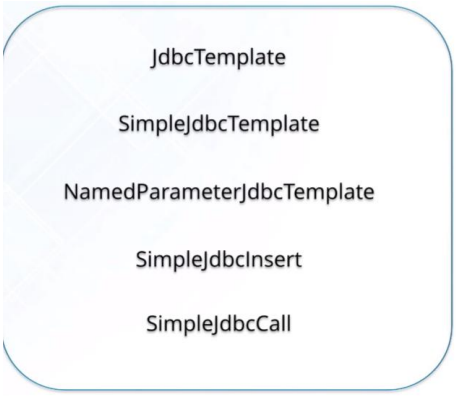
@Service:

@Required: required during injection in the spring config.

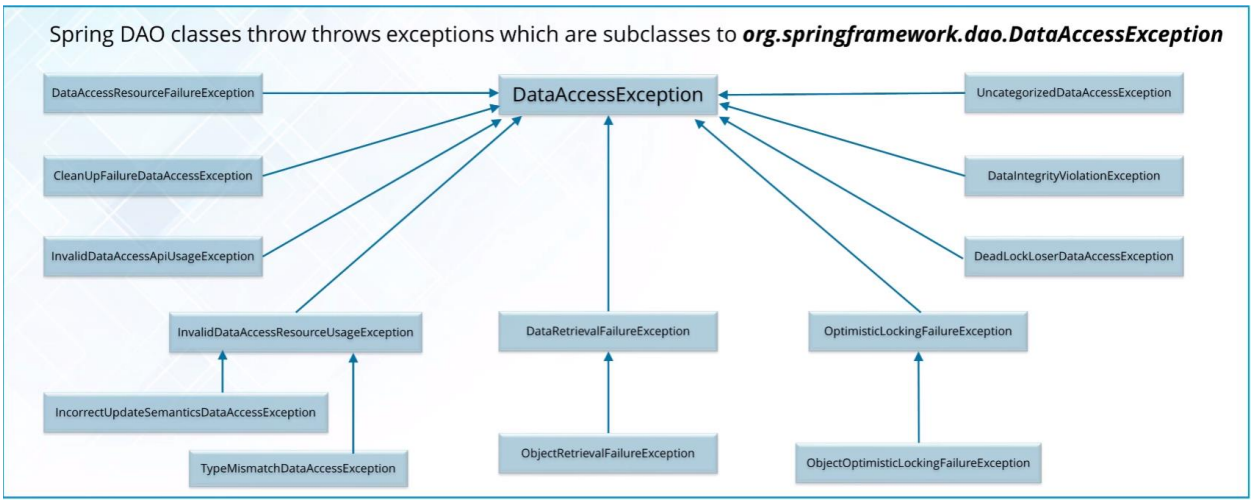
@Autowired:

@Qualifier:

@RequestMapping: to map incoming HTTP requests with method.



JDBC API



Spring AOP:

Aspect: concern with to weave in.

Join Point: A point during execution of a program such as execution of a method or handling of the exception.

Advice: Action taken by an aspect at a particular join point. Such as: around, before, after

Spring AOP vs AspectJ:

Spring AOP: **Runtime** weaving through **Proxy**. Support only **method** level PointCut. DTD Based.

AspectJ: compile time weaving through AspectJ, Support Field Level pointcuts. Schema based and Annotation configuration.

11. Hadoop

12. REST

Representational State Transfer. – architectural style for designing network applications.

Stateless, client-server protocol.

Whenever you ask for a resource from REST API and provide http header "**accept: application/json**", you will get back the json representation of resource.

@RestController = @Controller + @ResponseBody

```
@RestController
public class EmployeeRESTController
{

 @RequestMapping(value = "/employees")
    public EmployeeListVO getAllEmployees()
    {
    }
  @RequestMapping(value = "/employees/{id}")

   public ResponseEntity<EmployeeVO> getEmployeeById (@PathVariable("id") int id)
   {
   }
}
}
```

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

13. JSON/Protobuff

JSON stands for **JavaScript Object Notation**. JSON objects are used for transferring data between server and client.

JSON data: It basically has key-value pairs:

```
var chaitanya = {
   "firstName" : "Chaitanya",
   "lastName" : "Singh",
   "age" :  "28"
};
```

- It is light-weight
- It is language independent and so scalable with diff platform (Ruby, Python,Perl, Java, Ajax etc)
- Easy to read and write
- Text based, human readable data exchange format
- Standard Structure
- JSON Parser available to marshal and unmarshal data.

XML is lot more verbose with elements & attribute tagging.

14. Design Patterns

Creational:

Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder; Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton: Ensure a class only has one instance, and provide a global point of access to it.

Structural:

Adapter: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge: Decouple an abstraction from its implementation so that the two can vary independently.

Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Façade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

FlyWeight: Use sharing to support large numbers of fine-grained objects efficiently.

Proxy: Provide a surrogate or placeholder for another object to control access to it.

Behavioral:

Command Pattern: Encapsulate a request as an object, thereby allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.

Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Interpretor: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento Pattern: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

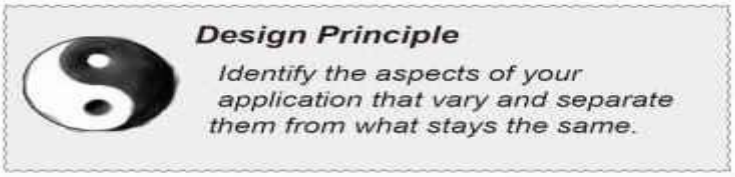
State: allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. It is quite similar to the State pattern.

Template: 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.

Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Core Design Patterns:

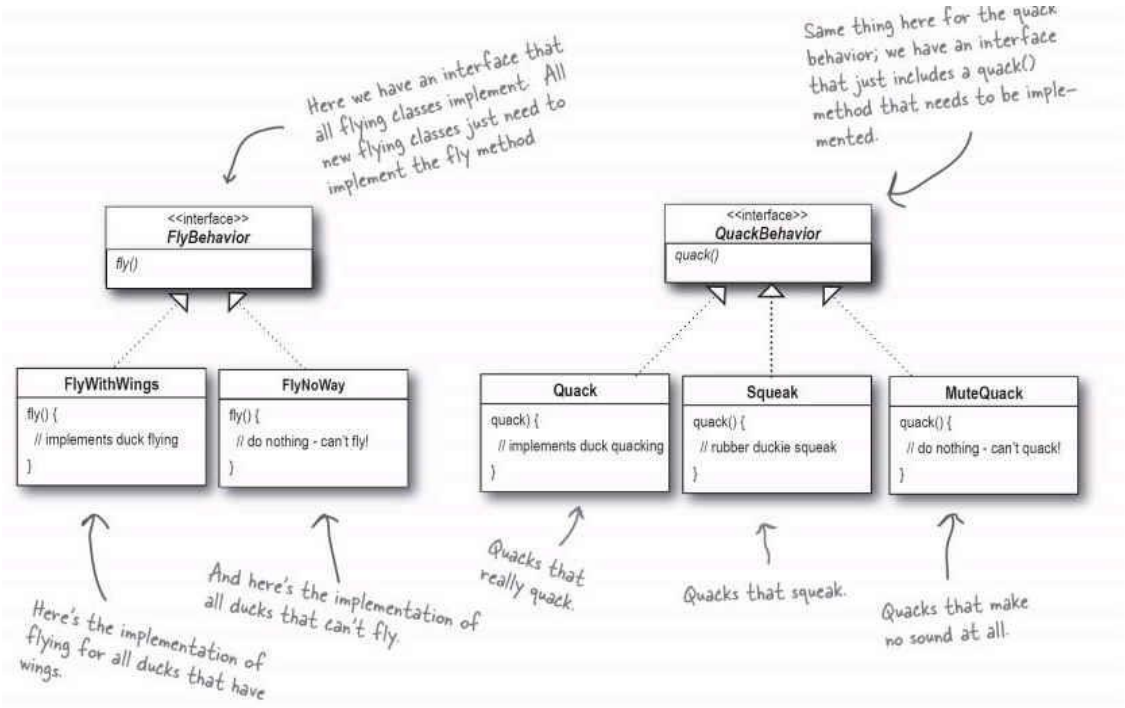


Our first of many design principles. We'll spend more time on these thruoghout the book.

Take what varies and "encapsulate" it so it won't affect the rest of your code.

The result? Fewer unintended consequences from code changes and more flexibility in your systems!

Here's 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.***

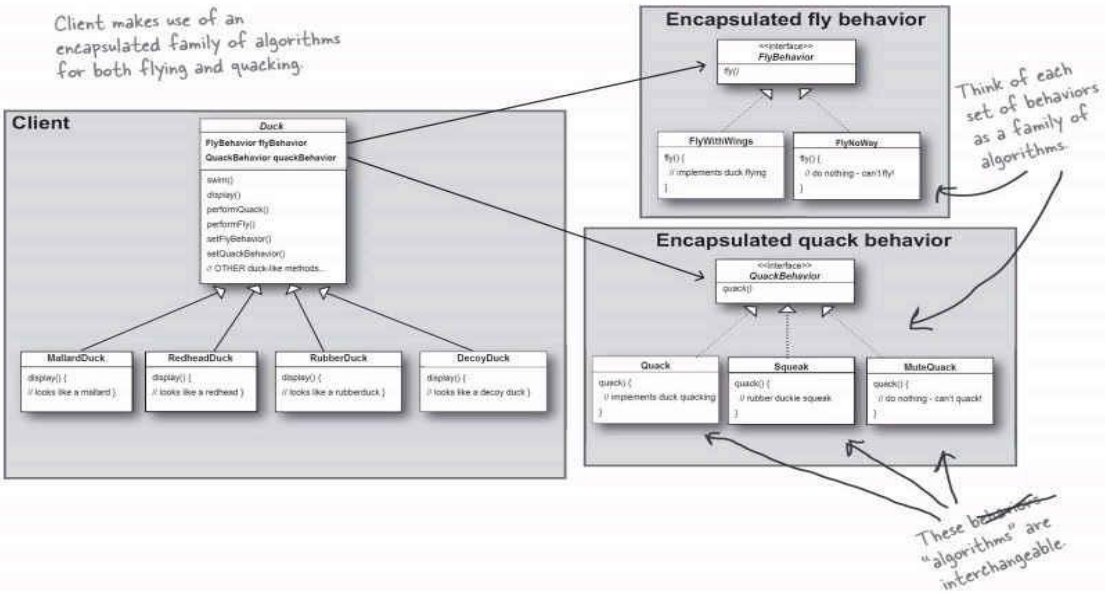


With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes!

And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that *use* flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with inheritance.

Pay careful attention to the *relationships* between the classes. In fact, grab your pen and write the appropriate relationship (IS-A, HAS-A and IMPLEMENTS) on each arrow in the class diagram.



You just applied your first design pattern—the STRATEGY pattern. That's right, you used the Strategy Pattern to rework the SimUDuck app. Thanks to this pattern, the simulator is ready for any changes those execs might cook up on their next business trip to Vegas.

Now that we've made you take the long road to apply it, here's the formal definition of this pattern:

> **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Use THIS definition when you need to impress friends and influence key executives.

HAS-A can be better than IS-A

The HAS-A relationship is an interesting one: each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.

When you put two classes together like this you're using ***composition***. Instead of *inheriting* their behavior, the ducks get their behavior by being *composed* with the right behavior object.

This is an important technique; in fact, we've been using our third design principle:

Design Principle

Favor composition over inheritance.

As you've seen, creating systems using composition gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you ***change behavior at runtime*** as long as the object you're composing with implements the correct behavior interface.

Composition is used in many design patterns and you'll see a lot more about its advantages and disadvantages throughout the book.

in your OO toolbox; let's make a list of them before we move on to Chapter 2.

- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.
- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers.

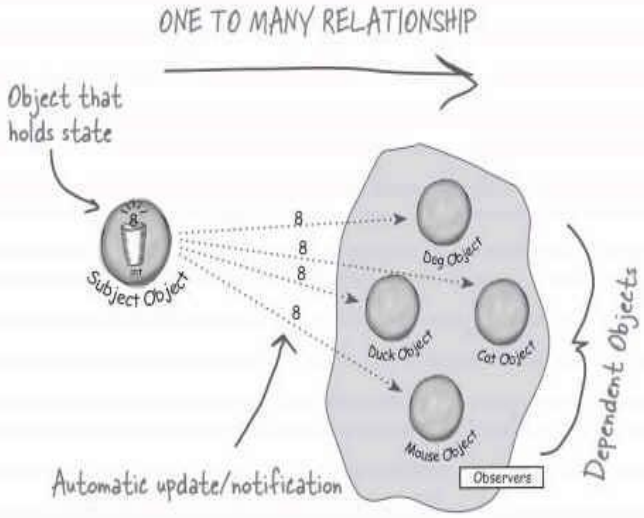
Sharpen your pencil

Which of the following are disadvantages of using subclassing to provide specific Duck behavior? (Choose all that apply.)

- ☑ A. Code is duplicated across subclasses.
- ☑ B. Runtime behavior changes are difficult.
- ☐ C. We can't make duck's dance.
- ☑ C. Hard to gain knowledge of all duck behaviors.
- ☐ D. Ducks can't fly and quack at the same time.
- ☑ E. Changes can unintentionally affect other ducks.

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Let's relate this definition to how we've been talking about the pattern:

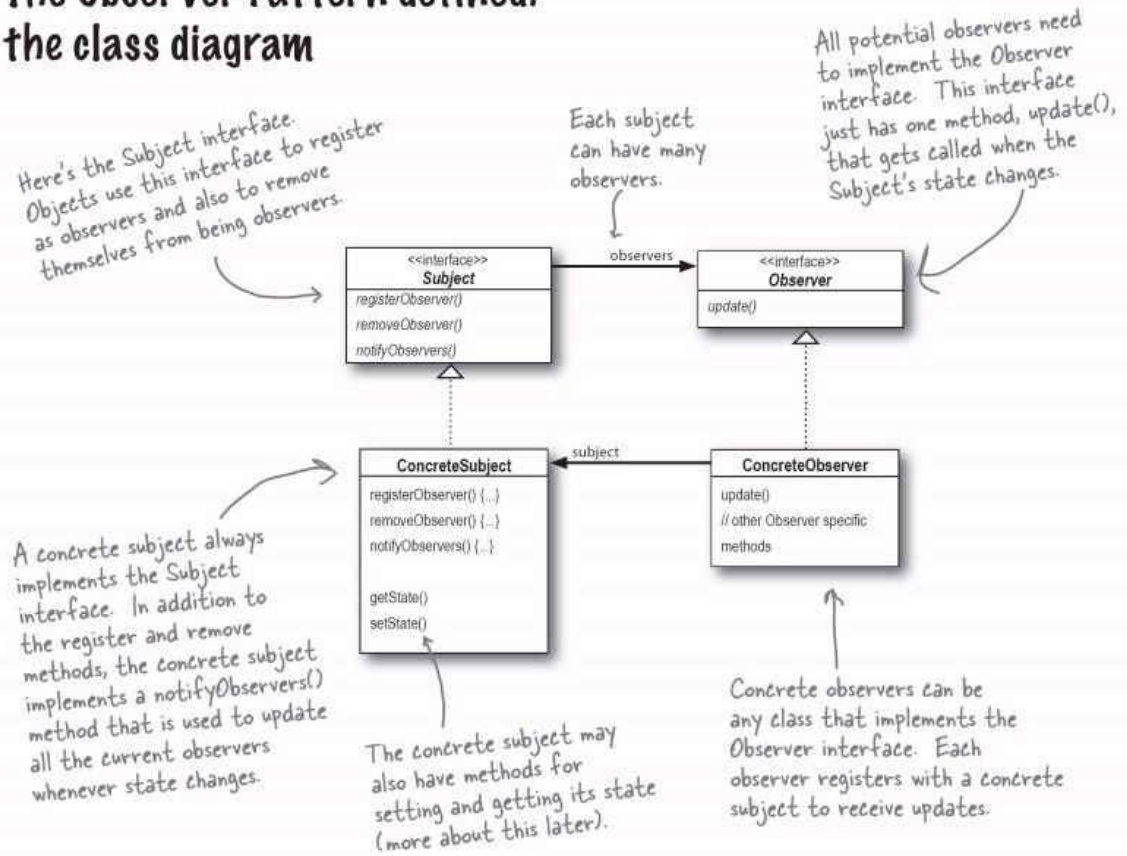


The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.

The Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

The Observer Pattern defined: the class diagram



Q: What does this have to do with one-to-many relationships?

A: With the Observer pattern, the Subject is the object that contains the state and controls it. So, there is ONE subject with state. The observers, on the other hand, use the state, even if they don't own it. There are many observers and they rely on the Subject to tell them when its state changes. So there is a relationship between the ONE Subject to the MANY Observers.

Q: How does dependence come into this?

A: Because the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

The power of Loose Coupling

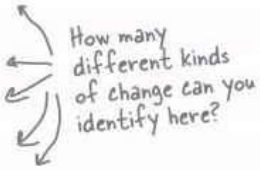
When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Why?

The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.



We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type, all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.

Design Principle

Strive for loosely coupled designs between objects that interact.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

there are no Dumb Questions

Q: Is update() the best place to call display?

A: In this simple example it made sense to call display() when the values changed. However, you are right, there are much better ways to design the way the data gets displayed. We are going to see this when we get to the model-view-controller pattern.

Q: Why did you store a reference to the Subject? It doesn't look like you use it again after the constructor?

A: True, but in the future we may want to un-register ourselves as an observer and it would be handy to already have a reference to the subject.

The Open-Closed Principle

Grasshopper is on to one of the most important design principles:

Design Principle

Classes should be open for extension, but closed for modification.

Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.

Sorry, we're *closed*. That's right, we spent a lot of time getting this code correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

Okay, here's what we know so far...

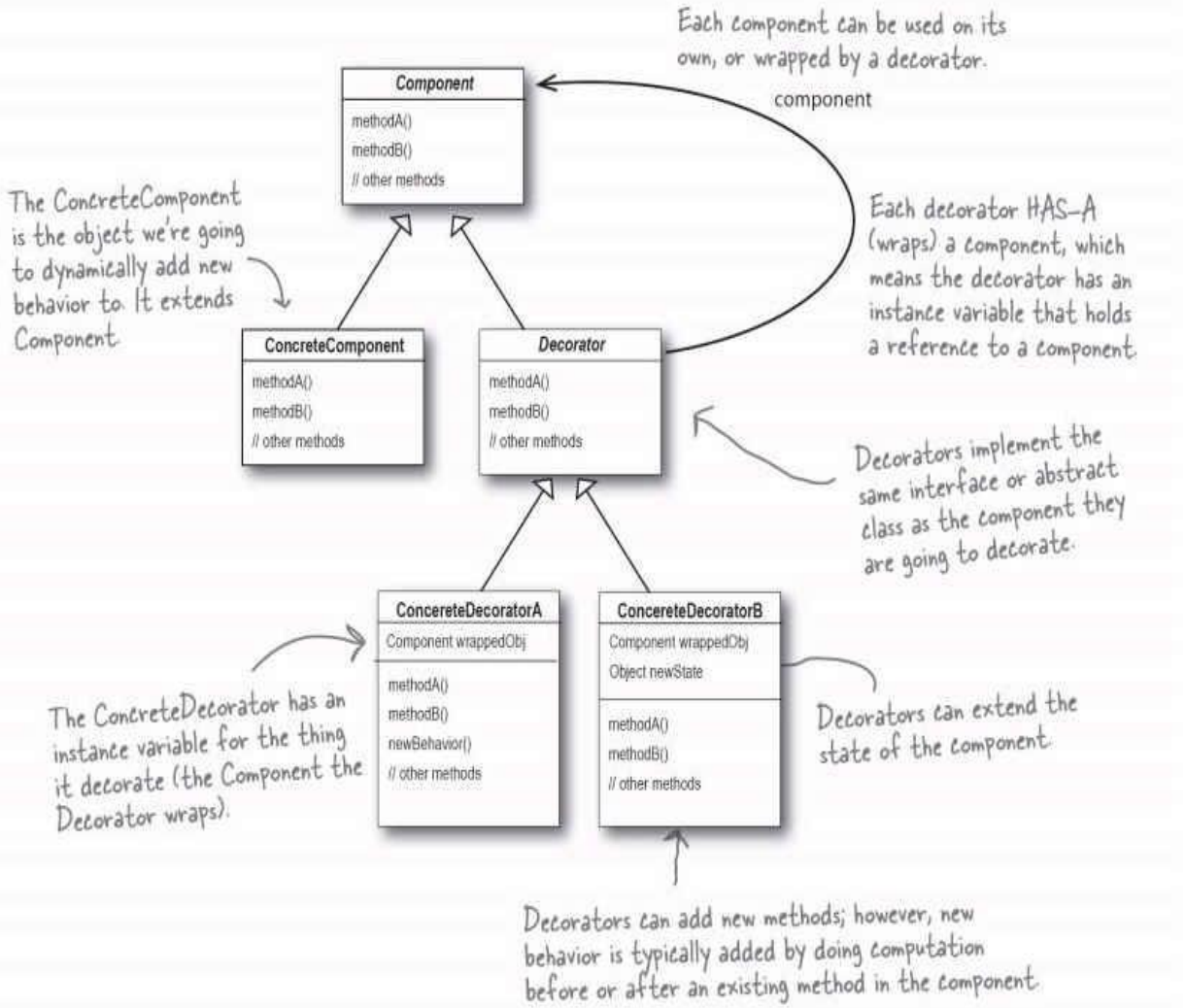
- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Key Point!

Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code.

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

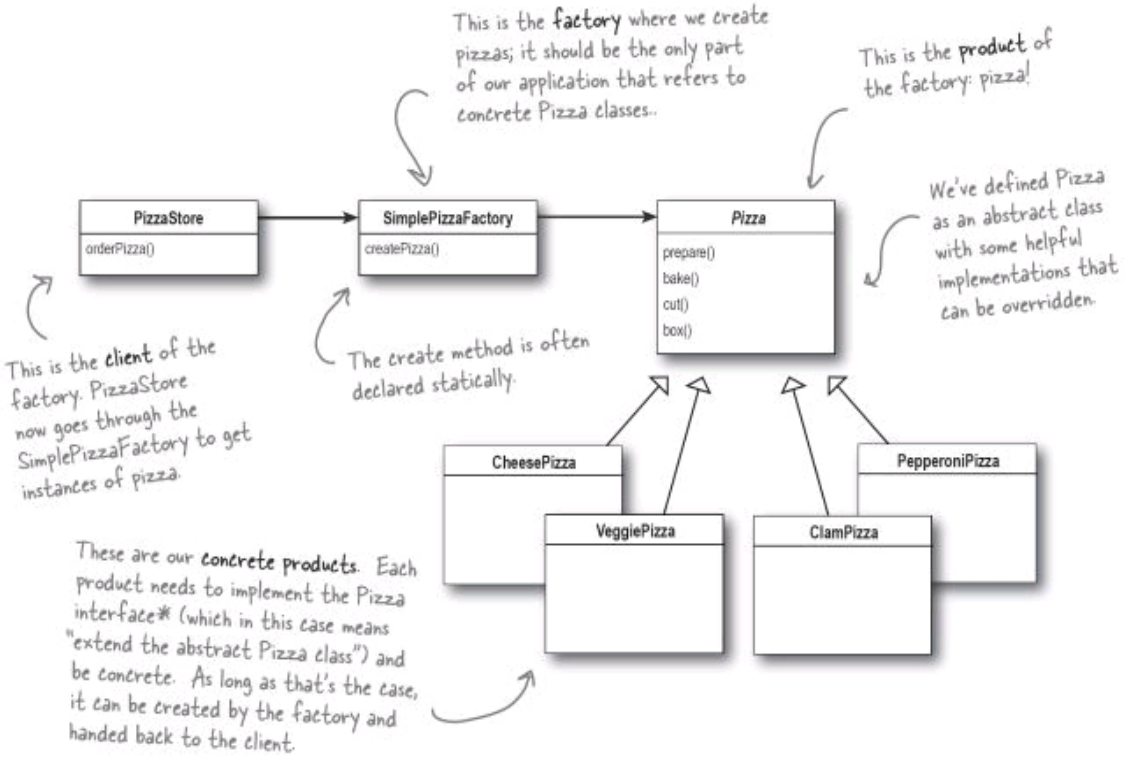
While that describes the *role* of the Decorator Pattern, it doesn't give us a lot of insight into how we'd *apply* the pattern to our own implementation. Let's take a look at the class diagram, which is a little more revealing (on the next page we'll look at the same structure applied to the beverage problem).



The Simple Factory defined

The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom. But it is commonly used, so we'll give it a Head First Pattern Honorable Mention. Some developers do mistake this idiom for the "Factory Pattern," so the next time there is an awkward silence between you and another developer, you've got a nice topic to break the ice.

Just because Simple Factory isn't a REAL pattern doesn't mean we shouldn't check out how it's put together. Let's take a look at the class diagram of our new Pizza Store:



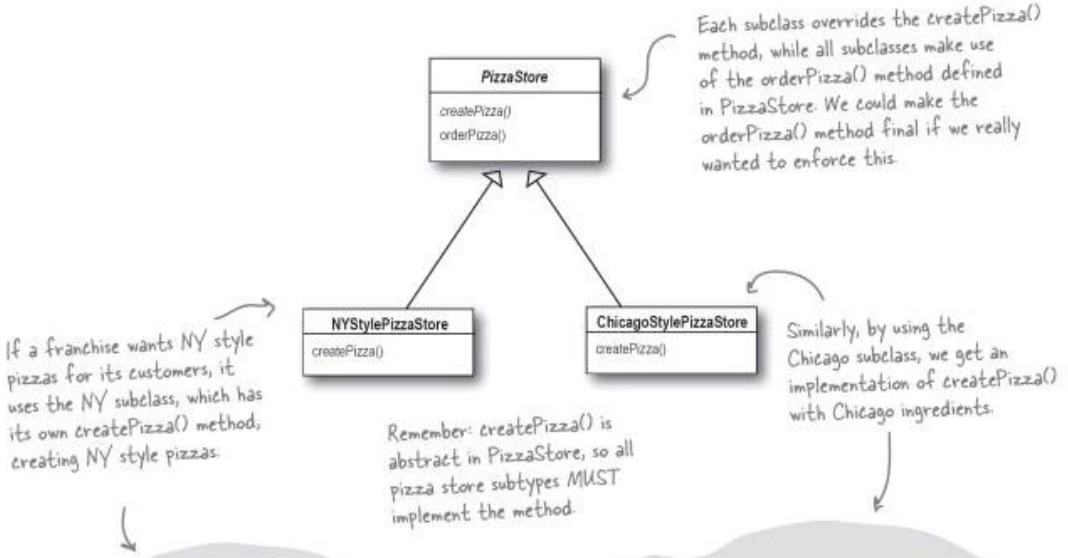
Think of Simple Factory as a warm up. Next, we'll explore two heavy duty patterns that are both factories. But don't worry, there's more pizza to come!

*Just another reminder: in design patterns, the phrase "implement an interface" does NOT always mean "write a class the implements a Java interface, by using the "implements" keyword in the class declaration." In the general use of the phrase, a concrete class implementing a method from a supertype (which could be a class OR interface) is still considered to be "implementing the interface" of that supertype.

Allowing the subclasses to decide

Remember, the PizzaStore already has a well-honed order system in the orderPizza() method and you want to ensure that it's consistent across all franchises.

What varies among the regional PizzaStores is the style of pizzas they make – New York Pizza has thin crust, Chicago Pizza has thick , and so on – and we are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of PizzaStore define what the createPizza() method looks like. So, we will have a number of concrete subclasses of PizzaStore, each with its own pizza variations, all fitting within the PizzaStore framework and still making use of the well-tuned orderPizza() method.



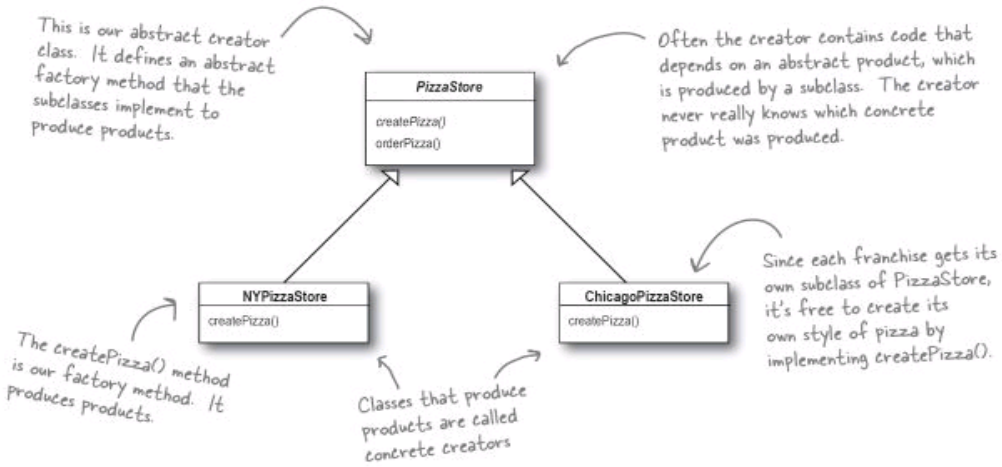

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

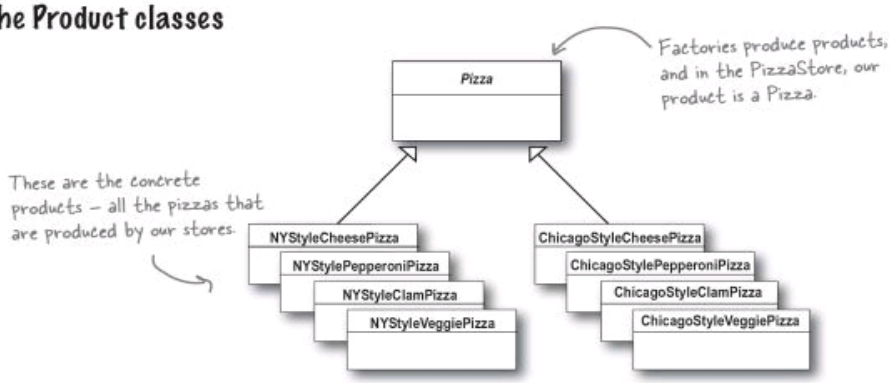
It's finally time to meet the Factory Method Pattern

All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this pattern:

The Creator classes



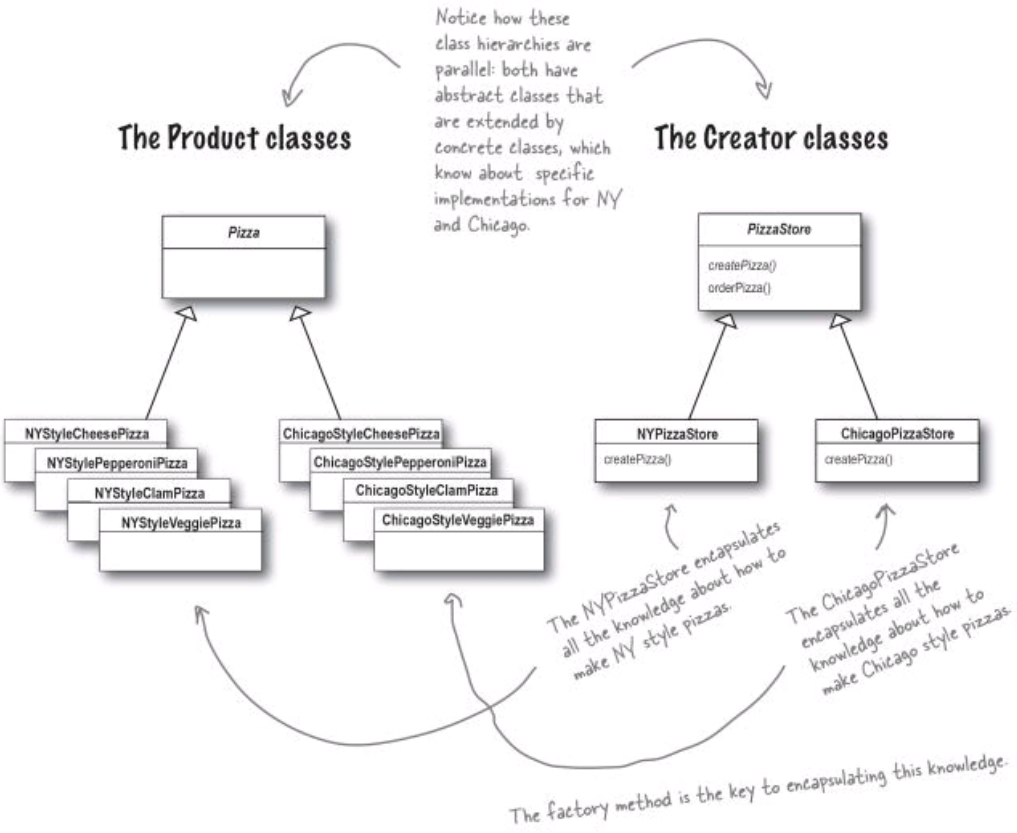
The Product classes



Another perspective: parallel class hierarchies

We've seen that the factory method provides a framework by supplying an orderPizza() method that is combined with a factory method. Another way to look at this pattern as a framework is in the way it encapsulates product knowledge into each creator.

Let's look at the two parallel class hierarchies and see how they relate:



Factory Method Pattern defined

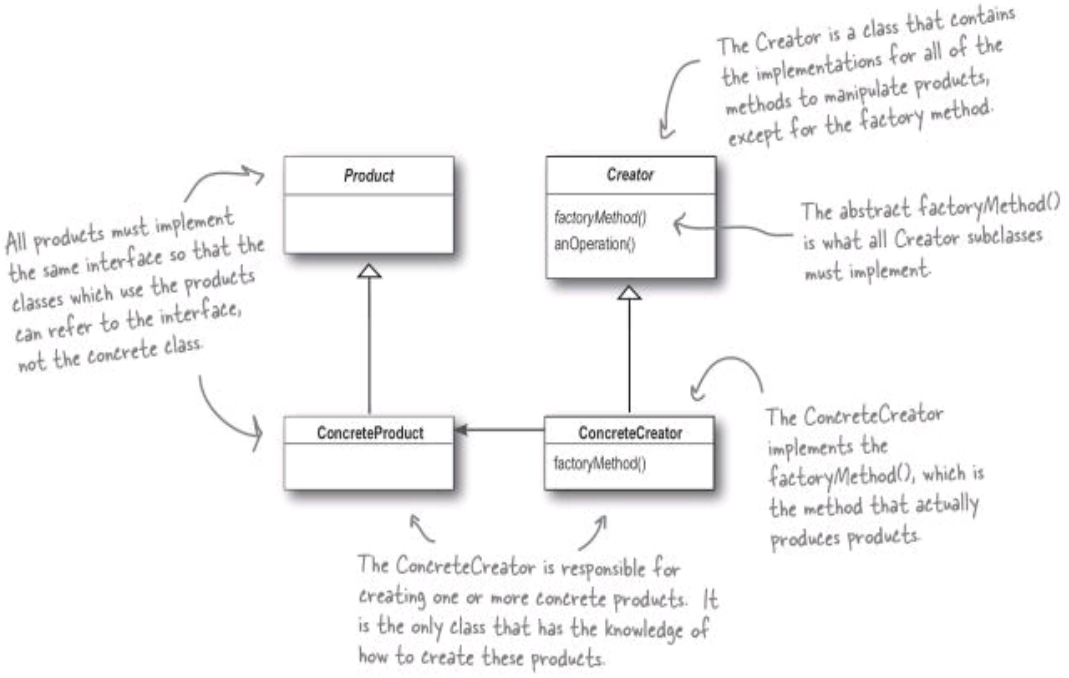
It's time to roll out the official definition of the Factory Method Pattern:

> **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator gives you an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

As in the official definition, you'll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

You could ask them what "decides" means, but we bet you now understand this better than they do!



The Dependency Inversion Principle

It should be pretty clear that reducing dependencies to concrete classes in our code is a "good thing." In fact, we've got an OO design principle that formalizes this notion; it even has a big, formal name: *Dependency Inversion Principle*.

Here's the general principle:

Yet another phrase you can use to impress the execs in the room! Your raise will more than offset the cost of this book, and you'll gain the admiration of your fellow developers.

Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

At first, this principle sounds a lot like "Program to an interface, not an implementation," right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

A "high-level" component is a class with behavior defined in terms of other, "low level" components.

For example, PizzaStore is a high-level component because its behavior is defined in terms of pizzas – it creates all the different pizza objects, prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

But what the heck does that mean?

Well, let's start by looking again at the pizza store diagram on the previous page. PizzaStore is our "high-level component" and the pizza implementations are our "low-level components," and clearly the PizzaStore is dependent on the concrete pizza classes.

Now, this principle tells us we should instead write our code so that we are depending on abstractions, not concrete classes. That goes for both our high level modules and our low-level modules.

But how do we do this? Let's think about how we'd apply this principle to our Very Dependent PizzaStore implementation...

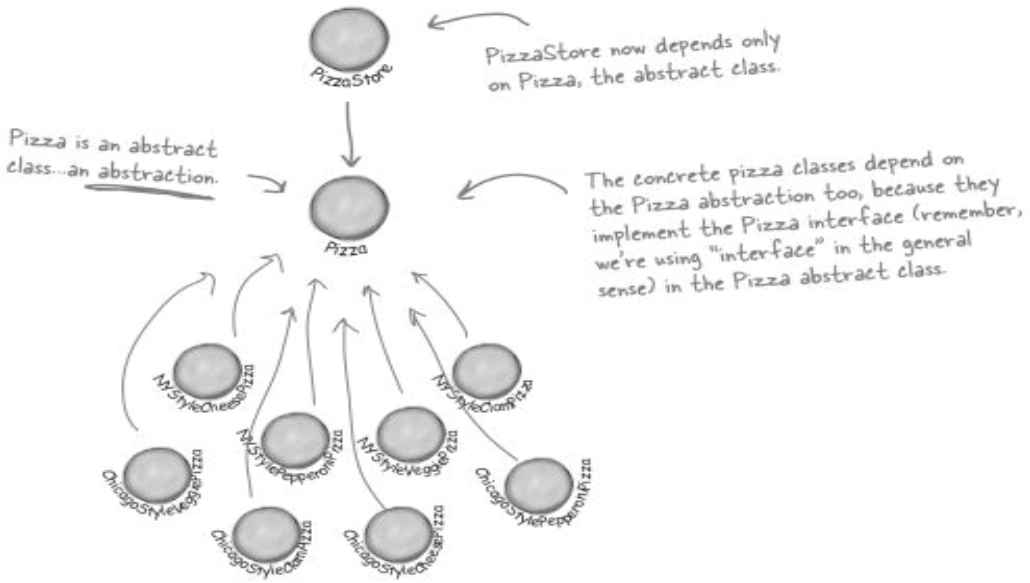
Applying the Principle

Now, the main problem with the Very Dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its orderPizza() method.

While we've created an abstraction, Pizza, we're nevertheless creating concrete Pizzas in this code, so we don't get a lot of leverage out of this abstraction.

How can we get those instantiations out of the orderPizza() method? Well, as we know, the Factory Method allows us to do just that.

So, after we've applied the Factory Method, our diagram looks like this:



After applying the Factory Method, you'll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction. Factory Method is not the only technique for adhering to the Dependency Inversion Principle, but it is one of the more powerful ones.

A few guidelines to help you follow the Principle...

The following guidelines can help you avoid OO designs that violate the Dependency Inversion Principle:

- No variable should hold a reference to a concrete class.

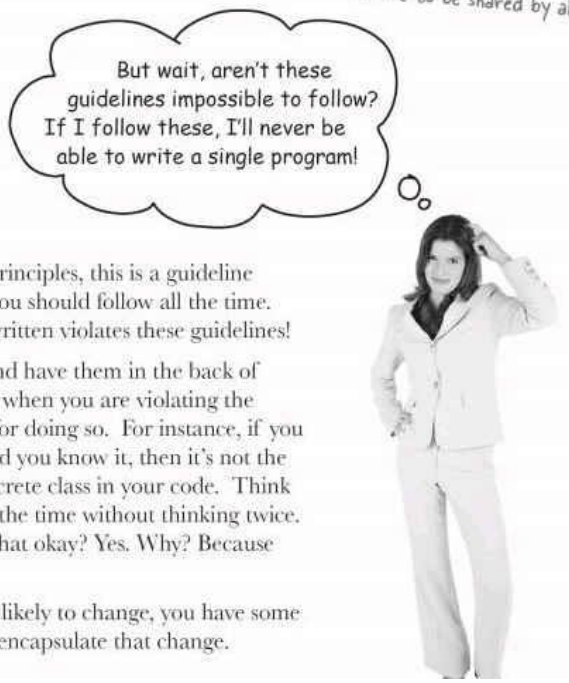
If you use **new**, you'll be holding a reference to a concrete class. Use a factory to get around that!

- No class should derive from a concrete class.

If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.

- No method should override an implemented method of any of its base classes.

If you override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.



You're exactly right! Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time. Clearly, every single Java program ever written violates these guidelines!

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate String objects all the time without thinking twice. Does that violate the principle? Yes. Is that okay? Yes. Why? Because String is very unlikely to change.

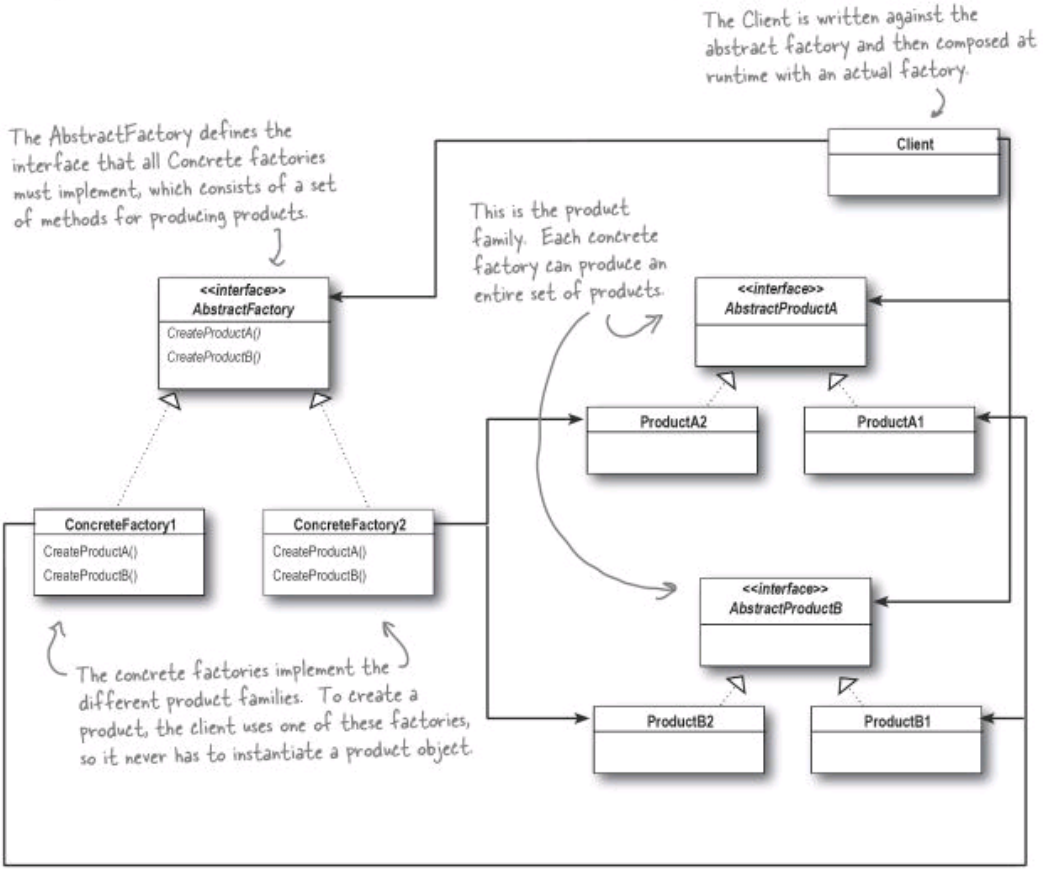
If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.

Abstract Factory Pattern defined

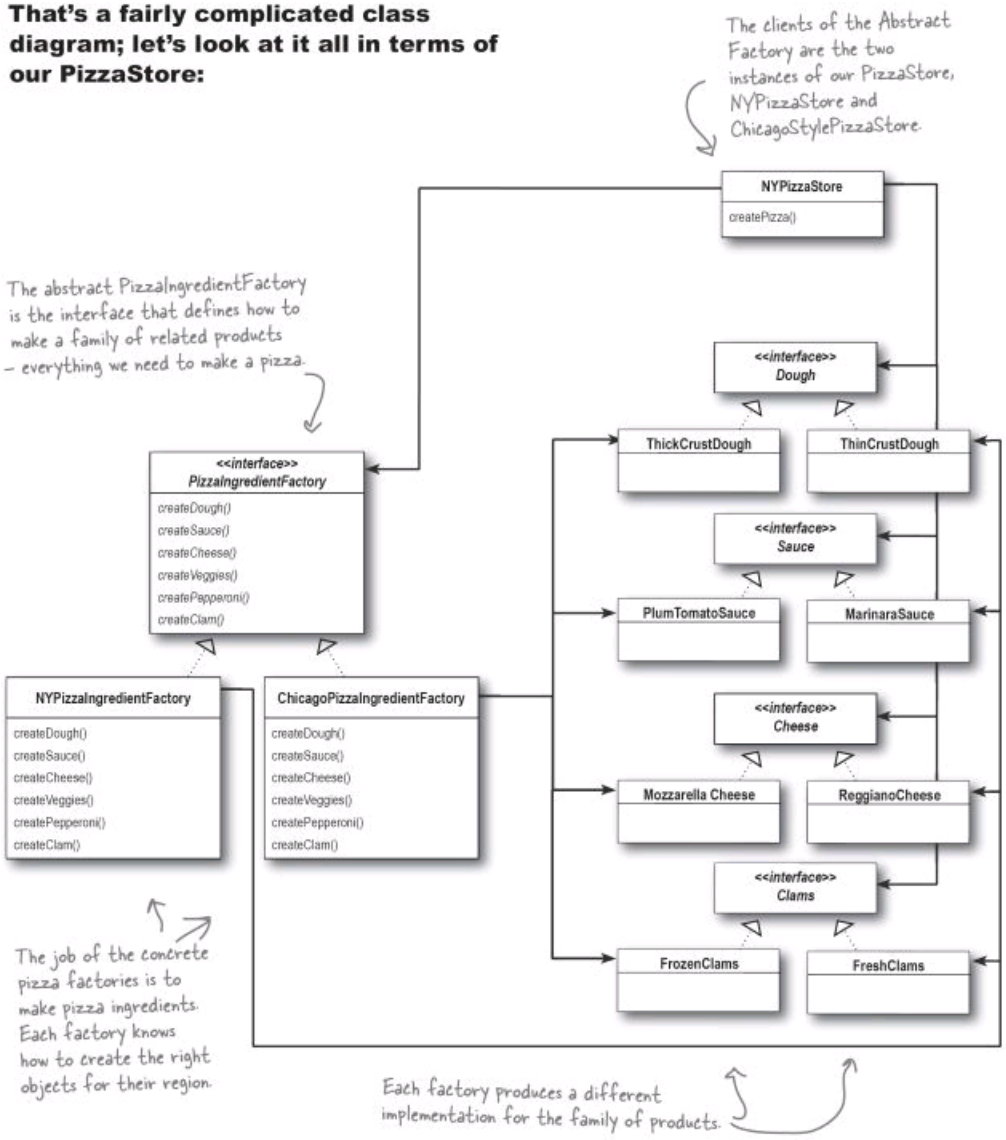
We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:

> **The Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.

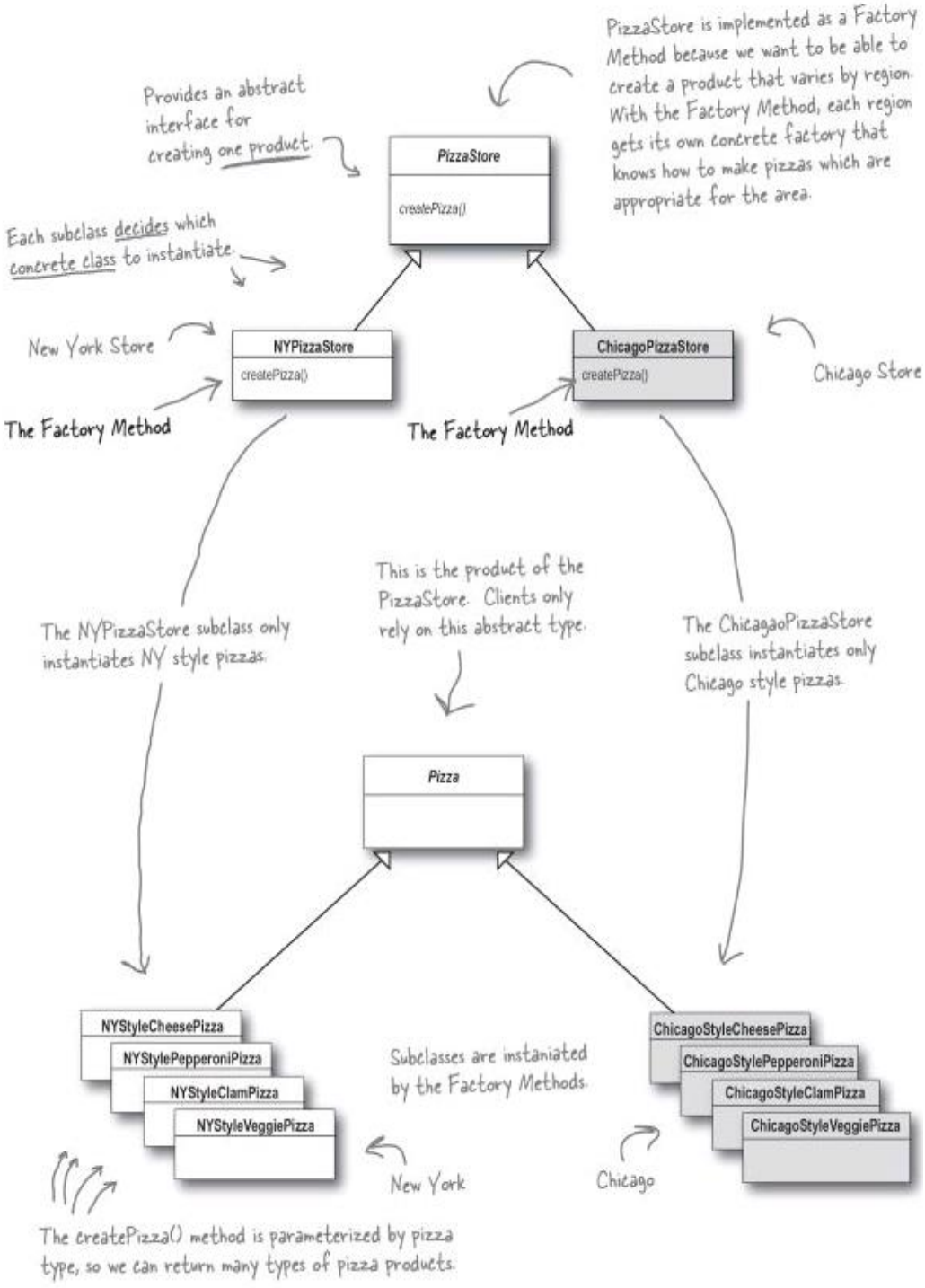
We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together:

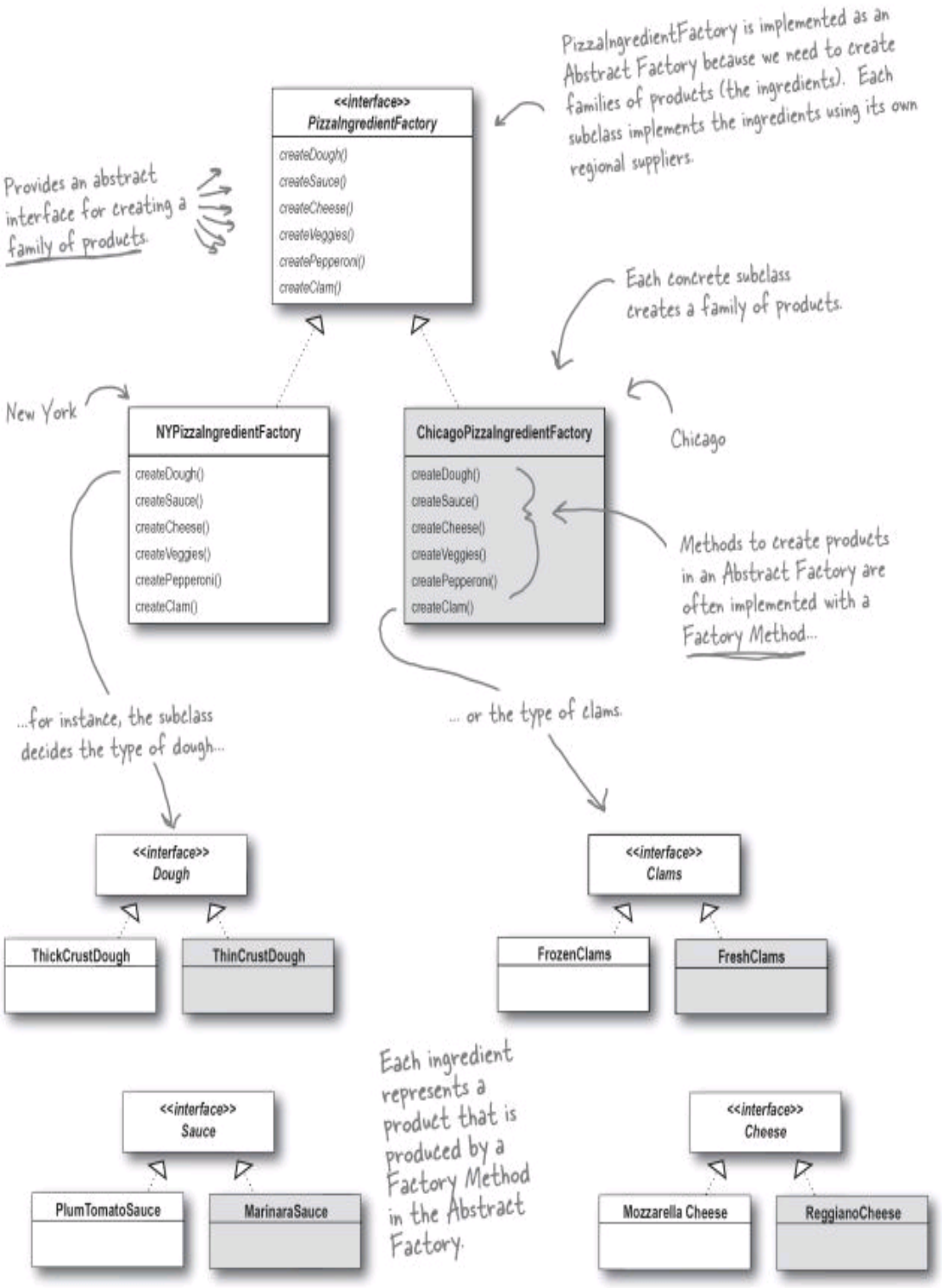


That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:



Factory Method and Abstract Factory compared




PizzaIngredientFactory is implemented as an Abstract Factory because we need to create families of products (the ingredients). Each subclass implements the ingredients using its own regional suppliers.
<<interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
Provides an abstract interface for creating a family of products.
Each concrete subclass creates a family of products.
New York
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
Chicago
Methods to create products in an Abstract Factory are often implemented with a Factory Method...
...for instance, the subclass decides the type of dough...
... or the type of clams.
<<interface>>
Dough
ThickCrustDough
ThinCrustDough
<<interface>>
Clams
FrozenClams
FreshClams
Each ingredient represents a product that is produced by a Factory Method in the Abstract Factory.
<<interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
<<interface>>
Cheese
Mozzarella Cheese
ReggianoCheese

Singleton Pattern defined

Now that you've got the classic implementation of Singleton in your head, it's time to sit back, enjoy a bar of chocolate, and check out the finer points of the Singleton Pattern.

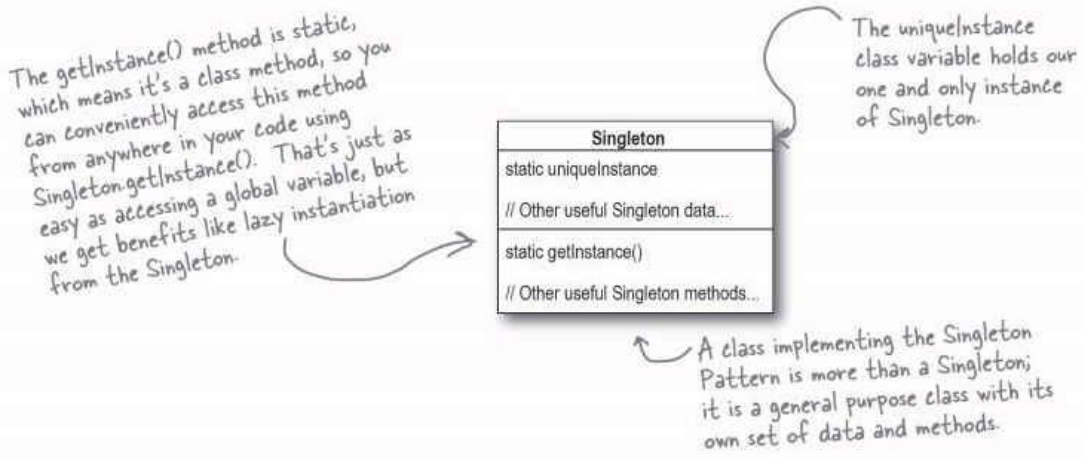
Let's start with the concise definition of the pattern:

> **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

No big surprises there. But, let's break it down a bit more:

- What's really going on here? We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.
- We're also providing a global access point to the instance: whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the Singleton is created in a lazy manner, which is especially important for resource intensive objects.

Okay, let's check out the class diagram:



Dealing with multithreading

Our multithreading woes are almost trivially fixed by making getInstance() a synchronized method:

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

Can we improve multithreading?

For most Java applications, we obviously need to ensure that the Singleton works in the presence of multiple threads. But, it looks fairly expensive to synchronize the getInstance() method, so what do we do?

Well, we have a few options...

1. Do nothing if the performance of getInstance() isn't critical to your application

That's right; if calling the getInstance() method isn't causing substantial overhead for your application, forget about it. Synchronizing getInstance() is straightforward and effective. Just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high traffic part of your code begins using getInstance(), you may have to reconsider.

2. Move to an eagerly created instance rather than a lazily created one

If your application always creates and uses an instance of the Singleton or the overhead of creation and runtime aspects of the Singleton are not onerous, you may want to create your Singleton eagerly, like this:

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

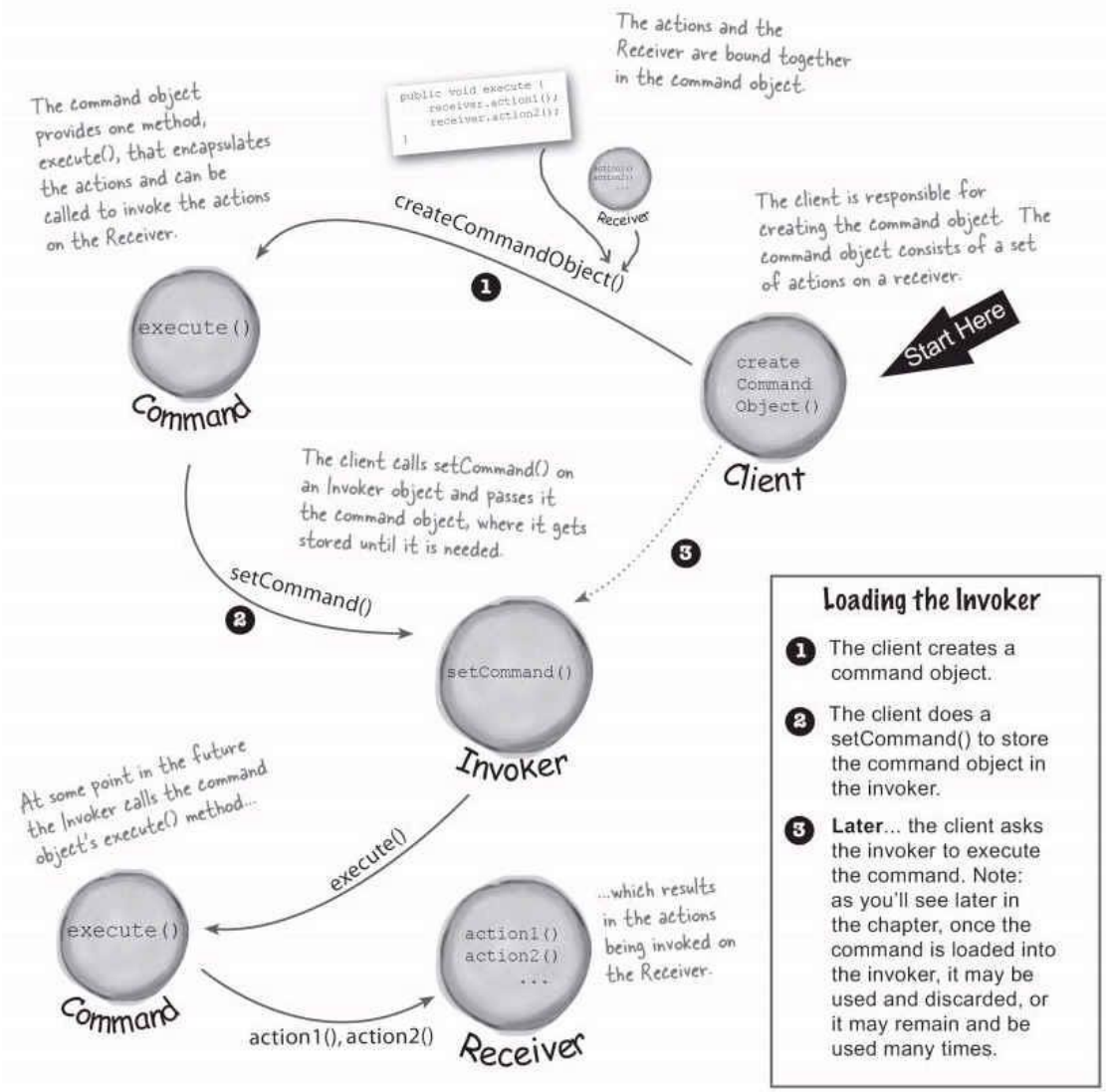
You've now added another pattern to your toolbox. Singleton gives you another method of creating objects – in this case, unique objects.

BULLET POINTS

- The Singleton Pattern ensures you have at most one instance of a class in your application.
- The Singleton Pattern also provides a global access point to that instance.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.
- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded!).
- Beware of the double-checked locking implementation; it is not thread-safe in versions before Java 2, version 5.
- Be careful if you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.
- If you are using a JVM earlier than 1.2, you'll need to create a registry of Singletons to defeat the garbage collector.

As you've seen, despite its apparent simplicity, there are a lot of details involved in the Singleton's implementation. After reading this chapter, though, you are ready to go out and use Singleton in the wild.

Okay, we've spent enough time in the Objectville Diner that we know all the personalities and their responsibilities quite well. Now we're going rework the Diner diagram to reflect the Command Pattern. You'll see that all the players are the same; only the names have changed.



Our first command object

Isn't it about time we build our first command object? Let's go ahead and write some code for the remote control. While we haven't figured out how to design the remote control API yet, building a few things from the bottom up may help us...

Implementing the Command interface

First things first: all command objects implement the same interface, which consists of one method. In the Diner we called this method orderUp(); however, we typically just use the name execute().

Here's the Command interface:

```
public interface Command {
    public void execute();
}
```

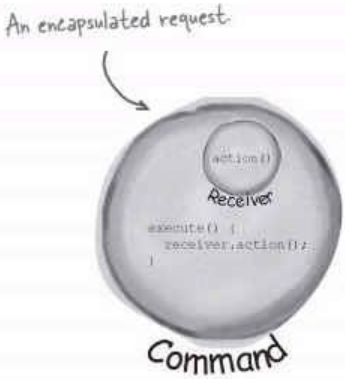
Simple. All we need is one method called execute().

The Command Pattern defined

You've done your time in the Objectville Diner, you've partly implemented the remote control API, and in the process you've got a fairly good picture of how the classes and objects interact in the Command Pattern. Now we're going to define the Command Pattern and nail down all the details.

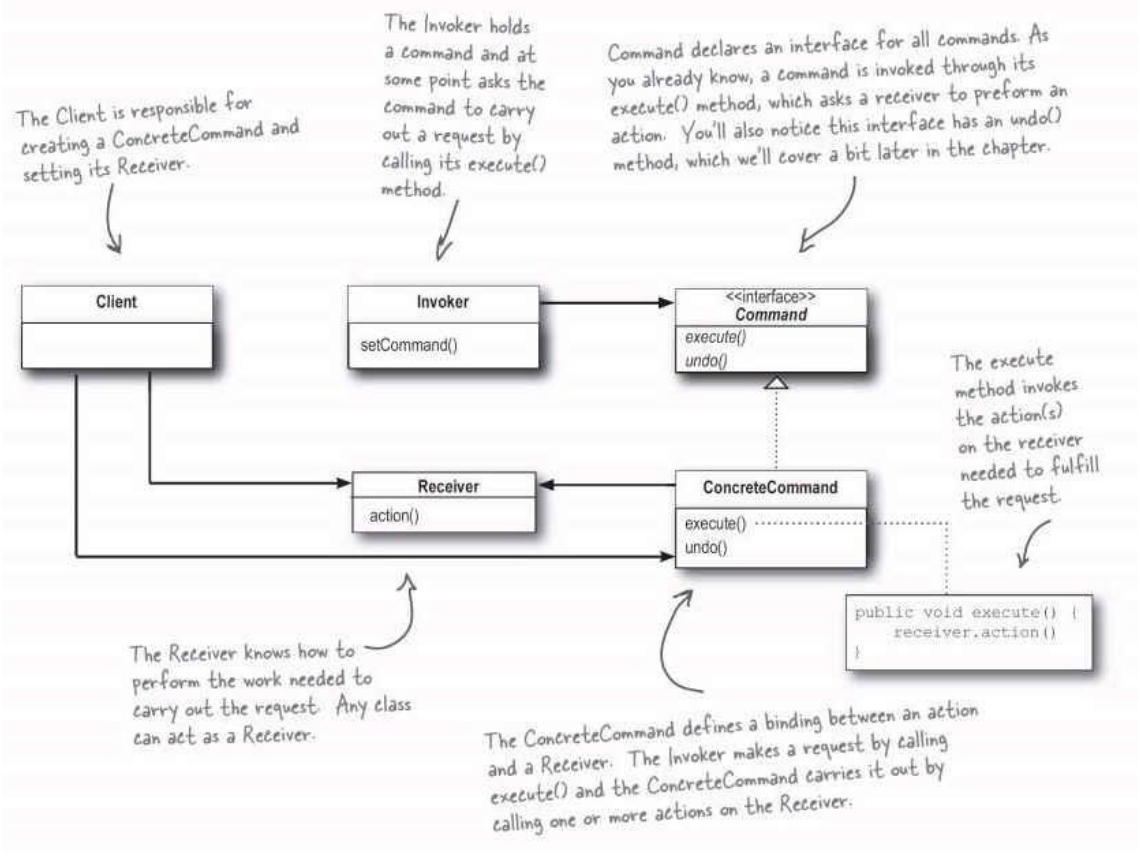
Let's start with its official definition:

> **The Command Pattern** encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



Let's step through this. We know that a command object *encapsulates a request* by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute(). When called, execute() causes the actions to be invoked on the receiver. From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the execute() method, their request will be serviced.

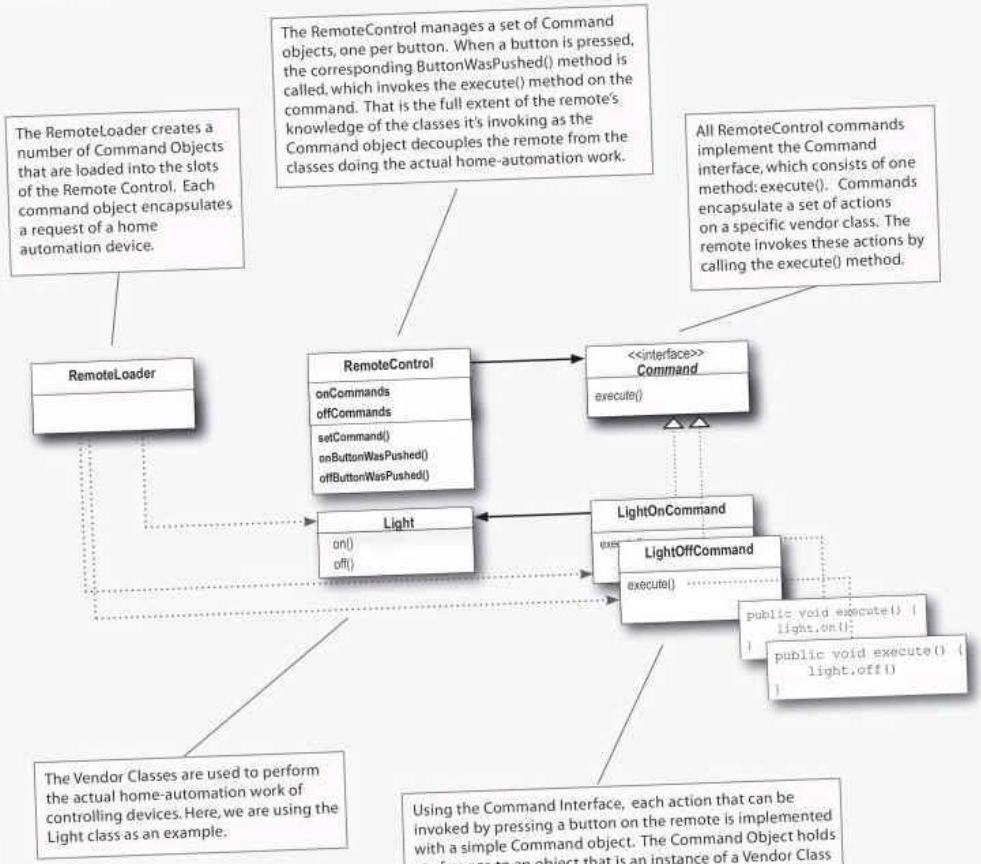
The Command Pattern defined: the class diagram



Remote Control API Design for Home Automation or Bust, Inc.,

We are pleased to present you with the following design and application programming interface for your Home Automation Remote Control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we have employed the Command Pattern to logically decouple the RemoteControl class from the Vendor Classes. We believe this will reduce the cost of producing the remote as well as drastically reduce your ongoing maintenance costs.

The following class diagram provides an overview of our design:



What are we doing?

Okay, we need to add functionality to support the undo button on the remote. It works like this: say the Living Room Light is off and you press the on button on the remote. Obviously the light turns on. Now if you press the undo button then the last action will be reversed – in this case the light will turn off. Before we get into more complex examples, let's get the light working with the undo button:

1. When commands support undo, they have an undo() method that mirrors the execute() method. Whatever execute() last did, undo() reverses. So, before we can add undo to our commands, we need to add an undo() method to the Command interface:

```
public interface Command {
    public void execute();
    public void undo();
}
```

Here's the new undo() method.

That was simple enough.

Now, let's dive into the Light command and implement the undo() method.

Mary's idea is to make a new kind of Command that can execute other Commands... and more than one of them! Pretty good idea, huh?

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
```

Take an array of Commands and store them in the MacroCommand.

When the macro gets executed by the remote, execute those commands one at a time.

Q: Do I always need a receiver? Why can't the command object implement the details of the execute() method?

A: In general, we strive for "dumb" command objects that just invoke an action on a receiver; however, there are many examples of "smart" command objects that implement most, if not all, of the logic needed to carry out a request. Certainly you can do this; just keep in mind you'll no longer have the same level of decoupling between the invoker and receiver, nor will you be able to parameterize your commands with receivers.

there are no Dumb Questions

Q: How can I implement a history of undo operations? In other words, I want to be able to press the undo button multiple times.

A: Great question! It's pretty easy actually; instead of keeping just a reference to the last Command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo() method.

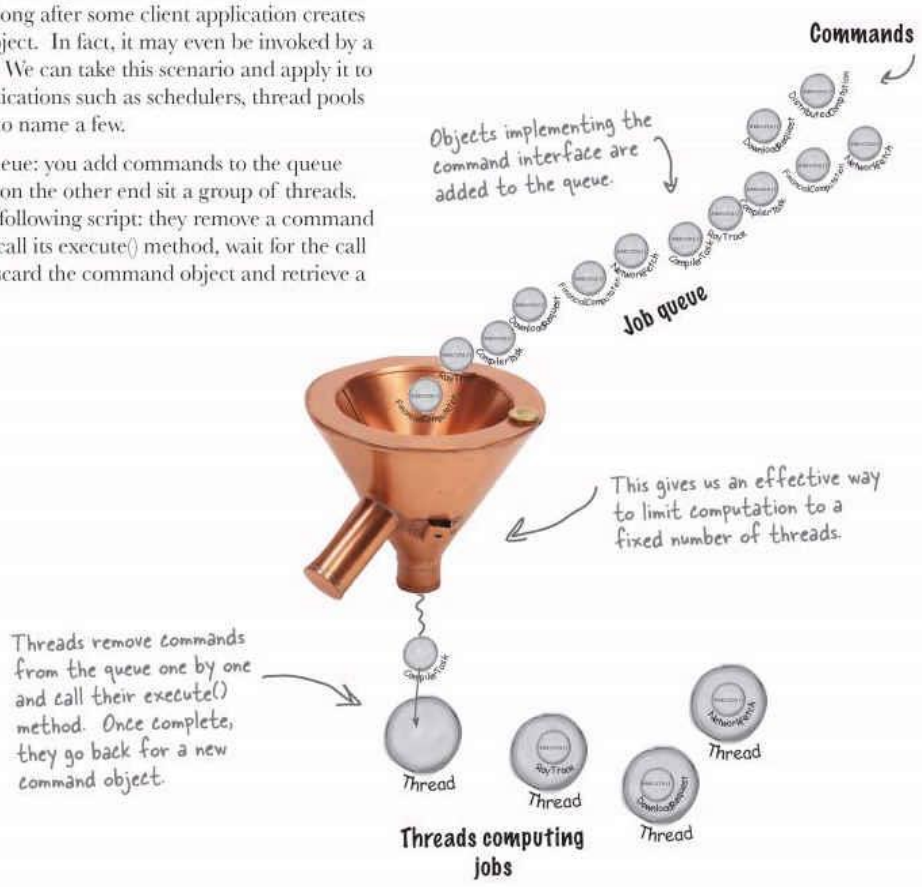
Q: Could I have just implemented Party Mode as a Command by creating a PartyCommand and putting the calls to execute the other Commands in the PartyCommand's execute() method?

A: You could; however, you'd essentially be "hardcoding" the party mode into the PartyCommand. Why go to the trouble? With the MacroCommand, you can decide dynamically which Commands you want to go into the PartyCommand, so you have more flexibility using MacroCommands. In general, the MacroCommand is a more elegant solution and requires less new code.

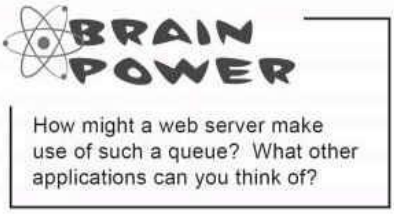
More uses of the Command Pattern: queuing requests

Commands give us a way to package a piece of computation (a receiver and a set of actions) and pass it around as a first-class object. Now, the computation itself may be invoked long after some client application creates the command object. In fact, it may even be invoked by a different thread. We can take this scenario and apply it to many useful applications such as schedulers, thread pools and job queues, to name a few.

Imagine a job queue: you add commands to the queue on one end, and on the other end sit a group of threads. Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.



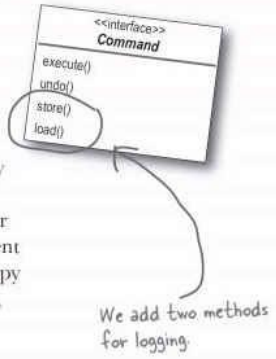
Note that the job queue classes are totally decoupled from the objects that are doing the computation. One minute a thread may be computing a financial computation, and the next it may be retrieving something from the network. The job queue objects don't care; they just retrieve commands and call execute(). Likewise, as long as you put objects into the queue that implement the Command Pattern, your execute() method will be invoked when a thread is available.

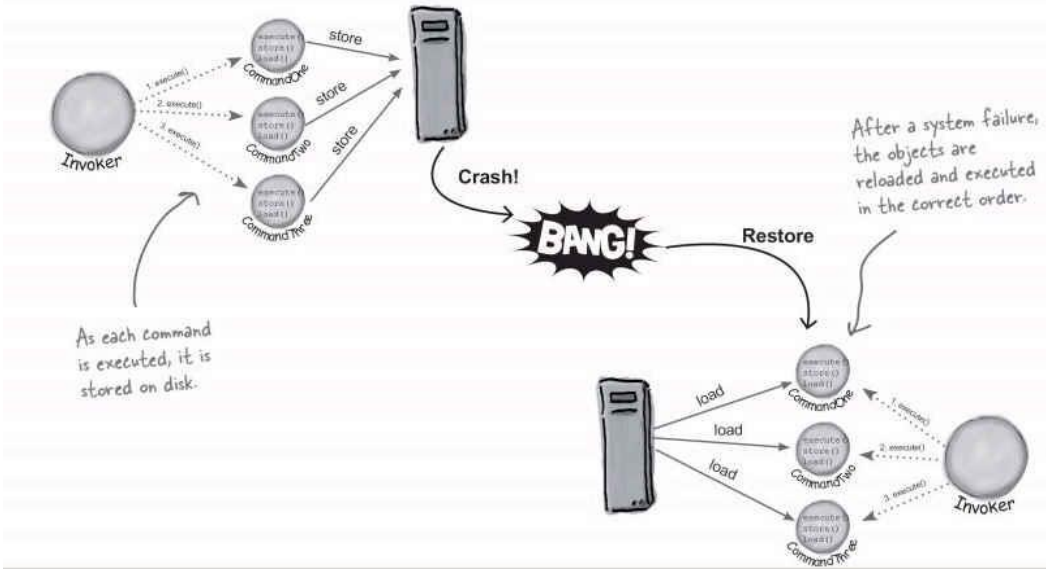


The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions. The Command Pattern can support these semantics with the addition of two methods: store() and load(). In Java we could use object serialization to implement these methods, but the normal caveats for using serialization for persistence apply.

How does this work? As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their execute() methods in batch and in order.

Now, this kind of logging wouldn't make sense for a remote control; however, there are many applications that invoke actions on large data structures that can't be quickly saved each time a change is made. By using logging, we can save all the operations since the last check point, and if there is a system failure, apply those operations to our checkpoint. Take, for example, a spreadsheet application: we might want to implement our failure recovery by logging the actions on the spreadsheet rather than writing a copy of the spreadsheet to disk every time a change occurs. In more advanced applications, these techniques can be extended to apply to sets of operations in a transactional manner so that all of the operations complete, or none of them do.





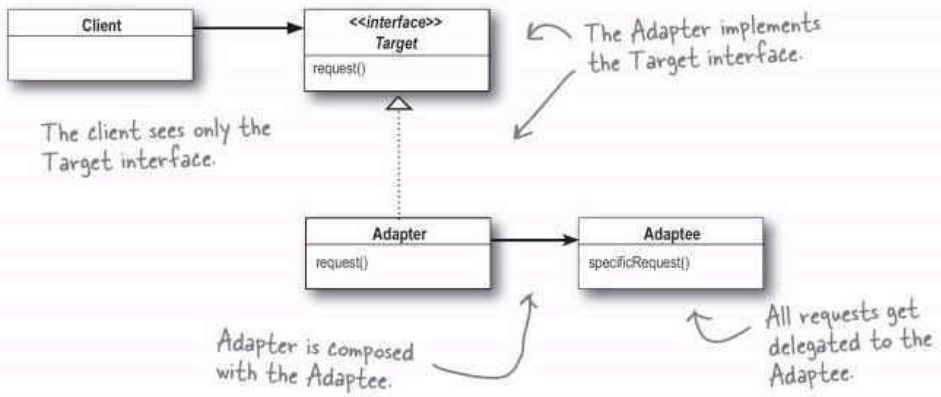
Adapter Pattern defined

Enough ducks, turkeys and AC power adapters; let's get real and look at the official definition of the Adapter Pattern:

> **The Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Now, we know this pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

We've taken a look at the runtime behavior of the pattern; let's take a look at its class diagram as well:

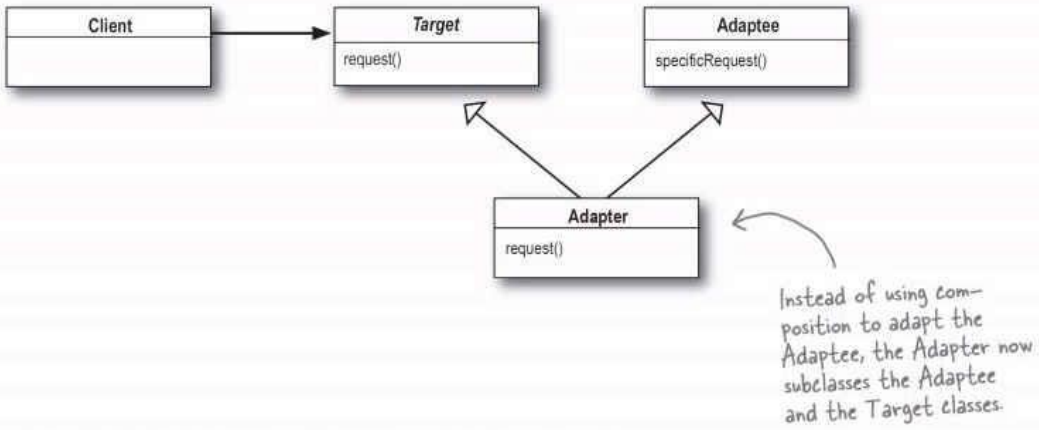


The Adapter Pattern is full of good OO design principles: check out the use of object composition to wrap the adaptee with an altered interface. This approach has the added advantage that we can use an adapter with any subclass of the adaptee.

Object and class adapters

Now despite having defined the pattern, we haven't told you the whole story yet. There are actually *two* kinds of adapters: *object* adapters and *class* adapters. This chapter has covered object adapters and the class diagram on the previous page is a diagram of an object adapter.

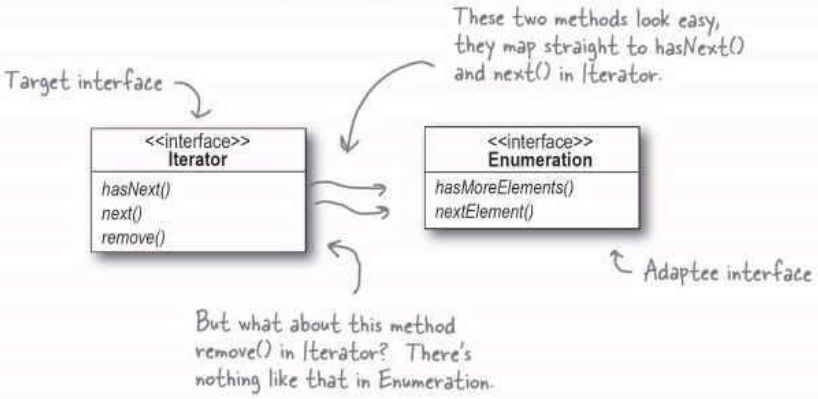
So what's a *class* adapter and why haven't we told you about it? Because you need multiple inheritance to implement it, which isn't possible in Java. But, that doesn't mean you might not encounter a need for class adapters down the road when using your favorite multiple inheritance language! Let's look at the class diagram for multiple inheritance.



Look familiar? That's right – the only difference is that with class adapter we subclass the Target and the Adaptee, while with object adapter we use composition to pass requests to an Adaptee.

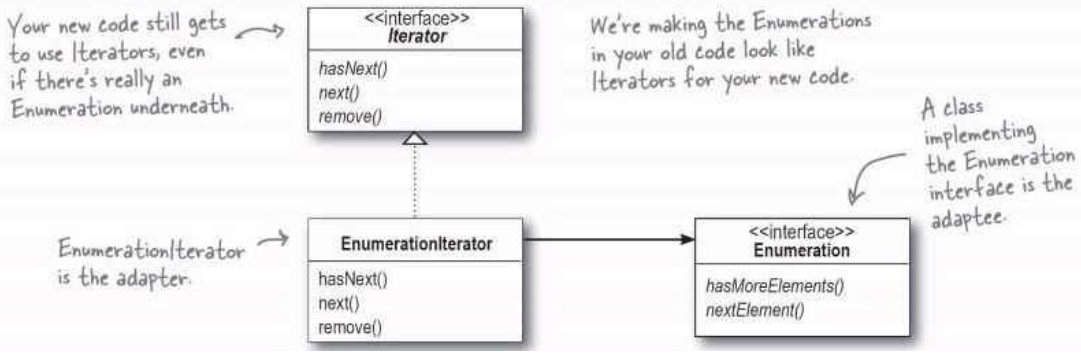
Adapting an Enumeration to an Iterator

First we'll look at the two interfaces to figure out how the methods map from one to the other. In other words, we'll figure out what to call on the adaptee when the client invokes a method on the target.



Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the Target interface and that is composed with an adaptee. The hasNext() and next() methods are going to be straightforward to map from target to adaptee: we just pass them right through. But what do you do about remove()? Think about it for a moment (and we'll deal with it on the next page). For now, here's the class diagram:



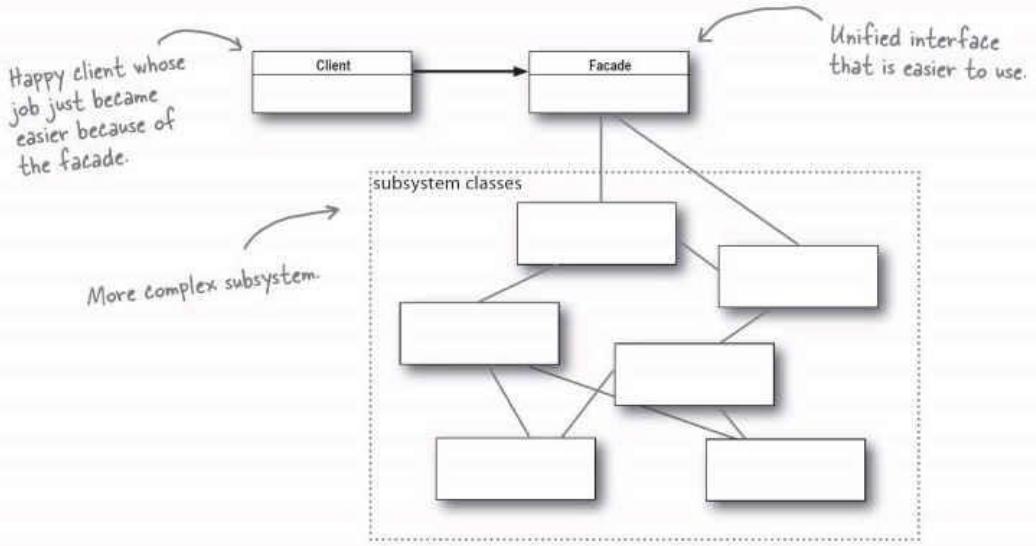
Facade Pattern defined

To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem. Unlike a lot of patterns, Facade is fairly straightforward; there are no mind bending abstractions to get your head around. But that doesn't make it any less powerful: the Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object oriented principle.

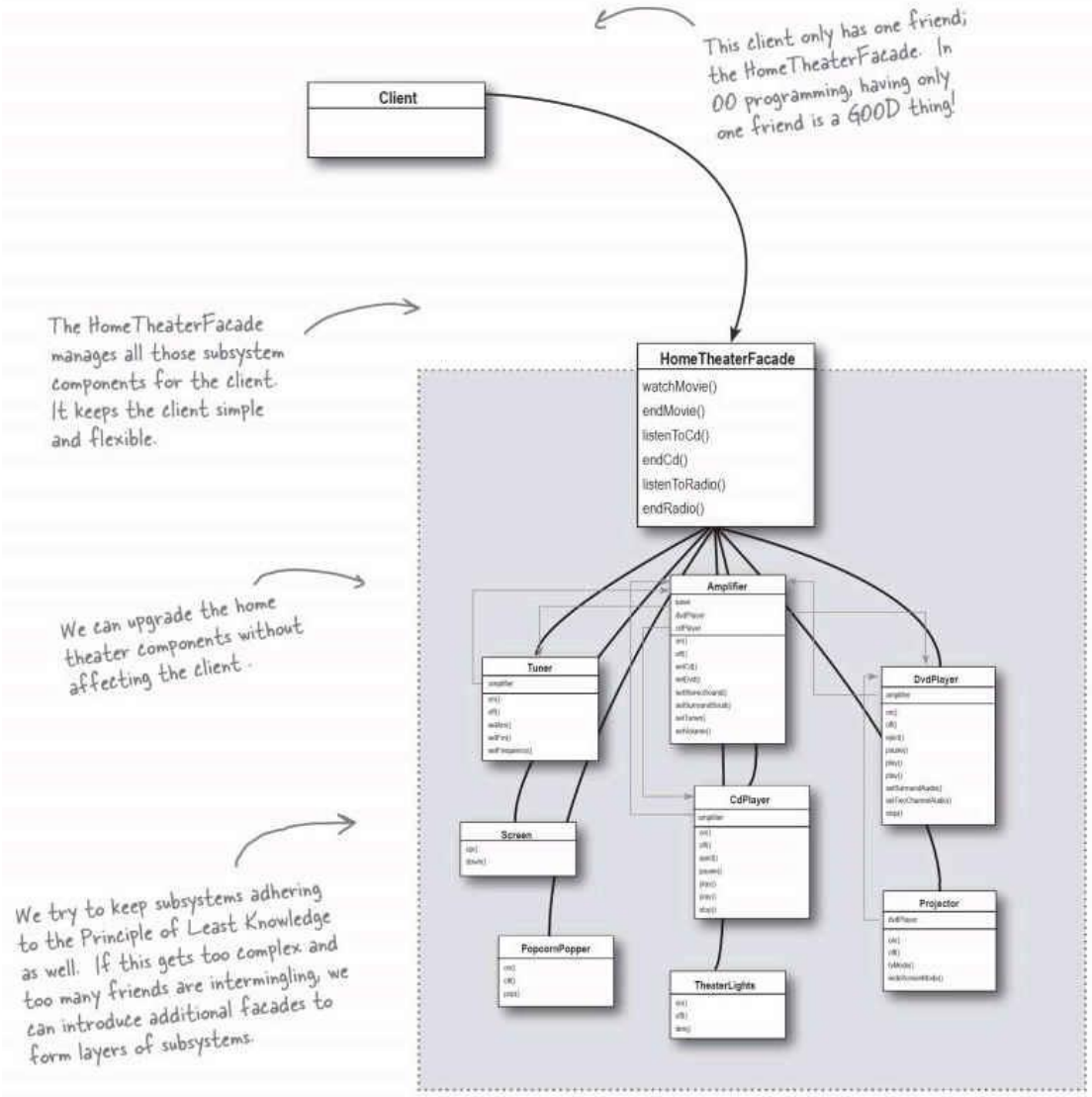
Before we introduce that new principle, let's take a look at the official definition of the pattern:

> **The Facade Pattern** provides a unified interface to a set of interfaces in a subsytem. Facade defines a higher-level interface that makes the subsystem easier to use.

There isn't a lot here that you don't already know, but one of the most important things to remember about a pattern is its intent. This definition tells us loud and clear that the purpose of the facade it to make a subsystem easier to use through a simplified interface. You can see this in the pattern's class diagram:



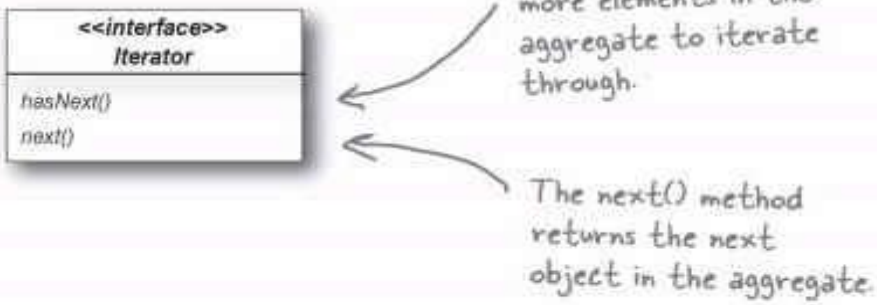
The Facade and the Principle of Least Knowledge



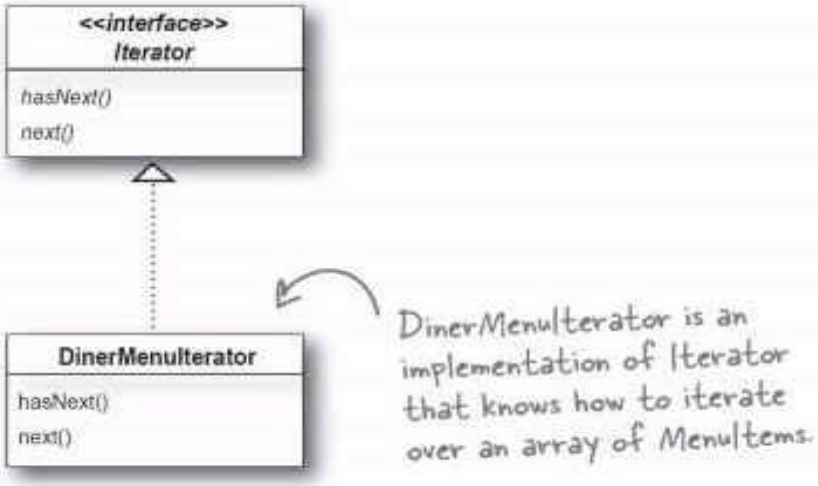
Meet the Iterator Pattern

Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is a Design Pattern called the Iterator Pattern.

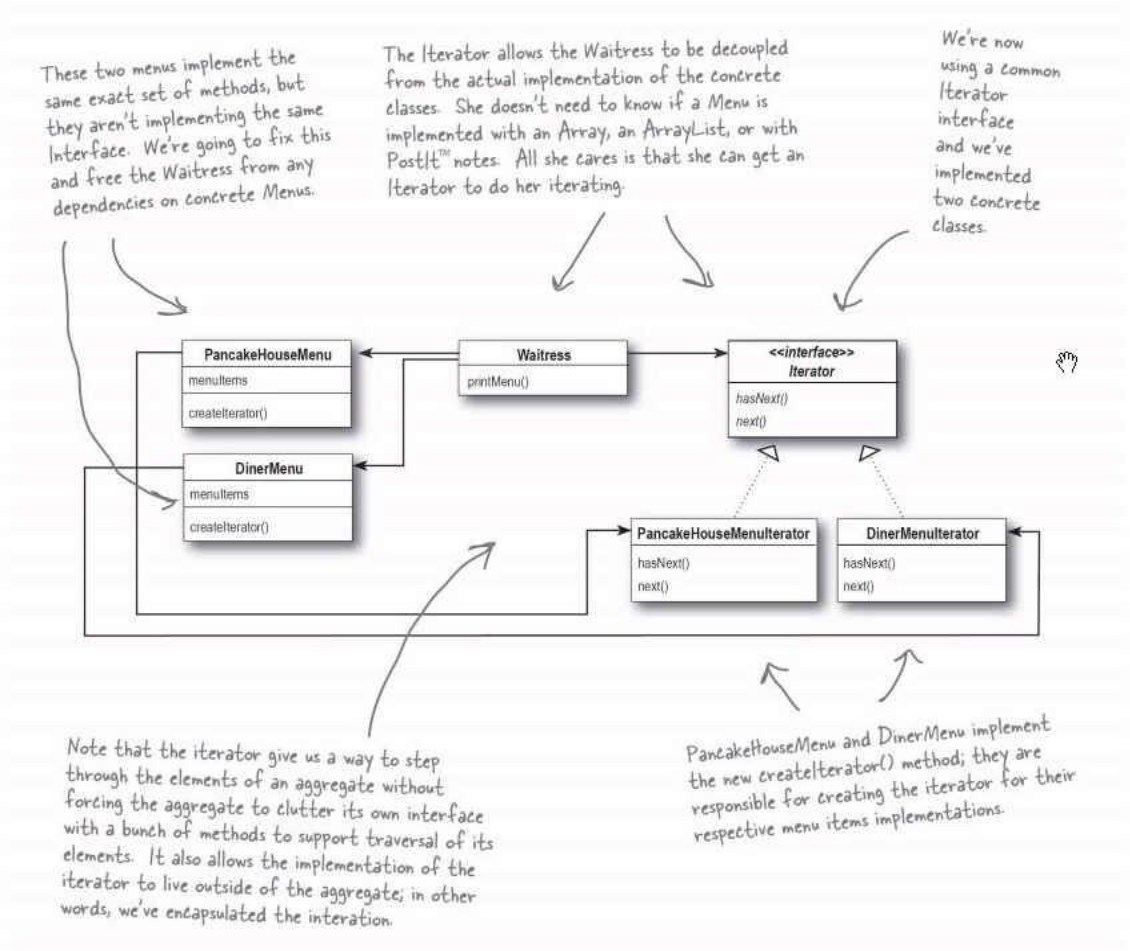
The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:



Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hashtables, ...pick your favorite collection of objects. Let's say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:

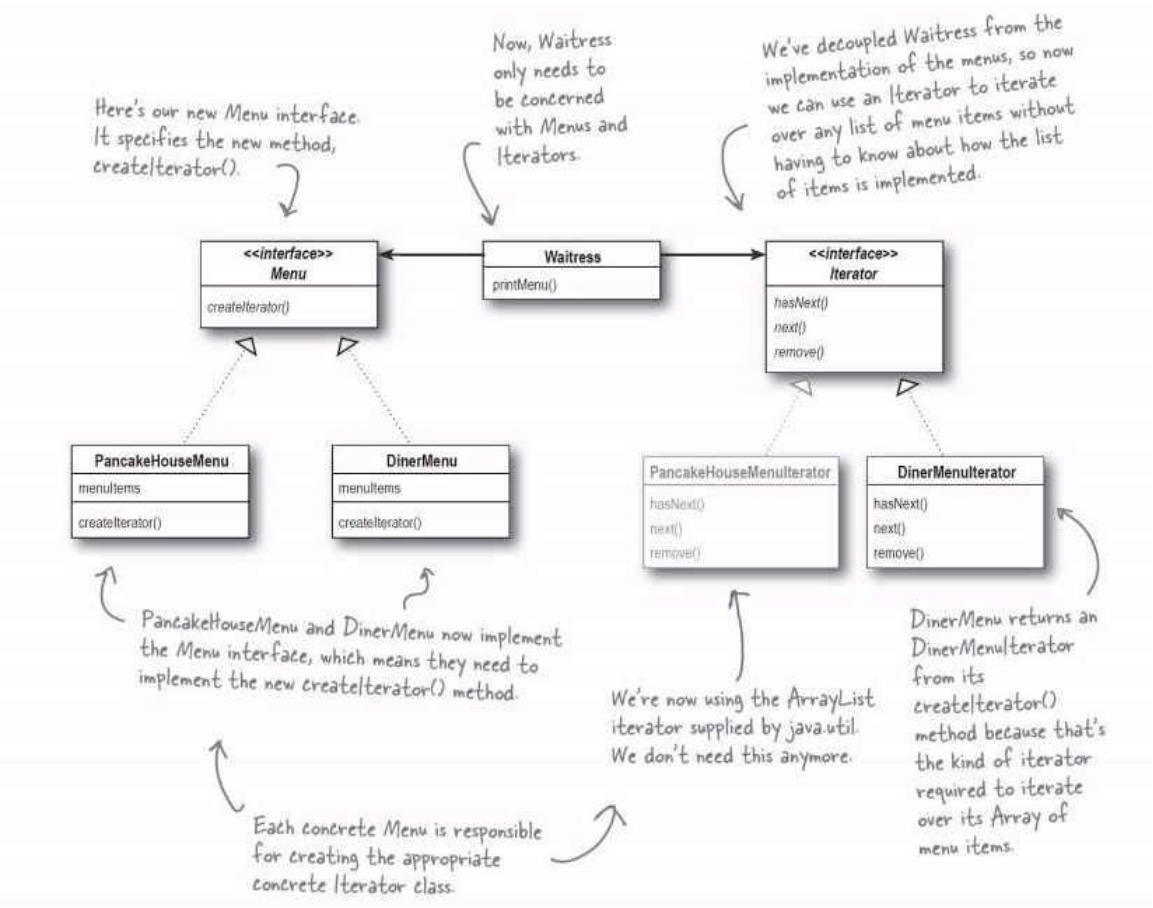


Before we clean things up, let's get a bird's eye view of our current design.



The new Menu interface has one method, createIterator(), that is implemented by PancakeHouseMenu and DinerMenu. Each menu class assumes the responsibility of creating a concrete Iterator that is appropriate for its internal implementation of the menu items.

This solves the problem of the Waitress depending on the implementation of the MenuItems.



Iterator Pattern defined

You've already seen how to implement the Iterator Pattern with your very own iterator. You've also seen how Java supports iterators in some of its collection oriented classes (the ArrayList). Now it's time to check out the official definition of the pattern:

> **The Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

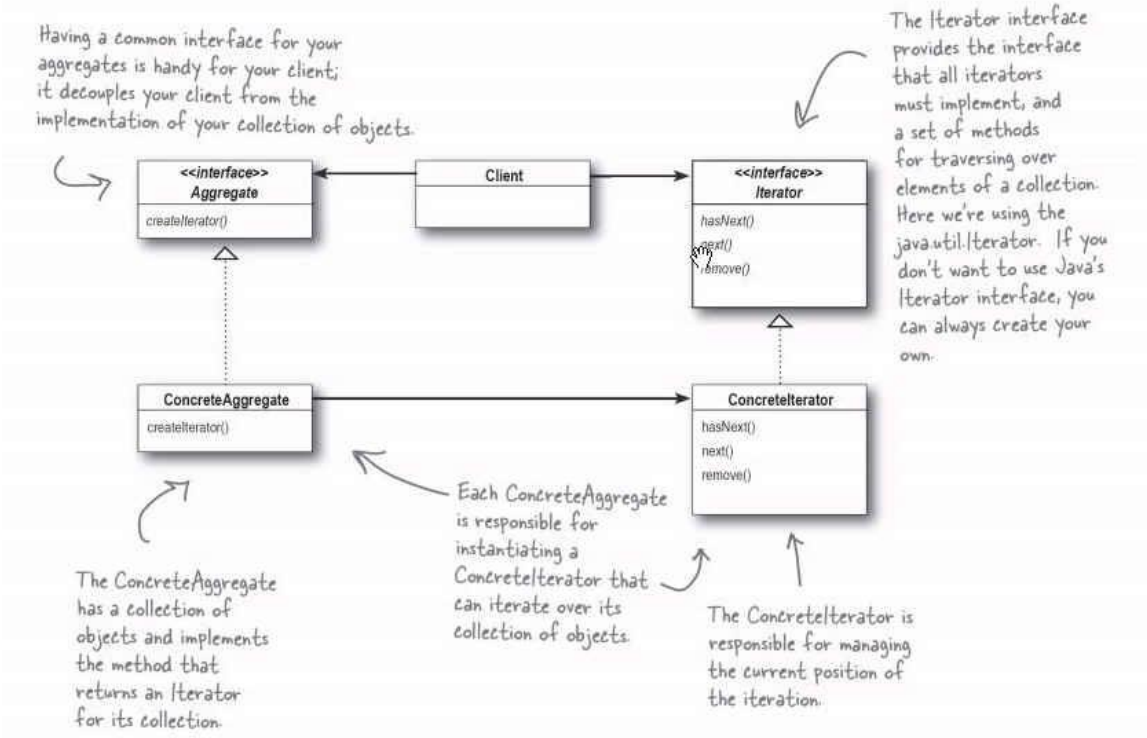
This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with *any* of these aggregates – just like the printMenu() method, which doesn't care if the menu items are held in an Array or ArrayList (or anything else that can create an Iterator), as long as it can get hold of an Iterator.

The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

Let's check out the class diagram to put all the pieces in context...

The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation.

It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.



Single Responsibility

What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods? Well, we already know that would expand the number of methods in the aggregate, but so what? Why is that so bad?

Well, to see why, you first need to recognize that when we allow a class to not only take care of its own business (managing some kind of aggregate) but also take on more responsibilities (like iteration) then we've given the class two reasons to change. Two? Yup, two: it can change if the collection changes in some way, and it can change if the way we iterate changes. So once again our friend CHANGE is at the center of another design principle:

Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

This principle guides us to keep each class to a single responsibility.

Design Principle

A class should have only one reason to change.

We know we want to avoid change in a class like the plague – modifying code provides all sorts of opportunities for problems to creep in. Having two ways to change increases the probability the class will change in the future, and when it does, it's going to affect two aspects of your design.

The solution? The principle guides us to assign each responsibility to one class, and only one class.

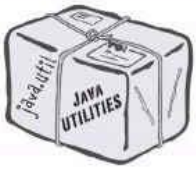
That's right, it's as easy as that, and then again it's not: separating responsibility in design is one of the most difficult things to do. Our brains are just too good at seeing a set of behaviors and grouping them together even when there are actually two or more responsibilities. The only way to succeed is to be diligent in examining your designs and to watch out for signals that a class is changing in more than one way as your system grows.

Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

We say that a module or class has *high cohesion* when it is designed around a set of related functions, and we say it has *low cohesion* when it is designed around a set of unrelated functions.

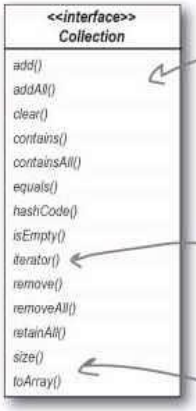
Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

Iterators and Collections



We've been using a couple of classes that are part of the Java Collections Framework. This "framework" is just a set of classes and interfaces, including ArrayList, which we've been using, and many others like Vector, LinkedList, Stack, and PriorityQueue. Each of these classes implements the java.util.Collection interface, which contains a bunch of useful methods for manipulating groups of objects.

Let's take a quick look at the interface:



As you can see, there's all kinds of good stuff here. You can add and remove elements from your collection without even knowing how it's implemented.

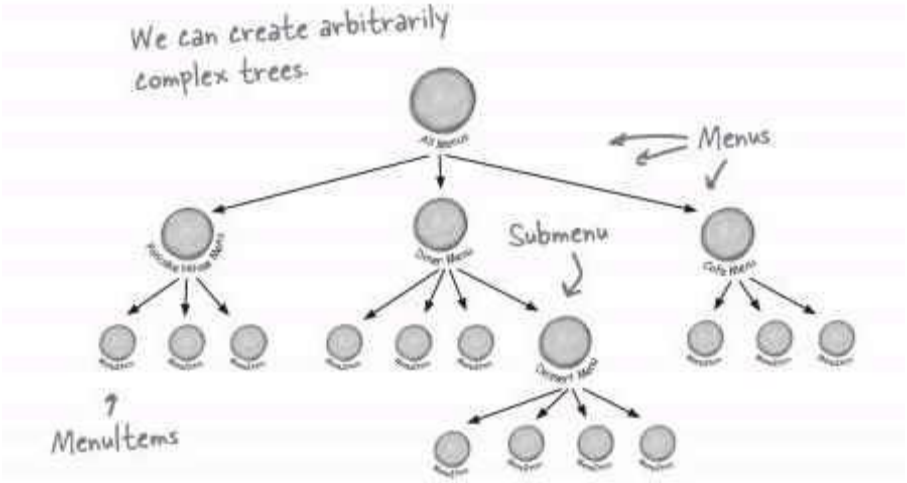
Here's our old friend, the iterator() method. With this method, you can get an Iterator for any class that implements the Collection interface.

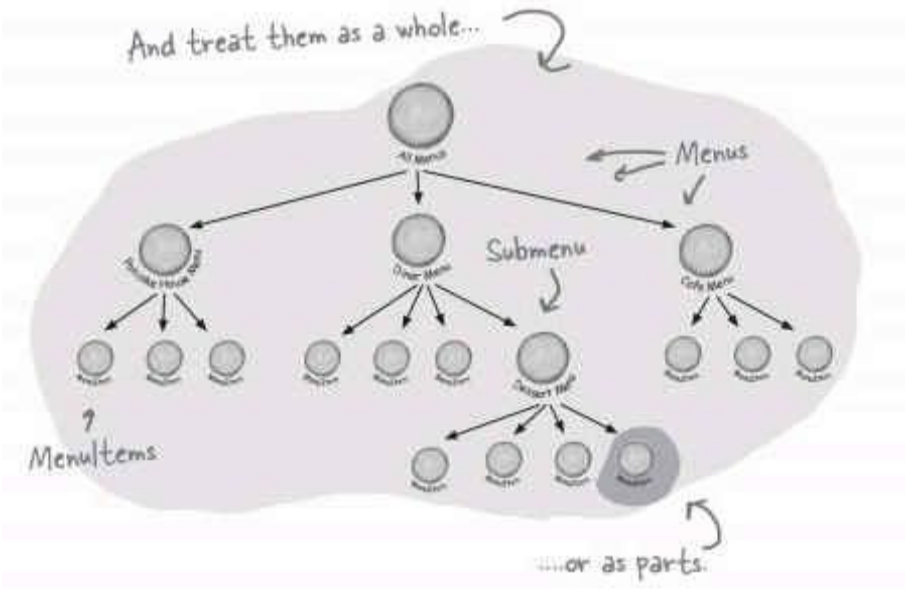
Other handy methods include size(), to get the number of elements, and toArray() to turn your collection into an array.

Watch it!

Hashtable is one of a few classes that *indirectly* supports Iterator. As you saw when we implemented the CafeMenu, you could get an Iterator from it, but only by first retrieving its Collection called values. If you think about it, this makes sense: the Hashtable holds two sets of objects: keys and values. If we want to iterate over its values, we first need to retrieve them from the Hashtable, and then obtain the iterator.

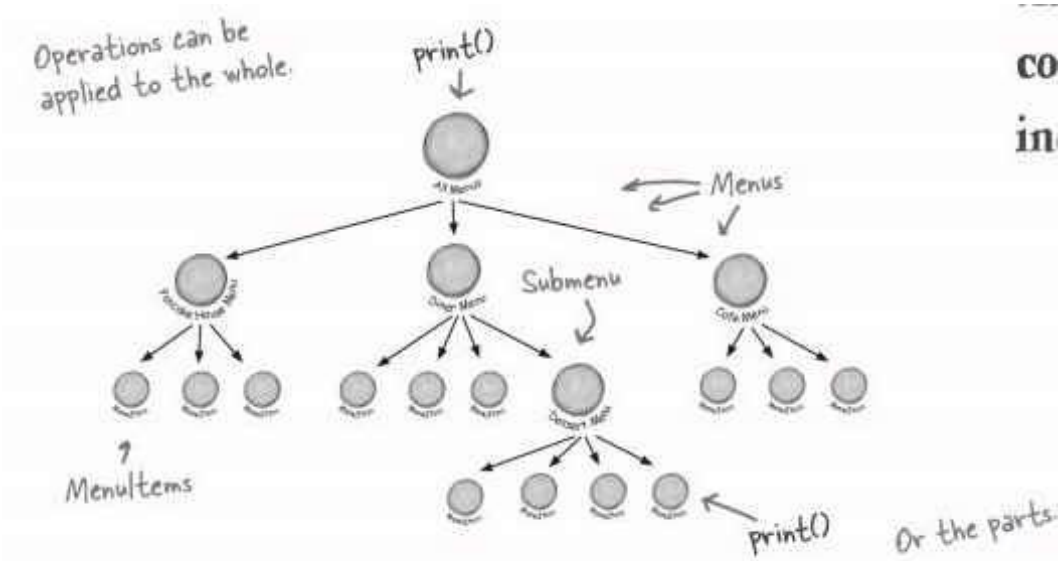
The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



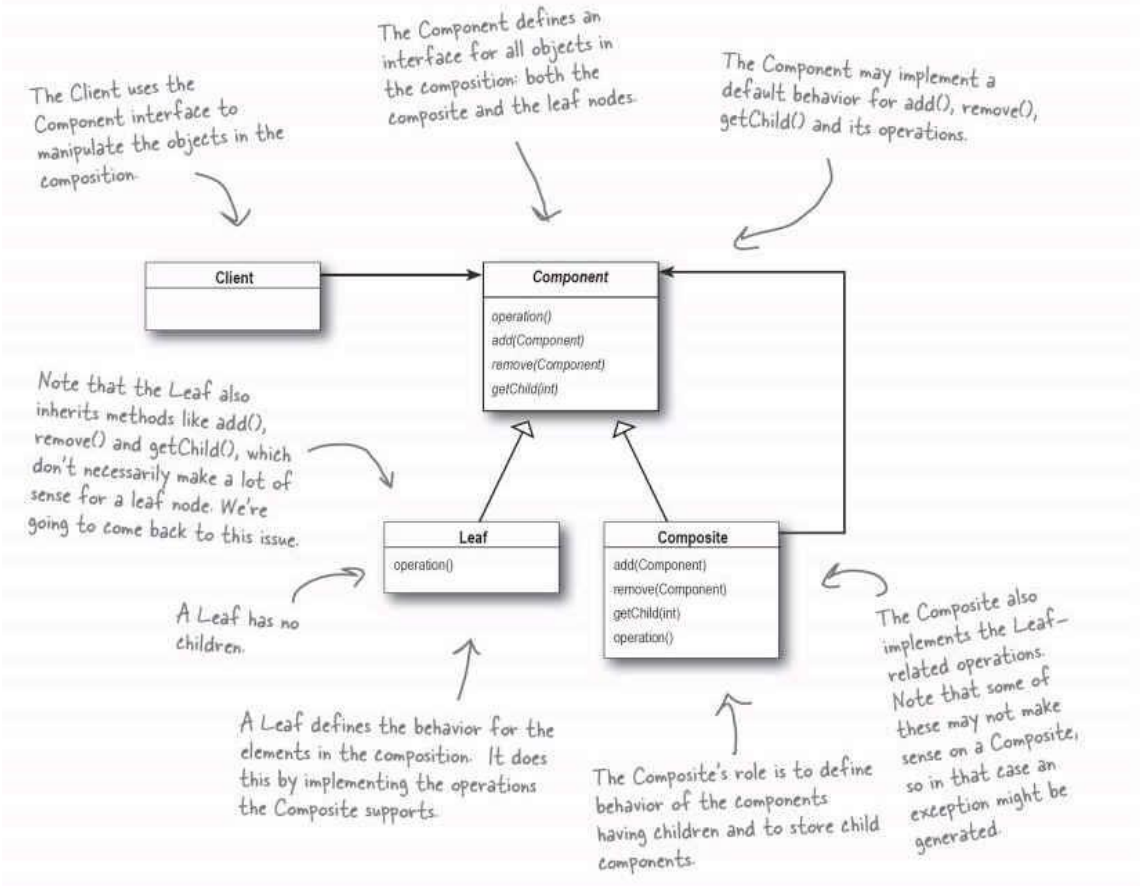


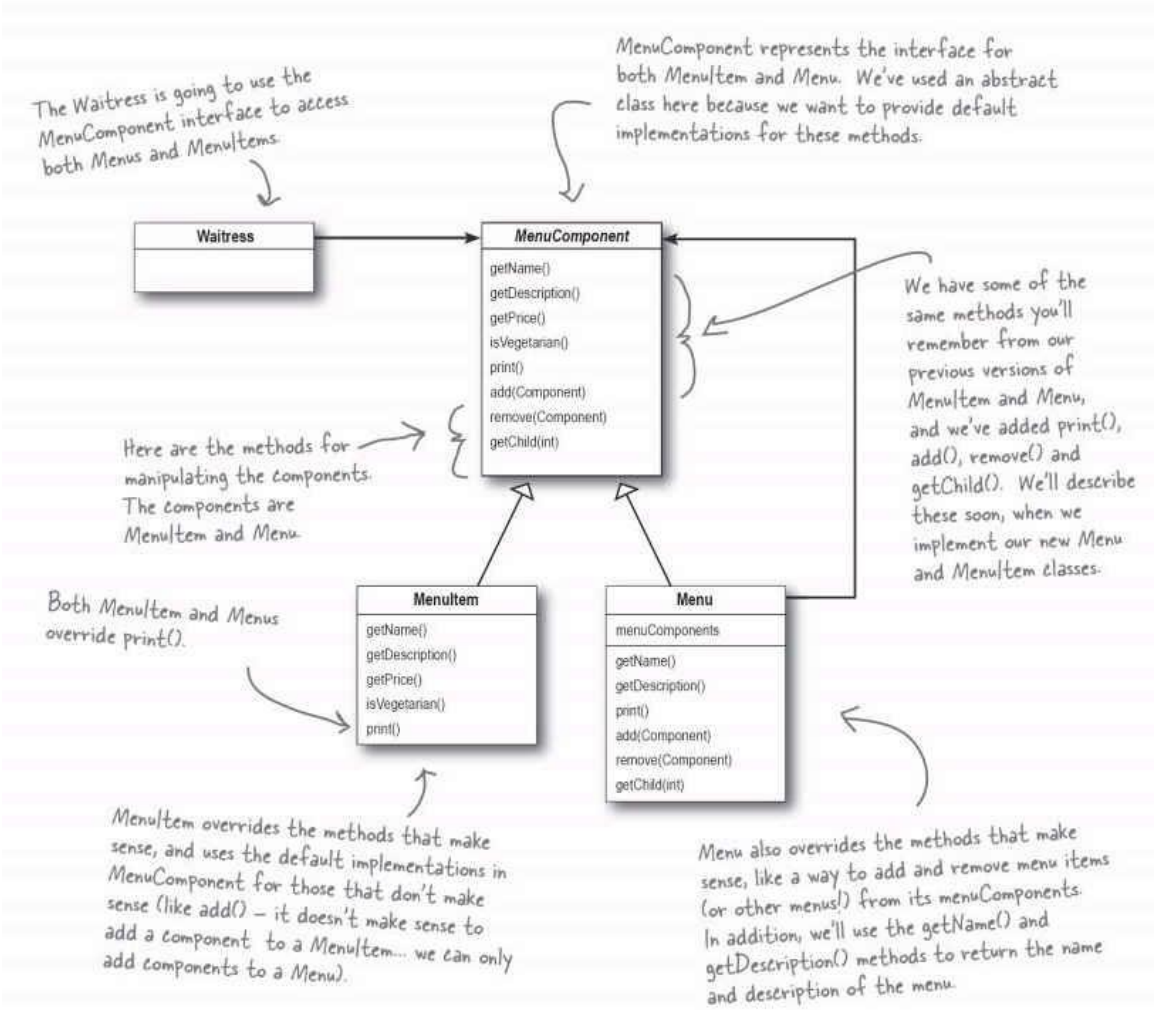
The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



***composite pattern** class diagram*





All components must implement the MenuComponent interface; however, because leaves and nodes have different roles we can't always define a default implementation for each method that makes sense. Sometimes the best you can do is throw a runtime exception.

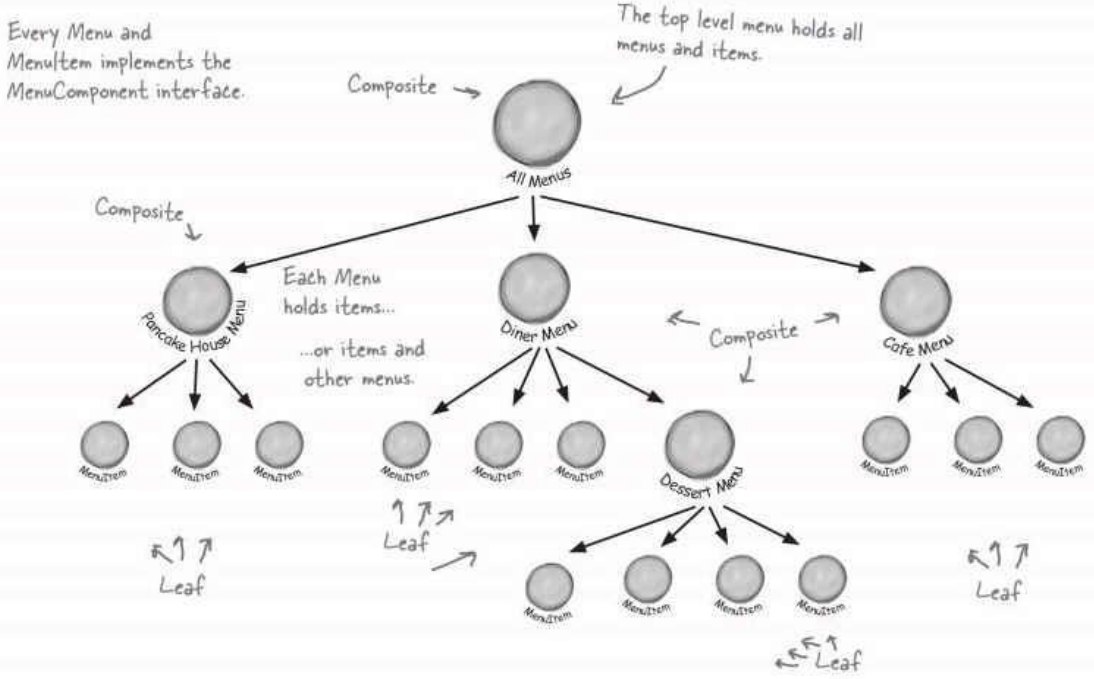
```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItems.

Look! We get to use an Iterator. We use it to iterate through all the Menu's components... those could be other Menus, or they could be MenuItems. Since both Menus and MenuItems implement print(), we just call print() and the rest is up to them.

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

Okay, one last thing before we write our test drive. Let's get an idea of what the menu composite is going to look like at runtime:



There is some truth to that observation. We could say that the Composite Pattern takes the Single Responsibility design principle and trades it for *transparency*. What's transparency? Well, by allowing the Component interface to contain the child management operations *and* the leaf operations, a client can treat both composites and leaf nodes uniformly; so whether an element is a composite or leaf node becomes transparent to the client.

Now given we have both types of operations in the Component class, we lose a bit of *safety* because a client might try to do something inappropriate or meaningless on an element (like try to add a menu to a menu item). This is a design decision; we could take the design in the other direction and separate out the responsibilities into interfaces. This would make our design safe, in the sense that any inappropriate calls on elements would be caught at compile time or runtime, but we'd lose transparency and our code would have to use conditionals and the `instanceof` operator.

So, to return to your question, this is a classic case of tradeoff. We are guided by design principles, but we always need to observe the effect they have on our designs. Sometimes we purposely do things in a way that seems to violate the principle. In some cases, however, this is a matter of perspective; for instance, it might seem incorrect to have child management operations in the leaf nodes (like add(), remove() and getChild()), but then again you can always shift your perspective and see a leaf as a node with zero children.

recursion is my friend."

WATCH OUT: RECURSION ZONE AHEAD

```
import java.util.*;

public class CompositeIterator implements Iterator {
    Stack stack = new Stack();

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Like all iterators, we're implementing the java.util.Iterator interface.

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext()...

If there is a next element, we get the current iterator off the stack and get its next element.

If that element is a menu, we have another composite that needs to be included in the iteration, so we throw it on the stack. In either case, we return the component.

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

Otherwise there is a next element and we return true.

We're not supporting remove, just traversal.

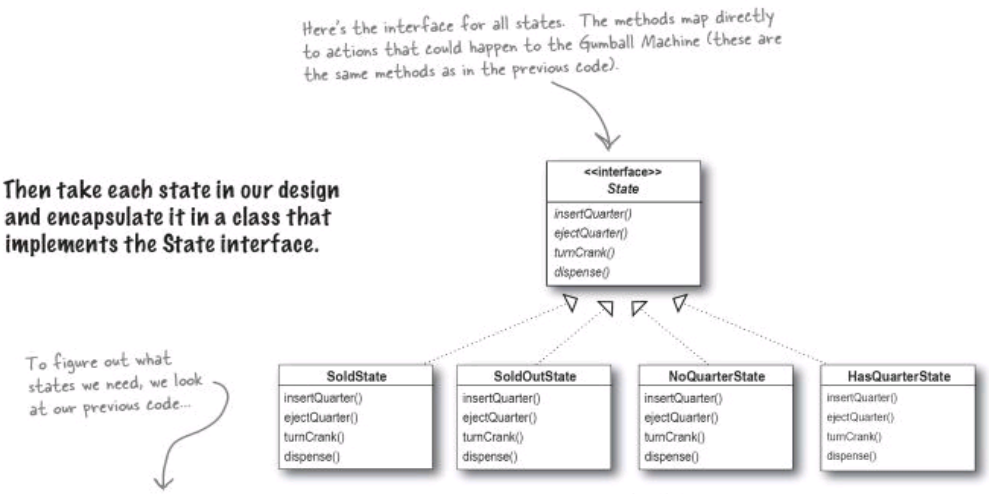
1. **First, we're going to define a State interface that contains a method for every action in the Gumball Machine.**
2. **Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
3. **Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.**

Defining the State interfaces and classes

First let's create an interface for State, which all our states implement:

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).

Then take each state in our design and encapsulate it in a class that implements the State interface.

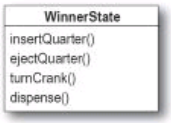


To figure out what states we need, we look at our previous code...

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

... and we map each state directly to a class.

Don't forget, we need a new "winner" state too that implements the state interface. We'll come back to this after we reimplement the first version of the Gumball Machine.



The State Pattern defined

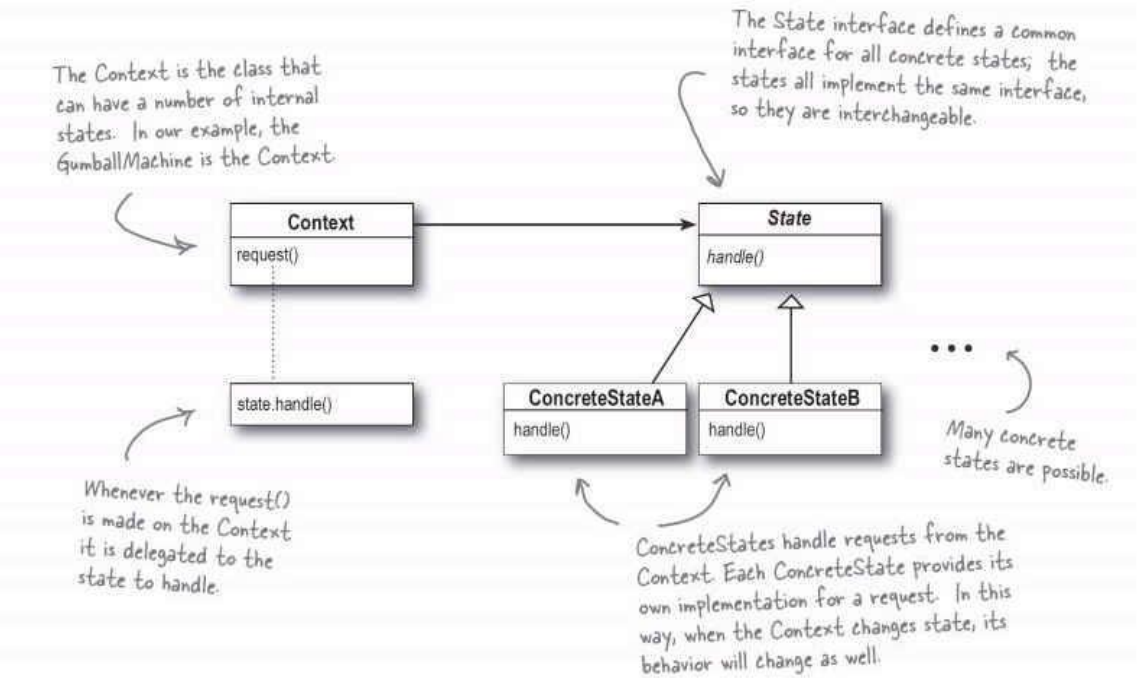
Yes, it's true, we just implemented the State Pattern! So now, let's take a look at what it's all about:

> **The State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The first part of this description makes a lot of sense, right? Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state. The Gumball Machine provides a good example: when the gumball machine is in the NoQuarterState and you insert a quarter, you get different behavior (the machine accepts the quarter) than if you insert a quarter when it's in the HasQuarterState (the machine rejects the quarter).

What about the second part of the definition? What does it mean for an object to "appear to change its class?" Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

Okay, now it's time to check out the State Pattern class diagram:



You've got a good eye! Yes, the class diagrams are essentially the same, but the two patterns differ in their *intent*.

With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

With Strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the flexibility to change the strategy object at runtime, often there is a strategy object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

The Proxy Pattern defined

We've already put a lot of pages behind us in this chapter; as you can see, explaining the Remote Proxy is quite involved. Despite that, you'll see that the definition and class diagram for the Proxy Pattern is actually fairly straightforward. Note that Remote Proxy is one implementation of the general Proxy Pattern; there are actually quite a few variations of the pattern, and we'll talk about them later. For now, let's get the details of the general pattern down.

Here's the Proxy Pattern definition:

> **The Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.

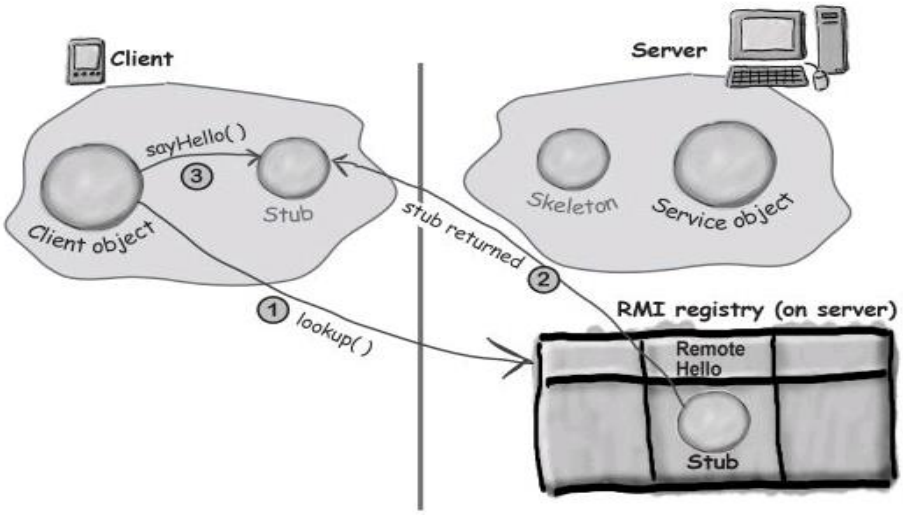
Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.

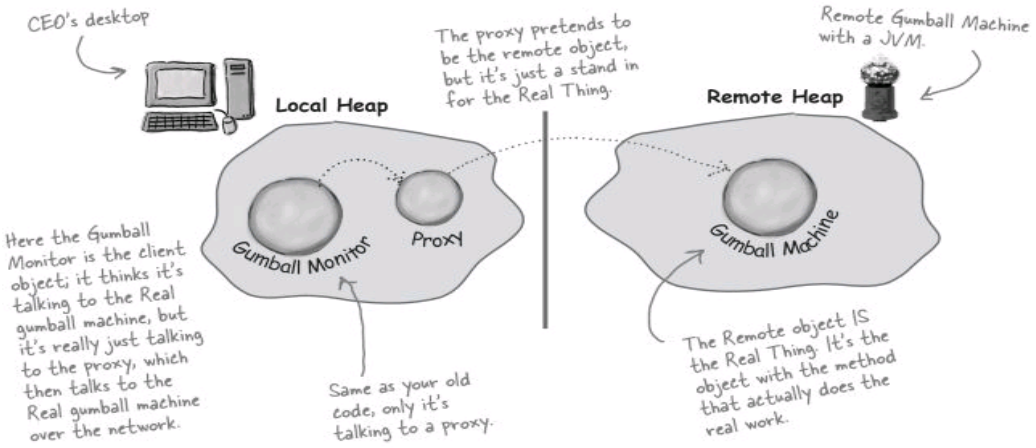
Well, we've seen how the Proxy Pattern provides a surrogate or placeholder for another object. We've also described the proxy as a "representative" for another object.

But what about a proxy controlling access? That sounds a little strange. No worries. In the case of the gumball machine, just think of the proxy controlling access to the remote object. The proxy needed to control access because our client, the monitor, didn't know how to talk to a remote object. So in some sense the remote proxy controlled access so that it could handle the network details for us. As we just discussed, there are many variations of the Proxy Pattern, and the variations typically revolve around the way the proxy "controls access." We're going to talk more about this later, but for now here are a few ways proxies control access:

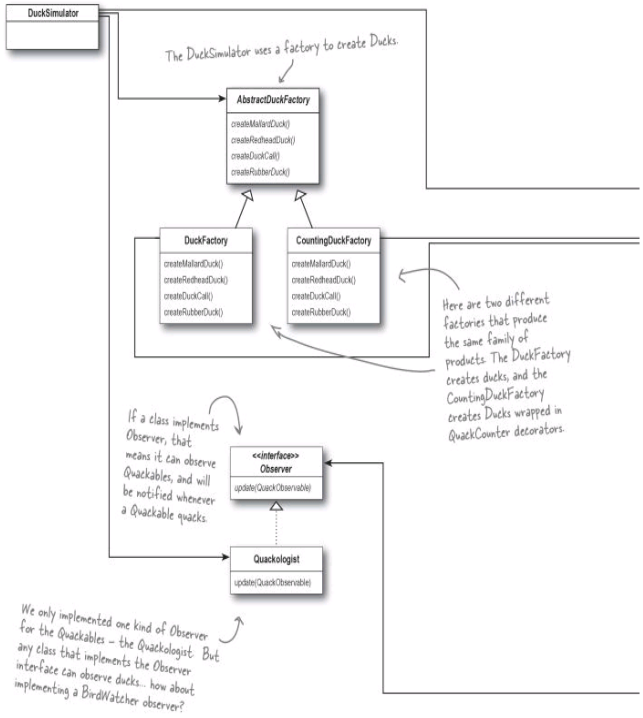
- As we know, a remote proxy controls access to a remote object.
- A virtual proxy controls access to a resource that is expensive to create.
- A protection proxy controls access to a resource based on access rights.

Now that you've got the gist of the general pattern, check out the class diagram.

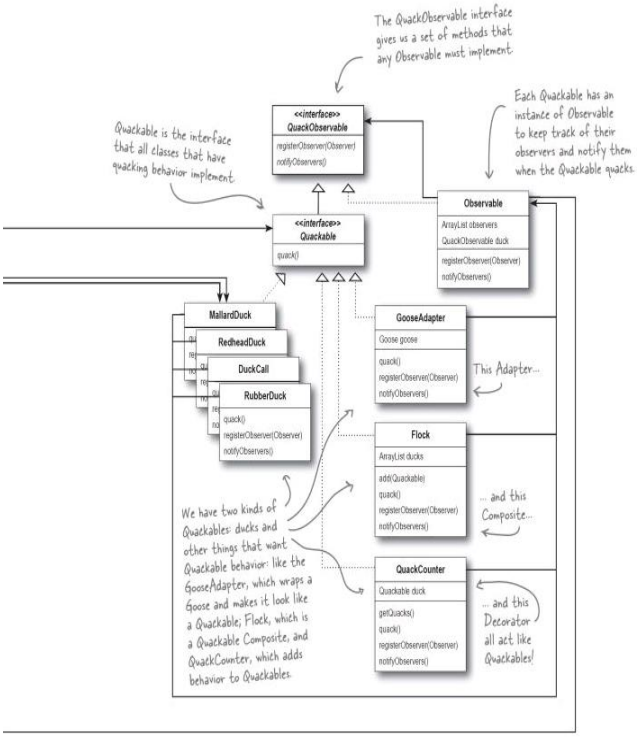




Your client object acts like it's making remote method calls. But what it's really doing is calling methods on a heap-local 'proxy' object that handles all the low-level details of network communication.



24 Chapter 12



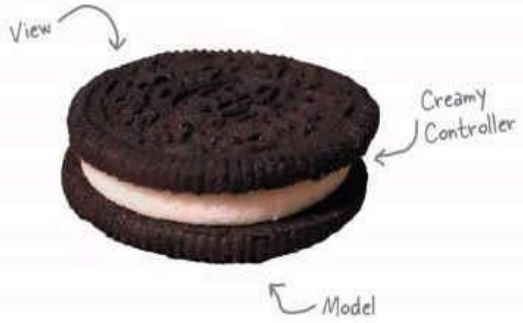
The King of Compound Patterns

If Elvis were a compound pattern, his name would be Model-View-Controller, and he'd be singing a little song like this...

Model, View, Controller

Lyrics and music by James Dempsey.

MVC's a paradigm for factoring your code
into functional segments, so your brain does not explode.
To achieve reusability, you gotta keep those boundaries clean
Model on the one side, View on the other, the Controller's in between.



Model View, it's got three layers like Oreos do
Model View Controller
Model View, Model View, Model View Controller

Model objects represent your applications raison d'être
Custom objects that contain data, logic, and et cetera
You create custom classes, in your app's problem domain
you can choose to reuse them with all the views
but the model objects stay the same.

To synchronize the data of the two

You can model a throttle and a manifold
Model the toddle of a two year old
Model a bottle of fine Chardonnay
Model all the glottal stops people say
Model the coddling of boiling eggs
You can model the waddle in Hexley's legs

Model View, you can model all the models that pose for GQ
Model View Controller

So does Java!

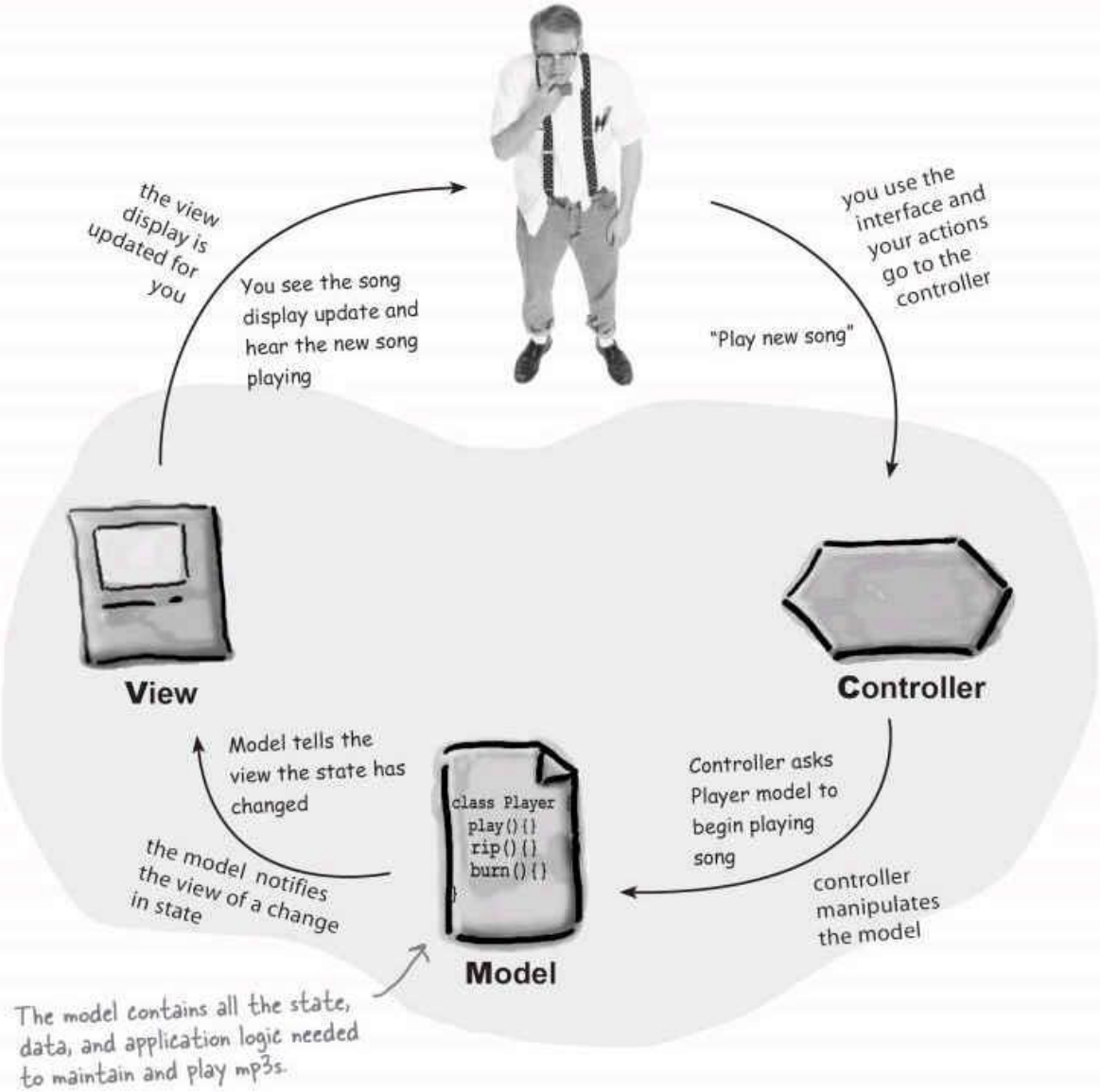
View objects tend to be controls used to display and edit
Cocoa's got a lot of those, well written to its credit.
Take an NSTextView, hand it any old Unicode string
The user can interact with it, it can hold most anything
But the view don't know about the Model
That string could be a phone number or the works of Aristotle
Keep the coupling loose
and so achieve a massive level of reuse

Model View, all rendered very nicely in Aqua blue
Model View Controller

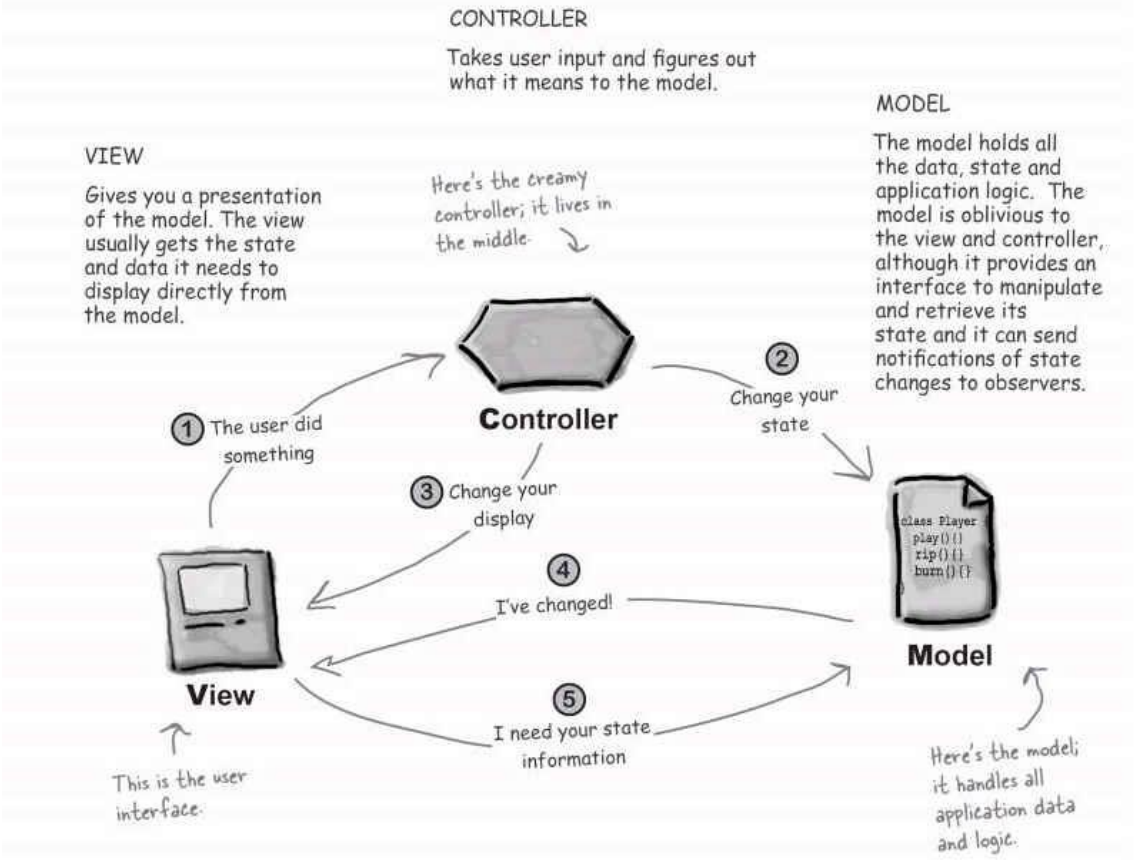
You're probably wondering now
You're probably wondering how
Data flows between Model and View
The Controller has to mediate
Between each layer's changing state

Imagine you're using your favorite MP3 player, like iTunes. You can use its interface to add new songs, manage playlists and rename tracks. The player takes care of maintaining a little database of all your songs along with their associated names and data. It also takes care of playing the songs and, as it does, the user interface is constantly updated with the current song title, the running time, and so on.

Well, underneath it all sits the Model-View-Controller...



The MP3 Player description gives us a high level view of MVC, but it really doesn't help you understand the nitty gritty of how the compound pattern works, how you'd build one yourself, or why it's such a good thing. Let's start by stepping through the relationships among the model, view and controller, and then we'll take second look from the perspective of Design Patterns.



Looking at MVC through patterns-colored glasses

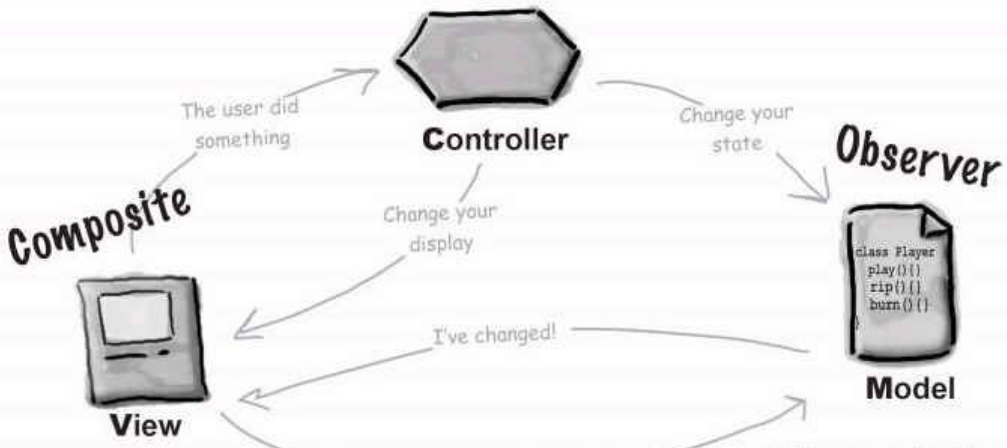
We've already told you the best path to learning the MVC is to see it for what it is: a set of patterns working together in the same design.

Let's start with the model. As you might have guessed the model uses Observer to keep the views and controllers updated on the latest state changes. The view and the controller, on the other hand, implement the Strategy Pattern. The controller is the behavior of the view, and it can be easily exchanged with another controller if you want different behavior. The view itself also uses a pattern internally to manage the windows, buttons and other components of the display: the Composite Pattern.

Let's take a closer look:

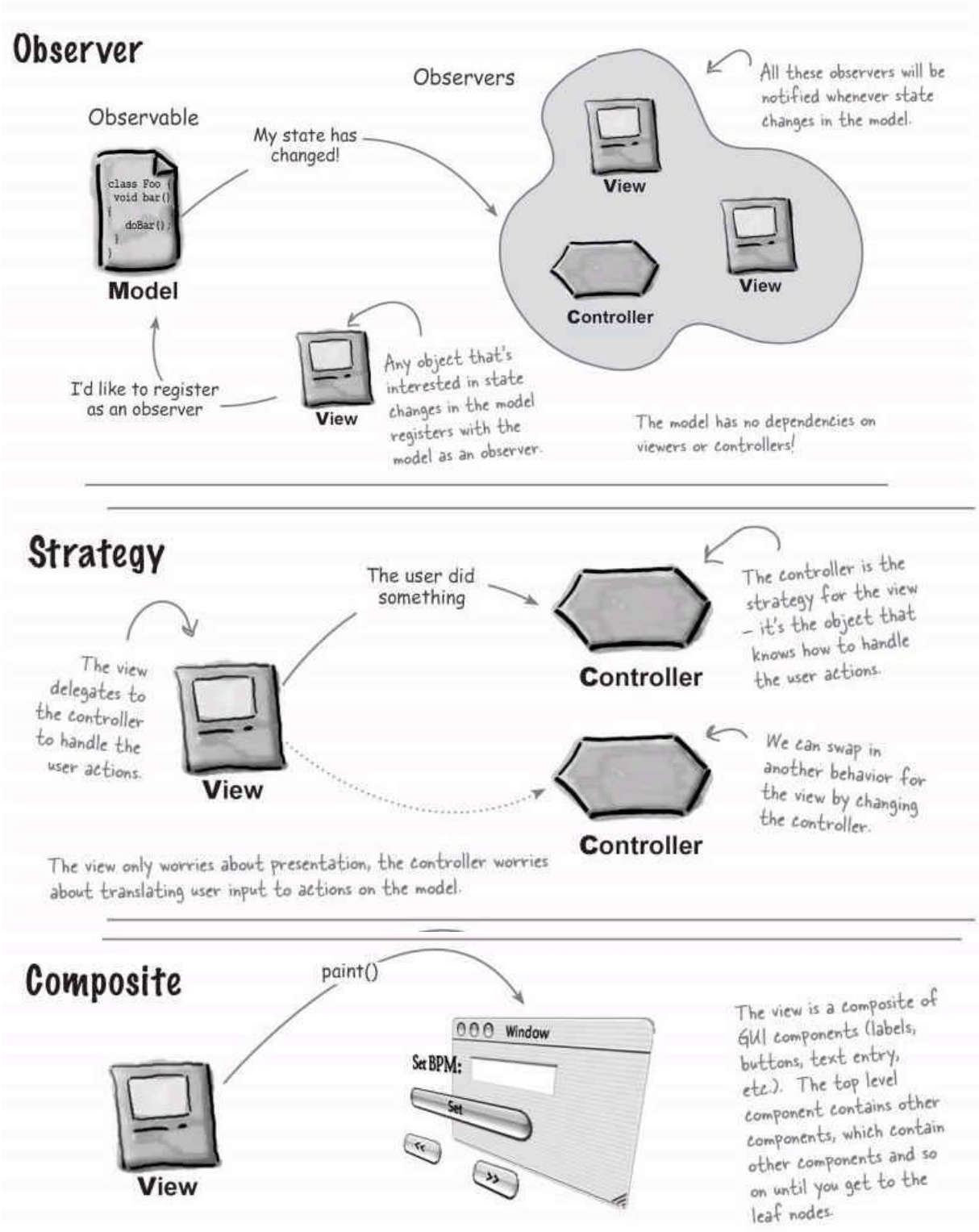
Strategy

The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

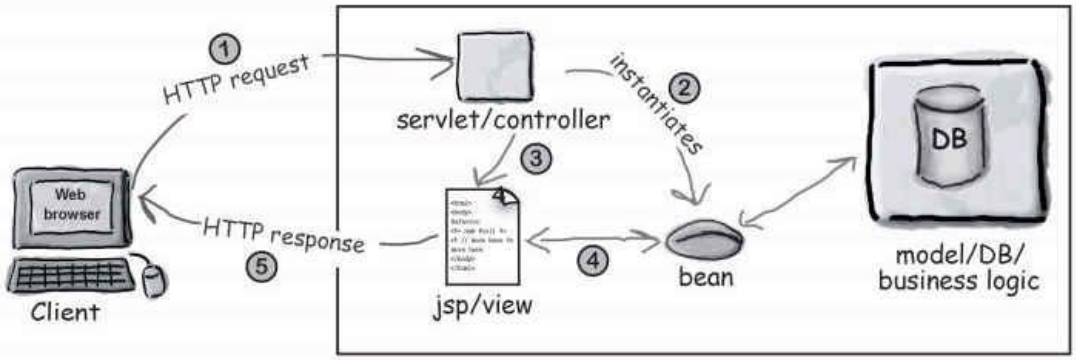
The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.


Observer
Observable
My state has changed!
Observers
All these observers will be notified whenever state changes in the model.
View
View
Controller
Model
I'd like to register as an observer
View
Any object that's interested in state changes in the model registers with the model as an observer.
The model has no dependencies on viewers or controllers!
Strategy
The user did something
The controller is the strategy for the view – it's the object that knows how to handle the user actions.
Controller
The view delegates to the controller to handle the user actions.
View
We can swap in another behavior for the view by changing the controller.
Controller
The view only worries about presentation, the controller worries about translating user input to actions on the model.
Composite
paint()
Window
Set BPM:
Set
View
The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.

MVC and the Web

It wasn't long after the Web was spun that developers started adapting the MVC to fit the browser/server model. The prevailing adaptation is known simply as "Model 2" and uses a combination of servlet and JSP technology to achieve the same separation of model, view and controller that we see in conventional GUIs.

Let's check out how Model 2 works:



1. **You make an HTTP request, which is received by a servlet.**
   Using your web browser you make an HTTP request. This typically involves sending along some form data, like your username and password. A servlet receives this form data and parses it.

2. **The servlet acts as the controller.**
   The servlet plays the role of the controller and processes your request, most likely making requests on the model (usually a database). The result of processing the request is usually bundled up in the form of a JavaBean.

3. **The controller forwards control to the view.**
   The View is represented by a JSP. The JSP's only job is to generate the page representing the view of model (4 which it obtains via the JavaBean) along with any controls needed for further actions.

5. **The view returns a page to the browser via HTTP.**
   A page is returned to the browser, where it is displayed as the view. The user submits further requests, which are processed in the same fashion.

After implementing the DJ Control for the Web using Model 2, you might be wondering where the patterns went. We have a view created in HTML from a JSP but the view is no longer a listener of the model. We have a controller that's a servlet that receives HTTP requests, but are we still using the Strategy Pattern? And what about Composite? We have a view that is made from HTML and displayed in a seb browser. Is that still the Composite Pattern?

Model 2 is an adaptation of MVC to the Web

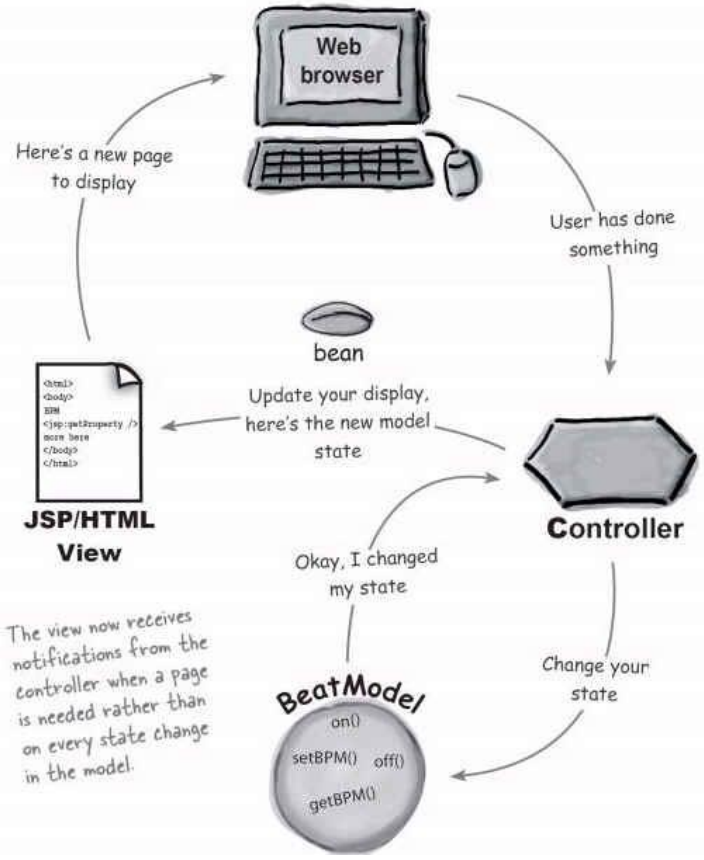
Even though Model 2 doesn't look exactly like "textbook" MVC, all the parts are still there; they've just been adapted to reflect the idiosyncrasies of the web browser model. Let's take another look...

Observer

The view is no longer an observer of the model in the classic sense; that is, it doesn't register with the model to receive state change notifications.

However, the view does receive the equivalent of notifications indirectly from the controller when the model has been changed. The controller even passes the view a bean that allows the view to retrieve the model's state.

If you think about the browser model, the view only needs an update of state information when an HTTP response is returned to the browser; notifications at any other time would be pointless. Only when a page is being created and returned does it make sense to create the view and incorporate the model's state.

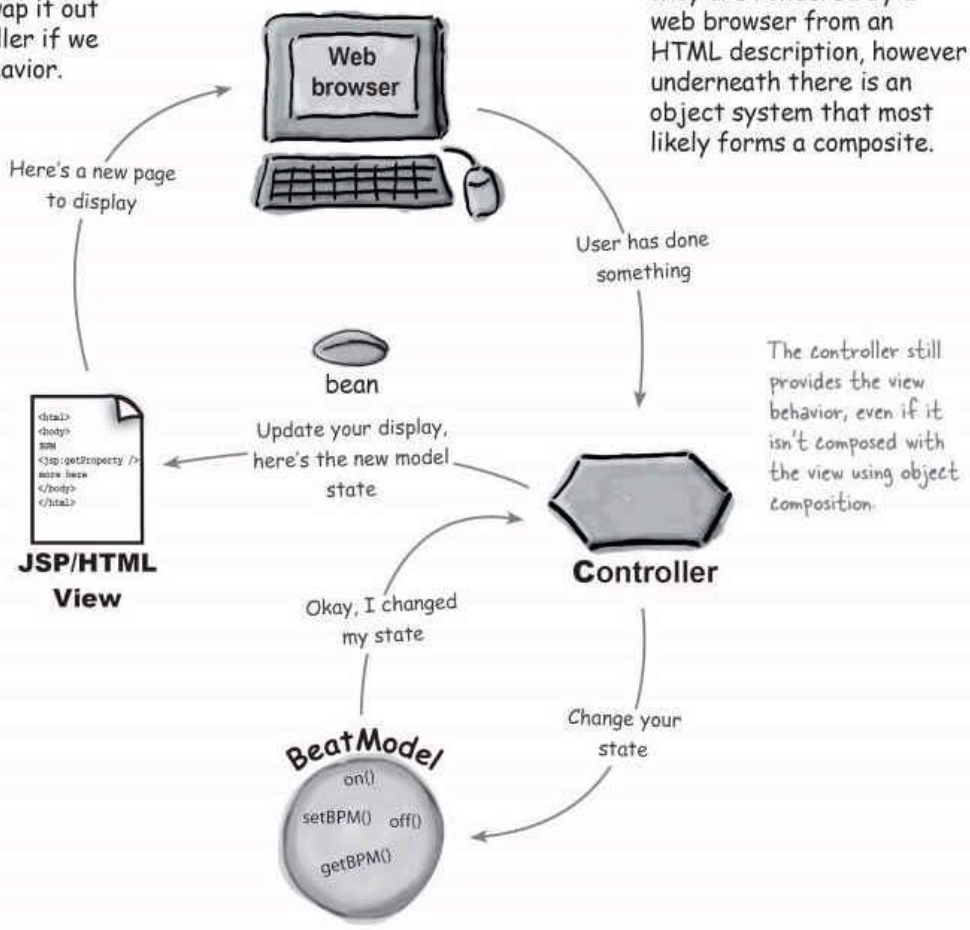


Strategy

In Model 2, the Strategy object is still the controller servlet; however, it's not directly composed with the view in the classic manner. That said, it is an object that implements behavior for the view, and we can swap it out for another controller if we want different behavior.

Composite

Like our Swing GUI, the view is ultimately made up of a nested set of graphical components. In this case, they are rendered by a web browser from an HTML description, however underneath there is an object system that most likely forms a composite.

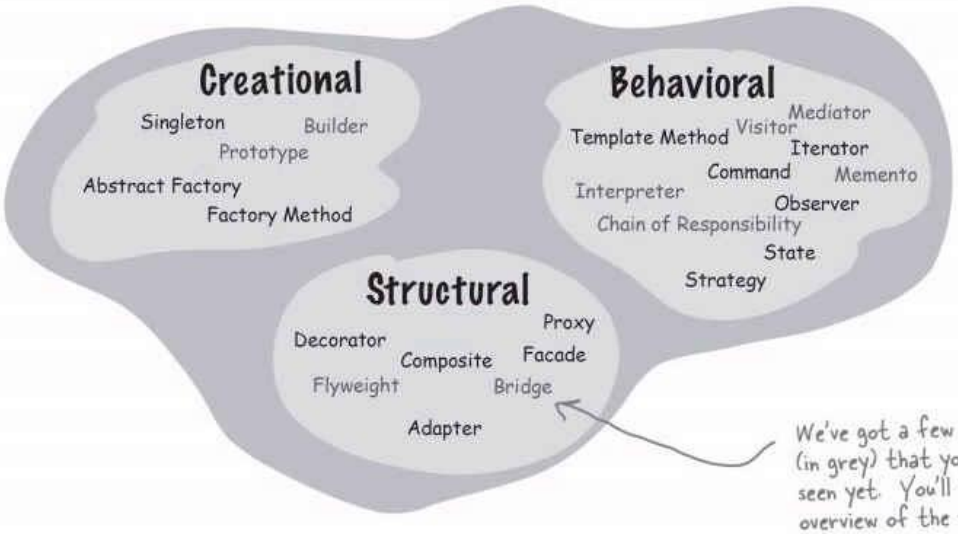


Solution: Pattern Categories

Here's the grouping of patterns into categories. You probably found the exercise difficult, because many of the patterns seem like they could fit into more than one category. Don't worry, everyone has trouble figuring out the right categories for the patterns.

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



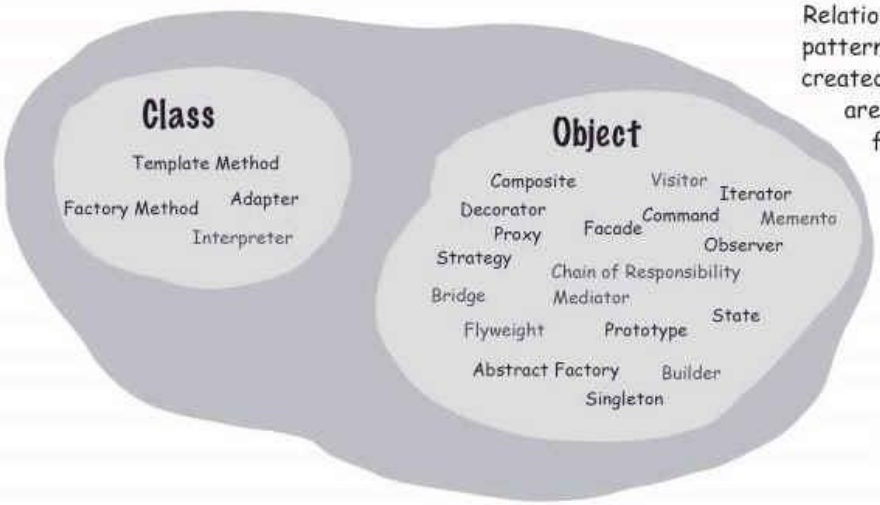
We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of the these patterns in the appendix.

Structural patterns let you compose classes or objects into larger structures.

Patterns are often classified by a second attribute: whether or not the pattern deals with classes or objects:

Class patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Object patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.



Notice there's a lot more object patterns than class patterns!

Q: Are these the only classification schemes?

A: No, other schemes have been proposed. Some other schemes start with the three categories and then add subcategories, like "Decoupling Patterns." You'll want to be familiar with the most common schemes for organizing patterns, but also feel free to create your own, if it helps you to understand the patterns better.

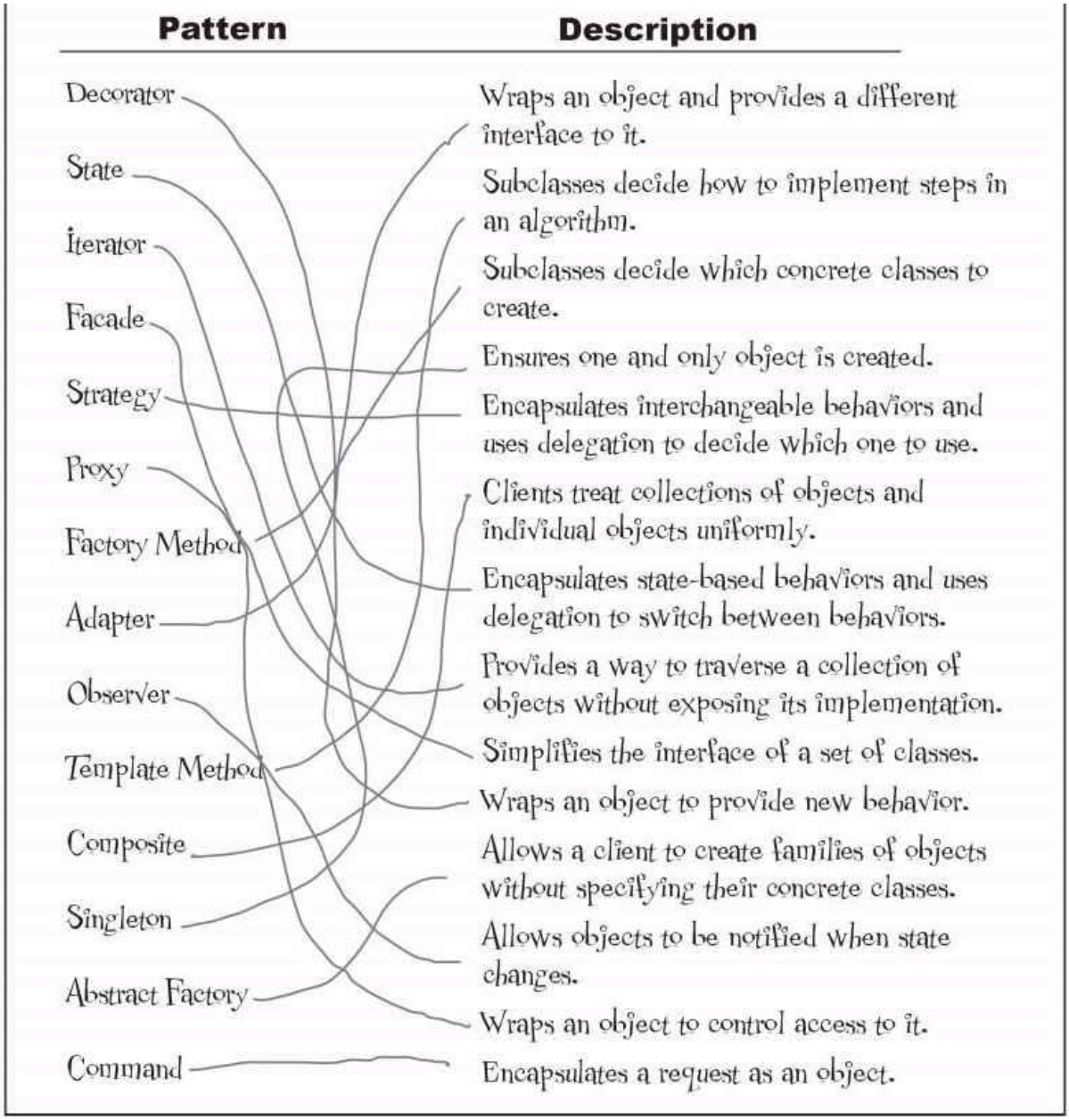
Q: Does organizing patterns into categories really help you remember them?

there are no Dumb Questions

A: It certainly gives you a framework for the sake of comparison. But many people are confused by the creational, structural and behavioral categories; often a pattern seems to fit into more than one category. The most important thing is to know the patterns and the relationships among them. When categories help, use them!

Q: Why is the Decorator Pattern in the structural category? I would have thought of that as a behavioral pattern; after all it adds behavior!

A: Yes, lots of developers say that! Here's the thinking behind the Gang of Four classification: structural patterns describe how classes and objects are composed to create new structures or new functionality. The Decorator Pattern allows you to compose objects by wrapping one object with another to provide new functionality. So the focus is on how you compose the objects dynamically to gain functionality, rather than on the communication and interconnection between objects, which is the purpose of behavioral patterns. It's a subtle distinction, especially when you consider the structural similarities between Decorator (a structural pattern) and Proxy, (a behavioral pattern). But remember, the intent of these patterns is different, and that's often the key to understanding which category a pattern belongs to.

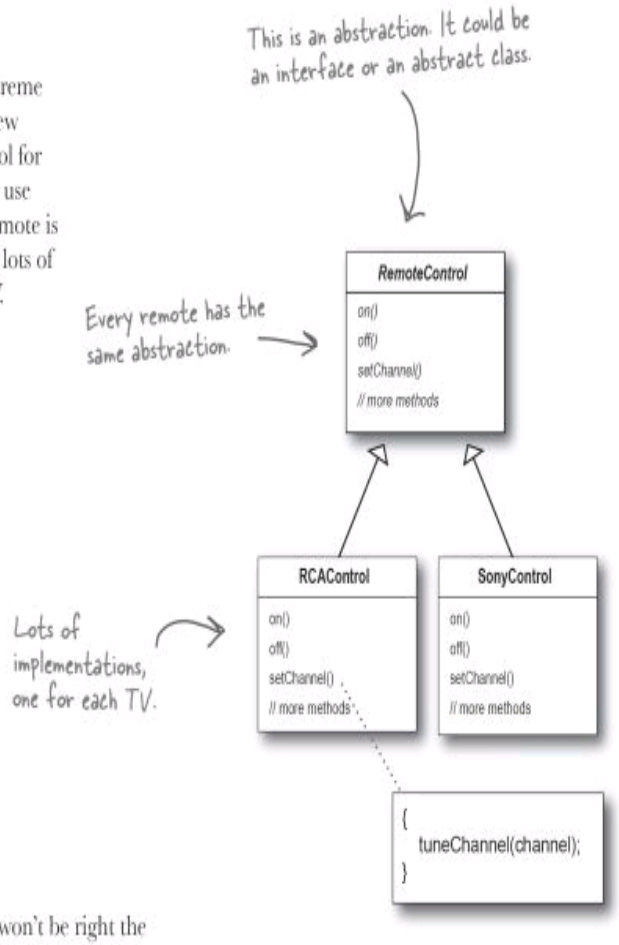

Pattern
Description
Decorator
State
Iterator
Facade
Strategy
Proxy
Factory Method
Adapter
Observer
Template Method
Composite
Singleton
Abstract Factory
Command
Wraps an object and provides a different interface to it.
Subclasses decide how to implement steps in an algorithm.
Subclasses decide which concrete classes to create.
Ensures one and only object is created.
Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Clients treat collections of objects and individual objects uniformly.
Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Provides a way to traverse a collection of objects without exposing its implementation.
Simplifies the interface of a set of classes.
Wraps an object to provide new behavior.
Allows a client to create families of objects without specifying their concrete classes.
Allows objects to be notified when state changes.
Wraps an object to control access to it.
Encapsulates a request as an object.

Bridge

Use the Bridge Pattern to vary not only your implementations, but also your abstractions.

A scenario

Imagine you're going to revolutionize "extreme lounging." You're writing the code for a new ergonomic and user-friendly remote control for TVs. You already know that you've got to use good OO techniques because while the remote is based on the same *abstraction*, there will be lots of *implementations* – one for each model of TV.



Your dilemma

You know that the remote's user interface won't be right the first time. In fact, you expect that the product will be refined many times as usability data is collected on the remote control.

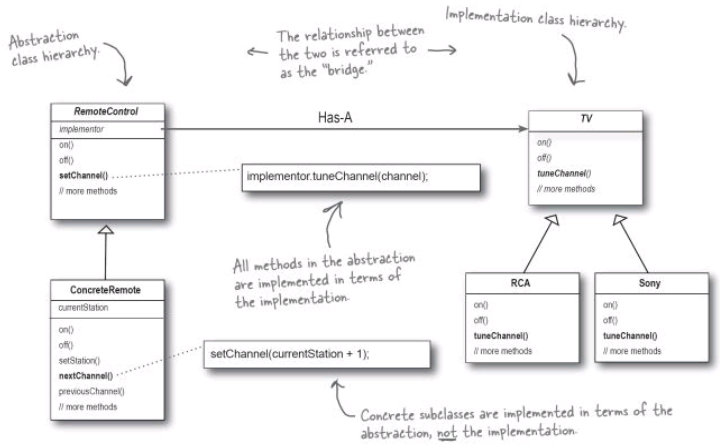
So your dilemma is that the remotes are going to change and the TVs are going to change. You've already *abstracted* the user interface so that you can vary the *implementation* over the many TVs your customers will own. But you are also going to need to *vary the abstraction* because it is going to change over time as the remote is improved based on the user feedback.

So how are you going to create an OO design that allows you to vary the implementation *and* the abstraction?

Using this design we can vary only the TV implementation, not the user interface.

Why use the Bridge Pattern?

The Bridge Pattern allows you to vary the implementation *and* the abstraction by placing the two in separate class hierarchies.



Now you have two hierarchies, one for the remotes and a separate one for platform specific TV implementations. The bridge allows you to vary either side of the two hierarchies independently.

Bridge Benefits

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don't affect the client.

Bridge Uses and Drawbacks

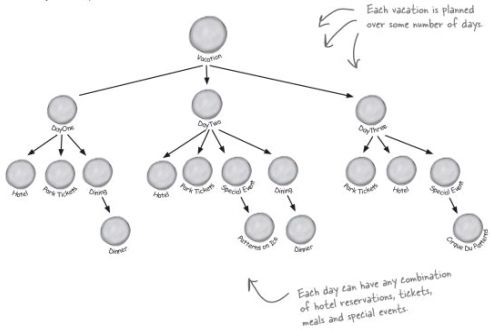
- Useful in graphic and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity.

Builder

Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

A scenario

You've just been asked to build a vacation planner for Patternsland, a new theme park just outside of Objectville. Park guests can choose a hotel and various types of admission tickets, make restaurant reservations, and even book special events. To create a vacation planner, you need to be able to create structures like this:



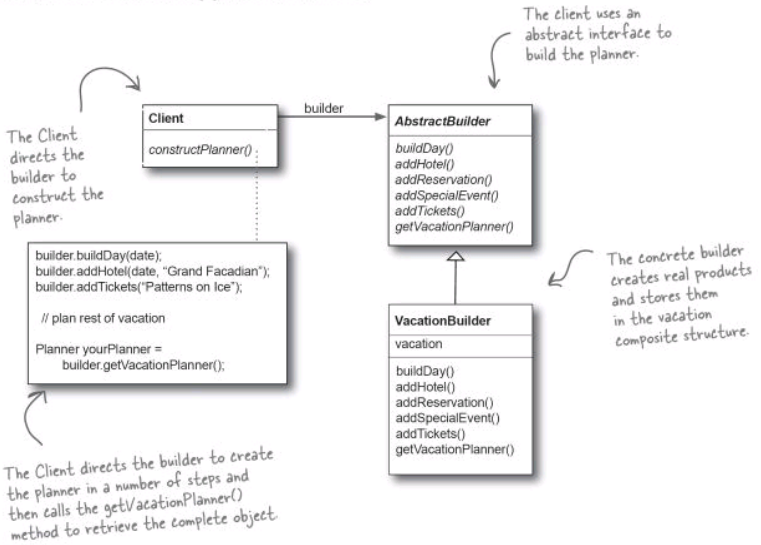
You need a flexible design

Each guest's planner can vary in the number of days and types of activities it includes. For instance, a local resident might not need a hotel, but wants to make dinner and special event reservations. Another guest might be flying into Objectville and needs a hotel, dinner reservations, and admission tickets.

So, you need a flexible data structure that can represent guest planners and all their variations; you also need to follow a sequence of potentially complex steps to create the planner. How can you provide a way to create the complex structure without mixing it with the steps for creating it?

Why use the Builder Pattern?

Remember Iterator? We encapsulated the iteration into a separate object and hid the internal representation of the collection from the client. It's the same idea here: we encapsulate the creation of the trip planner in an object (let's call it a builder), and have our client ask the builder to construct the trip planner structure for it.



Builder Benefits

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.

Builder Uses and Drawbacks

- Often used for building composite structures.
- Constructing objects requires more domain knowledge of the client than when using a Factory.

Chain of Responsibility

Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request.

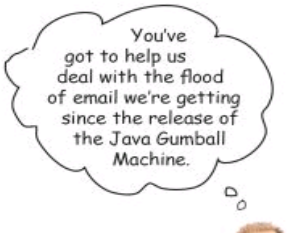
A scenario

Mighty Gumball has been getting more email than they can handle since the release of the Java-powered Gumball Machine. From their own analysis they get four kinds of email: fan mail from customers that love the new 1 in 10 game, complaints from parents whose kids are addicted to the game and requests to put machines in new locations. They also get a fair amount of spam.

All fan mail needs to go straight to the CEO, all complaints go to the legal department and all requests for new machines go to business development. Spam needs to be deleted.

Your task

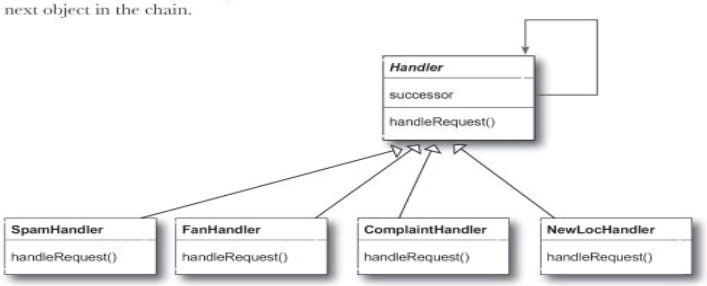
Mighty Gumball has already written some AI detectors that can tell if an email is spam, fan mail, a complaint, or a request, but they need you to create a design that can use the detectors to handle incoming email.



How to use the Chain of Responsibility Pattern

With the Chain of Responsibility Pattern, you create a chain of objects that examine a request. Each object in turn examines the request and handles it, or passes it on to the next object in the chain.

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



As email is received, it is passed to the first handler: the SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain – although, you can always implement a catch-all handler.

Chain of Responsibility Benefits

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Chain of Responsibility Uses and Drawbacks

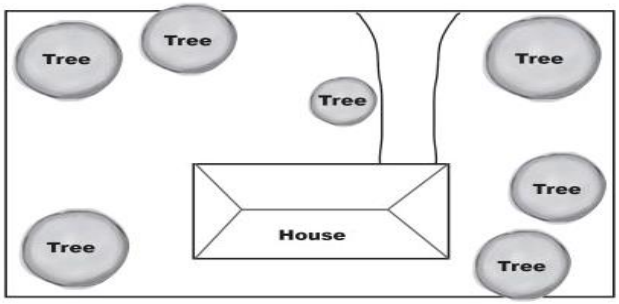
- Commonly used in windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe the runtime characteristics and debug.

Flyweight

Use the Flyweight Pattern when one instance of a class can be used to provide many "virtual instances."

A scenario

You want to add trees as objects in your hot new landscape design application. In your application, trees don't really do very much; they have an X-Y location, and they can draw themselves dynamically, depending on how old they are. The thing is, a user might want to have lots and lots of trees in one of their home landscape designs. It might look something like this:



Each Tree instance maintains its own state.

```
Tree
xCoord
yCoord
age
display() {
  // use X-Y coords
  // & complex age
  // related calcs
}
```

Your big client's dilemma

You've just landed your "reference account." That key client you've been pitching for months. They're going to buy 1,000 seats of your application, and they're using your software to do the landscape design for huge planned communities. After using your software for a week, your client is complaining that when they create large groves of trees, the app starts getting sluggish...

Why use the Flyweight Pattern?

What if, instead of having thousands of Tree objects, you could redesign your system so that you've got only one instance of Tree, and a client object that maintains the state of ALL your trees? That's the Flyweight!

All the state, for ALL of your virtual Tree objects, is stored in this 2D-array.

```
TreeManager
treeArray
displayTrees() {
  // for all trees {
    // get array row
    display(x, y, age);
  }
}
```

One, single, state-free Tree object.

```
Tree
display(x, y, age) {
  // use X-Y coords
  // & complex age
  // related calcs
}
```

Flyweight Benefits

- Reduces the number of object instances at runtime, saving memory.
- Centralizes state for many "virtual" objects into a single location.

Flyweight Uses and Drawbacks

- The Flyweight is used when a class has many instances, and they can all be controlled identically.
- A drawback of the Flyweight pattern is that once you've implemented it, single, logical instances of the class will not be able to behave independently from the other instances.

Interpreter

Use the Interpreter Pattern to build an interpreter for a language.

A scenario

Remember the Duck Pond Simulator? You have a hunch it would also make a great educational tool for children to learn programming. Using the simulator, each child gets to control one duck with a simple language. Here's an example of the language:

```
right;
while (daylight) fly
quack;
```

Turn the duck right

Now, remembering how to create introductory programming classes

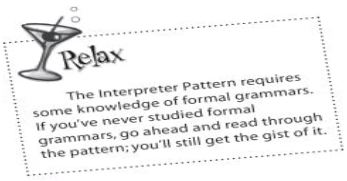
```
expression ::=  <com
sequence ::= <expres
command ::= right |
repetition ::= while
variable ::= [A-Z,a-
```

A program is an expression consisting of sequences of commands and repetitions ("while" statements).

A sequence is a set of expressions separated by semicolons.

We have three commands: right, quack, and fly.

A while statement is just a conditional variable and an expression.

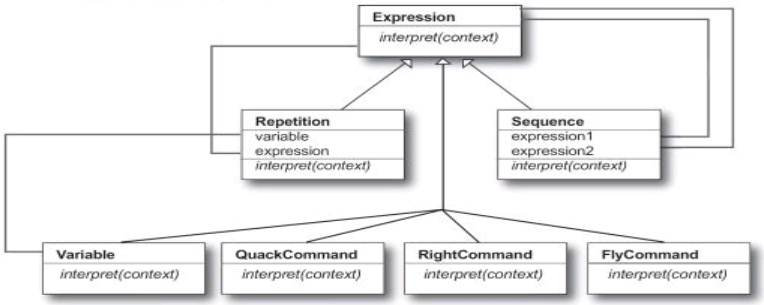


Now what?

You've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.

How to implement an interpreter

When you need to implement a simple language, the Interpreter Pattern defines a class-based representation for its grammar along with an interpreter to interpret its sentences. To represent the language, you use a class to represent each rule in the language. Here's the duck language translated into classes. Notice the direct mapping to the grammar.



To interpret the language, call the interpret() method on each expression type. This method is passed a context – which contains the input stream of the program we're parsing – and matches the input and evaluates it.

Interpreter Benefits

- Representing each grammar rule in a class makes the language easy to implement.
- Because the grammar is represented by classes, you can easily change or extend the language.
- By adding additional methods to the class structure, you can add new behaviors beyond interpretation, like pretty printing and more sophisticated program validation.

Interpreter Uses and Drawbacks

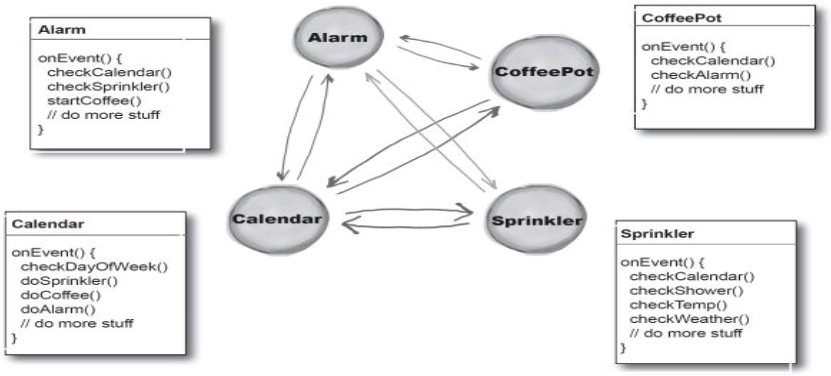
- Use interpreter when you need to implement a simple language.
- Appropriate when you have a simple grammar and simplicity is more important than efficiency.
- Used for scripting and programming languages.
- This pattern can become cumbersome when the number of grammar rules is large. In these cases a parser/compiler generator may be more appropriate.

Mediator

Use the Mediator Pattern to centralize complex communications and control between related objects.

A scenario

Bob has a Java-enabled auto-house, thanks to the good folks at HouseOfTheFuture. All of his appliances are designed to make his life easier. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing. Even though life is good for Bob, he and other clients are always asking for lots of new features: No coffee on the weekends... Turn off the sprinkler 15 minutes before a shower is scheduled... Set the alarm early on trash days...



HouseOfTheFuture's dilemma

It's getting really hard to keep track of which rules reside in which objects, and how the various objects should relate to each other.

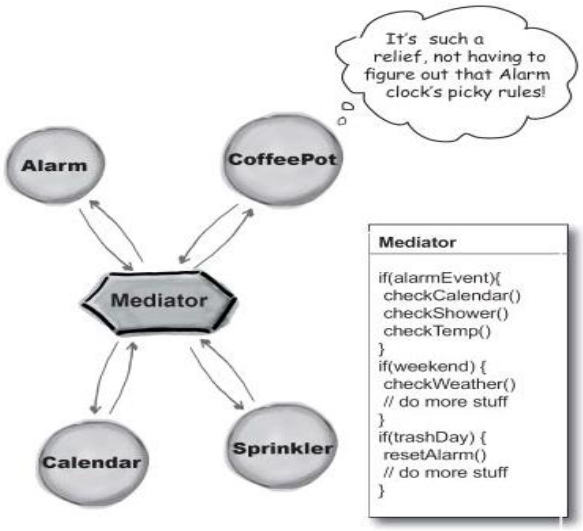
Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified:

- They tell the Mediator when their state changes.
- They respond to requests from the Mediator.

Before adding the Mediator, all of the appliance objects needed to know about each other... they were all tightly coupled. With the Mediator in place, the appliance objects are all *completely decoupled* from each other.

The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.



Mediator Benefits

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system.
- Simplifies maintenance of the system by centralizing control logic.
- Simplifies and reduces the variety of messages sent between objects in the system.

Mediator Uses and Drawbacks

- The Mediator is commonly used to coordinate related GUI components.
- A drawback of the Mediator pattern is that without proper design, the Mediator object itself can become overly complex.

Memento

Use the Memento Pattern when you need to be able to return an object to one of its previous states; for instance, if your user requests an "undo."

A scenario

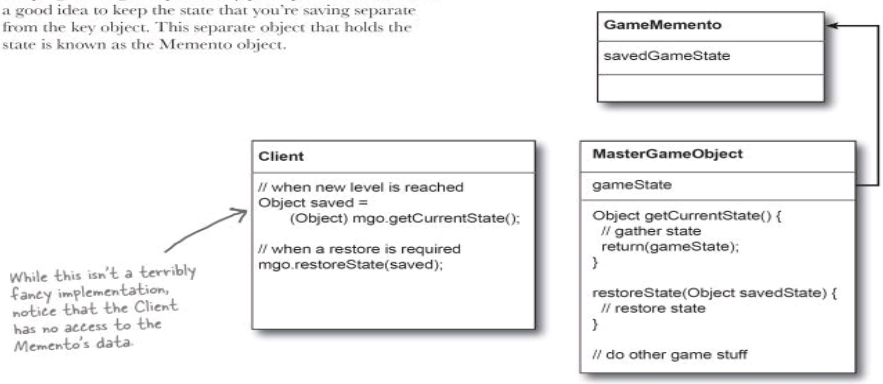
Your interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled "level 13." As users progress to more challenging game levels, the odds of encountering a game-ending situation increase. Fans who have spent days progressing to an advanced level are understandably miffed when their character gets snuffed, and they have to start all over. The cry goes out for a "save progress" command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The "save progress" function needs to be designed to return a resurrected player to the last level she completed successfully.

The Memento at work

The Memento has two goals:

- Saving the important state of a system's key object.
- Maintaining the key object's encapsulation.

Keeping the single responsibility principle in mind, it's also a good idea to keep the state that you're saving separate from the key object. This separate object that holds the state is known as the Memento object.



Memento Benefits

- Keeping the saved state external from the key object helps to maintain cohesion.
- Keeps the key object's data encapsulated.
- Provides easy-to-implement recovery capability.

Memento Uses and Drawbacks

- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time consuming.
- In Java systems, consider using Serialization to save a system's state.

Prototype

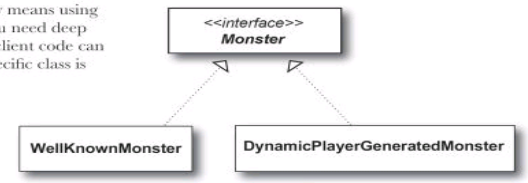
Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.

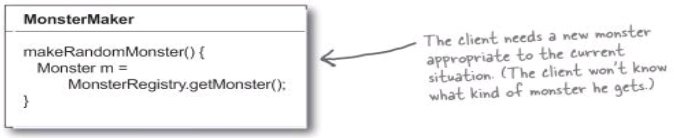
A scenario

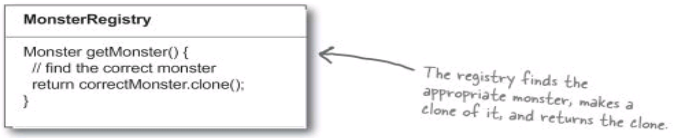
Your interactive role playing game has an insatiable appetite for monsters. As your heros make their journey through a dynamically created landscape, they encounter an endless chain of foes that must be subdued. You'd like the monster's characteristics to evolve with the changing landscape. It doesn't make a lot of sense for bird-like monsters to follow your characters into underseas realms. Finally, you'd like to allow advanced players to create their own custom monsters.

Prototype to the rescue

The Prototype Pattern allows you to make new instances by copying existing instances. (In Java this typically means using the clone() method, or de-serialization when you need deep copies.) A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated.







Prototype Benefits

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Prototype Uses and Drawbacks

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

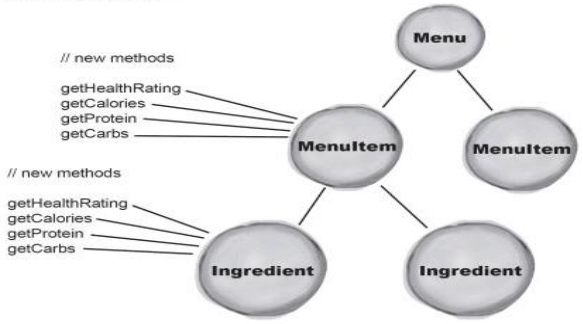
Visitor

Use the Visitor Pattern when you want to add capabilities to a composite of objects and encapsulation is not important.

A scenario

Customers who frequent the Objectville Diner and Objectville Pancake House have recently become more health conscious. They are asking for nutritional information before ordering their meals. Because both establishments are so willing to create special orders, some customers are even asking for nutritional information on a per ingredient basis.

Lou's proposed solution:

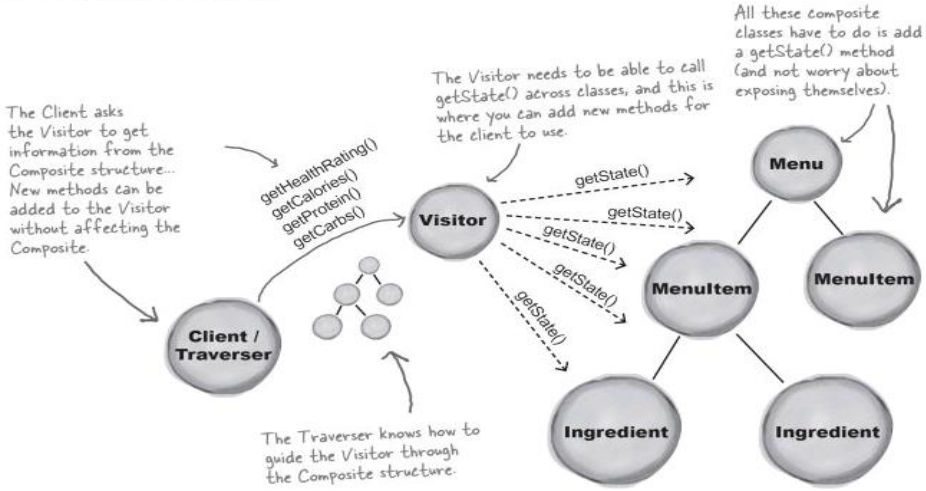


Mel's concerns...

"Boy, it seems like we're opening Pandora's box. Who knows what new method we're going to have to add next, and every time we add a new method we have to do it in two places. Plus, what if we want to enhance the base application with, say, a recipes class? Then we'll have to make these changes in three different places..."

The Visitor drops by

The Visitor must visit each element of the Composite; that functionality is in a Traverser object. The Visitor is guided by the Traverser to gather state from all of the objects in the Composite. Once state has been gathered, the Client can have the Visitor perform various operations on the state. When new functionality is required, only the Visitor must be enhanced.



Visitor Benefits

- Allows you to add operations to a Composite structure without changing the structure itself.
- Adding new operations is relatively easy.
- The code for operations performed by the Visitor is centralized.

Visitor Drawbacks

- The Composite classes' encapsulation is broken when the Visitor is used.
- Because the traversal function is involved, changes to the Composite structure are more difficult.

Core J2EE:

Presentation Tier:

Intercepting Filer: You want to intercept and manipulate a request and a response before and after the request is processed. Use an Intercepting Filter as a pluggable filter to pre and post process requests and responses. A filter manager combines loosely coupled filters in a chain, delegating control to the appropriate filter. In this way, you can add, remove, and combine these filters in various ways without changing existing code.

Front Controller: Centralized Access: Use a Front Controller as the initial point of contact for handling all related requests. The Front Controller centralizes control logic that might otherwise be duplicated, and manages the key request handling activities.

Application Controllers: Centralized and modularize action and view management. Use an Application Controller to centralize retrieval and invocation of request-processing components, such as commands and views.

View Helper: separate view from processing logic: Use Views to encapsulate formatting code and Helpers to encapsulate view-processing logic. A View delegates its processing responsibilities to its helper classes, implemented as POJOs, custom tags, or tag files. Helpers serve as adapters between the view and the model, and perform processing related to formatting logic, such as generating an HTML table.

Composite View: Use Composite Views that are composed of multiple atomic subviews. Each subview of the overall template can be included dynamically in the whole, and the layout of the page can be managed independently of the content.

Service To Worker: Use Service to Worker to centralize control and request handling to retrieve a presentation model before turning control over to the view. The view generates a dynamic response based on the presentation model.

Business Tier:

Business Delegate: Use a Business Delegate to encapsulate access to a business service. The Business Delegate hides the implementation details of the business service, such as lookup and access mechanisms.

Service Locator: *transparently locate business components and services in a uniform manner****.*** Use a Service Locator to implement and encapsulate service and component lookup. A Service Locator hides the implementation details of the lookup mechanism and encapsulates related dependencies.

Session Façade: *expose business components and services to remote* ***clients*** Use a ***Session Façade*** to encapsulate business-tier components and expose a coarse-grained service to remote clients. Clients access a ***Session Façade*** instead of accessing business components directly

Application Service: (a.k.a App controller) *centralize business logic across several business-tier components and services.* Application Service to centralize and aggregate behavior to provide a uniform service layer.

Business Object: **Business** Objects to separate business data and logic using an object model

Composite Entity: *entity beans to implement your conceptual domain model. (local entity bean)* Use a Composite Entity to implement persistent Business Objects using local entity beans and POJOs. Composite Entity aggregates a set of related Business Objects into coarse-grained entity bean implementations

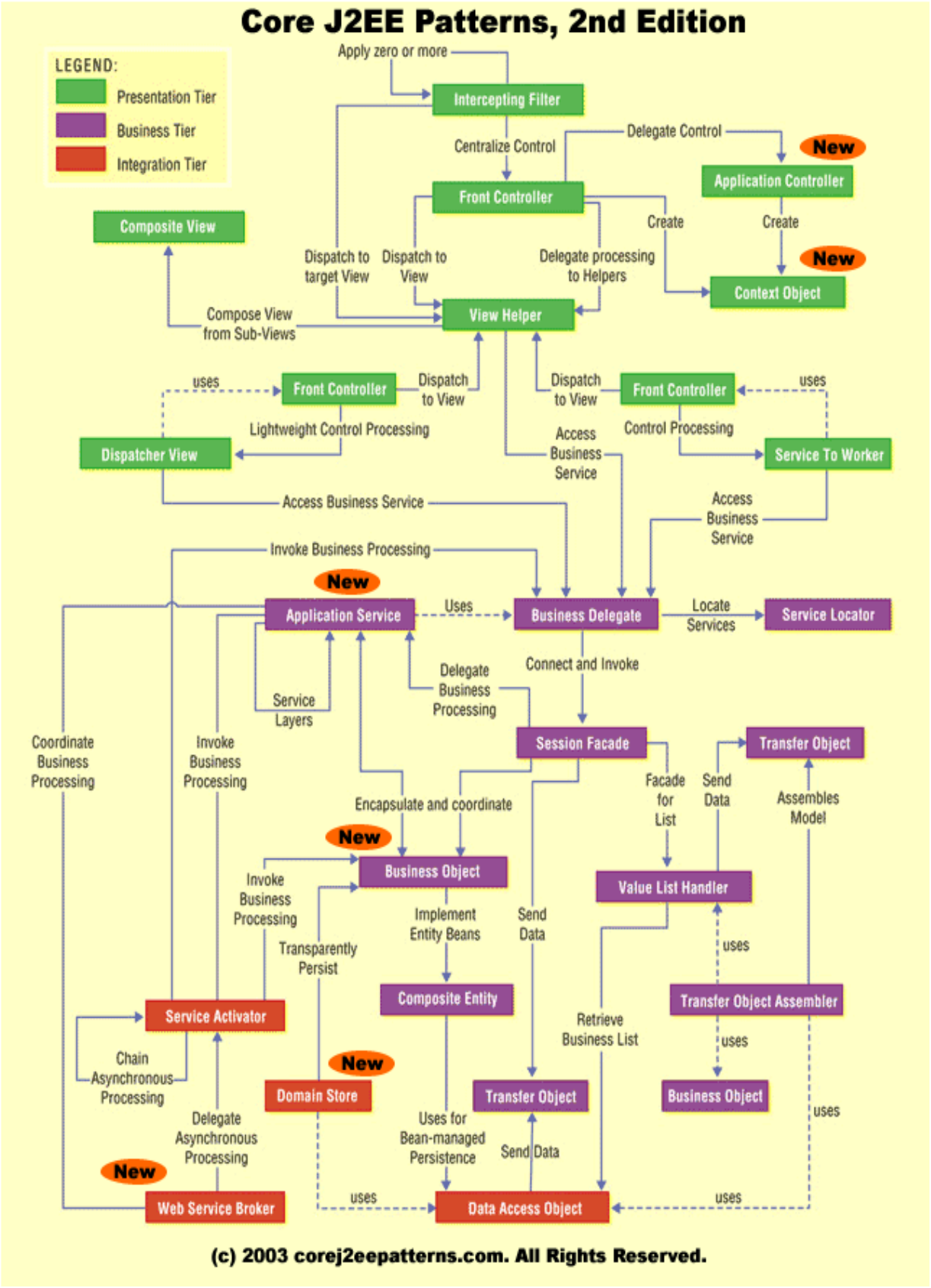
Transfer object: *transfer multiple data elements over a tier.*

Value List handler: Use a ***Value List Handler*** to search, cache the results, and allow the client to traverse and select items from the results.

Integration Tier:

Data Access Object*: You want to encapsulate data access and manipulation in a separate layer.* Use a *Data Access Object* to abstract and encapsulate all access to the persistent store. The *Data Access Object* manages the connection with the data source to obtain and store data.

Web Service Broker: Use a ***Web Service Broker*** to expose and broker one or more services using XML and web protocols.


Core J2EE Patterns, 2nd Edition
LEGEND:
Presentation Tier
Business Tier
Integration Tier
Apply zero or more
Intercepting Filter
Centralize Control
Delegate Control
New
Application Controller
Front Controller
Composite View
Create
Create
New
Dispatch to target View
Dispatch to View
Delegate processing to Helpers
Context Object
View Helper
Compose View from Sub-Views
uses
Front Controller
Dispatch to View
Dispatch to View
Front Controller
uses
Lightweight Control Processing
Control Processing
Dispatcher View
Access Business Service
Service To Worker
Access Business Service
Access Business Service
Invoke Business Processing
New
Application Service
Uses
Business Delegate
Locate Services
Service Locator
Service Layers
Delegate Business Processing
Connect and Invoke
Coordinate Business Processing
Invoke Business Processing
Session Facade
Transfer Object
Facade for List
Send Data
Assembles Model
Encapsulate and coordinate
New
Business Object
Value List Handler
Invoke Business Processing
Implement Entity Beans
Send Data
Transparently Persist
uses
Composite Entity
Transfer Object Assembler
Service Activator
Retrieve Business List
Chain Asynchronous Processing
New
Domain Store
uses
Transfer Object
Business Object
uses
Delegate Asynchronous Processing
Uses for Bean-managed Persistence
Send Data
New
Web Service Broker
uses
Data Access Object
uses
(c) 2003 corej2eepatterns.com. All Rights Reserved.

Intercepting Filter:

Problem

You want to intercept and manipulate a request and a response before and after the request is processed.

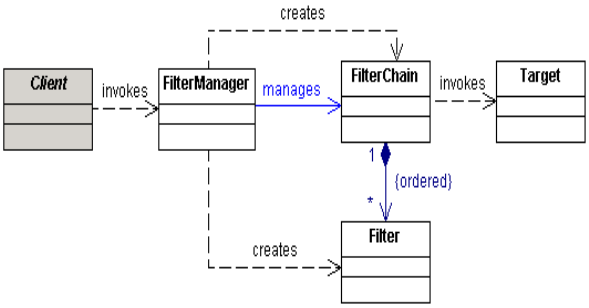
Forces

- You want centralized, common processing across requests, such as checking the data-encoding scheme of each request, logging information about each request, or compressing an outgoing response.
- You want pre and postprocessing components loosely coupled with core request-handling services to facilitate unobtrusive addition and removal.
- You want pre and postprocessing components independent of each other and self contained to facilitate reuse.

Solution

Use an Intercepting Filter as a pluggable filter to pre and postprocess requests and responses. A filter manager combines loosely coupled filters in a chain, delegating control to the appropriate filter. In this way, you can add, remove, and combine these filters in various ways without changing existing code.

Class Diagram



Front Controller:

Problem

You want a centralized access point for presentation-tier request handling.

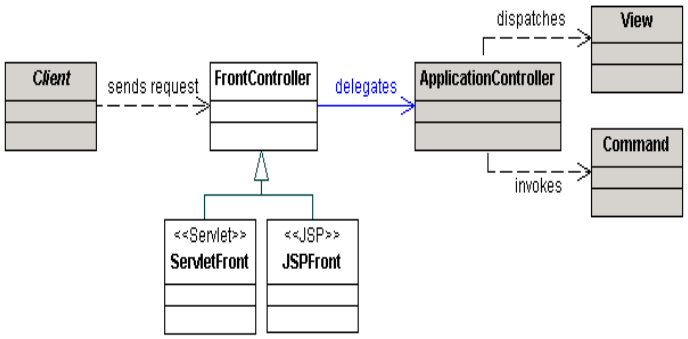
Forces

- You want to avoid duplicate control logic.
- You want to apply common logic to multiple requests.
- You want to separate system processing logic from the view.
- You want to centralize controlled access points into your system.

Solution

Use a Front Controller as the initial point of contact for handling all related requests. The Front Controller centralizes control logic that might otherwise be duplicated, and manages the key request handling activities.

Class Diagram



Application Controller:

Problem

You want to centralize and modularize action and view management.

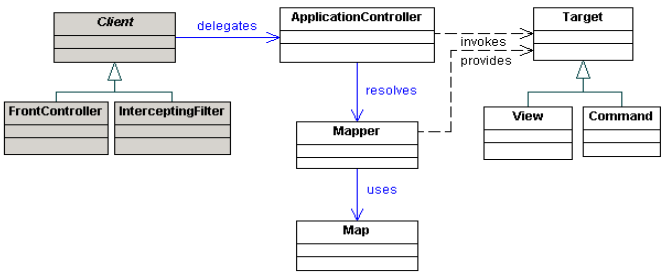
Forces

- You want to reuse action and view-management code.
- You want to improve request-handling extensibility, such as adding use case functionality to an application incrementally.
- You want to improve code modularity and maintainability, making it easier to extend the application and easier to test discrete parts of your request-handling code independent of a web container.

Solution

Use an Application Controller to centralize retrieval and invocation of request-processing components, such as commands and views.

Class Diagram



Service To Worker:

Problem

You want to perform core request handling and invoke business logic before control is passed to the view.

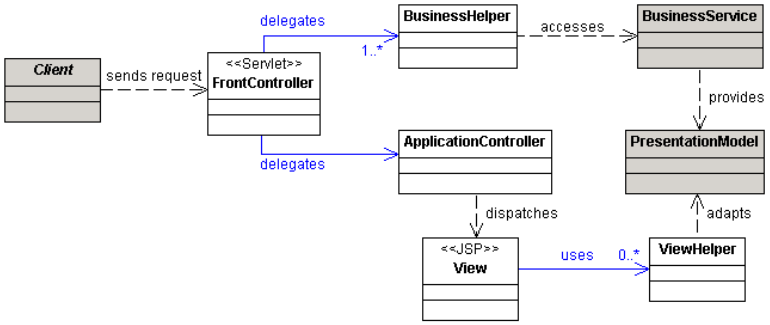
Forces

- You want specific business logic executed to service a request in order to retrieve content that will be used to generate a dynamic response.
- You have view selections which may depend on responses from business service invocations.
- You may have to use a framework or library in the application.

Solution

Use Service to Worker to centralize control and request handling to retrieve a presentation model before turning control over to the view. The view generates a dynamic response based on the presentation model.

Class Diagram



Business Delegate:

Problem

You want to hide clients from the complexity of remote communication with business service components.

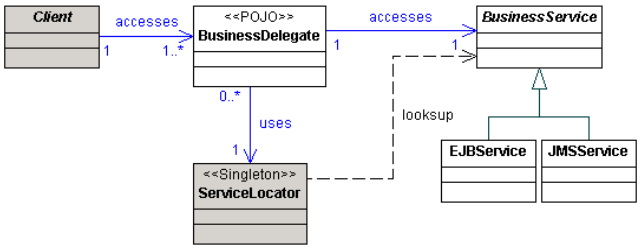
Forces

- You want to access the business-tier components from your presentation-tier components and clients, such as devices, web services, and rich clients.
- You want to minimize coupling between clients and the business services, thus hiding the underlying implementation details of the service, such as lookup and access.
- You want to avoid unnecessary invocation of remote services.
- You want to translate network exceptions into application or user exceptions.
- You want to hide the details of service creation, reconfiguration, and invocation retries from the clients.

Solution

Use a Business Delegate to encapsulate access to a business service. The Business Delegate hides the implementation details of the business service, such as lookup and access mechanisms.

Class Diagram



Service Locator:

Problem

You want to transparently locate business components and services in a uniform manner.

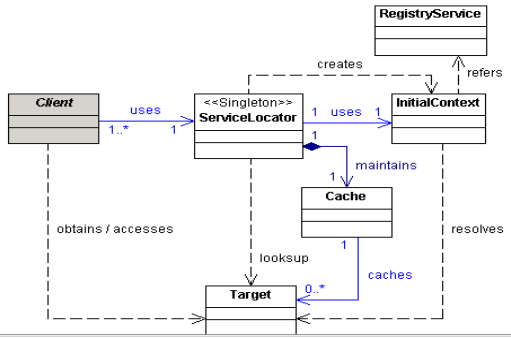
Forces

- You want to use the JNDI API to look up and use business components, such as enterprise beans and JMS components, and services such as data sources.
- You want to centralize and reuse the implementation of lookup mechanisms for J2EE application clients.
- You want to encapsulate vendor dependencies for registry implementations, and hide the dependency and complexity from the clients.
- You want to avoid performance overhead related to initial context creation and service lookups.
- You want to reestablish a connection to a previously accessed enterprise bean instance, using its Handle object.

Solution

Use a Service Locator to implement and encapsulate service and component lookup. A Service Locator hides the implementation details of the lookup mechanism and encapsulates related dependencies.

Class Diagram



Session Façade:

Problem

You want to expose business components and services to remote clients.

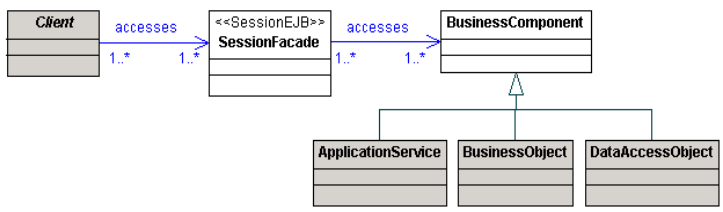
Forces

- You want to avoid giving clients direct access to business-tier components, to prevent tight coupling with the clients.
- You want to provide a remote access layer to your Business Objects (374) and other business-tier components.
- You want to aggregate and expose your Application Services (357) and other services to remote clients.
- You want to centralize and aggregate all business logic that needs to be exposed to remote clients.
- You want to hide the complex interactions and interdependencies between business components and services to improve manageability, centralize logic, increase flexibility, and improve ability to cope with changes.

Solution

Use a *Session Façade* to encapsulate business-tier components and expose a coarse-grained service to remote clients. Clients access a *Session Façade* instead of accessing business components directly.

Class Diagram



Data Access Object:

Problem

You want to encapsulate data access and manipulation in a separate layer.

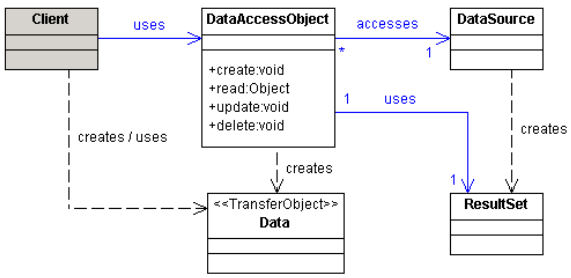
Forces

- You want to implement data access mechanisms to access and manipulate data in a persistent storage.
- You want to decouple the persistent storage implementation from the rest of your application.
- You want to provide a uniform data access API for a persistent mechanism to various types of data sources, such as RDBMS, LDAP, OODB, XML repositories, flat files, and so on.
- You want to organize data access logic and encapsulate proprietary features to facilitate maintainability and portability.

Solution

Use a *Data Access Object* to abstract and encapsulate all access to the persistent store. The *Data Access Object* manages the connection with the data source to obtain and store data.

Class Diagram



Business Object

See Core J2EE Patterns, 2nd Edition for full description of this pattern and its strategies.

Problem

You have a conceptual domain model with business logic and relationship.

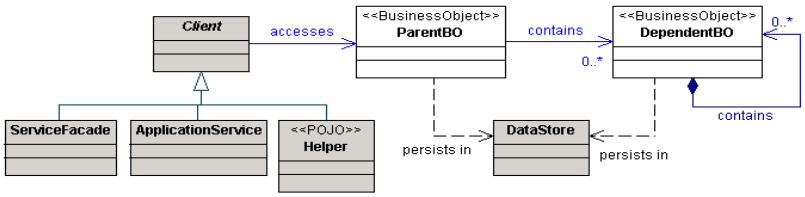
Forces

- You have a conceptual model containing structured, interrelated composite objects.
- You have a conceptual model with sophisticated business logic, validation and business rules.
- You want to separate the business state and related behavior from the rest of the application, improving cohesion and reusability.
- You want to centralize business logic and state in an application.
- You want to increase reusability of business logic and avoid duplication of code.

Solution

Use Business Objects to separate business data and logic using an object model.

Class Diagram



Transfer Object

See Core J2EE Patterns, 2nd Edition for full description of this pattern and its strategies.

Problem

You want to transfer multiple data elements over a tier.

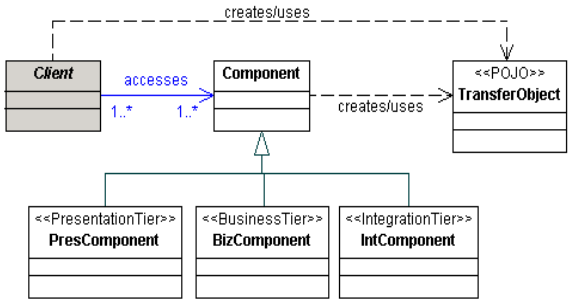
Forces

- You want clients to access components in other tiers to retrieve and update data.
- You want to reduce remote requests across the network.
- You want to avoid network performance degradation caused by chattier applications that have high network traffic.

Solution

Use a Transfer Object to carry multiple data elements across a tier.

Class Diagram



Web Service Broker:

Web Service Broker

See Core J2EE Patterns, 2nd Edition for full description of this pattern and its strategies.

Problem

You want to provide access to one or more services using XML and web protocols.

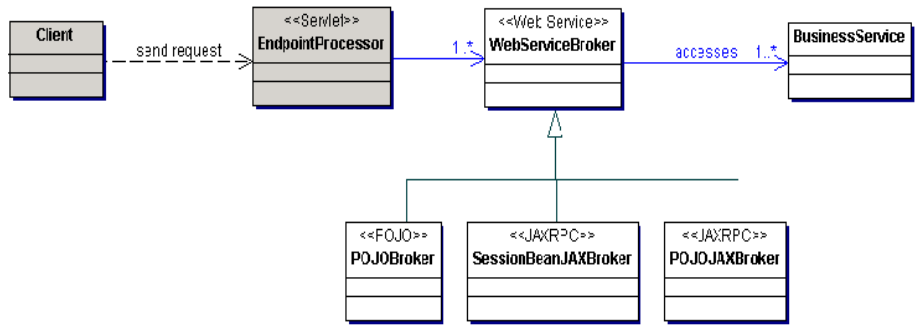
Forces

- You want to reuse and expose existing services to clients.
- You want to monitor and potentially limit the usage of exposed services, based on your business requirements and system resource usage.
- Your services must be exposed using open standards to enable integration of heterogeneous applications.
- You want to bridge the gap between business requirements and existing service capabilities

Solution

Use a *Web Service Broker* to expose and broker one or more services using XML and web protocols.

Class Diagram



15. Algorithms

1. You have a ladder n-steps in height. You can either take one step or two steps up the ladder at a time. How can you find out all the different combinations up the ladder? Then figure out an algorithm that will actually print out all the different ways up the ladder. ie: 1,1,2,1,2,2... etc...

2. Given the root node to a singly linked list, reverse the last 5 nodes in the list.

3. Given the root node to a singly linked list, write an algorithm to see if it loops back on itself somewhere in the middle.

4. Write an algorithm to calculate the square root of a number.

5. Given a const null terminated string containing a sentence, can you print out the words of the sentence in reverse without changing any data and without creating a temporary string buffer? Do not use any helper functions, write it all out on your own. (ie "The dog is fast" becomes "fast is dog The")

6. Implement a deque. Think about what properties a deque must have. Implement it as a base class that can be extended and as a template. In general, start thinking about what's underneath all the other STL data types and how they work.

7. Given a char pointer to large buffer of memory, write your own version of my_malloc and my_free without using any system calls. Make it as robust as possible. How would you minimize memory fragmentation?

8. Implement quicksort().

9. Given the function: "bool numExists(int array[], int array_len, int num)" where "array" is a sorted array of integers. Determine if "num" exists in the array. This is basically a binary search question. Do a recursive version and an iterative version. Writing binary search is easy but getting it the most efficient is hard. Can you do it with the minimum number of operations possible?

10. The standard library function of atoi() is not very robust. How would you design/implement a better version of it?

11. Implement a Singleton. It's the most over used and mis-used design pattern. Everyone knows the "textbook" implementation of the singleton but think of all the different ways you can implement it and what are their pros/cons? Make a thread-safe version.

Basic Java (to be used for more junior candidates)

1. Hashcode and equals
   1 pt
   Follow up Q: what will happen if hashcode always returns same value for all objects?

2. When will you use, LinkedList, ArrayList, HashMap
   2 pts

Advanced practical Java (to be used for Sr. candidates)

3. Multithreading: how would you implement a system that that work in multithreading environment? What classes Java offers to help with this?
   5 pts
   A: Callable, Runnable, Thread, synchronization package (locks, mutexes, etc)

4. How you implemented REST in your application? (spring or jersey or java-rx)
   5 pts
   A. Implement @RestController

   Follow up Q: What are the annotations for rest in spring?
   A : @requestMapping and @requestBody, and more

5. Have you followed TDD , BDD. 5 pts
   - Junit, testNg, Mockito, mockMvc, CoreMatchers.
   - How will you segregate the testing based on smoke, integration, unit?
   - Coverage tools.
     - Jacoco, surefire, etc

Theory

6. What is Lamba expression and how can it be used?
   4 pts
7. Inheritance or composite? what would he/she prefer? 2 pts
8. What is micro service? 2 pts
9. Patterns:
   - What is circuit breaker 3 pts
     Spring circuit breaker(hystrix): When this pattern is applied to a method in muService, Hystrix watches for failing call to that method, and it trips the circuit open and subsequent calls are redirected to the fallback method. This currently only works in a class marked with @Component or @Service.

Add @HystrixCommand with a parameter fallbackMethod to the Service class (e.g. @HystrixCommand(fallbackMethod = "return")

Implement the specified fallback method in the above annotation

Add @EnableCircuitBreaker to the Application class

- Decorator. Do you know any Java classes that follow this pattern? 2 pts
- Factory 2 pts

10. What are the recommended communications styles between muServices?
    - Synchronous communication based on HTTP/REST/JSON 1 pts
    - Asynchronous communication based on AMQP (Rabbit) 3 pts
11. Difference between SOAP and REST? 3 pts

A:
SOAP has its own protocol.
REST is over HTTP
REST is preferable in most cases
SOAP has transactions, etc.

12. Have used spring, what spring component you have used. 7 pts
    - Spring –netflix (hystrix, eureka, turbine, zuul)
    - Spring-cloud (spring profile and configurations)
    - Spring dependency injections and inversion of control.
    - Spring batch
    - data modelling and persistence using spring extra 3 pts

      A: Spring data, Spring-jpa , crudRepository and @entity, in memory database like H2.

13. How do you secure your rest URLs (CRUD) 5 pts

A:

HTTPS
Secure Token
Spring Security
Java Authorization and Authentication
OAUTH

14. Distributed caching and storage (Gemfire)
    3 pts
    Gfsh cli
    Regions
    Members
    functions
    Gemfire pulse

15. Build tools knowledge 2 pts
    Jenkins
    Concourse
    Maven , working with maven plugins

Practical

1> Write a java data structure to declare and initialize the above data?
1 pt

A B C D

E F G H

I J K L

A: char[4][4]

2> Fill in this method:

public String cat(String[][] data) {

// Return a comma separated string. eg A,B,C,D,E,F,G,H,I,J,K,L

}

3> Draw a simple block diagram that lets me use 3 different types of cache libraries by some means of external configuration.

4> Reverse the string (how many ways, he/she can solve this)

A: Loop, recursive, call api?

Optional Q. (use if the person does not know other topics)

1. What is AOP?
   3 pts.

Additional Micro Service Tech:

1) What does @ComponentScan annotation do?
   a) This tells Spring to look for classes with @Component, @Configuration, @Repository, @Service, and @Controller and wire them into app context as beans.
2) What does @Enable AutoConfiguration do?
   a) This turns on Boot's auto-configuration behavior for adding extra beans. This decision is made using the classpath and settings found inside application.properties file.
3) What Are The Issues With Sticky Session?
   a) There are few issues that you may face with this approach. The client browser may not support cookies, and your load balancer will not be able to identify if a request belongs to a session. This may cause strange behavior for the users who use no cookie based browsers. In case one of the machine fails or goes down, the user information (served by that machine) will be lost and there will be no way to recover user session.
4) How would you implement simple workflows using asynchronous communications?
   a) Mainly correlation IDs and distributed transactions should be used for tying up various microservice calls together.

1. Java Question "With the list of ranges given, find the overlapped ranges"

2. SQL Question "Given the three tables (student, subject, and score with no common column between student and subject) list all students for each subject"

1. Explain how to do the authentication in a J2EE application, including both UI section and and server side.

1. Give an array [1,2,3,4,5,6,7,8,9], output result should be production without current member, like [2*3*4*5*6*7*8*9, 1*3*4*5*6*7*8*9, ..., 1*2*3*4*5*6*7*8].

No division allowed and must consider the large production number bigger than integer limitation. Also need to consider how to handle big size of input, like one million elements inside the input array

3. Use the regular expression to classify the string with 256 ASCII codes. like given a string This is a Bloomberg's Java8 question score 345 &with \t $100 worth.\n".

In the output you should put all the words together, and all number together, then all special character together.

4. What is autoboxing in Java and write some codes to explain it.

5. Given two lists (A->B->C->D and E->Y->H->C->D), then output the common tail "CD", the time complexity shouldn't be larger than O(n) and space complexity should be O(1)

The question and answer can be found here

http://www.geeksforgeeks.org/design-a-data-structure-that-supports-insert-delete-search-and-getrandom-in-constant-time/

Maths Problems:

5) Generate power set of given set

int A[] = {1,2,3,4,5}; int N = 5; int Total = 1 << N; for (int i = 0; i < Total; i++) { for (int j = 0; j < N; j++) { if ((i >> j) & 1) cout << A[j]; } cout << endl; }

2. Find if the given expression contains redundant parantheses. ex : if expr = a+(b*c) , print false, if expr = a+((b*c)), print true.

3. Given a set of intervals like 5-10, 5-10, 8-12, 9-15. Find the ith smallest number in these intervals.

for eg:-Suppose we have intervals like 5-10, 8-12. Then total numbers in these two intervals would be: {5,6,7,8,8,9,9,10,10,11,12} So, 1st smallest number: 5 4th smallest number: 8 5th smallest number: 8

4. Add two numbers without using + operator.

5. Negate a number without using just +operator. Eg: negate(5) = -5 ; negate(-5)=5.

6. Find number smallest number K , if exists, such that product of its digits is N. Eg: N = 6, smallest number is k=16(1*6=6) and not k=23(2*3=6). int getnumber(int n) { if(n==0|| n==1) int i=9; int num=0; int pow=1; while(i>;0) { while(n%i==0 && i>;1) { num=i*pow + num; pow=pow*10; n=n/i; } i--; } if(n!=1) return -1; else return num;

7. How to find the smallest number with just 0 and 7 which is divided by a given number?

8. Count the trailing zeros in factorial of a number N? (5! Has number of 0's = 1)

9. Check if the bit representation of a number is palindrome or not? Eg . 9 => 1001 > is palindrome. Function

boolean isPalindrome(int anumber) { uint numberOfBits = sizeof(uint)*8; for(uint i=1; i<= numberOfBits/2; ++i) { if(((aNumber & (1<<(numberOfBits-i)))? 1:0) == ((aNumber & (1<<(i-1)))? 1:0)) { continue; } else { return false; }
} return true; }
10. Generate random number generator (mod 7) using (mod 5). Understand the trick then you can generate any random number generator using a given generator. int rand7() { int vals[5][5] = { { 1, 2, 3, 4, 5 }, { 6, 7, 1, 2, 3 }, { 4, 5, 6, 7, 1 }, { 2, 3, 4, 5, 6 },

{ 7, 0, 0, 0, 0 } }; int result = 0; while (result == 0) { int i = rand5(); int j = rand5(); result = vals[i-1][j-1]; }

11. Get the least number after deleting *k* digits from the input number. For example, if the input number is 24635, the least number is 23 after deleting 3 digits.

http://codercareer.blogspot.in/2013/11/no-48-least-number-after-deleting-digits.html

check first decreasing number, and remove that from the number. Keep doing it until kth element.

12. Find how many squares are present on a chessboard.
http://www.programmerinterview.com/index.php/puzzles/find-number-of-squares-chessboard/

Number of squares for an NxN chessboard?

What if we want to find the number of squares for a chessboard of size NxN, where N is any number. Do you notice a pattern in the results in the table above? Well, if you take a look at the results that we showed above for the normal-sized 8×8 chessboard, you can see that the number of squares is equal to the sum of squares from 1^2 to 8^2, where 8 is equal to N. So, we can say in more general terms that the sum of squares for a chessboard of size NxN is equal to $n^2 + (n-1)^2 + (n-2)^2 + \ldots. (1)^2$.

Linked Lists:

13. Reverse a doubly linked list. (Just don't ignore the problem because it sounds simple). Code it and be careful at edge cases.

14. Merge two sorted linked lists (**in-place**) , don't form third linked list.

15. Quick-Sort on Doubly linked list. Also try it on Singly linked list.

16. Reverse singly linked list without using three pointers. You can use only two pointers. eg: p-> points to head ; q-> points to head->next then, while(q) { q = p ^ q ^ (q->next) ^ (q->next=p) ^ (p=q) ; } At the end, p points to newHead.

17. Loop in linked list. Start of loop, if loop exists.

18. Find middle of singly linked list. Break Circular linked list into two different circular linked lists.

19. Number is represented in linked list such that each digit corresponds to a node in linked list. Add 1 to it. For eg: 1999 is represented as : **(1->9 -> 9 -> 9) + 1** => **(2->0->0->0)** Extension : Don't reverse linked list.

20. Reverse nodes in linked list problems. Like : k-alternate node reverse **OR** reverse k nodes, then next k nodes and so on..

Arrays Problems:

21. You have an array which has a set of positive and negative numbers, print all the subset sum which is equal to 0.

22. Divide a given array into two subarray (not necessary to be continuous) such that difference between sum of both array is minimum.

23. Find number of inversions in an array. Inversion means a pair where a[i] > a[j] and i <j in O(nlogn).

24. Triplet sum problem. Find all pairs of numbers such that a[i]+a[j]+a[k]=N.

25. A stream of integers is coming. Find median of numbers received till now. Hint: Two heaps required, one minheap and other one maxheap.

26. Find equilibrium index in array. Its the index i where sum till i from 0th index = sum from ith index to last index.

27. Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted. Eg: (0 4 12 3 15 16 18 17 19).. Seems like mean length unsorted subarray is from 14 to 17, but its not. Because if you sort that partition you get, (0 4 3 12 15 16 17 18 19) which is not sorted. Think of all cases and find minimum length such that , if we sort that contiguous subset, we get sorted array. Hint : In above example, once you find shaded part, just get min and max in that shaded part and check if anything in left of that partition is not greater than min and anything on right side of that part is not less than max, so that once you sort that partition, you will get sorted array.

28. Numbers in array are repeated thrice except one number which is present just once. Find the one? Hint : A number appears three times, each bit (either 0 or 1) also appears three times. If every bit of numbers appearing three times is added, the sum of every bit should be multiple of 3.

29. Write heap functions to build heap, add new element in heap.

30. Find peak element in array. A[i] is peak element, a[i-1] < a[i] > a[i+1].. For corner elements a[0] is peak, if a[0] > a[1] **or** a[length-1] > a[length-2] then a[length-1] is peak.

There can be many peaks in array. Find anyone. No repetition of elements in array. Try it on O(logn).

31. There are 0's and 1's in array. Bring all 0's on odd positions and 1's at even positions. If 1's or 0's are more, leave the remaining as it is after arranging at even/odd positions. Hint: Keep two pointers. Odd and even position pointers. Both will move +2 steps at a time. Keep moving odd pointer ahead till u get a place where 0 is not present. Do the same for

even pointer. Then exchange values of these two pointers. Continue till anyone of them reaches end of array.

32. Given an array of integers, sort the array according to frequency of elements. For example, if the input array is {2, 3, 2, 4, 5, 12, 2, 3, 3, 3, 12}, then modify the array to {3, 3, 3, 3, 2, 2, 2, 12, 12, 4, 5}. Hint: Insert elements into maxheap. Condition for maxHeap is frequency of elements. So root will contain element that has maximum no of repetitions.

33. The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days. For example, if the given array is {100, 180, 260, 310, 40, 535, 695}, the maximum profit can earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

34. You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum. For example, if the given array is {-2, -5, 6, -2, -3, 1, 5, -6}, then the maximum subarray sum is 7. Dont use Kadane.. Try with Divide and conquer approach. Refer geeksForGeeks.org

35. Find maximum difference in indexes. Say, (j-i) is answer if i,j are both indexes such that i<j and A[i]>A[j]; .. This problem is different from the one below..

36. Maximum difference in two elements in arrays such that the larger elements appear on right-side.

37. Suppose array values represent height of histograms. Find largest area rectangle formed by some contiguous histograms.

38. Imp problem: Find number which is repeated more than n/2 times in array where n is length of array. Extension: Now find element which is repeated more than n/4 times, n/8 times and so on. Hint: http://www.cs.utexas.edu/~moore/best-ideas/mjrty/ This will help you find n/2 times repeated element. To find(n/4 or more)times repeated element, repeat same algorithm but this time ignore the one which is already found as repeated n/2 times.

39. Find number of elements on right side, which are greater in value then current element. Find this for all elements of array. (Hint: Use BST tree).

Stacks and Queues Problems:

40. Simplest and famous one: Design stack for top(),push(),pop() and getMin() operations in O(1) time complexity. Hint: Two stacks. Beware of edges cases like when

last element is popped out. Extension: How would you design for getMax() operation also.

41. Find the next smallest element on right side for every element a[i] in array a[].

42. Design queue for getMiddle() in O(1) time complexity. Hint : Think in terms of how to implement queue. How about LinkedList. Can we maintain one additional pointer to middle element always and update it whenever new element is added at rear.

43. Design Queue for getMin() in O(1) time complexity. Hint: Use two queue.

44. Inorder/postorder **iterative** using explicit stack. This problem is important as this technique is required in many problems.

45. Find maximum number in sliding window of k . eg : if array is { 1,5,7,2,1,3,4} and k=3, then first window is {1,5,7} so maximum is 7, print 7, then next window is {5,7,2} , maximum is 7, print 7. So final output is : { 7,7,7,3,4 } This is very famous problem usually asked in big companies during initial screening rounds.

46. Celebrity problem : There are n number of people in party and only one celebrity. If celebrity is the person, who knows no one but everyone knows him, find who is the celebrity? Given: a function is provided which tells whether ‘a’ knows ‘b’ or not? doesAknowB(A,B). Hint: Push all the elements into stack. Remove top two elements(say A and B), and keep A if B knows A and A does not know B. Remove both A,B is both know each other.

Trees Problems:

47. Find any two elements in BST such that sum of those two nodes = K. Hint: Try O(logN). Do 2 iterative inorders, one for left-root-right and other following right-root-left. Add those two values and check cases like <k, >k,==k.

```
inroderRevInorderBoth(root) { done1, done2 = false; while(1) { while(done1==false) {
while(curr1) { s1.push(curr1); curr1=curr1->left; } if(s1.empty()) done1 = true; else { val1
= s1.pop(); curr1 = val1->right; done1= true; } }
while(done2==false) { while(curr2) { s2.push(curr2); curr2=curr2->right; } if(s2.empty())
done2 = true; else { val2 = s2.pop(); curr2 = val2->left; done2= true; } } if(val1->data +
val2->data==target) return;/*print answer val1, val2*/ if(val1->data + val2->data < target)
done1=false; else done2=false; } }
```

48. Convert a binary tree to BST. Extra space allowed. (Can you maintain same shape of the original tree and still get BST. Answer is no..(http://stackoverflow.com/questions/3531606/convert-binary-tree-bst-maintaining-original-tree-shape)

49. Given a BST, delete a node in BST. (Refer to example below for hint. Node with value 6 is going to be deleted from its original position).

50. Given a BST, and a node in BST, make that node, as new root of the tree. But still maintain the tree as BST after making this node as root. For eg: In tree below, say given node is 6, then change the tree to make 6 as root of tree.. Hint: Take 6 as root, as its on left of original root(8) so, make 8 as right child of 6 and 3 as left child node. Now, there are two cases: - If 6 is leaf, then no worries at all - if 6 is internal node, remove 6 and replace it with (either max in left substree or min in

right subtree)

51. Find the inorder successor of BST in O(logN). By using inorder, you can find it in O(n) but then you are not using its BST properties. public class inorderSuccessor {

public static TreeNode iterativeSolution(TreeNode root, TreeNode target) {
if (root == null || target == null) return null; boolean targetFound = false;
TreeNode currNode = root, candidate=null; while (currNode != null) { if (currNode.val == target.val) { targetFound = true; currNode = currNode.right; } else if (currNode.val > target.val) { /* only update candidate here as we r looking for successor.. it will always be greater.*/ candidate = currNode; currNode = currNode.left; } else {// currNode.val < target.val currNode = currNode.right; } } return targetFound ? candidate : null; // important check to return candidate. }
}

52. Determine if a tree is a valid BST with no duplicated values. (This means that if the binary tree has a duplicated number it should return "invalid" even if it's an actual BST)

Hint: Check if left subtree or right subtree is invalid or not. Also keep track of prev node visited in inorder traversal and check if value is repeated by comparing previous visited node with current node.

53. Implement a function which returns list of all nodes in a binary tree having a given number of leaves, say k . Also mention complexity. Eg: k=4, ans is {4,5} .. k=2 ans is {8,9,10,11,12,7} Hint: Think bottom-up, so at every node, getall nodes from left and right subtree and see if its k, then print the node.

54. Replace each node with the sum of all greater nodes in a given BST? Hint-1 : If converted to Doubly linked list, then you can take prefix sum, starting from tail to head. But this will modify the tree. Hint-2: Try reverse inorder(right, root,left). Take sum of all nodes in right first, add current node's value to it and pass to left subtree. Total sum returned by left subtree can be passed to root. Recursion is the key.

55. Do **iterative** inorder, preorder **without** using stack. No extra space allowed. Hint : Morris traversal.

56. Find least common ancestor in tree for given nodes 'a' and 'b'. But time complexity should be O(n) and note its binary tree and not BST.

57. Print all paths in trees (not just root to nodes). Hint: O(n^2) to get all pairs of nodes and to print path between them, you can find LCA(a,b) between nodes 'a' and 'b' to get path between them, which will take O(n) so, total complexity O(n^3). Can you get better?

58. Find median in BST. There are can be different approaches. Extension to question: Suppose 'K' number of queries are coming in such a way that they ask for i-th smallest element in BST. Hint: For extension, you can preprocess tree such that, it stores extra information about number of children in left/right subtrees so that, we can traverse tree based on value of 'i'. Eg: if a node has 5 nodes in left subtree and 3 on rightside, and i=7, then at node you know, you have to go in right substree. So, for 'K' queries, O(logn) will be required to get i-th element.

59. Two nodes in BST are incorrectly placed. So, swap the two nodes such that BST is formed correctly.

60. Form BST from given preorder traversal output. eg: Preorder given : 8,3,1,6,4,7,10,14,13 ... Think recursion, 0th element is root, then find elements that qualify for left subtree and right subtree and go recursive to get actual trees. Output:

61. Find distance between two keys in Binary tree. (not BST). Hint: LeastCommonAncestor

62. Given a binary tree, a target node in the binary tree, and an integer value k, print all the nodes that are at distance k from the given target node. No parent pointers are available. Hint: Refer geeksForGeeks.org

Given a binary tree where each node has an additional pointer filed *'sibling'* (in addition to the *'left'* & *'right'* fields), connect all the siblings. Note, the tree may not be a complete binary tree. Hint-1: Do level order traversal and previous node is connected to current node in level order traversal. Hint-2: For every node, (if right child exists),connect its left-child to right-child, if not, connect left child to (find node by traversing, sibling pointer of current-node, and finding appropriate next sibling for left child). For right child, find sibling by traversing, sibling of current node and finding required node. eg. In example tree of question 60, at node 8, connect 3's sibling pointer to 10. At 3, connect 1's sibling to 6, and for 6, traverse sibling of 3 which is 10, and get 10's right child(14) as left is not present. Attach sibling of 6 as 14. And so on.

Strings Problems:

63. Find longest repeating substring in string. Eg; abcdsabcaewcabcad ..Here ans is ‘abc’. Hint: http://en.wikipedia.org/wiki/Longest_repeated_substring_problem

64. Find longest palindrome substring in O(n2) without extra space. Hint: No dynamic programming required. - For odd length palindromes, From each element, moves both directions left+right and check if palindrome is formed and record length if maximum. - Also, for even length palindromes, start from between every two elements and go both directions till palindrome property is formed.

65. Famous problem for interviews: Given a string ‘str’and some other string ‘str2’, find smallest substring in str which will contain all characters of ‘str2’ even if they are in any order. eg: str=”bzdsdaddwcfaobwww” str2=”abc”. Highlighted part is answer.

66. KMP search : Find if given ‘pattern ‘ is present in given string “str”, if present return the index of start of pattern in str. For eg: str=”asdqeaicicaiciciaaw”, pattern=’icici’, then ans is 11. One of the Important questions, understand KMP search properly as its frequently asked.

67. Design String class of our own. Be careful to design understanding all properties of Java String class like immutable. (Hint: See how to make class immutable and check Java String class properties)

68. There is a dictionary of billion words and there is one method provided "String getWord(int index);". We can give it index and it will return the String on that index. Hint: Use Binary search if end index is known. Or else try Fibonacci search to reach a point where string at that index is greater than given word.

69. String compression**(in-place)**. O(n) required. For given string compress using following logic: case1 : input string : aaawwwbbbcc compressed output: a3w3b3c2 case 2: input string : acccbww compressed output: a1c3b1w2 Assume that every input string has infinite amount of memory. Hint: Count number of places where no repetition is present but don’t insert 1 now, just count. - For characters where repetition is present, replace duplicate characters by digits. At last, let compressed string be called str(as its in-place). - To insert 1’s at places where no repetition is there, start copying each char from end of str to str.length+count position. This is new length of string. - While copying for characters for which number is not present in-front of them, it means, they appeared once, so insert 1 and then the character. - Eg: str=acccb After transformation, str=ac3b and new_length=4 and count=2 So, keep pointer at end of str, at ‘b’, and copy it to new_length position in str. But we observe that for ‘b’ , no number appeared before it so, its not repeated, so we insert 1 at that new position. Then, we get number, so for numbers, just copy them to new postion as it is. (new position -1) is to be done every time. Now, for c, we saw a number just before ‘c’ so, it means, c was repeated so copy ‘c’ as it is.

Now, comes ‘a’ and we see no number associated with it, so insert ‘1’ and then ‘a’. So final answer : a1c3b1 **Dynamic Programming(Only frequently asked):**

70. Find length of longest series questions : - Longest common subsequence in two strings - Longest common substrings in two strings - Longest increasing subsequence in O(nlogn) Refer : http://www.algorithmist.com/index.php/Longest_Increasing_Subsequence.cpp

71. You are given an array A with elements 0 to n-1, numbers can be repeated in the array. Create n sets where S[i]={a[i],a[a[i]],a[a[a[i]]]…}. Set has all elements unique. Find the size of the largest set. Input: First line contains n, size of the array. n<1000 Next lines contains n numbers, each element of the array Output Prints one number: Size of the largest set. Sample Test Case: Input: {3,1,2,0} Output: 2 Explanation: Four possible sets are {3,0},{1},{2}{0,3} Refer: http://www.careercup.com/question?id=4724898538717184

72. Find optimal binary search tree that can be formed from given sorted array of values. Better explanation for dynamic programming given at : http://orion.lcg.ufrj.br/Dr.Dobbs/books/book2/algo032a.htm

73. Check if there exists, **non-contiguous** subset in array such that sum of subset is exactly total_sum/2. Where total_sum = sum of all elements in array.

74. Mostly asked in written test of Amazon: Given coins of certain denominations and their infinite supply, find minimum number of coins required to get required ‘K’ amount of money. Variation: There can be only one coin of each denomination, now tell if K sum amount can be formed or not.

75. Word break problem. Eg: Iamlive, this string can be broken,i am live, which are three valid words in dictionary. Figure out , if a string can be broken down into valid words or not? Function written should just return true/false.

76. A frog can jump 1 or 2 steps at a time. In how many different ways can he reach distance K. Hint: Is Fibonacci related to this problem?

77. There are N houses in a row. Each house can be painted in either Red, Green or Blue color. The cost of colouring each house in each of the colours is different. Find the colour of each house such that no two adjacent houses have the same colour and the total cost of colouring all the houses is minimum. Hint:

- If you are calculating cost of colouring till ith house and you know the minimum cost of colouring the (i-1)th house and color of (i-1)th house, then you can color ith house to keep cost minimum. There will always be two colors by which you can paint ith house except 0th house(0th you can color in 3 ways). So at every house, there will be two values and you need to consider both of them when calculating the cost of next one. At last house, we can paint it using any two colours again, choose the one which gives minimum cost. - Point is only at last house, you can choose only one colour, the colour that gives minimum cost. At all other houses, you have to maintain two values for two colors, you are giving them. Because, if at every house, you keep choosing only the color that gives minimum that is greedy approach and may not always give correct answer.

78. Edit distance problem. (http://en.wikipedia.org/wiki/Edit_distance) Solution at http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/

79. Matrix chain multiplication problem. Very famous problem.

80. You are given a boolean expression consisting of a string of the symbols 'true', 'false', 'and', 'or', and 'xor'. Count the number of ways to parenthesize the expression such that it will evaluate to true. For example, there are 2 ways to parenthesize 'true and false xor true' such that it evaluates to true. Hint: http://people.cs.clemson.edu/~bcdean/dp_practice/dp_9.swf

Miscellaneous problems:

81. Find median,if K sorted arrays of length N are merged. Hint: MinHeap. Take min element from each array and add to minHeap. Remove min element from root of heap and insert new element into heap from the array to which root belonged. Keep on doing this, till you get to median. (Median => if K*N is odd, then median is (K*N)/2 + 1 th element, and if (K*N) is even, then ((K*N)/2 + ((K*N)/2)+1))/2 th element.) Check this condition once.

82. Find top 10 frequent words in a file with millions of words. Hint: Trie and minheap. Size of heap will be 10. Insert word into trie, the last node containing last character of word will have count which will incremented by 1 , everytime word is inserted. If count > (the minimum value in minheap(i.e. root's count) then remove root and insert this word into heap. MinHeap is arranged based on number of occurances a word had till now.

Food for thought: Can we used doubly linked list instead of minHeap to reduce time complexity. Extension : Instead of top 10 words, it can top 10%. In that case, the minheap size will change such that everytime it will store 10%(high-frequency words) of the words read till now.

83. Implement Least Recently used cache. Write code for it too.

84. Find maximum sum rectangle in matrix of size m*n with 1's and 0's. Hint: You will need Kadane algorithm to reduce time complexity from O(n^4) to O(n^3). Refer geeksforgeeks.org for solution.

85. Given a number find the next smallest palindrome. eg: 123 -> 131, 991 -> 1001, 121 -> 121 itself. Hint: - There are two cases, whether the number of digits in the number is odd or even. We'll start with analyzing the odd case. - Let's say the number is ABCDE, the smallest possible palindrome we can obtain from this number is ABCBA, which is formed by mirroring the number around its center from left to right (copying the left half onto the right in reverse order). - This number is always a palindrome, but it may not be greater than the original number. If it is then we simply return the number, if not we increase the number. But which digit to increment? - We don't want to increment the digits on the left half because they increase the number substantially, and we're looking for the smallest palindrome larger than the number. - The digits on the right half increase the value of the number much less, but then to keep the number palindrome we'll have to increment the digits on the left half as well, which will again result in a large increase. So, we increment the digit just in the middle by 1, which corresponds to adding 100 in this case. - This way the number stays a palindrome, and the resulting number is always larger than the original number.

Courtesy: Ardent (http://www.ardendertat.com/2011/12/01/programming-interview-questions-19-find-next-palindrome-number/)

86. Backtracking problems: N-Queens problem. Place N queens in such a way that, they can't kill each other.

87. One knight is placed at some position Y in chessboard, find in how many minimum moves it will move to some given position X in chessboard. By position X or Y means, it will have row and column number.

88. Print all permutations of number in sorted order formed by last digits. Eg: number=123, then numbers formed are: 123,132,213,231,312,321. Hint: To get next number which is greater than given number. 1. Start from rightmost digit and go left till you find ith element such that, digit_at_i < digit_at_i+1. 2. Now go find on right-side of ith element, find the element(say at position 'j') that is greater than ith element but smallest among all those elements which are greater and on right side of ith element. 3. Swap the digits at position 'i' and 'j'. 4. Sort digits from i+1th position to end.

89. Given a text file and a word, find the positions that the word occurs in the file. You'll be asked to find the positions of many words in the same file. Hint: Do some precomputation which will help to reduce complexity of getting position of word in text. So kind-of inverted index (word -> position mapping). Choose datastructure. It can be hashtable or trie(at whose end node will have List<Integer> indexes where the word occurred.)

90. Given a matrix of integers and coordinates of a rectangular region within the matrix, find the sum of numbers falling inside the rectangle. Our program will be called multiple times with different rectangular regions from the same matrix. Hint: Refer (http://www.ardendertat.com/2011/09/20/programming-interview-questions-2-matrix-region-sum/)

91. Implement Trie operations -> insertion, deletion and lookup operations.

92. Stream of 0's and 1's is coming, find whether the number formed together by this binary bits is multiple of 3 or not. The stream will keep sending 1 bit at a time. At every bit, tell the new number formed is multiple of 3 or not. 1) Get count of all set bits at odd positions (For 23 it's 3). 2) Get count of all set bits at even positions (For 23 it's 1). 3) If difference of above two counts is a multiple of 3 then number is also a multiple of 3.

93. From interview standpoint, always know and have understanding about problems like Reader-writers problem and Producer-consumer problem. Also know their solutions using semaphores or monitors.

94. Train station timings are given. That is, at a station arrival and departure times of trains is given. Find minimum number of platforms to be constructed at that train station so that trains at similar interval range can be accommodated. Eg: Train 1 Arrival:9:30am Depart: 10:30am Train 2 Arrival:9:45am Depart: 10:10am Train 3 Arrival:10:35am Depart: 11:30am Train 4 Arrival:11:55am Depart: 12:10pm Train 5 Arrival:11:50am Depart: 12:15pm Ans: 2 (Train 1,2 and Trains 4,5 overlap, so two platforms are required) Hint: The required platforms is the number of trains whose timings overlap. If none overlaps, then we can do with one platform. - Sort all the timings including arrival and departure timings by keeping them in one single array. And maintain that a particular timing is for arrival and departure. So, if n trains are present, you sort an array with 2n timings(arrival and departure time for each train). - Once you sort these 2n values, start from 0th index and increment count as you encounter timing value(marked as ‘arrival’) and decrement count for timing (marked as ‘departure’). - Also, maintain the maximum value obtained by count somewhere because at last this max value achieved by count is the answer.

Design Questions

95. Design an elevator system. Hint: Think in terms of algorithm to be used for elevator movement. Different usecases. Focus on classes and interaction between them.

96. Design spell-checker for MS-word. Hint: Think about data-structure like trie or ternary search tree.

97. Design Chess game. Define classes and interactions.

98. Design web-application that will accept pincode as input and will return list of departmental stores in that pincode. Interviewer will not say anything, after this line, he will accept you to come up with databse design and what data you will store in db to figure out which stores lie in a particular pincode. Hint: Also, don’t forget its web-application, so do focus on caching, multiple servers, load balancing for additional points □□

99. Implement a T9 dictionary. Its keypad of mobile. When you press certain numbers, you need to suggest words that are possibly formed from characters represented by those numbers.

100. Imagine you're designing a Web Service for a phone application that returns a list of suggested Words that may complete a given string the user types. For example, if the user writes "ap", a list of suggested words may contain ["apple", "application", ,aptitude",...]Assume English only words and no misspelling.

1. Explain how to do the authentication in a J2EE application,
> including both UI section and and server side.
> 2. give an array [1,2,3,4,5,6,7,8,9], output result should be
> production without current member, like [2*3*4*5*6*7*8*9,
> 1*3*4*5*6*7*8*9, ..., 1*2*3*4*5*6*7*8]. No division allowed and must
> consider the large production number bigger than integer limitation.
> also need to consider how to handle big size of input, like one
> million elements inside the input array
> 3. use the regular expression to classify the string with 256 ASCII
> codes. like given a string "This is a Bloomberg's Java8 question score
> 345 &with \t $100 worth.\n". In the output you should put all the
> words together, and all number together, then all special character
> together.
>
>
> Second round:
> 1. what is autoboxing in Java and write some codes to explain it.
> 2. given two lists (A->B->C->D and E->Y->H->C->D), then output the
> common tail "CD", the time complexity shouldn't be larger than O(n)
> and space complexity should be O(1)
1. Max continuous sum of sub array.
2. Longest distance between two nodes in a tree
3. number of paths in a matrix

Most of the questions were verbal on thread synchronization, particularly synchronized vs lock and various producer-consumer scenarios, member and variable modifiers such as final vs finally vs finalize

There were only 3 written questions:
1. given a class with a list member how to ensure it cant be modified - answer copy it in the getter method
2. Spin lock example, pluses and minuses
3. the question and answer can be found here http://www.geeksforgeeks.org/design-a-data-structure-that-supports-insert-delete-search-and-getrandom-in-constant-time/

Here are the questions I could remember.
*
*
Question 1:
int a=4, b=0;

try{

Begin forwarded message:

> *From:* Lakshmi Menon <lakshmi.menon87@gmail.com
> <mailto:lakshmi.menon87@gmail.com>>
> *Date:* November 6, 2016 at 8:34:06 PM EST
> *To:* Tom Morgan <tom@pencom.com <mailto:tom@pencom.com>>
> *Subject:* *Bloomberg Qs*
>
> Hi Tom,
>
> Following are the questions I was asked at the KYC interview:
>
> 1. Design a channel subscription site.
>
> 2. Reverse a string using built-in javascript functions (Hint: start
> by converting it into an array).
>
> 3. Write a javascript function that takes an array of numbers and a
> value 'n' as input and returns an array of arrays where the size of
> each inner array is at most 'n'.
>
> Example:
>
> (i) input_array = [2, 3, 4, 5, 6, 7, 8, 9] and n = 3
>
> output_array = [[2, 3, 4], [5, 6, 7], [8, 9]]
>
> (ii) input_array = [2, 3, 4, 5, 6, 7, 8, 9] and n = 2
>
> output_array = [[2, 3], [4, 5], [6, 7], [8, 9]]
>
> 4. What is the output of the following (Hint: variable hoisting):
>
> var foo = 5;
>
> function bar(){
> if(!foo){
> var foo = 10;
> }
> alert(foo);
> };
>
> bar();
>
>

> 5. Write a javascript function to print out the following given input
> 'n' where 'n' is the number of levels:
>
> n = 3
>
> #
> ##
> ###
>
> n = 5
>
> #
> ##
> ###
> ####
> #####
>
>
> 6. Using jQuery write code to create a checkbox hierarchy:
>
> Parent checkbox
> Child checkbox A
> Child checkbox B
> Child checkbox C
>
> - Selecting the parent checkbox will select all child checkboxes.
> - Unselecting the parent checkbox will unselect all child checkboxes.
> - Unselecting any child checkbox will uncheck the parent checkbox.
> - Selecting all 3 child checkboxes will check the parent checkbox.

Kathryn DiBari wrote:
> 1 You are given an array of integers(Both Positive and Negative), Find
> the continuous sequence with the largest sum and return the sum.
>
> 2. . Design a system which represents a flight radar. There are planes
> which are flying in every direction (without any predictable route) and
> a radar that can track their location. System should have a function
> that returns location of ten closest to the radar planes every time it
> is called. The function is called very frequently by clients and should
> return data immediately.
>
> 3.
> There is an array where i-th element is a closing price of a given
stock
> on day number i. Write a program to make maximum profit if you can buy
> and sell stock only once. For example for array {10,3,5,2,8,7} maximum

> profit will be 6 if you buy on day 3 (price 2) and sell on day 4
(price 8).
>
>> On Apr 18, 2016 10:23 AM, "Joseph Mack" <joe@pencom.com
>> <mailto:joe@pencom.com>> wrote:
>> Here are some real questions asked by the MARS team-that will help
you -
>>
>> 1. You are given an array of integers(Both Positive and Negative),
>> Find the continuous sequence with the largest sum and return the sum.
>>
>> 2. . Design a system which represents a flight radar. There are planes
>> which are flying in every direction (without any predictable route)
>> and a radar that can track their location. System should have a
>> function that returns location of ten closest to the radar planes
>> every time it is called. The function is called very frequently by
>> clients and should return data immediately.
>>
>> 3.
>> There is an array where i-th element is a closing price of a given
>> stock on day number i. Write a program to make maximum profit if you
>> can buy and sell stock only once. For example for array {10,3,5,2,8,7}
>> maximum profit will be 6 if you buy on day 3 (price 2) and sell on day
>> 4 (price 8).
>>
>>
>> These are specific questions that were asked from this group in the
>> last few weeks.
>> Thanks
>> 1. The first was to implement a connected graph in java and have a
clone
>> method in the class with returns a deep copy of the graph.
>> I think I did ok on this one.
>>
>> 2. You are given an array of integers(Both Positive and Negative),
Find
>> the continuous sequence with the largest sum and return the sum.
>> Example: input[2,-8,3,-2,4,-10] -> output: 5 (from the continuous
>> sequence 3,-2,4).
>> I was trying to find an efficient way to do this so it took me a
little
>> while, but I ended up doing a nested for loop to check all the
>> combinations which gave O(n^2).
>>
>> he gave a matrix, (B means cannot go through)
>>

>>> 000
>>> BGG
>>> G00
>>>
>>> i need calculate the shortest path between each 0 to G, so result will be
>>>
>>> 211
>>> BGG
>>> G11
>>>
>> *1. Below are questions that I have been asked during today's
>> technical interview on HackerRankX - to add to your questions'
>> collection, which was very useful for interview prep.*
>>
>> 1. Create class which stores big integers (of any length) and
>> implement addition operation.
>> To store the number I used array of integers, one array element stores
>> one digit. Digits are stored in reverse order.
>> Method add() adds digits one by one from right to left, which is easy
>> to implement because array has digits in reverse order. The only trick
>> part is handling overflowing (9+3 = 12, 2 should be written to current
>> position and 1 should be carried over to next digit to the left)
>> The interviewer wanted the fully functional code, created a few test
>> cases and made sure that code returns correct result.
>>
>> 2. Print all items on k-th level of a binary tree.
>> I implemented this with level order tree traversal which stops at
>> level k. Then interviewer asked me to do the same using recursion (you
>> need to pass the current level number to each recursive call and
>> compare it with k)
>> I wasn't asked to actually run the code, but interviewer was diligent
>> to make sure that there are no errors in my code.
>>
>>
>> *
>> 2. Below are questions that I was asked during on-site interview.
>> Hopefully they will be useful for other candidates:*
>>
>> 1. There is an array where i-th element is a closing price of a given
>> stock on day number i. Write a program to make maximum profit if you
>> can buy and sell stock only once. For example for array {10,3,5,2,8,7}
>> maximum profit will be 6 if you buy on day 3 (price 2) and sell on day
>> 4 (price 8).
>>
>> 2. There is a linked list in which each node has a pointer to next

>> node and some random node. Write a code that clones such linked list
>> in O(n) time.
>>
>> 3. Design a system which represents a flight radar. There are planes
>> which are flying in every direction (without any predictable route)
>> and a radar that can track their location. System should have a
>> function that returns location of ten closest to the radar planes
>> every time it is called. The function is called very frequently by
>> clients and should return data immediately.
>>
>> 4. Implement multiple-readers/single-writer lock in Java
(multithreading)
>>
>> 5. There is a binary tree in which each node has three pointers:
>> pointer to left child, pointer to right child and pointer to next node
>> on the same level. Write a code to clone such tree.
>>
>> *
>> Given a sorted array, find out how many times a given integer appears;
>> clone (deep copy) a linked list ; set right neighbor for all nodes
in a
>> binary tree.
>>
>>
>> *These were some of the questions they asked me?*
>>
>>> 1. What was the first thing I did when I saw the problem?
>>> 2. How did I go about reading the problem?
>>> 3. How did I go about designing for the problem?
>>> 4. What kind of designs you thought of?
>>> 5. What was the motivation of using one design over another?
>>> 6. What are some of the things you would change?
>>> 7. If a new requirement came in for alerts, how does it affect your
>>> design?
>>> 8. Why did I use a certain data structure?
>>> 9. coupling vs cohesion?
>>> 10. Then they asked about specific lines in the code
>>> 11. Then they asked, if I had to build this into an enterprise
system,
>>> what are some of the layers that I would need?
>>> 12. Then they asked, how would I design a database to hold some of
>>> this information?
>>> 13. Then they went into light database questions. Foreign keys,
schema
>>> design, data coupling
>>> 14. Do I know about unit testing?

>>> 15. How would I go about testing my code?
>>>
>> *The interviewer asked about how to implement a cache and asked me
>>
>>> specific questions about how the cache could be changed to
accommodate
>>> of unexpected functionality.*
>>>
>> *#1*
>> What is fork(), exec()?
>> What does open() return?
>>
>> C++:
>> Copy constructor, overload assign operator?
>> Open question: what is the difference between passing a pointer as a
>> parameter vs passing a reference?
>> Open question: do you prefer passing a file name or passing a ostream
>> as a parameter?
>> How can you erase an element from a vector that meets certain
>> condition using a for loop and an iterator?
>>
>> Multi-thread:
>> What is a spin lock? What is its side effect?
>> How to avoid deadlock?
>>
>>
>> *#2-
>> -* grep, wc, cat*-
>> *
>> 1. Code to check if binary tree is BST. Order nof vertices and with
>> constant space
>> 2. Reverse words in a given string
>> 3. Queue implementation using stacks
>> 4. Big/little endian code
>> 5. Monte Hall Problem
>> 6. inheritance and virtual functions
>> *
>> #3-
>> *difference between pass by reference and pointer
>> data structure questions, hashmap and map in STL
>> IPC: mutex and semaphore uses, how to design multi thread system
>> use of grep on Linux
>> how to fork, select on Linux
>> use of file descriptor
>> linklist vs array list, design a queue using either list
>> look a sample code, check if an object can be passed by value.

>> how to call c++ function from c
>> how to debug a deadlock
>> *
>>
>> #4-
>> *1. Would the following program compile and what would happen if it ran?
>>
>> void main()
>> {
>> return main();
>> }
>>
>> (Answer: it will overflow the stack when running.)
>>
>>
>> 2. You have a malicious program that writes a 10GB file to your hard
>> drive, then sleeps for 1 minute and forks (i.e., splits off a copy of
>> itself) and repeats. When you run one copy, after 10 minutes, your hard
>> drive is completely full. If you ran two copies, how long would it take
>> for your hard drive to be full?
>>
>> Answer: 9 minutes. You are starting with two copies so you are on minute
>> ahead of the game as when you just started one copy.*
>>
>> #5-
>> *I just finished the interview. It looks it went Ok. Got few tech C++
>> question
>>
>> 1. a variation of atoi function (convert string to an int), first I
>> gave a C version, then rewrote it C++
>>
>> 2. implement hasSum function. The idea is to check if in an array there
>> are two numbers, summing them up produce the expected result.
>> I first gave them a brute force version with n2 complexity (loop +
>> inner loop over the array), they didn't like it, then through a series
>> of hits, I improved it to sort the array first, then the inner loop
>> doesn't have to iterate the array completely, only from the point of the
>> previous iteration.
>>
>> Talked about my experience, asked questions about tech they use.

>>
>> Didn't get any complex questions, like tree/graph traversal, I was
>> expecting to get
>> *
>> **#6*
>> Questions he asked. He only asked me to code the first
>> one.
>> The
>> other two I answered verbally.
>>
>>
>> 1. The second question you gave me about reversing an integer he
asked me.
>>
>> 2. You have two linked lists. One is infinite meaning when you start
>> at the beginning and go to the next node you will always get a next
>> node. One is circular meaning the last element points to the first
>> element. For this list the same condition holds - you will always get
>> a next node.
>>
>> Given one of these lists, how do you determine if it's the circular
>> one or not?
>>
>> 3. You have a 10 leg flight itinerary, say from NY to Sydney. The
>> tickets or legs are not in any particular order. How would you put
>> them in the correct order? *
>> *
>> *#7-
>> *
>> 1) Remove duplicates from a singly linked linked list.
>>
>> Give runtime and space complexity for any solution you come up with
>>
>> 2) Reverse an integer.
>> Give runtime and space complexity.
>> *#8-
>> *1. Given two bonds and an incoming amount, calculate a prorate so
that
>> all of the incoming amount is distributed among the two bonds. Ex: If
>> BondA's balanceInCents is 100$ and BondB's balanceInCents is 200$ and
>> the incoming amount is 100$. The prorate will be 33.33% and the final
>> balanceInCents of BondA will be 67$ and BondB will be 133$.
>> 2. Given an Integer, reverse it.
>> 3. If in a program, a reference to an object is passed to a methodB
from
>> a methodA and the methodB sets the reference to null, will the methodA

>> also have null for the reference?
>> 4. C program in which one variable is stored on Stack and the other on
>> heap. The question was to predict the values of the variables.*
>>
>> #9-
>> *For the written test problem, the first problem is 1900 feet of 1cent
>> coin into a 2*2*2 cubic feet box. Beside correct calculation, there is
>> one situation needs to be covered: When a stack of coin put into the
>> box, because the coin is a circle, so between stacks of circles, there
>> are 4 corner spaces that actually cannot be used. So to solve the
>> problem correctly, we have to assume it is a stack of square being put
>> into the box, where the square covers a coin including 4 corners.
>>
>> *10.*
>> Here are some news about my phone interview. I think it went much
>> better than my interview at Amazon and I think the additional time to
>> prepare was very beneficial. I had three questions:
>> 1. I was asked about the different type of memory that could be used
>> in c++. I was, at first, a little puzzled by the question but then I
>> realized that I was referring to the stack and heap memory. He asked
>> about the differences between them, what each was used for, and what
>> part needed to be managed in c++. He, then, asked me about my
>> experience with exception handling in c++ and what type of memory I
>> would use to implement exceptions.
>> 2. The next topic dealt with binary trees. He asked me to draft a very
>> simple BinaryTreeNode class and then asked me to implement a method to
>> verify that a binary tree was a binary search tree.
>> 3. Lastly, he asked me to design a method to get rid of duplicates in
>> a sorted array in place. For example, starting from:
>> [2, 3, 4, 5, 12, 12, 12, 20, 20, 34, 34, 34, 90, 100, 100] would
>> produce the following array: [2, 3, 4, 5, 12, 20, 34, 90, 100, ?,
>> {duplicates}, ?].
>>
>> *
>>
>> *
>>
>> The questions I had during my interview were not extremely difficult
>> but I expected questions on algorithm analysis whereas I had specific
>> C++ questions.. on maps and structures that I do not use on a daily
>> basis :-\
>> I had to ask my interviewer some questions about the syntax because I
>> forgot some details... I hope it went well...
>>
>> ---
>>

>> Here are the questions I had yesterday:
>>
>> 1) determine how many bytes will be used in the following structure
>>
>> *struct ObjectID {*
>> * const char name[9];*
>> * unsigned short m1; *
>> * unsigned short m2;*
>> *};*
>>
>> Here, you have to know that a char takes 1 bytes and an unsigned (same
>> for signed) short integer takes 2 bytes.
>>
>> 2) Let's consider this new structure
>>
>> *struct ObjectID {*
>> * std::string name;*
>> * unsigned short m1; *
>> *};*
>> *
>> *
>> and let's consider a map that uses this structure as the key:
>>
>> *std::map<ObjectID, int> myMap;*
>> *
>> *
>> What do you need to write to make this map work?
>> -> you need to write a comparator because the map does not know how to
>> compare 2 ObjectID (this would not have been an issue if the key was
>> an integer or a string for instance)
>>
>> 3) Write a comparator for this map:
>>
>>
>> here is the code I wrote:
>>
>> *// structure that contains 2 attributes*
>> *struct ObjectID {*
>> * // name*
>> * std::string object_name;*
>> * // a short integer*

>> * unsigned short m1; // 2 bytes*
>> * // overload smaller operator*
>> * bool operator<(ObjectID& o2);*
>> * bool operator<(ObjectID const & o2) const ;*

>> *};*
>>
>> we overload the < operator.
>> NB: something that I did not see but I saw after the interview is that
>> you need to implement the operator that takes a constant parameter
>> because it is how it is written originally for the < operator,
>> something important to remember when you overload any operator or
method.
>>
>> *bool*
>> *ObjectID::operator<(ObjectID& o2){*
>> * // compare the name*
>> * if(object_name < o2.object_name){*
>> * return true;*
>> * }*
>> * // else if the name are identical*
>> * else if(object_name == o2.object_name){*
>> * // then compare m1*
>> * if(m1 < o2.m1){*
>> * // if m1 smaller, return true*
>> * return true;*
>> * }*
>> * // else m2 is smaller, and return false*
>> * else{*
>> * return false;*
>> * }*
>> * }*
>> * // else my object name is larger than the object name of o2*
>> * else{*
>> * return false;*
>> * }*
>> * // end of method*
>> *}*
>> Be careful, you need to compare all the attributes of the structure in
>> order to make a full comparison!
>>
>> 4) Let's test it!
>>
>> i*nt main()*
>> *{*
>> * // instantiate an object of type std::map*
>> * std::map<ObjectID, int> myMap;*
>> * // instantiate a ObjectID object*
>> * ObjectID myobj1;*
>> * // ...and set the attributes*
>> * myobj1.object_name = "cat";*

```
  >> * myobj1.m1 = 2;*
  >> * myMap.insert(std::pair<ObjectID, int>(myobj1, 12));*
  >> * // instantiate a ObjectID object*
  >> * ObjectID myobj2;*
  >> * // ...and set the attributes*
  >> * myobj2.object_name = "dog";*
  >> * myobj2.m1 = 1;*
  >> * myMap.insert(std::pair<ObjectID, int>(myobj2, 1));*
  >> *
  >> *
  >> * // print my map*
  >> * // loop over the elements of my map*
  >> * for(std::map<ObjectID, int>::const_iterator It=myMap.begin();
  >> It!=myMap.end(); ++It){*
  >> * // ...and print*
  >> * std::cout<< It->first.object_name << " " << It->second <<
std::endl;*
  >> * }*
  >> *
  >> *
  >> * return 0;*
  >> *}*
  >>
  >
```

Largest Rectangular Area in a Histogram | Set 1