module-based-textbook

Textbook

Each chapter should summarise one module. You do not need to break it down by lecture - simply write a helpful summary of the content from that module.

Helpful here means:

1. *help your classmates understand the material when they are first learning it*
2. *help your classmates when they are revising and studying the material*
3. *help your classmates when they are in the exam*

You do not need to sumarise what is *said* in each *lecture* as you did in your first year courses (you can do that in your personal lecture blogging if you wish). Instead in this course we want you to briefly **explain** the concepts in each module (in your own words or in mine whichever you find most helpful). Notice that this is a higher level thing to do that the simple **describing** we asked you to do in first year.

Details of who does what chapter are on the Classes link above

---

Table of Contents

Introduction

1.0.0 About The Course

This course is about solving design problems with your wits, not your technical mastery. Java is taught, but it's not about Java. It's more about Object Orientation, but it's not about Object Orientation. It's really about design and becoming a designer. We'll design complex systems, and we'll Object Orientation to do it, and Java to do that. **This is not a Java course.**

The previous courses (Computing 1, Computing 2) have taught craftsmanship and science. This course teaches design. You can already get the little bits right, but it's always too easy to make a mess of the big picture. Software projects quickly become unmaintainable and confusing.

Being able to critique something, study it and understand it doesn't make you good at it. Like poetry - you can know everything about poems but that doesn't mean you can write a good poem.

We teach C before Java because there's nothing scarier than someone who thinks they know the big picture but has no concept of the implementation.

More than half of software projects fail.

1.0.1 Solving Problems

Software development is hard. We can make cars and bridges really well, but we're not good at writing software. Industry still hasn't gotten it right. Even the biggest companies make dodgy software.

Scientific approaches solve small problems, like sorting numbers, not large, ill-defined, real-world problems.

Google is digitising every game in the Universe. They're all going to get digitised. The real problem isn't algorithms - it's what the hell to do. The problem is massive - there's no starting point. When we have a problem like this, we tend to have a lot of good ideas about things we can do, and we try doing all of those and don't succeed at any of them. Furthermore, every time we show what we've done to our boss, they realise they want something slightly different, and the spec changes!

Agile

1. Cleverly select the most important feature, and talk the client in to it.
2. Get that working.
3. Ask the client what to do next.
4. Then add those features.
5. Go to Step 2.

By the deadline, at least you have something.

The Project - Roma

The last version you'll make is version 3.2, but imagine you're going to version 5.0 - you aren't expected to get all the features in by the end of the course. Each week you'll pick the most important features and implement them. Provide a list of what's new, known bugs, etc.

You'll be working with a partner - don't work with a friend, it could strain your friendship. Could choose randomly - good if you don't know anyone. Go with someone who'll put in the same enthusiasm. You could do it by your marks from last year.

Good Design

Lollipop - "GO" on one side "STOP" on the other - operator can't screw it up and crash cars.

1.0.1.1

Intro to Java

- We are using **Java 1.6 (aka Java 6)** in this course
- **Java programs end with .java** (surprise!)
    - has to be named with camel case
- C convention for naming files;
    - if file contains an ADT, file is named after said ADT.
    - since ADTs are named with a capital letter 1st, the files are too.
- Similar thing for Java
    - if the file defines a class (java equivalent of an ADT) file must start with capital letter
    - However, almost all .java files contain classes - so, **almost all .java files will start with caps**

Compiling a Program

- In c, compiling a .c with gcc produces a .exe (executable)
- In order to compile a .java (e.g. HelloWorld.java) , type into terminal;
  `javac HelloWorld.java`
    - **This will produce a HelloWorld.class**
    - Assertions are compiled in, but you decide at runtime whether or not you want them in
    - Don't need flags to compile with javac. Java is a lot simpler than C
- In the C world, there're 2 steps to compilation;
    - .c gets compiled into .o (object file)
    - All these object files get linked together to make the .exe
    - GCC does these 2 steps in 1
    - Each .c file is converted into .o (machine code), and linkup makes .exe
- With java, we don't do this together
- Whilst C compiles into machine code, Java compiles into "special Java code"
    - So, we must run it through a "Java interpreter" to run program

Running a Program

- For C;
  `./HelloWorld`
- For java;
  `java HelloWorld`

Looking a the HelloWorld.java Program

```
class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello world!"); //awwyea
    }

}
```

- `class HelloWorld{...`
    - This file defines a class called HelloWorld

    - In C, all programs are public and static anyway
    - Usually 2 args for main in C, argc and argv
    - Here, we just pass in an array of strings as arg - we're not passing in size of array
        - In java, unlike in C, arrays know their size!
        - if you try to access outside bounds of array in java, runtime sys will pick it up
- `system.out.println`
    - println = C's printf
    - There's the extra overhead in C of having to #include stuff
        - harder to express simple things, but easier to express more complex things
        - e.g. 'ugg' vs. Shakespeary English
    - System.Out says "This is where you look to find println"
        - Sort of like `#include <stdlib.h>`

1.0.2

Richard Suggests No More Pyramids

- Previously, we programmed with pyramids
- 1st year, if there were problems, Richard would've said "get problems & break into sub-problems"
    - write top-level routine that calls functions related to each of those sub-problems
    - etc. etc.
- Creates tree - guaranteed to work. Nice way to get started.
    - top-down design, or hierarchical programming
- Good for small problems, but really bad for large problems
- Problems with top-down design in big program.
    - Really elaborate structure is all based on 1 problem. If this 1 thing changes, you're fucked - 'brittle system'
    - Focused on process of solving a problem
        - However, irl, usually solve more than 1 problem, and these problems change over time
        - You want this system to be able to solve
- Point = in any complex system, the elements of the system stay stable, though how you want to use these elements change over time.
    - Write a system that encapsulates these relationships between things and record that - this is much more stable.
        - This is the idea behind Object-Orientation

Process Vs. Data

- Traditional C program;
    - lots of little pyramids (functions) in the static half
        - static = happens at compile-time
        - dynamic = run-time
    - Data is given types
    - typedef defining pointer to struct (ADT)
    - in dynamic-land, the functions act on the data
        - functions do exciting stuff, and data is just what stuff is happening to.
    - At runtime, the functions turn into processes
        - the data becomes an area of memoery
- Object-oriantation = data is not subordinate to functions, but the other way around!

How Object-Orientation Works

- Focuses on program as modelling something (instead of a series of processes)
  - elements of program are components of this system
  - describe relationship with each other
  - components look after themselves in a decentralised way

1.0.3

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Source: http://www.sh1v.com/tubes/access.png

Interfaces

Lecture 1.1.0

Recap

- *General*: In this lecture Richard talked a mainly about the history and motivation of Java.

Java
Intro

- Java was invented by James Gosling - released in 1995, developed at Sun Microsystems.
  Motivation
- At the time, there were many different architectures of of microchips and whenever a program needed to be on another system, a lot of porting had to be done, this was expensive and more often than not, the program had to be rewritten.
- Gosling dreamed of the ability to write once, and run it everywhere.
- The Java Virtual Machine was born.

JVM

- The JVM was simply an emulator which would be run on every chip.
- This meant that everytime you had a new chip, all you needed to do was to write a JVM for it.
- Once a chip had a JVM, ALL java programs would be able to run on it. "write once, run anywhere".
- Now you could write a program to run on whatever machine had a JVM

Problems

- Corporations didn't like this idea: they wanted money and you can't earn money when a

  Netscape die off.
- Microsoft tried to do a similar thing with java - took their code (java made their source free) and microsoft tried to change it for their own uses and their version of Java run only on their machines.
- Java sued, Java won.
- Java is slower than other languages: garbage collection, uses a virtual machine
- Java is VERY large -> lots of libraries.

Benefits of Java

- Oddly enough, beauty is in the eye of the beholder and so are features. Garbage collection is great -> no need to remember allocation memory and freeing it -> java does all that for you.
- Java is VERY large -> Lots of functionality. You can stand on the shoulders of giants.
- Java is open source, everyone can use it
- Java is OO - object orientated language, so object orientated designs and flourish in Java. Also means that abstract data types are a lot more controlled and variables are a lot safer (public/private)
- Java has extensive IDEs to help developers (eclipse/Intellij).
- Public variables are good for interfacing. Private variables are for abstraction. Java fully secures this and enforces this abstraction.

Design

Just a closing note: What actually is good design? Can we quantify it? It it more of an art? Is there even an answer?

Lecture 1.1.1

Recap

- *4:39* - **Fields**: When we declare variables in a class in Java, they are stored in the Heap and persist. Unlike local variables (written in functions) which are temporary and stored in the Stack.
    - Inside a class, when we call a field, it assumes we're asking for a local field.
    - The constructor of a class mallocs memory for us. Java abstracts away memory management so that we don't have to worry about malloc-ing or free-ing memory. How it works is that when Java sees that there is no longer a reference to an object, it deletes it. This is known as 'garbage collection'.
    - We can have multiple definitions for a constructor as detailed below.

```
public Circle(int radius)
public Circle(double circumference)
```

- *14:03* - We always get references or pointers to objects in Java. i.e. Game game; (this command sets aside space for a pointer to a game object but points to 'null' initially)
- *17:46* - What happens if we do game2 = game1? game2 now points to game1. game1 is lost (no reference to it) so garbage collection removes it

Lecture 1.1.2

Recap

group in the course made their own hunter.c which had the AI for a hunter to compete in the game. Since all the hunter.c AI's have to send actions to the game, we could make a common **interface** of type Hunter which would detail the functions every team's hunter should have in order to be able to be of the type Hunter.

  - E.g. Lock Acceptance Tests: Rupert wanted an array of tests. Every test is a class but they are not of identical type Test so we can't make an array of them. Interfaces let us write different implementations of a class yet still share the same type even though the functions and fields are different for each implementation.
    - An interface looks just like a header file in C and is defined like so:

```
public Interface Test {
        public void run();
}
```

To use this interface we write:

```
public class TestEasy implements Test {
    public void run() {
        Lock aLock = new Lock(5,7,4,7,3,7);

        ...
    }
}
```

- Now TestEasy is of type TestEasy and Test. So Rupert wrote an array of type Test.

- *25:00ish* - e.g. We could have a car transporter which is holds an array of type car.
  - Now we can implement the Car interface for different types of cars e.g. Prius, Peugeot
  - Also, we can implement multiple interfaces.
    - A bus lane could only accept vehicles of type BusLaneVehicle.
    - We want to model a taxi which is a type of car and can go in the bus lane. So we implement both the Car and BusLaneVehicle interfaces.

- 32:54 - Every type comes with a built in equality checker.
  - C checks if types are equal if they point to the same thing in memory and the same behaviour is present in Java.
  - A common pitfall is with Strings demonstrated below. This occurs because when strings are compared using the == operator, their memory addresses are compared rather than the actual content of the String.

```
String name = "john";

name == "john"; // will return false

name.equals("john"); // will return true
```

Design

Lecture 1.2.0

Lecture Notes

We are expected to take good, detailed lecture notes (3 or 4 hours of work each) - they should

Object Oriented Programming: 'is a' and 'has a'

This lecture is brought to you by 'is a' and 'has a'. When seen from above, an object oriented (OO) program looks like a series of objects interacting with each other. Our challenge as programmers is figuring out what the objects are, and then determining the relationship between these objects. The standard relationships in OO programming are 'is a' and 'has a'.

For example, a car class has wheels, has an engine, has a sunroof and conforms to the vehicle interface. A car can also be turned on and off and has pedals to control speed.

Questions:

- What are the objects?
- What are their relationships?
- What responsibilities do they have?

In Java, objects 'have' fields (or variables) and methods (or functions). If an object has a 'has a' relationship, you do this in code by giving a field inside the object, for example:

```
public class Car {
   SteeringWheel ...;
}
```

This leads to a nice approach called delegation. For example the combination lock: we said it had a fence, notches, pins, wheels.

There are 3 approaches to the problem:

- C approach, Monolithic - I'll create a Lock class with huge functions that the interface requires and they'll be really huge and it will do all the work
- Wheel class, lock class, Delegation - A lock *has* some wheels (3 or 4 in this case). This is better than the above approach. This means that you don't have to keep track of things like where each wheel is. You let the wheel look after itself.
    - Analogy: Richard always gets stressed running courses, so he employs tutors and course admin, and he delegates to them and they do things for him.

Richard thinks 'has a' is nicer than 'is a', but 'is a' is where everyone trips up in OO.

Richard's 'has a' example:

```
public class Car {
    private boolean isOpen;
    private Engine engine;
    private Speedometer speed;
    private Wheels wheels;
    private Accelerator accelerator;
    private Brake brake;

    public static void main(String[] args) {
        System.out.println("Hello from Car!");
    }

    public Car () {
        this.isOpen = false;

        // when we create a car, create an engine
    public boolean isRunning() {
        return engine.isRunning();
    }
}
```

A car *has a* Brake, Engine, Speedometer, isRunning method, etc.

Subtyping

Subtyping emerges from the idea that in OO it is possible for an object to have more than one type.

For example a car carrier carries cars. But if a new variety of car is a subtype of car, then we don't have to recall all car carriers every time someone thinks of a new type of car - it's better that cars have a standard way of fitting in things. This was done with interfaces in week 1.1.

Subtyping is cool because things can be substituted for each other.

Subtyping only goes one way. For example if you have a human type and a librarian type and something wants a human, you can give it a librarian. You can't give a human type to something that wants a librarian though.

Subtyping makes new things a runtime problem rather than a compile time problem, so we don't have to go through fixing our code. It also supports something neat called polymorphism, which we'll talk about later.

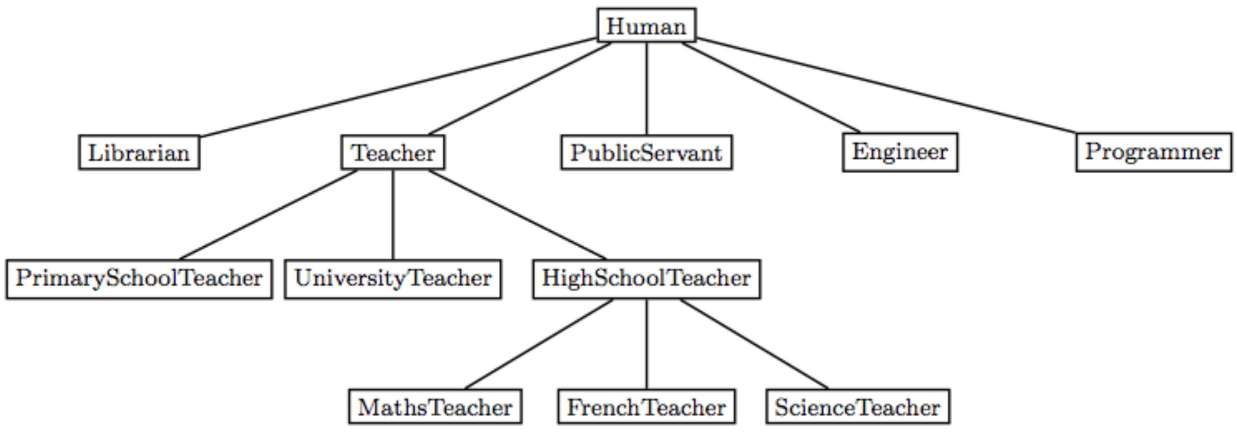
Subtyping and subclassing differences

A type is an abstract thing we have in our head, a class is more of a syntactical thing and a class can have multiple types.

Librarians are a subclass of humans if everything you can do with a human can be done with a librarian (but librarians can still do extra stuff too if you want them to!). This sort of subclassing tries to capture the 'is a' relationship.

Java has inheritance that sets up a subclassing heirachy. You have a series of objects and they're arranged in a tree.

For example:



  a Programmer or Human.
- If you want a MathsTeacher you can only use a MathsTeacher, you can't use a any Teacher, HighSchoolTeacher or a ScienceTeacher.

Not everything can be captured this way, but a lot of things can.

Historical example: Military Chain of Command

In the army you trust people increasingly as you go up higher and higher. This is how secrecy is guaranteed in the army. They found out during the second world war that that didn't work very well. You might not want to tell someone at the very top your secret, for example the Enigma Machine secret. Churchill thought this was probably the most important secret of the war (that they'd cracked the Allies' code). This was secured by another type of secret called multilateral security or codeword security (or role based access control in modern computer systems). "Magic" might be the keyword. Then only people who knew the keyword could find out the secret. It's an ad hoc model - you might or might not be cleared for "magic". Churchill thought it was really important that no one found out about "magic", so no one who knew about "magic" could leave the country or go to Germany.

Codeword secrecy is like implementing an interface, hierarchical model is like subclassing.

Subclassing is like saying that the subclasses share a common interface.

Subclassing is subtly different to subtyping. Subtyping is 'it's safe to use a taxi instead of a car'. Subclassing is 'it's legal to use you over you'.

For example, a square is a subclass of a rectangle: a square has all properties of a rectangle and has all the functions: getWidth, getHeight, getArea. But a square has different rules to a rectangle (all sides must be equal), and a square can't use the same constructor as a rectangle (int width, int height) because that would allow you to create a square that wasn't square. For interrogation properties a square is a rectangle, but not for changing properties. Richard's take is that an immutable square is an immutable rectangle. But a mutable square is not a mutable rectangle.

Another example, if you have a stack and a hopeless stack with the same functions, but hopeless stack just throws things away when it is passed them, then syntactically they are the same but not semantically. Types are about meaning, classes are about syntax. Just because things have the same syntax and functions doesn't mean they can be used interchangably.

Deciding whether something is a subtype

To determine if something is a subtype of something else, use the Liskov substitution principle: wherever you expect an `a`, you should be able to pass it a `b`. If this holds, `b` is a legitimate subtype of `a`.

One way of thinking about it is that a type is a set of all values that can exist in a type, subtyping is a subset of these values. This is harder to do in Java since it's not just getters but also setters.

It is undecidable to work out if two functions can be substituted for each other - it converts into the halting problem (substitute the function with "will halt"). Even worse is if you have non-deterministic programs - we have no canonical version to tell if they're the same.

Checking for subclassing a compiler can do - it just checks type signatures. Subtyping is up to

Design approaches

- Model the real world
- KISS
  - Only do exactly what is asked, just refactor when things are needed
  - Don't make it more general or abstract than it has to be
  - Just write what you need now, don't anticipate problems (there are too many), keep unit tests so you can refactor quickly later.

Counterexample of KISS, the beer pouring Rube Goldberg machine - solving something in the most complicated way you can. (video)

Primitive types and classes

Java has primitive types and classes, but for efficiency and speed they have primitive types like integers and chars, as well as classes. Java has set sizes for types unlike C - an int is always 32 bits.

Extra Information

`byte`: The byte data type is an 8-bit signed two's complement integer. [-128, 127]

`short`: The short data type is a 16-bit signed two's complement integer. [-32 768, 32 767]

`int`: The int data type is a 32-bit signed two's complement integer. [-2 147 483 648, 2 147 483 647]

`long`: The long data type is a 64-bit signed two's complement integer. [-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]

`float`: The float data type is a single-precision 32-bit IEEE 754 floating point.

`double`: The double data type is a double-precision 64-bit IEEE 754 floating point.

`boolean`: The boolean data type has only two possible values: true and false.

`char`: The char data type is an unsigned 16-bit integer representing an Unicode character. [0, 65 535]]

Unlike C, you cannot stick modifiers in front of a primitive type, so `unsigned int` or `long double` is invalid syntax. There is no way to get 8-, 32- or 64-bit unsigned integers.

Design Task 3

See if you can record something that makes sense forwards as well as backwards, like double-backmasked tracks. The "Buckaroo Banzai" effect.

Things will go wrong - no-one thought to check that the machines in the labs could play sounds. When you do your presentations, everything will go wrong. Everything that could possibly go wrong will go wrong, so be prepared.

You can quantify the chance something will go wrong as a number between 0.3 and 17.5 (where 17.5 means it will almost definitely go wrong) using this formula.

[illegible]

can do anything. Use the Java documentation - Google it, make sure you get version 1.6.

`Integer` is a subclass of `Number`, `Number` is a subclass of `Object`. Everything is a subclass of Object! That means you always get equals, clone, toString, etc.

Java packages

In C we have libraries - lots of useful things that people might want, that are squished together into one unit that you can #include. It's a good idea, so Java does a similar thing - common things in Java are lumped together into packages. If you say nothing, you get put into the 'Default Package'. Package names start with lowercase letters (by convention). Technically you should say `java.lang.String.blah()` - that tells you which String to talk about, but Java has a rule that if you leave off the package name it always searches in `java.lang`.

The default object is defined in `java.lang`.

Java importing

Java has importing which is different to C #includes - in C a `#include` just dumps everything into your code - in Java importing tells Java somewhere else to search, e.g. `import java.io.*` lets you use everything in `java.io` without using the package name

Java strings

In Java strings are immutable - if you want to modify a string, you just make a new one. Also in Java the plus operator will concatenate strings:

e.g. `System.out.println("Hi" + 6 + "freeman")` prints *Hi6freeman*

`"Hi" + objectThatIsNotAString` means concatenate `"Hi"` and `objectThatIsNotAString.toString()`

java.util.Scanner()

`java.util.Scanner()` is really good at parsing - given strings it's really good at getting stuff out of them

javadoc

We all have to submit code and explain our design but the documentation is always a little bit behind. We have a strong incentive to get the code right but not to get the documentation right. Knuth suggested literate programming - your program is the documentation. All lines are comments unless otherwise specified.

The Java idea is `javadoc` - it comes with the JDK, along with `javac` and `jar`.

To generate Java documentation: `javadoc -public *.java`

This creates a collection of HTML pages documenting the `.java` files.

Lecture 1.2.1

Lecture Notes

abstraction you are modelling, generally the better the design will be. Anybody who modelled the quarks in the lock task overcomplicated it.

Exceptions - What happens when something crazy happens

- You can have lots of error detection and exception handling
- Or you can expect no problems to occur (and explode when they do)
- Or in java we can have something called exceptions, which lives between the two

Exception handling is a clever idea that says 'let's just say we know there will be problems, and when there's a problem we'll call it an exception'.

For example, how might we handle a divide by 0 exception? We pass it up a chain of command until someone can handle it. This is like when a receptionist is given an poorly completed form, so they call their supervisor or their manager and so on, until someone can handle it. In this case the exception handler might ask the user to enter a different number or just assume 1 and carry on - it depends on what the program is for.

In systems we create we want to handle exceptions as gracefully as possible. If something goes wrong we should make it fail safely - for example this may mean rolling back changes carefully and aborting.

In Java we have:

```
try {
    myObject.doSomething();
} catch (Exception e) {
    System.out.println("Exception: " + e);
}
```

All exceptions are subclasses of `Exception` and they can get quite specific, e.g. `InvalidArgumentException`.

In the above code, it will run the code in the try segment. If an exception occurs, Java will generate an Exception object and will check all catch exceptions to see if there is a match for the object. It will then run that code instead. So if myObject were null, it would run the try code, report a null object trying to be printed as an exception, check if we considered that as an exception. We did so it will run the System.out.println().

Note that `e` above contains a description of the exception that occurred. When you print e, Java implicitly calls e.toString() and prints out a nice error message with information about what the exception was.

After the curly bracket of the catch statement, the program will continue as normal. If you don't want it to, you have to put an abort or something in the catch statement.

Functions which contain code that may throw an exception, but which do not catch it must state this explicitly. If you do not declare it, the compiler will complain that no-one is responsible for when something goes wrong.

```
void doSomethingToMyObject() throws Exception {
    myObject.doSomething();
}
```

Exceptions can be used as a `goto` or for handling internal user behaviours that you're in control of, but Richard feels it's nicer not to use exceptions this way, and to deal with this sort of units (such as the standard library).

For more information, here is a nice explanation. Exceptions In Java

Reading and Writing Sound Files and the Secret to Java Programming

Have a look at the reading and writing functions: Reading and Writing Sound Files

You'll need to find the sample size and reverse the array, then output it to a file.

But you'll need an AudioInputStream. The secret to Java programming is to find the way to convert what you have into what you need. Learn to navigate the java docs.

```
byte[] -> ByteArrayInputStream -> AudioInputStream
```

Lecture 1.2.2 + 1.2.3

Choosing objects

A good way to figure out what objects to model is to choose the concrete nouns: car, speedometer, accelerator, etc. You might end up with extra objects than these (since some things that need to be modelled won't be concrete), and some of these nouns won't need to be modelled because they have no interesting relations or properties, but it's a useful starting point.

Naming conventions

Booleans should be named after predicates, so that if you use it in an English sentence it's obvious what it means, for example

```
if (sunroofOpen()) ...
if (sunroofIsOpen()) ...
```

Choosing functions

The methods an object has are the verbs - what each object can do.

For example, a driver can turn the engine on and off, open and close the sunroof, check if the engine is running, check if the sunroof is open, check the speed of the car, press or release the brake and accelerator pedals.

Inheritance

Video 1.3.0

Design Journal

Every week write down some things that have happened

- don't need to write a lot
- not a transcript
- thoughts about design, project, design task

Inheritance is a way to re-use code, like re-using structural design in physical engineering, struts, arches, etc. Very often a software problem has been solved before and starting a solution from scratch is wasted effort. Inheritance seeks to solve this.

Open/Closed Principle

Bertrand Meyer - The implementation of a class should only be modified to correct errors

- New or changed features should be put into another class
- Open to extension
    - E.g. AudioReverser reverses .wav files, wish to extend to also reverse .mp3 files
- Closed to modification
    - Abstraction might be violated, original functionality might break, overall - don't want everything fiddled with unnecessarily

Inheritance - Chess

If we have some ChessThinker class and wish to extend its functionality to solve some chess problem, e.g. Complete piece, N-Queens, Knight's Tour, etc., we would extend ChessThinker.

Inheritance Keyword - **extends**

ChessBing **extends** ChessThinker

- ChessBing inherits all public and protected members and methods of ChessThinker
- Private members and methods are not inherited
    - However, if there are protected or public methods to access private members in ChessThinker, ChessBing can access the private members through those methods

Inheritance allows us to achieve two key things

- Code re-use
    - Code from ChessThinker is automatically now in ChessBing
- Interface inheritance
    - Forms an 'is a' relationship
    - ChessBing is a ChessThinker ChessBing is a **subclass** of ChessThinker However, ChessBing is not necessarily a **subtype** of ChessThinker
- As ChessBing has all publicly accessible methods of ChessThinker it can be used in place of a ChessThinker
    - Syntactic equivalence
- If some functionality of ChessBing did not match that of ChessThinker, then it would not be a subtype
    - i.e. it wouldn't behave in the same was as ChessThinker (
- A subclass is simply a class formed using inheritance.
- A subtype is is class that can be substituted (using polymorphism) with no observable effect.) **IMPORTANT**

Use Inheritance when you want BOTH code re-use and if there is proper relation between the subclass and the superclass or bad things happen.

e.g.

- Create a working PriQ class
- Want to create a Queue class but feeling lazy so decide to use inheritance

    - Due to interface inheritance, whenever something expects a PriQ, it can still take in Queue (but shouldn't!)

If code re-use is wanted but not interface inheritance then we use delegation of 'has a' i.e. Queue has a PriQ member, and we use the PriQs methods from the member. (In software engineering, the **delegation** pattern is a design pattern in object-oriented programming where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object.)

Summary

- Code re-use, no Interface Inheritance- Delegation
- Code re-use, Interface Inheritance- Inheritance (extends)
- No Code re-use, Interface Inheritance - Interface Implementation

Private / Public / Protected: Functions/Methods/Variables

Private

- Only the class can access & modify
- Other objects in the same class can access private methods (in that class, everything is public to every thing else in that class)
- Sub-classes cannot access

Public

- Anything inside or outside the class can access or modify it

Protected

- Public for the current class and any sub-classes as well as package
- Private for all other classes
- Generally bad design (*why?*)
    - a hint that the superclass has too many responsibilities and the subclasses are likely to be coupled to its internal design, rather than to its external interface/type

Default

- Public for code within the same package
- Private elsewhere

Jargons

- subclass
- superclass
- parent
- child
- abstract

Abstract class

- Can implement some methods but not all of them.
- Cannot make an instance of an abstract class
- The classes that extend this have an obligation to implement the other stuff.

    - Only do it when there is an "is a" relationship
    - e.g. `String toString(){ return "Hello" }`
        - this function can override the initial function getting from the parent class

Super

- super.object = object of parent
    - e.g. in queue class, you can overide the priority queue's enQue function and when you want to use the parent's enque function, you have to use super.enQue

final Design question: when is it best to use which feature?

- should question oneself over the rest of the course + programming career

Final

- this is the end

Video 1.3.1

- Take pair programming seriously
- RoboCode
- Implement an interface. Extend - traditional inheritance.
- Tight coupling - two classes - want them to be loosely coupled. But inheritance is tight coupling.
- Inheritance pros:
    - Quick method of code re-use without rewrite
    - Sub-classes can be used anywhere the parent class was used.
    - The parent class does not have to be changed. This means that old code dependent on the parent class won't break.
- Disadvantages
    - Can violate the principle of abstraction
    - this can be neat but also means the child needs to know how the parent works. and if the parent changes any of it's internals even if it still complies with its public contracts, it can break the child. i.e. the child can rely on more than just the public contract.
    - e.g. the child can access internal methods and attributes of its parent if these attributes/methods have protected access rather than Private
    - e.g. if the child replaces any of its parent's public methods which are called by other methods in the parent.
- So far - not that much need for inheritance.
- Useful packages:
    - `Java.util.random`
    - `Java.util.ArrayList`
- Interfaces can also inherit. E.g. list interface, sorted interface (interface sortedList extends list)
- No attributes in interfaces but you can have constants.
    - Debates - should it have constants?
    - http://www.theserverside.com/discussions/thread.tss?thread_id=19221

Video 1.3.2

incredibly hard to represent the state of the chess using objects. This is because there are a lot of ways that you can represent the game. It is hard to code the interaction that is why it's so hard.

Who knows where the pieces are:

1. Just the board?
2. Or the piece?
3. Or the board and the piece?
4. What if they each think different things?

This task, the lock, and task 2.0 is the crux of the course. It's hard because there's lots of thinking involved.

Although in practice in work, you can't spend forever to think about coding. What matters is that you put a lot of effort into thinking about it. Look at the model solutions and then think about it.

Work out how much time and resources you can spend and allocate the tasks. Then decide what you can do and do it perfectly.

Back to Design

Google has uncensored China! What do we think about censoring the internet?

- there are bad things that are harmful to society
- think of the children

The idea behind censoring is that there are things that aren't good about society and we shouldn't know about them/think about them

The danger: Someone has to decide what's right

- This is a centralised/paternalistic approach

NOT WHAT WE WANT Programming in OO is

- Decentralised
- Trust the underlings
- Don't control everything

CRC Cards (Class, Responsibilties, Collaborations)

There are many way you can represent the class hierachy:

- inheritance
- extensions
- abstract classes
- interfaces
- private
- public
- protected
- patterns

The trick is working out what the relationships are up front

1. Start talking about what program has to do.
2. Everytime they get to a noun or a possible class, they write the name of the class on a piece of paper
3. Then they list its responsibilities ie Chess board has the responsibility of showing the pieces and remembering the pieces
4. And then you list its Collaborations ie Car needs to know about acceleration, brakes, sunRoof
5. After you think about it, if you don't want it you put it aside (easier to do than putting aside code you have written)

How much is too much responsibility?
Too much to fit on the card. If there are too many responsibilities move it to another class

Why is it bad to have a card that does too much?

- It makes it complicated
- There are too much interactions
- It doesn't take half the time to debug 100 lines of code compared to 200 lines
    - Because we can't fit it all in our head (cognitive ability)
- It's bad presentation

It's very nice for each class to be responsibile for just one concept

Testing Driven Development

1. Only write what you have to test
2. write the bare minimum to make those tests pass, even if you are just echoing it
3. then you refactor
4. write more tests
5. add more code

The steps you're doing are so small the design of the class emerges as you write rather than upfront designing

Review

We talked about Greeks because they were good designers

- political system
- transfering power

Suppose you had a Pawn Class, a Rook Class, and a Queen Class. You also have a ChessThinker class with a canMoveTo(Piece, destination) function that does a lot of else ifs and in the end, returns the answer.

This is the Egyptian way of thinking, with a single function that does everything. We should instead delegate. Every piece should have a canMoveTo function.

Things that we should do:

- The code for a class should all fit on page or
- Each class should have no more than ten methods, and each method should have no more than ten lines
- [illegible]

Are called to create an object, but you have to ask them to do it ie "new" is a constructor function

Purpose of a constructor is different to a method. Purpose of a method is to do something Purpose of a constructor is to set something if you don't write a constructor in any class, java sticks one automatically in there for you It takes no argument, but you can create one that does (java won't do this automatically)

Constructing an animal

Conditions:

1. Animals have a name
2. Every animal makes a sound

Three different constructors for the animals:

1. specify the name and the sound
2. specify the sound and give it a default name
3. construct an animal with a default name and a default sound

It's duplication of code to do write all that set up stuff in all three of them. Sometimes it's convenient to get one of the constructors to do a little special thing and then call the other constructor. You can do this, the idea being that you call the other constructor first, and then you call your own constructor to add in special stuff

You can use the word "this" to refer to another constructor of that same object, and you can only do it in the first line of the constructor, after that the compiler starts to grumble.

Intro to Inheritance

Class b extends class a. Class b inherits all of class a's methods. **BUT** Class b doesn't inherit Class a's constructors. Class b must work out how to make Class b. If you want to call your parent's constructor, you can (only in the first line of your constructor) using "super".

Example: mammal and animal. Mammal extends an animal. The source code for mammal only has functions that make it unique to the mammal. If you don't put it in a constructor, Java will do it automatically (calling it "super"), but it will only call your parent's constructor with no arguments.

Called: Auto magic.

See Richard's Animal.java and AnimalUser.java.

Notes on this program

Our constructor takes in arguments, so the default constructor that Java gives you will not work (will complain during compile time).

See Richard's Mammal.java.

It has an extra function that gives it hair colour and overwrites the parent's function toString. Previously, there was no constructor in cluded. Java then put in the default super, which has no arguments, and so the compiler complained. This means that you must call the parent's

Video 1.3.3

A good warmup to get a sense of how one constructor calls another:

1. Write a two paramater constructor and default constructor for animals
2. Write a constructor where you specify noise and name
3. Write a constructor didn't specify either and got given default values
4. Practice calling them
5. Repeat for mammals

Queues

Richard's Queue.java
We are now in the position where someone has a class already written that does everything we want and we want to add extra functionality to that by extending. This will be done without looking at the implementation of the code and only the javadoc on it, writing code from the perspective of a user.

See Richard's TestQueue.java.

This is a queue that models a queue of objects, so you can put any object in it. The idea is that this is a queue at a hospital. These are the people who have come into the emergency room and we deal with them in the order they come in (does not model real life). Because of the context, it is quite serious if we lost someone from the queue. Richard's test adds patients onto the queue then removes patients from the queue, testing that they are removed in the correct order then that the queue is empty.

Notes on this program

1. The function q.add adds an object to the rear of the queue (as seen in javadoc).
2.

```
assert (q.head() == patients[0]);
```

This compares that you have two references **to the same string**, and not the contents of two different strings.

Inheritance

Flueue (Flaky Queues)

This is a queue that forgets every fourth element. Instead of rewriting all that code out, we're going to inherit a queue (as previously tested) to make it. The point of this exercise is a practical example of the theory Richard discussed in Video 1.3.0, about how people may be tempted to use inheritance for code re-use but there is not a genuine 'is a' relationship leading towards catastrophe.

See Richard's Flueue.java.

Notes on this program

This program has overwritten the parent's add function by adding an extra parameter before far as java is concerned, a flueue is an example of a queue ("is a" relation). **However**, it does not behave like a queue and will create runtime problems if used as a queue. It will work fine if you comply to the interface, but in reality your extended code is no longer the original object.

Conclusion: Inheritance is a bad thing.

Flueue2

As mentioned in video 1.3.0, the preferred method of having the convenience of code re-use without a genuine 'is a' relationship is delegation. Richard does this by declaring that Flueue2 is a Flueue2, and not extending queue, thus leading to the "has a" relation (ie Flueue2 has a Queue). This means that no one can use a Flueue2 as a Queue and the tests will have to be different.

See Richard's Flueue2.java and TestFlueue.java

Notes on this program

Before, we called the parent's function with super.add(object). However, in this program, we store the queue within Flueue2, which means that queue is no longer the parent function, but a function stored within Flueue2. You will also need to write out all the original functions within the queue.

The toString function also needs to be passed through from the Queue class otherwise it will use the object toString function (which prints out rubbish).

Back to Design

Design Principles

Good principle from Rupert: Write your code so you can understand it when you're sick. Also on that note: write comments as you're coding it really will help in the long run, especially for the project. Now it seems as though both you and you're partner will remember everything you're doing but writing concise, effective comments as you go will be ideal in the exam.

Don't put in code that you think you will need in the future: You aren't going to need it. This is taken care of by clients and spec changes. Just code what you need now.

To read or borrow:

- Test Driven Development by Kent Beck
  - He uses small bits of code to form a final unique piece of code

Generalisation

When you are writing tests, write something specific to pass it. There is no need for generalisation until you have more than one thing to test.

Thursday lecture

Lock Design task

  - But in this course: don't code things you aren't going to need - don't pre-emptively/defensively
  - To make things easier later on: high quality code with low coupling
  - Roma won't be to anything you've defensively coded. Success will be based on quality of your code.
  - There were three wheels and they had names.
    - Your first design should've been that.
    - Later on they may be more wheels.
  - Everyone who did the array one: "it feels cleaner & nicer". Richard says BULL.
    - Concrete reasons why it might be nicer:
      - An array lets you iterate over things (repetition)
        - Repetition:
          - Creating the wheels (may be easier to create in loop)
          - Linking the wheels (may be easier in a loop)
        - Claim by Richard: named ones are equal in terms of amount of code
      - If you had a super monster master monolithic Egyptian Lock class
  - Better: Wheels only care about itself and whoever it's connected to. No boss.
- Everyone post your solution

Extra content / Admin stuff

- Black-box testing vs. white-box testing & unit testing vs. acceptance testing Two basic philosophies for testing

1. It's given to you in a black box, a box impervious to everything, and all you can see its externalities.

- You can check only its externalities
- ie TV (checking the volume control knob, press buttons, checking as far as you can see is that it works)
- Known as acceptance testing

1. It's give to you in a white box, but the box is transparent and you know everything that you are testing

- You can find out anything about it, all its internal components
- You have extra knowledge and can hypothesise about possible situations

Why each test is good

Black Box testing	**White Box testing**
Testing works even if the drunken guy moves to another room.	Useful for suggesting things that you could try for testing the contract
You can write it before you have finished the product.	Has a higher probability of letting you come across specific instances that could break your product
Tests can be reused for other products which may have different components but achieve the same result	Sometimes written knowing what has gone wrong
Don't make assumptions and just test	

- Visit the planner and use the workspaces.
- What is reflection? See forum.

The below shamelessly stolen from here

Summary on Inheritance

- Heirarchical reuse of behavior and attributes between similar classes.
- Deriving specialized versions of general classes.
- Overriding some of the behavior of a class.

Weaknesses of inheritance

- Inheritance violates abstraction.
    - Code reuse (via inheritance) is intrusive.
    - Polymorphism (via inheritance) is intrusive.
- Using inheritance for code reuse requires knowledge of the implementation details of the parent class; the child class cannot be designed and maintained independently of the parent class.
- Polymorphism is only possible for the child class if the parent class supports it; the parent class cannot be designed and maintained independently of the child class.
- Multiple inheritance is especially problematic due to occasional naming overlaps between the methods or attributes of the parent classes.
- In computer science, polymorphism is a programming language feature that allows values of different data types to be handled using a uniform interface.
- Delegation: When my object uses another object's functionality as is without changing it.
- Aggregation: My object consists of other objects which can live even after my object is destroyed.

Prefer delegation to inheritance

- Delegation supports abstraction.
    - Code reuse (via delegation) is non-instrusive.
    - Polymorphism (via delegated interfaces) is non-intrusive.
- Inheritance is falling out of favor, much as goto statements did in earlier decades.
- An alternative to inheritance is for the child class to have an instance of the parent class as a data member and to access the behavior and attributes of the parent class through the data member (via the public interface of the parent class).
- This changes the relationship between the child class and the parent class from "is-a" to "has-a", from generalization to composition.
- The design approach for this, delegation, helps make composition as powerful for code reuse as inheritance.
- Delegation is more flexible; you can delegate to a different object at run-time, whereas the inheritance of a class is fixed at compile-time.
- Interfaces, combined with delegation, make it easier to avoid inheritance.
- The child class implements the public interface of the parent class.
- The relevant method calls are delegated to the parent object.
- In this way, the instance of the child class can be used as if it were an instance of the parent class (which is one of the benefits of inheritance).
- Multiple interfaces can be used in place of multiple inheritance.
- Polymorphism is more natural with interface delegation than with inheritance; rather

Polymorphism

- (via inheritance) An object decides what method to apply to itself at run-time depending on its place in the inheritance hierarchy.
- (via delegation) One or more objects are accessed using the same public interface. The objects are of different classes or change to different classes over time, causing them to respond differently to the same actions.

OO Design

Nathan's Super-Happy-Fun-Time work list:
Here's a list of all the bolded points. Put your name next to a point to shotgun it. Ideally everyone in the tute should be in this table. If you think something is missing, add it to the table with your name. Also if you think a topic needs further development, add your name to the Done by as well.

I've added something new. The bounty bit here lists how difficult I think a certain topic is to cover properly. The next time we meet, I'll add up your bounty figures and the three people with the highest bounties each get an awesome edible gift. Candy in different shapes and sizes will be given for the next five people. If you edit a topic and can show why it is now better because of your edit, you get to share the bounty with the original author. If you write really well, you don't have to share with anybody. If you have any problems with the bounty figures, if you think they are too high or too low for the amount of work needed, I'll be happy to change them.

Remember to send me the text that you write to help make my life easier in working out the quality of writing. I always read my emails (even if I don't respond to them).

Topic	Notes	Done by	Bounty
Different Design Philosophies	Why shouldn't we be limited to one/Pick one and why it is suitiable to solve a problem		3.0
Feature creep and 2nd project syndrome	Problems caused with constantly adding things to your projects		3.0
Agile development	Links with the above point. How does it help/Does it help?		2.0
Refactoring: Developing further on existing classes	Why is this useful for us?		2.0
Forms of refactoring	Link to wiki with refactoring examples		1.0
Vectors	Code explaining how to use vectors	Steven Lou	3.0
Improving design	Eliminating magic numbers/Refactoring customer at 26:30 (2.0.0)		2.0
Spec changes: Be like water	Code so you never have to worry about rewrites 33:00 (2.0.0)	Albert Wang	2.0
Enumeration	Why is it there and why is it cool/Is this good style or bad style?	Patrick Wong & Febrianto Halim	4.0
Class coupling	If its not needed there, move it	Patrick Wong	2.0

Law of demeter or principle of least knowledge	Look at wikipedia on this, comment, reflect and post	Lasath	4.0
Explaining our focus on code	We focus on quality code that is easy to change and understand. Is this the most important design goal?		3.0
Abstract classes	Why are they useful and how do they affect our designs?/Code examples	Ryan	5.0
Code smells	What reeks of a needed refactoring? Give examples (switch case for instance)		4.0
Long functions are a warning flag	Long methods should be seperate smaller methods	Albert Wang	4.0
The state approach	Having objects change from one to another via attributes. Which design patterns does this conflict with?	Ryan	3.0
Switch cases	Should there be a refactoring?	Ryan	3.0
Set interfaces	Why many classes with the same interface is a bad idea. Give a code example if possible		5.0
Iterators	Sample code with an iterator. Try making the code as succinct as possible with plenty of notes		3.0
Final	Why it is used, its edge cases and what can trip programmers up	Xavier	3.0
Overloading methods	How to overload a method	Jacky & Xavier	3.0
Overriding methods	List of default functions that are good to overload for custom classes/Why they are needed	Jacky & Xavier	3.0
General editing and cleanup	Making this page pretty is just as important as its content	Robert Cen	3.0

Copy pasta'd Nathan's basic layout, everyone (Wednesday Bell 3-6pm tute) add/change their part as you finish them please (for the cohort if not for yourself =D):

2911-2.0.0
How design shouldn't be limited to one philosophy 3:00

- Different design philosophies

The gradual increase in work over time due to finding requirements 9:00

- Feature creep and 2nd project syndrome

Agile solutions 11:00 Further emphasis on that our final uses our existing project. Well written and designed projects lead to easier final exams, not because of projects that support the extra features that have not been asked to support, but instead if your project is well designed, it is much easier to modify.

Programming for tomorrow does not mean putting in extra features now, however it means writing well designed and flexible code that can be easily modified for changes. 12:00

Refactoring by Martin Fowler 15:00

feature, you need to make sure the code is correct by passing unit tests, however the incremental addition of the feature can be done in a messy way at first just to ensure correctness. Then you refactor it to make it a lot nicer. Never add all your features and then refactor everything as this is a nightmare. Always take baby steps where you add a small feature and then refactor it.

- Is there a Richard Buckland booklist?

Vectors 22:00

- Kin of the ArrayList, but used for stacks and queues
- Some sample code using a vector would be nice

Vector

Vector class is in java.util package of java. Vector is dynamic array which can grow automatically according to the required need. ArrayList and vector are similar but with two main differences. Vector is synchronised and allowed to optimize storage management, while ArrayList does not have these properties.

There are four constructors:

```
Vector()
Vector(Collection c)
Vector(int initialCapacity)
Vector(int initialCapacity, int capacityIncrement)
```

The initial capacity is 10 if there is not a specified one. Capacity will be incremented with a specified number when it is needed. If the capacityIncrement is not specified, vector defaults to doubling the size of its array.

Example of vector usage

Improving the design 26:30

- Eliminate magic numbers
- Refactoring customer to remove the math

For OO, very long functions are a warning flag 30:00

- Long methods should be smaller separate methods

Spec changes: Be like water (33:00)

When a spec changes and asks for additional or shifted requirements, there are increased risks of code duplication and messy changes. The problems presented by code duplication are an increased chance of bugs occurring and also duplicating the work load as new features are introduced. The idea behind refactoring code into a very beautiful form is to enhance the adaptability of your code. Generally the best method to go about it is to refactor your code as you write it.

Evidently we'll always be facing potential spec changes and additional feature requests (*looks at Richard*) so we must learn to deal with these situations. Ideally the way to handle this possible cases of how our specs can change. If we fall into this trap we can easily end up consuming all our time trying to make our code so flexible that it becomes unnecessarily bulky (arrays for two of the same element - an example would be the Lock design task, you may argue we should use an array so that we can handle x number of wheels, but its not sensible to use an array for 3 objects).

Classic example of class coupling 38:00

- If its not needed there, move it.
- This is a form of refactoring (moving method). The code shown on the lecture slide constantly accessed things from the rental class despite the method being inside the customer class. As such, it isn't needed in the customer class and should be moved into the rental class to group similar things together under one class. By doing this, code becomes much more simpler.

```
class Customer {
...
   private double amountFor(Rental aRental) {
      double result = 0;
      switch (aRental.getMovie().getPriceCode()) {
         case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2) {
               result += (aRentalgetDaysRented() - 2) * 1.5;
            }
            break;
         case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
         case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3) {
               result += (aRentalgetDaysRented() - 3) * 1.5;
            }
            break;
      }
      return result;
   }
...
}

inside Rental class:

double getCharge() {
   double result = 0;
   switch (getMovie().getPriceCode()) {
      case Movie.REGULAR:
         result += 2;
         if (getDaysRented() > 2) {
            result += (getDaysRented() - 2) * 1.5;
         }
         break;
      case Movie.NEW_RELEASE:
         result += getDaysRented() * 3;
         break;
      case Movie.CHILDRENS:
         result += 1.5;
         if (getDaysRented() > 3) {
            result += (getDaysRented() - 3) * 1.5;
   return result;
}
```

Enumeration 42:00

- Why is it there and what makes it cool?

Extract from Wikipedia: An enumerated type is a data type consisting of a set of named values called elements, members or enumerators of the type. For example, the four suits in a deck of playing cards may be four enumerators named CLUB, DIAMOND, HEART, SPADE, belonging to an enumerated type named suit.

```
 enum Cardsuit { CLUBS, DIAMONDS, SPADES, HEARTS };
```

The enumerator names are usually identifiers that behave as constants in the language.

- Is this an example of good or bad programming magic?

A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value.

```
 if (Cardsuit == DIAMONDS) {
    // do something
 }
```

In other words, an enumerated type has values which are different from each other, and which can be compared and assigned, but which do not have any particular concrete representation in the computer's memory; compilers and interpreters can represent them arbitrarily.

Can use enum - say, Cardsuit - instead of using arbitrary constants such as:

```
private final static int DIAMONDS = 0;
private final static int CLUBS = 1;
private final static int HEARTS = 2;
private final static int SPADES = 3;
```

Over-optimisation - What needs improvement is rarely where you expect 46:00

- Good examples of over-optimised code
- Write goodly first, correct for speed later
- HARDMODE: Profiling and why it's for attractive, cool people

2911-2.0.1

Law of Demeter or Principle of least knowledge 00:00

- Each class should know only enough to communicate with its partners
- No class should communicate with objects outside of its intended function
- More detail on the wiki page for Principle of Least Knowledge

The Law of Demeter (LoD) or Principle of Least Knowledge is a design guideline for developing software, particularly object-oriented programs. In its general form, the LoD is a specific case of loose coupling. The guideline was invented at Northeastern University towards the end of 1987, and can be succinctly summarized in one of the following ways:

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.

Only rule is to make your code easy to change and easy to understand 7:00 Reducing duplication in code 12:00

- Is this the most important design goal?

Abstract classes 14:20

- Why is it useful and how does it affect design?
- Code explaining how to use it

Switch cases - Should there be a refactoring? 18:30

- Not flexible enough to deal with a change to types of movies 22:00
- This makes sense to have a section on common areas in code that call for a refactoring.

The state approach 32:20

- Having object change from one type to another via an attribute

Uniform access principle 40:00

- Even inside a class, attempt to abstract away attibutes from being modifiable by internal functions if not needed

2911-2.0.2
Set interface 2:30

- The good, bad and the ugly of having many classes share the same interface. This is a recurring theme over many of Richards lectures

HashSet 5:30 Iterators 8:30

- Sample code with an iterator
- HARDMODE: Make its use clear with as few lines as possible. Shortest code is best code.

Iterators (and the for each loop)

Collection objects implement the Iterable interface, allowing for their elements to be iterated on.
The code below demonstrates an explicit use of an Iterator object.

```
import java.util.Set;
import java.util.Iterator;

public static void printPhoneList(Set<Phone> phones) {
    Iterator<Phone> all = phones.iterator();
    while (all.hasNext()) {
        Phone current = all.next();
        System.out.println("number: " + current.getNumber());
    }
}
```

An Iterator object does not need to be explicitly created for the Collection you want to iterate on, as shown below.
This noticeably cleaner code is equivalent to the code above, and also introduces the usage of a for each loop.

```
public static void printPhoneList(Set<Phone> phones) {
    for (Phone p : phones) {
        System.out.println("number: " + p.getNumber());
    }
}
```

A for each loop iterates on the elements of the object on the right of the semi-colon.
The above code thus means: "For each Phone p in phones, print "number: " + p.getNumber()"
Richard also notes that a for each loop can be used to iterate on the elements in an array, which is useful because arrays do not implement the Iterable interface.

Its strength is its weakness: The problem with objects 20:00

- Make sure that you're not modifying a reference you don't expect

More uses of final 28:00

Final means a different thing depending on its context:

- final class: if the final keyword is placed before a class declaration, then no other class can extend from the final class.
- final method: if the final keyword is used as part of a method's type signature, the method cannot be overriden by any subclass. A common use of this is to prevent logic critical to the operation of a class being overridden by a subclass.
- final variables: if the final keyword is used in a variable's declaration, the variable may only be assigned to once (the variable doesn't have to be assigned immediately on declaration). The main use for this is to create constants.

Overloading methods

Overloading method is type of polymorphism which is called Ad-Hoc Polymorphism. It means you can use the same function name for a group of similar functions which have different signatures (different parameters). For example we have a class called calculator and it has a method called add to add two numbers together:

```
public class Calculator {

   public int add(int a, int b) {
      return a + b;
   }
}
```

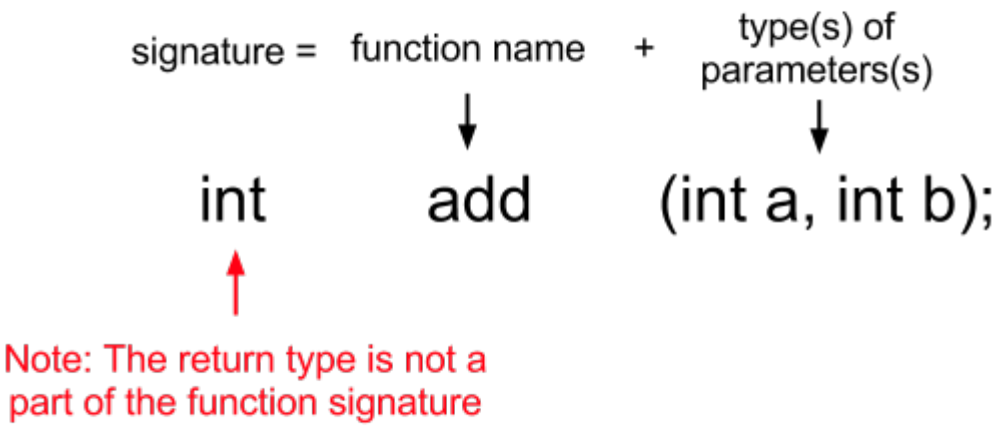
the problem of this calculator is it can’t calculate floating point numbers, so we can overload the add method to handle the floating point adding by doing:

```
public class Calculator {

   public int add(int a, int b) {
      return a + b;
   }

   public float add(float a, float b) {
      return a + b;
   }
```

name. In java, functions are differentiated by signatures. A function signature is illustrated by the following graph:


signature = function name + type(s) of parameters(s)
int add (int a, int b);
Note: The return type is not a part of the function signature

So a function signature only consists of the function name and the types of the parameters, so overloaded methods are different because they have different parameter types. Notice, the function signature doesn't include the return type so these two function are considered by the Java compiler that they are the same functions and it would refuse the compile:

```
public class Calculator {

   public int add(int a, int b) {
      return a + b;
   }

   public float add(int a, int b) {
      return (float) a + b;
   }

}
```

Why we need overload methods? Because that would allow us to use the same function name instead of come up with different, hard to memorize name like addInt addFloat and so on and we can just use add for adding all the data types. With the power of function overloading (or Ad-Hoc polymorphism), we can now do something neat like this:

```
public static void main(String[] args) {
   Calculator cal = new Calculator();
   cal.add(1,2);
   cal.add(2.3,2.2); // different types by same function
}
```

Overriding Method:

Use Annotation (@Override) right before your new function to inform the compiler this function is overriding its parent function. It's not necessary to use annotation but it's a good practice to use it because after u used annotation, the compiler can check if you are overriding the method correctly, it also inform the other programmer who is reading your code that you are overwriting a function.

[illegible]

This is the parent class of every other class in java and it provides concrete implementation for this two methods.

The source code of the original equals function is this:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

So it compares the reference (i.e. Address/Pointer) of these two Objects. So if you want your class to compare the data inside the class rather than the reference, then you need to override this function.

In the example in the lecture video, HashSet differentiate objects by calling equal functions on them and it also compares the hashCode of those objects and that's why Richard changes the equal function as well as the hashCode function in order to avoid duplicated instances are put into the Set.

Design Patterns

Lecture 2.1.0

- We'll be using SWING
- GUI: Graphical User Interface – Eg. Microsoft Windows
- Back in the day, it used to be text-based
- WIMP: Windows – Icons – Mouse – Pull down menus
    - Disadvantages: The user having to have a mental image of everything in his mind, remembering the state of everything, remember what the current directory is, what shortcut variables that were defined were. Have to process the information in terms of a mental imagery. This is not accessible to everybody.
- **GUI** developed by a picture being seen by the user which represented/corresponded with the mental image of the user. The picture maps our imagination. Very convenient to the user, because what the user is interacting with, is very close to what it really is in the back-end. In other words, the actions of the user correspond with what is happening behind the image/model.
    - Advantageous for complicated problems – if it is represented by modelling it and the components of the problem correspond to our mental image of how it works, then our intuition helps us with understanding the complexity of the problem, designing and building it. Hence, leading to a better understanding of things.
- OO allows us to physically model things.
- GUI allows us to view the underlying data structures.
- This is represented on the screen through what is called a "widget".
- It is better not to have the GUI and the back-end *tightly coupled* i.e. to bind each button to an action. It is far better to have a messenger between the logic and interface so that when one is modified, the other can remain unchanged. Encapsulation allows for this.
- Polling is where areas ask if resources are free to be used. another method is to instead divide slices of time out to each action making a request.

concurrency problem

- If each task is done one by one, we can easily make Que of waiting list, if hundreds of

Lecture 2.1.1

MVC

- MVC – Model, View, Controller. A design pattern often used for GUIs.

- GUIs have aspects:
    - Display
    - User input
    - Presentation logic
    - Domain logic
    - Data

- The main goal is to seperate the View from the Logic.

- MVC has three components
    - Model: holds the data and domain logic
    - View: manages the user's view of the system. Needs a copy of the data being displayed. It is an observer of the model, and so gets notified when certain things about the model change.
    - Controller: listens for changes in the view (i.e. user input) and processes those inputs and stores the data in the model.

- Richards's idea of what a design pattern is: a useful approach for solving a problem. Don't try to work out exactly what a design pattern means, just evaluate whether it will be useful for the task at hand. (Bush walk analogy)
    - Example: MVC is just three letters. There are many variations and you could nit-pick about exactly how the three parts should interact in "true MVC". Ignore this, just work out the best way to apply MVC to your problem. Know the disadvantages and advantages.

- Example of design patterns: stage lights. Richard and his friend had a particular way of lighting the stage with the limited resources of their poor school. Richard's friend went to the "seymour centre", where the lighting technician was lazy and simply used hundreds of lights (more than Richard and his friend had) in a way that was quick to set up and required no effort. Richard and his friend laughed about this because the guy was lazy.Another example given, three tier architecture that is 3 layers GUI, Object and Database layer, Gui Draws the screen and handles that, Object deals with run-time and structural values while the database layer deals with persistency.
    - They later came accross a situation where the "seymour centre" method was ideal, and they already had the language to describe what they needed. Eventually the two of them were able to talk to each other about "lighting design patterns" because they'd seen them before and understood their advantages and disadvantages. They had a vocabulary to describe design patterns
- Challenge: research the Three-tier architecture. Work out where the five GUI aspects fit (sounds like a good exam question)

Packages

- When you're developing in a small team, you know how all your classes interact, although you might not know how they work internally (encapsulation).
    - This is encapsulation at the class-level.

```
package mySup;
class Sup {
        // class stuff
}
```

- This is encapsulation at the package level. Only code within the mySup package can create a Sup object or and access Sup's fields. This allows other developers to use the functionality of your package (public classes) without knowing about all the inner workings of your package (default, protected, and private classes). Such classes are sometimes called 'helper' classes.

- In Eclipse, you can drag classes into a package and it will change the package identifier at the top automatically!

GUI and swing

- Swing: Java does everything, doesn't trust the OS with anything. AWT was the other way: support a lowest common denominator and just reduce everything to OS calls. AWT is the heavyweight way, Swing uses the lightweight way. Swing is great for portability

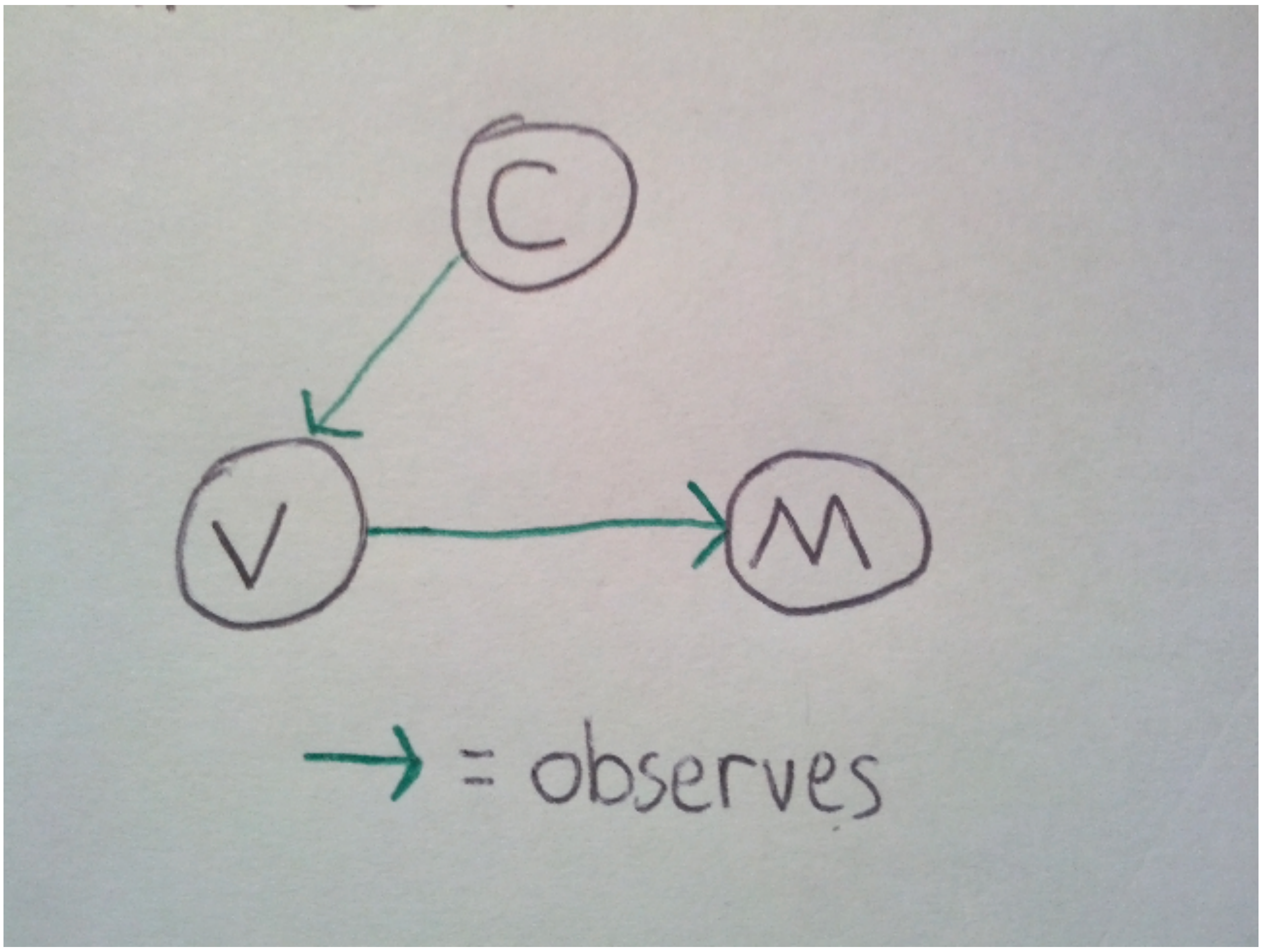
- Swing works by overriding: e.g. BasicWindow extends JFrame. (see code)

- Duplication: you have data on the screen (the widget has to know the data), and the model also has data.

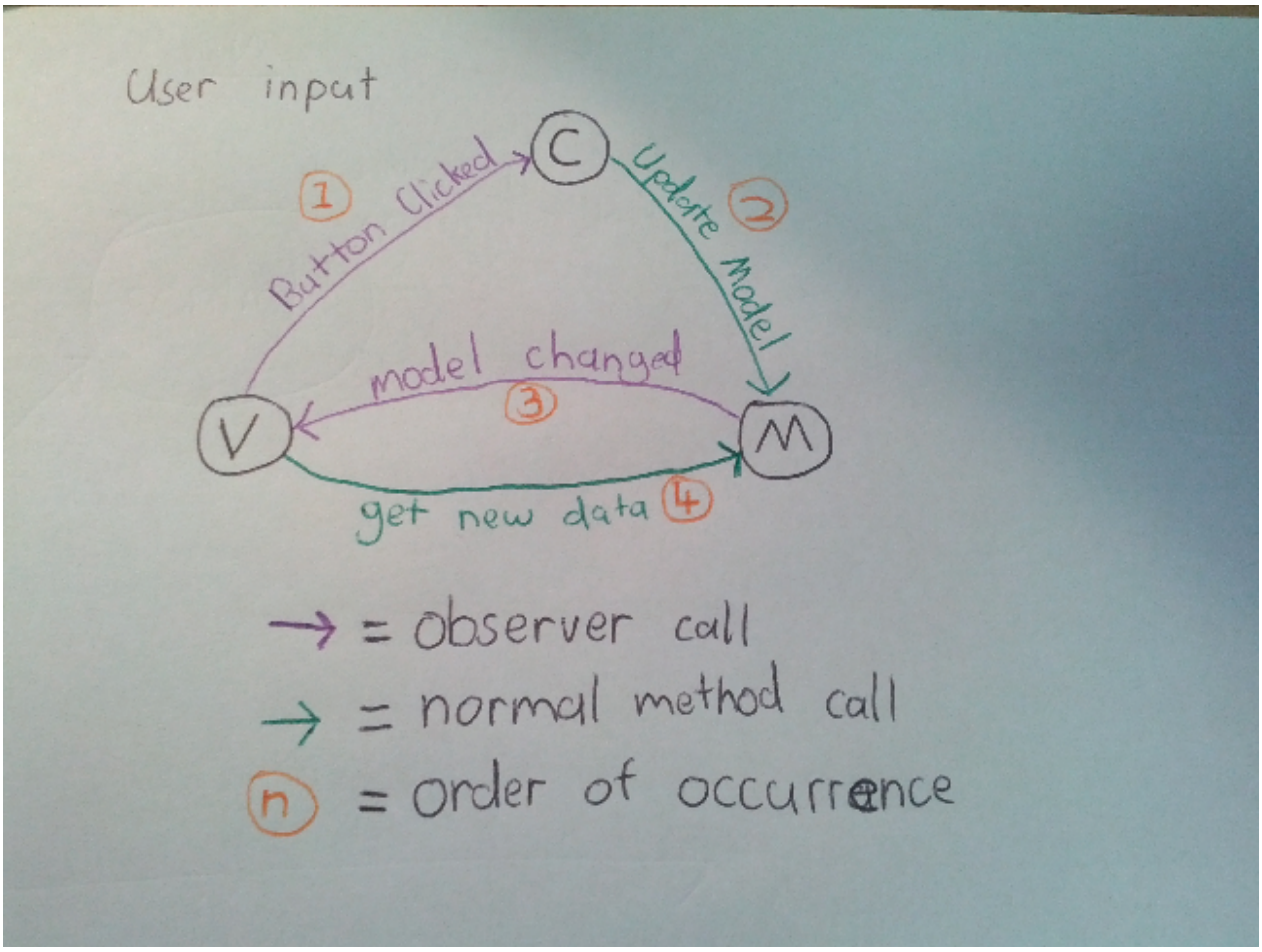
- ListenerGUI: has an underlying list model and a list widget.
    - Aside: put all your widgets into a panel. Even the default layout manager of a panel will automatically position the widgets so you don't have to think about it until you want to clean up your program layout.

- Richard uses his controller (ListUpdater listener;) as the event listener for the button. The button then updates the model (DefaultListModel model;). He glues all of this together in a class called ListenerGUI (the View). See diagrams:

How the example is set up



What happens when you click the button



- **ListUpdater**: The controller. Notice that it implements ActionListner and thus has the void actionPerformed(ActionEvent) method.
  - It contains a reference to the model (a DefaultListModel) which it can update when needed.
  - When the model is updated, it lets the view (ListenerGUI) know because ListenerGUI is an observer of the model.

Lecture 2.1.2: Design Patterns

Strategy for solving any problem: factor X is causing a problem. How can X be used in such a way that it is a good thing?

- Richard used to teach classes of ~20. This was good but later on he had to teach classes of hundreds. This was bad because he couldn't help all of them and they were all at different levels and there was simply too much work for a single teacher to do. So his strategy was to use the student cohort to teach each other, e.g. asking questions, asking students to explain something to the class, having the wiki where students can help each other out. Now it's actually better.

- Maths has design patterns, e.g. proofs, proof by contradiction, proof by induction.
- Category theory: basically design patterns formalised for maths. One way of solving a mathematical problem can be abstracted and used to solve problems from other fields

- The view gets all its data from the model (using getters).
- The model is passive, except when it changes, when it notifies the view of changes to the data (through an interface).

Revising the code from the previous lecture

- Richard points otu that due to all of the argument and diff version of MVC it really is not a design patent
- With MVC, the important distinction is between the model <-> (view, controller).
- Richard is suggesting to put as much of the UI/presentation logic as possible into the controller (and avoid putting them into view), because the view is quite difficult to test.
    - Things which are hard to test (the view) should have the least amount of functionality in them as possible.

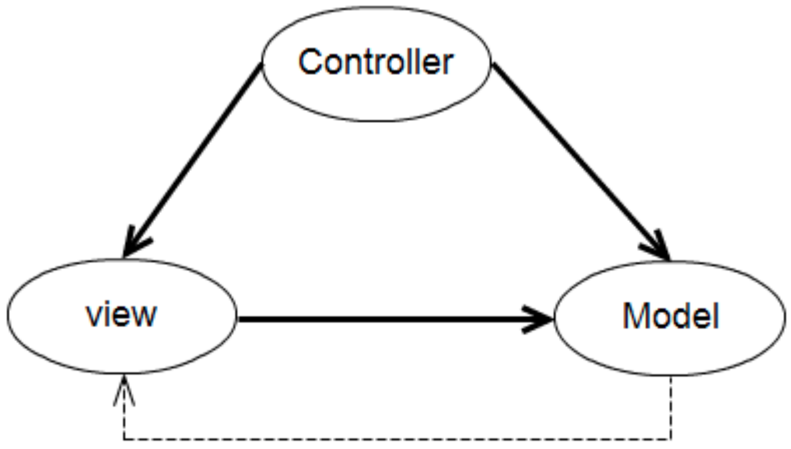
The observer pattern

pg 293 of the GO4 book.

- Starts with the intent, then describes how to use it.
- Consequences of the observer pattern. The subject doesn't need to know the types of the observers, or how many observers exist.
- Disadvantages
    - unexpected updates. A seemingly inconsequential data change in the subject can cause a huge cascade of updates in the observers, and if the relationships aren't well defined, it is difficult to reason about what consequence a particular change in the subject will have
    - There is no protocol for defining what has changed in the subject, so observers can be needlessly notified about irrelevant changes. In java we can pass a whole object as part of the notification to give information about the change.
    - Open to memory leaks because the garbage collector won't collect objects that were observing when they were destroyed.

The observer design pattern

- Observer relationship: When data in the 'model' have changed, it sends notifications to the 'view' to say it has changed. In other words "the controller watches the view". They don't know about each other, however, they behave as though they know about each other, in a way that when one gets modified, the other reflects those modifications.
- Whenever an action is performed, the controller acts appropriately.



notifications.
    - They are not tightly coupled: they do not mess around with each other’s internal logic; hence, they can be substituted and changed easily.

- Used in the following situations:

1. When an abstraction has two aspects: encapsulating these aspects in separate ways that you vary and reuse them independently of each other.
2. When a change to one object requires changing others and you don’t know how many objects need to be changed.
3. When an object should be able to notify other objects without making assumptions about who these objects are.

**** Advantages:**

- Abstract coupling between the subject and the observer – all the subject knows about is that it has a list of observers. The subject doesn’t know the concrete class of any observer.
- Freedom to add and remove observers at any time.

**** Disadvanages:**

- Unexpected updates: observers have no knowledge of each other’s presence; they can be blind to the ultimate cost of changing the subject. Updates which are hard to track down.

Lecture 2.1.3

Joshua Block

- Chief Java architect at Google, and previously distinguished Engineer at Sun. Strong advocate of Java.

Books that (he believes) every programmer should read:

- the GO4 book ('Design Patterns') is on the list.
    - Good examples, common vocabulary. However, its style & language is inconsistent, and it is beginning to show its age.
- 'The Elements of Style' by William Strunk, Jr. and E. B. White
    - Not a programming book, but recommended for us to read for writing reports
- 'The Hacker's Delight' by Henry S. Warren
- 'The Art of Computer Programming' by Donald Knuth
- 'The Elements of Programming Style' by Brian W. Kernighan and P. J. Plauger
    - Uses Fortran (an older procedural language) so its old, but still good
- 'The Mythical Man Month' by Fred Brooks
    - (Apparently it rambles on about God lots)

Programming isn't just about making something that works, but about making something readable, maintaineable & efficient

- cleaner & nicer programs are generally faster (and if not, easier to make faster)
- **"It is easier to optimise correct code than to correct optimised code"**

1. Know the problem that needs solving
2. It's a negotiation, not simply a problem given by the customer **Have use cases, not specification at the start**
3. How to write an API - write the code that uses the API before the code that implements it
    - In fact, write the code that uses the API before the spec is even fully fleshed out - otherwise you may be writing spec for something that is fundamentally broken
    - **This is test-first API design. You're testing your design, not just the implementation**
    - Fundamental theorem of API design - "When in doubt, leave it out!"
    - Disagrees with Fowler: tests are not a substitute for the spec. You need to test the spec using your tests.

Could java be used more? Could all C++ be replaced with Java?

- He's not religious. But believes it's far more efficient to use a modern language.
- Programmer time > computer time generally.
    - (Not necessarily true if running same program on many thousands of machines) Should you use modern tools?

What tools Block uses to program

- Does too much coding without modern tools - uses emacs.
- You should learn a good, modern IDE to do refactoring - it makes it 100000000 times easier
    - People now write cleaner code because they now to refactorings that they wouldn't have attempted before - can count on these tools to propagate changes without changing behaviour of code.

Should students learn a low level language (e.g. C) first and then a high level language?

- <u>Block:</u> No! Learn the high level language first
    - Believes lower-level stuff is indeed important to know, but students shouldn't have to worry about buffer overflows and allocation errors it in their 1st exposure to programming - just pure concepts instead.
- <u>Buckland:</u> Yes! Teach students to be aware of the fundamental workings of a computer and to program defensively from the beginning. Don't teach them to program in a forgiving environment and then spring C on them at the end.

Java was awesome because it was a clean slate. Is it now overrun and bloated? What was the impetus for adding generics?

- They represent a large addition to an already complex system.

Would java be better without generics?

- oh, I dunno.

Was there user pressure for generics?

- No, there wasn't. I'm guilty of putting it in because it was 'neat' and seemed nice.
- Triangulation: have three real use cases before you put something in.
- These days I keep a close eye on the complexity meter.

Two ways to design a system:

Networking

Layers

- Electronics over the wires
- Bits over the interface: error checking etc
- Packets, getting a packet between two computers over an interface
- Organising the flow of packets, routing them. there is no need for each packet in a series to travel the same route.
- Reassemble the series of packets into a connection, like a phone call
    - This is the level we deal with in java. Pretty much a unix pipe over the internet
- sockets - like a pipe between addresses on a network a socket is a way to send data between computers
    - Java has a class for sockets

-each computer has an address which is it's name on the network, and a series of ports, used to accept or send data -localhost is a way to send a message to yourself.

-Port 80 is the default port for HTTP traffic.

HTTP (Hyper Text Transfer Protocol)

- client-server relationship:
    - the client requests information
    - the server provides information
- HTTP allows one request and one response only. if you want two items, you must send two one requests

Request format:
request line
header information
<blank line>
extra data

response format:
response line inc. response code(e.g. 404 for file not found)
optional headers
<blank line>
requested information

proxies

A proxy is a virtual server which forwards requests from clients to external locations, and returns the requested data to the client. It may process the information in some way or another, such as censoring particular words(the Design Task this week)

Fooling on a network

- Web server starts looking for web files from the public html directory
- html file: has a head and body
- When we browse the web, we use addresses that link to html files on servers.
- Http protocol: request is sent and the web server returns a response. We can see the

consisting of initial line, header information, blank line etc. The servers response would include a response line, header information, blank line and the actual html data.

- We can inspect what is going on in the browser with the http request/response using java.
- Talking over http requires a tcp connection, and we are given this in java using sockets.
- To create a socket we need the name of the machine we want to talk to and the port we want to talk with of the server.

- TCP/IP networking has 4 layers of software, and each layer uses the one below it:
  - Link layer - sends packets between adjacent computers (i.e. same lan)
  - Internet layer - sends packets between any two computers (using layer 1 API). Packets may get lost of our of order.
  - Transport layer - send reliable stream of messages between any two computers (using layer 2 API).
  - Application layer - uses sockets to carry out a task e.g. a client (irefox) fetching a web page from a server. Requests and responses.

- Connection end points are called sockets
- HTTP = Hyper Text Transfer Protocol

- Example request:
  - GET/index.html HTTP/1.0
  - HOST: www.cse.unsw.edu.au
  - Accept-Language: en-us,en;q=0.5
  - Accept-Encoding: gzi,deflate
  - [BLANK LINE]
  - [EOF]

- Example response:
  - HTTP/1.1 200 OK
  - Date: Thu, 15 Apr 2010 03:24:11 GMT
  - Content-Type: text/html;charset=UTF-8
  - Content-Length: 29127
  - [BLANK LINE]
  - <html>
  - <head>

- If we send a request for a non-existent file the server returns a "404 Not Found error". If a file exists the server return "200 OK".
- The Content-Type in the response specifies the type of the data in the response and therefore how it should be handled. i.e. html, binary for images etc.
- We can setup a local server on the machine and get Firefox to send requests to it and simulate a response in java. We can speak to the server we setup on our machine using the following address: http://localhost.
  - an example request is http://localhost:2911/~cs2911/0s1/demo.html

- Proxy - if we try to Google something from the labs at uni, Firefox sends the message to a local machine at CSE which will then forward it to Google and the response comes back through that machine to us. **e.g. proxy is www-proxy.cse.unsw.edu.au**

g04 design patterns

Intent

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

AlsoKnown As

Kit

Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feelspecific classes of widgets throughout the application makes it hard to change the look and feel later. We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.

Diagram here

There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate widget for the look and feel. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetFactory returns a scroll bar for Presentation Manager. Clients create widgets solely through the WidgetFactory interface and have no knowledge of the classes that implement widgets for a particular look and feel. In other words, clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

A WidgetFactory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a MotifWidgetFactory.

Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint. l you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

==== structure ====

Participants

- AbstractFactory (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- AbstractProduct (Window, ScrollBar)
  - declares an interface for a type of product object.
- ConcreteProduct (MotifWindow, MotifScrollBar)

Refactoring

REFACTORING - WEEK 2.2

Definition : Code refactoring is a disciplined technique used to restructure an existing body of code by altering its internal structure without changing its external behaviour.

Lecture 2.2.0

Code Writing, What are our objectives?

- Aim
  - Speculate the problem and what is required to achieve a solution
  - Focus has changed from just achieving a solution to a problem to design and development of a solution to a problem

- Correctness
  - Solution which solves the problem both now, and in the future
    - Clear and simple solution
    - Low coupling
    - Reusability of the solution
    - No duplication of methods
    - Size and number of methods

- Easy to write, if a program is easy to write, we can write it quickly and therefore get more done in the timeframe
  - Easy to write now
  - Easy to write in the future, programs need to deal well with changes
  - Intermediate factors which lead to easy to write programs
    - Simplicity, an example is to have short methods
    - No duplication
      - Duplication can lead to problems with updates as the code needs to be changed in more than one place
      - Clarity, it is hard to maintain correctness of unclear code

- Refactoring
  - A program is never completely finished, instead it is constantly being changed/fixed
  - Restructuring only the internal structure of the solution
  - Improving the program’s quality but also keeping its correctness
  - Refactoring Plan
    - Apply small changes one at a time

        - 'Borrowing' from the future, debt due to poor design will have to be payed back eventually
        - Innovate present solutions with ideas from the future
        - Solutions which don't include ideas for the future will render the present solution useless which results in wasting valuable resources
        - Present solutions which incorporates ideas from the future will reduce the loss in valuable resources

- Review on Inheritance and OO Design
    - OO design reduces the length and complexity of a solution (if statements from C)
    - OO also allows delegation of a solution which produces more simple and clean code
    - OO removes the disadvantages of a pyramid structured solution
    - Dynamic binding results in methods being determined during run time
    - Overriding
        - Allows duplicate methods with same type signatures but offer 2 different classes due to inheritance
        - Child methods replaces the parent methods
    - Overloading
        - Allows multiple methods to be created with the same name, i.e. the same method name referes to multiple instances of that method. Methods must have a different parameter list
        - Should not be confused with forms of polymorphism (where the correct method will be chosen at runtime rather than statically)
        - Usually bad practice but useful and common for constructor methods
    - Shadowing
        - Like overriding for fields
        - Parent gives its methods a name and the child gives its method a name
        - If a child and its parent class both have a field sharing a common name, typecasting a child to a parent will return the parents field which is different to overloading (shadowing is dependent on type)
        - Shadowing shadows the outer class

Lecture 2.2.1

- Review on Casting Objects
    - Casting a child to a parent is possible and vice versa but is dangerous for the latter.
    - Child objects can be casted back to its parent without any problems
    - Avoid casting objects downwards i.e. a parent as a child as it causes undefined behaviour
- Testing
    - Tests are done by testing down from low level
    - Unit testing for a proxy server is hard but not impossible
    - Tight interaction between a program and another system makes it hard to test
    - Testing a program more than once will ensure correctness of code
- Mocking
    - Mocking is the implementation of an interface between two tightly coupled object
    - Allows the programmer to implement a mock class to substitute for one of the objects (mock objects mimics the behaviour of an object)
    - Mock objects allow the programmer to gather data on the program as well as tests
    - Mock objects allow the programmer to test the behaviour of another object
    - Mock objects are useful when a real object is impractical or impossible to incorporate into a unit test (hint for this week's design task)

    - In a unit test, mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:
        - supplies non-deterministic results (e.g. the current time or the current temperature);
        - has states that are difficult to create or reproduce (e.g. a network error);
        - is slow (e.g. a complete database, which would have to be initialized before the test);
        - does not yet exist or may change behavior;
        - would have to include information and methods exclusively for testing purposes (and not for its actual task).
    - For example, an alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the test must wait until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.
- Dependency Inversion Principle
    - Design of code is normally a top down design
    - Lower level code should not be dependent on higher level code or vice versa
    - Abstractions should not be dependent on details rather details be dependent on abstractions
    - This is due to fact that changes are risky, and by depending on a concept instead of on an implementation, you reduce the need for change at call sites.
    - DIP reduces coupling between different pieces of code.
    - Interfacing is important
    - Dependency inversion abstraction allows decoupling of tightly coupled objects
    - Lower level code should only dependent on higher level interfaces
    - Only use the dependency inversion principle if it is required
- Dependency Injection
    - Dependency injection is a software design pattern that allows a choice of component to be made at run-time rather than compile time. This can be used, for example, as a simple way to load plugins dynamically or to choose mock objects in test environments vs. real objects in production environments.
    - Dependency injection allows a choice of components to be made at runtime rather than compile time
    - Dependency injection is useful when mocking objects
    - It should never be the top level objects responsibility to decide on which object it should choose to implement its interface
    - Injection should be applied when constructing top level objects
    - Getters and setters will perform a similar job to injection Lecture 2.2.3

- Incremental Changes
    - Incremental changes are good on many levels however multiple incremental changes such as refactoring can lead to code smells
    - When implementing incremental changes such as refactoring avoid creating code and design smells
- Observer Pattern
    - Observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state change

automatically

- Code Smells
  - Code smells usually refer code which looks incorrect, brittle and which are hard to test (which normally occur with long methods or a lazy class)
  - Code smells occur due to:
    - Too many parameters: a long list of parameters in a procedure or function make readability and code quality worse.
    - Feature envy: a class that uses methods of another class excessively.
    - Inappropriate intimacy: a class that has dependencies on implementation details of another class.
    - Code being protected since the user will not be able to see what a class is doing (the class, package and subclasses can access a certain variable but the world cannot see it)
    - Long methods also cause code smells because long methods become untidy and normally destroys the aim of OO programming
    - Duplicated code may also cause code smells since OO programming should reduce duplicated methods and can normally be assigned to a class to avoid duplication
    - Static fields can also cause code smells since static fields are associated with classes and only allows the values to be shared amongst all objects of the same class type
    - Large classes defeats the purpose of OO programming as it forces a certain class to do more work than it is required and normally will result in duplicate code which makes the functionality of the class ambiguous
- Design Smells
  - Design smells refers to when the design or implementation of the program affects the layout of the program
  - Design smells results in a program which is hard to test
  - Design Smells occur due to :
    - Ambiguous names for classes also cause design smells as it makes the functionality of the class ambiguous and any later changes that may be applied to the class or the program itself will be hard due to the ambiguous naming of its classes
    - Lazy classes since classes should have defined functionality rather than a class that does either too much or too little
    - Conditional complexity cause a lot of problems and is often a sign of design smell as it generally increases the complexity of the program and will normally induce duplicated code
    - Switch statements are also a sign of design smell even though there are many advantages of using a switch statement, it is also the main cause of program bugs
    - Public attributes are also a case of design smell as it allows the world to see the attribute and, will cause problems for the methods and classes that relies on the public attribute Plan - Lem

The following is how I'm planning to layout this Lecture content. As it is an interview I've noticed there are quite a few interesting points raised by Vickery but kinda all over the place (as expected when people recall thoughts and ideas). So I've tried to rearrange them into the following: Richard Vickery short profile + possible link to any site with more information about him Vickery's personal preferences for games and therefore potential bias that said we can generally respect his opinion because he is the Chair of Board Games Australia Aspects of games defined as GOOD Aspects of games defined as BAD How does Vickery know what's

Interview with Game Designer Richard Vickery

Richard Vickery

Scientist, Researcher, Academic

Chair of Board Games Australia http://www.boardgamesaustralia.org.au/bga-committee (link is down though it seems)

Lem’s Rough Notes

Lem: I’m just putting my rough notes here, I’ll refine it if I get round to it :s

Design Interview with Richard Vickery

Design and Board Games

Scientist

Researcher

Academic

Chair of Board Games Australia

Lobbying for board games everywhere!

Design ideas and analysis: how to tell what's a good design and what's a bad design

Analysing design in other disciplines to see if there are common attributes

In this case we're looking at game design

Vickery's preferences

<u>Go:</u>

- enjoyable and simple
- a quality that makes it feel like it "MUST exist"
- feels like a game that was discovered rather than invented

<u>Dune:</u>

- scheming and backstabbing
- representing of different factions
- win by predicting when someone else is going to win

- mind games are fun

Tigris and Euphrates http://boardgamegeek.com/boardgame/42/tigris-euphrates

- elegant?

Trends or Changes

- Shorter playing times
- Simpler components
- less calculations

Design Principles: What makes a game good? Bad?

<u>Bad:</u>

- don't hit game objective is bad
    - "game will take 20 minutes but takes all day is a disaster"
- broken
    - game that doesn't end
        - gap in the rules
        - unclear game end conditions
            - didn't do enough blind play testing
            - didn't test it against people who didn't have inside knowledge
            - too much white box testing not enough black box testing
        - contradicting game rules
- too complex - concept too hard
    - required simplification

<u>Good:</u>

- flexibility
- able to support many players
- able to support variable players
- good aesthetics - some people like nice looking stuff
- replay value

Movie: watch it once and be happy, give it a good rating and never watch it again

A game has to make you want to play it again and again without stop

- Needs to provide a different experience
    - versus a puzzle, do it once, doing it again is the same thing
    - Puzzle may still work given that there are many variations and creating the puzzle isn't hard, but it is enjoyable
- randomness - dice
- board can be different each time
- asymmetry - eg different factions with different gameplay objectives
- helps if the initial state is different or diverges very fast

<u>Bad:</u>

- too fiddly
    - game is meant to be enjoyable
    - effort > reward
    - tracking states of the game etc

<u>Monopoly</u>

- [illegible]

        - Unbalanced as any small mistake at the beginning of the game heavily reflects on ending performance
    - winning and start winning very crazily
        - Generally only the people that win enjoy it
    - asymmetry joy from the game
        - pleasure from causing others pain
    - very materialistic

Good:

- even if you're losing you still feel good
    - Settlers of Catan http://en.wikipedia.org/wiki/The_Settlers_of_Catan
    - always moving forward
        - always building villages/cities
    - always very close until the end
        - ”In the hunt right until the end"
        - The game is balanced due to randomness, though skill does give an advantage
    - games with hidden victory conditions
    - games with a story
        - winning isn't the main objective of the game
    - survival games where you know you're going to die/lose eventually
- The enjoyment someone gets from a game is more important than the goal of winning

Versus Monopoly

- start moving backwards when you start selling stuff

Monopoly is simple so people have a fond memory of it

Design smells

aren't always bad but you try to avoid them

- anything too visually familiar aren't as interesting
    - eg anything that looks like monopoly or rhymes with an existing game
    - indicates that the maker hasn't experimented with many games
- dice are a smell
    - games with lots of dice are generally ok
    - but games with one or two dice aren't
    - reason: if a game is boring you can put dice in to introduce randomness and make it more exciting
- special powers
    - it's clear that the base game is very dry and that the special power is just applied over the top
    - the "special power" is more of a patch rather than fixing the base design issues

Vickery's Roma I opinion:

- released with a design flaw - the merchant was too over powered
- forums feel overpowered
- game is unbalanced
- icons were very annoying on the cards - descriptions were better

- extra dice disc slot
    - can be activated with any dice
    - have to pay the dice value in money

Why don't people realise their own bad design?

- people design games when they don't play games
    - writing a novel when you've only read "Harry Potter"
    - lacking experience
    - people get stuck into an idea, and have difficulty seeing other variations

- standing on giant's shoulders
    - reuse of good ideas
    - avoiding clearly failed ideas or smells where possible

- games tell a story in mathematics versus stories that are told in words
    - you're creating a space that people explore
        - want to make sure that players reach all the different states quite evenly
    - got to calculate all the branches of a tree
        - most games where the rules are incomplete indicate that the designers didn't analyse all the possible game paths well enough

- a fundamental weak understanding of the variety of game states is a smell
    - leads to games that grind to a halt or loop infinitely
        - with so much experience Vickery can generally see such cases from the rules

- how to avoid falling into these traps
    - computer programming is an iterative process where you're constantly showing and receiving feedback from people
- a designer and a developer
    - "for a loss of 10% of enjoyment, you reduce the complexity by 50%"
- designer can lose track of the game objectives
    - lots of testing before and after

What's a classic game or can classic games still be made?

- lots of classic games are in the public domain
- arimaa http://arimaa.com/arimaa/
    - rules are free - can be played with a chess set
    - yearly $10,000 prize challenge? or tournament?
- Trivial Pursuit
- Magic the Gathering
- Settlers of Catan
- Risk - simplistic
- Sodoku - recent, simple and is very popular

Vickery's opinion of the trend of computerising board games over the internet

- slight loss of social interaction
- being able to see your opponents eyes as you destroy something of theirs
- rarely is a player fully focusing on the game
    - a magic circle contract that all the participants are fully involved with the game
        - physically this can be seen and felt
        - over network, not so apparently and possibly not there

- there was a major fear of that - things come and go in waves
    1. dungeons and dragons and other role playing games wiped out war games
    2. magic the gather and other trading card games wiped out role playing games
    3. computer games came along and wiped out everything
- the whole gathering at a table and socially interacting directly will never be replaced by computer gaming
- parents concerned about their child constantly in front of a computer
    - board game sales are rising

Games exploring mathematical space, Diplomacy board game - psychological and mathematical

- have to set up the interaction between players that allow for a higher psychological plane to be developed
- Give players more power, and choice in their potential actions

Game becoming more simple -> any redeeming qualities of a game that is too fiddly?

- (Advanced) Squad Leader board game
    - rules and details for every single different gun ever created
    - descriptions and calculations for almost every possible scenario
- Only small population like it

Any games that impress Vickery recently?

- Cooperative games - where the players work together to beat the system
    - Pandemic
    - Ghost stories
    - Battle Star Galatica
    - Lord of the Rings game
- Interesting because there isn't necessarily a winner or everyone is a winner

---

Algorithm Design

Lecture 2.3.0

Design Smells

- Design Smells are a symptom that there may be a deeper design problem with the design of the system.

Interfaces

- Program to an **interface**, not an implementation. "Abstraction should not depend on details. Details should depend on abstractions."
    - This means you are using a **type** and not a concrete class. It will allow you to swap classes in the future: you can write a new class with a totally different implementation, but using the same interface so swapping is easy.
- Favor object composition rather than inheritance.
    - When adding new features or extending a class a bit, don't use inheritance. Instead, use delegation to compose classes together to add the functionality.

```
class Fruit {

    //...
}

class Apple {

    private Fruit fruit = new Fruit();
    //...
}
```

Inheritance, class Apple is related to class Fruit by inheritance, because Apple extends Fruit. In this example, Fruit is the superclass and Apple is the subclass

```
class Fruit {

    //...
}

class Apple extends Fruit {

    //...
}
```

Constructors

- Strangely, constructors can be design smells, because they can only be overloaded, not given unique names like methods.
- Constructors are not abstract. When you inherit from another class, you don't inherit their constructors, you give your own constructors. If you want to re-use theirs, you have to explicitly do so.
- Constructors cannot be renamed, they must be overloaded. Overloading does not change the name of the constructor so you cannot attach semantic meaning to what exactly it is constructing.
- Overloaded constructors are a code smell, there is most likely code duplication which can be refactored into a more generalized constructor.
- Sometimes constructor details give away the implementation details.

- **Chaining constructors together** (a refactoring method)
    - Find two constructors that are similar
    - Rewrite one of them and get it to call the other constructor, adding the extra thing in it
    - Run tests again
    - Rewrite to make everything beautiful
    - Repeat
- Beware of increasing complexity when having multiple constructors though
- Named constructors
    - These are just normal methods that return objects
    - Everyone can call the named constructor to make an object and now you can make the normal constructor private, forbidding others from using it.
    - It is better to use a named constructor because it has a better name, and the details of how to make the thing you want are inside the constructor, away from prying eyes. It does the heavy lifting for you.

Summary

- Big Design vs Incremental Design
- Construction methods
    - Static
    - Factories
- Factory methods
- Template pattern
    - Hooking

Examples

- Static Construction Method
- Factory Methods
- Template Pattern - Hook

Overall Information

Big Design vs Incremental Design

Big Design

- This style is not used so much now days
- All the designing is done up front
- If everything goes well it can be fantastic
- Otherwise it will likely end up as a messy, broken and poorly implemented

Incremental Design

- More popular today
- Design choices are done as you go
- Will almost certainly return an acceptable program
- Is less able to return something as awesome as a good big design

Balance of Planning

- Up-front design plans are good because they can guide you along the whole work process.
- Up-front design plans are bad because they limit your development path and can be a waste of time if you get them wrong.
- Up-front design should be balanced between too much and too little, by:
    - Your ability to produce plans.
    - The size and complexity of the project.

Construction Methods

- Methods that replace constructors.
- Constructions methods can be static, or made into factories.

    - The details of the construction can be totally encapsulated into the construction method (increasing abstraction).
    - Constructors that may have been hidden and privatised by the construction methods can be changed (implementation wise) without breaking anything (decrease in coupling).

Factory Methods

- Type of construction method, which usually groups related construction methods together.
- They group construction methods together because these construction methods cannot be mixed and matched, they combine to produce a high-level product.
- When a factory method is called it returns a type(interface) as opposed to a concrete class which allows for flexibility.

Template Pattern

- Designing classes like so:
    - Create core methods which call and control sub-methods.
    - Create sub-methods to do various tasks.
- This allows children of this parent class to inherit and override those sub-methods, use the bulk of the parent's work and add slight tweaks and variations.
- Template method defines the skeleton of Algorithm(steps of algorithm) in a method, differing some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Advantages :
    - No code duplication between the classes
    - Inheritance and Not Composition.
    - By taking advantage of polymorphism the superclass automatically calls the methods of the correct subclasses.

Hook

- A hook is s particular type of template method where sub-methods are wrapped in front and behind a core method, but these sub-methods are written to do nothing in the parent, hence are not used at that level.
- The parent is written carefully this way to envisage future use of those empty sub-methods.
- From there the children can inherit and override(inject) their additional functionality (hooking into the parent).
- Hook is a method that is declared in abstract class, but only given empty or default implementation. This gives the subclasses the ability to hook into the algorithm at various points, if they wish. Subclass is also free to ignore the hook method.

Template Method Pattern : Subclasses decide how to implement steps in an algorithm.

Strategy Pattern : Encapsulates interchangable behaviours and use delegation to decide which behaviour to use.

Factory Method: Subclasses decide which concrete classes to create.

Abstraction

- want abstraction so objects aren't bound or coupled so the codes is flexible for changes
- can't create instance of the abstract class

Factory Method Pattern

The method of creating a super class that contains the basic functions and then having the subclasses of the super class overwrite existing functions or incorporated new functions.

Factory methods eliminate the need to bind application-specific classes into your code.

The code only deals with the product interfaces; therefore, it can work with any user-defined concrete product classes.

Factory methods connect parallel class hierarchies in such a way that it localizes the knowledge of which classes belong together.

Factory methods provide hooks for sub-classes to create different concrete products.

- Examples:
  - Maze/BasicVersion - hard coding the Maze
  - MazeGame.java - factory method
  - BombedMazeGame.java - factory method that allows changes
- creating an object using the naming method instead of a constructor
- allow the creation child function that can overwrite parent function it is an example of the template method

Abstract Factory Pattern

The only difference from factory method is that there is a client class that tells the abstract factory to create the product. The factory is abstract because the client class don't know how the factory class creates the product

- Examples:
  - MazeGame.java - client using the abstract factory
  - MazeFactory.java - abstract factory (interface)
  - BombedMazeFactory.java - concrete factory

- This pattern is good when we don't want the client class to know about the factory.
- abstract factory acts as an interface and the client class just tells what to create.
- Abstraction is mainly meant for removing the code duplication and also for supporting polymophism and also code reusability.

Pattern

example

- parent got all the fundamental function
- the child just overwrite the steps function
- this is an example of a template Method

Design Algorithm and Data Structure

Brute force is a method to find an algorithm solution by searching through all the possible combinations.

- Can be used for small complexity program
  - Use other method if possible.
- Not really useful for big complexity program
  - It will slow down the performance

the shoulders of giants

- standing on the shoulders of giants - reusing other people's idea to build upon your idea
- Example of idea reuses:
  - beethoven's idea for music was based on Mozart's music
  - back to the future - mcfly played rock and roll back in the 50's

---

Lecture 2.3.3

Design Process

In lecture 2.3.3, Richard explains a general design process for any problem. In the lecture itself he uses a graduation speech he has to deliver as an example, but the general design process can be listed as follows:

1. **Understand your problem** - This is the most crucial step, and the hardest one. The best way to tackle this is not to try and understand the entire problem, but break it down and solve small problems in the incremental design we've been using over the semester. Understanding a small problem is really easy compared to understanding a big problem e.g. Big problem - "How do i make a car?" compared to small problem - "How do i attach a door to the frame?".
2. **Data collection** - Look at several examples of problems similar to yours (great speeches in history) or exactly like yours (the terrible speech he saw at an earlier graduation). When researching cast a wide net and try to find many examples because the more data you have, the better your results can be. But also be wary of what data you consider otherwise you fall into the trap of Garbage In Garbage Out.
3. **Propose solutions** - Think about various solutions to the problem (Various different speeches you can give). Analyse the data before you and use maybe a criteria you formed when understanding the problem. In various forms of design you can always find different methods to the same problem and it is important to be able to analyse which one's are good and which one's are bad and combine all the good and throw out the bad.
4. **Designing the details** - Try to design small simple items at a time rather than create large things all at once (Only show 1 or 2 points). This has been a tagline of this course, and we've seen it in OO design but it applies universally because humans do not deal well with multidimensional problems. If we try to consider too many factors at once and solve the entire system, the solution, whatever form it may be in (speech, program, industrial machine) will be ripe with errors and flaws. The Lock was a great example where by considering what each wheel does, makes the program really easy to program and to understand, but if a person were to try program the entire lock without considering small simple items, it would be a complete nightmare.

Lecture 2911-3.0.0

Notes

- Ensure blog entries have no general information unless you can back it up.
- Ensure your blogs are thoughtful.
- Design has a lot of things in common throughout the different areas of design.
- We now have to use rationality and reasons to solve problems.
- How do we solve algorithmic design problems?
- We want to look at how the process of solving problems.
- The trying everything approach is the brute force approach which will come up with a solution although it is usually slow
- Sometimes you only have local information which is hard to make a strategy which is the optimum one for the whole solution.
- Greedy Strategy You make a series of decisions so that you have enough to make a complete solution.
- Coin Change problem. Greedy Strategy, choose the largest coin first, etc. This doesn't work for all denominations.
- Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data.
- Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.
- When someone gives you a question you have to see if there is a way to check first to see if there is a way to unsolved the solution.
- Design methodologies.
- Steps to solving an algorithm:
    - Is an exhaustive search viable.
    - Can i break it into parts.
    - Then try and see method

To sum up:

- The greedy method is awesome. If you can use it, you should definitely use it. It is simple to code and produces *major* speedups.
    - Greedy algorithms basically pick the best thing they can at each stage, and hope this is sufficient to produce an optimal solution.
- A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage[1] with the hope of finding a global optimum.
- On some problems, a greedy strategy need not produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution.
- Only issue with this: greedy doesn't always work. A simple example of this is the coins problem. It's possible to define a set of coins where the greedy method won't find the optimal solution; for example, if you define the set of coins as (15,10,8,1) and try to make 18 using the greedy method, the algorithm will give the solution (15,1,1,1). The optimal solution, however, is clearly (10,8).

Lecture 2911-3.0.1

Notes

- Low coupling ways are usually faster
- You want parts of the solution to be coherent
- Exponential is never a satisfactory solution its way to big.
- A Greedy Algorithm would be to work out the first component the second component and so on.
- Bruteforce is enumerate over all solution states till you find a answer state.
- a problem state is a partial constructed solution state.
- The greedy method constructs a solution in problem states.
- If there is no coupling between the solution states means you can solve it in a piecemeal way.
- You see a problem for the first time you have to be able to break the whole answer into a series of sub answers, then you have to have the solution of this part is not depended on future parts, means you can do the greedy method.
- If each of the parts is sufficient to solving the whole solution.
- selection sort uses the greedy method.
- If you can do a greedy method do it.

To sum up:

- similar principles apply to both algorithm design and object-oriented design. They often involve breaking down problems into little self-contained pieces which combine to form a solution.
- The less coupling in an algorithm, the faster it generally is.
- Some examples of greedy algorithms:
    - Selection sort (Complexity is O(n^2))
    - Prim's and Kruskal's algorithms, as well as Dijkstra's

Lecture 2911-3.0.2

Notes

- Information content, is useful in compression, as if we know the content and rewrite it in another way, then you can compress it.
- Divide and conquer is an alternative to greedy.
- It works by solving the same problem for smaller sizes and then combine the answers for the whole solution.
- When it works its fantastic but it isn't really common to use it
- Time to sort numbers is n^2 divide and conquer notices when the problem is greater than linear it is faster to divide it in half.
- ie T(n/2) = (n^2)/4 and then the total would be n^2/2 and along as the work to combine them is liner then T(n) becomes n^2/2 + n vs n^2.
- Merge sort uses this method and complexity is below

```
T(n)=T(n/2)  + T(n/2) + n;
T(n)=2T(n/2) + n;
T(n)=2[2T(n/4) + n/2] + n;
T(n) = 4T(n/4) + 2n;
T(n) = 8T(n/8) + 3n;
T(n) = 2^k T(n/2^k) + kn

n = 2^k k = log2(n) T(n) = n + nlogn;
```

Lecture 2911-3-Johnny (How to use

- Download zip,
- enable -ea
- Check John garlands nots for making a runnable jar
- Put a jar in the tested folder
- So tests don't work properly yet
- "If you can't implement time machine don't worry about it" Quote Acceptance test caesar
- You can't use intellij with acceptance tests Actually you can you make a a jar with the acceptance tests and your implementation.
- You keep implementing things till there are no assertion errors
- Use adapters so you don't have to change all of your game
- So don't really directly implement interfaces

Combinatorial Search

3.1

3-1-Fogg-Problem

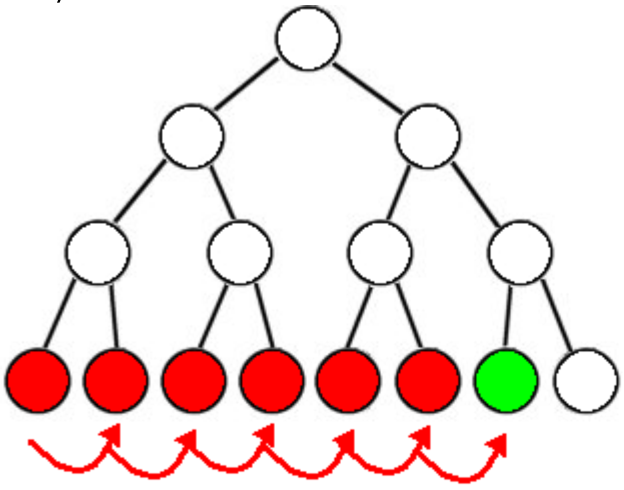
Richard presents the Phileas Fogg problem, wherein you are trapped in a room with many doors. One of these doors will lead to an exit, the others will simply be cupboards. Each door has 3 locks that must be opened using a certain key. In this example, there are 4 doors and 12 keys.

The answer state is the correct way of solving the problem. In this problem, the Answerstate is represented as (D2, K2, K3, K7), which is opening door #2 using keys 2, 3 and 7 for locks A, B and C, respectively. This will open door #2, which will lead to an exit.

The solution state is a potential answer that has not yet been verified. One such state can be represented as (D1, K1, K2, K3). Since there are 4 doors and 3 locks than can be opened with one of 12 keys, there number of solution states is (4x12x12x12) = ~6000. For a human being fitting keys into locks, this would take far too long.

Approach #1: Brute Force

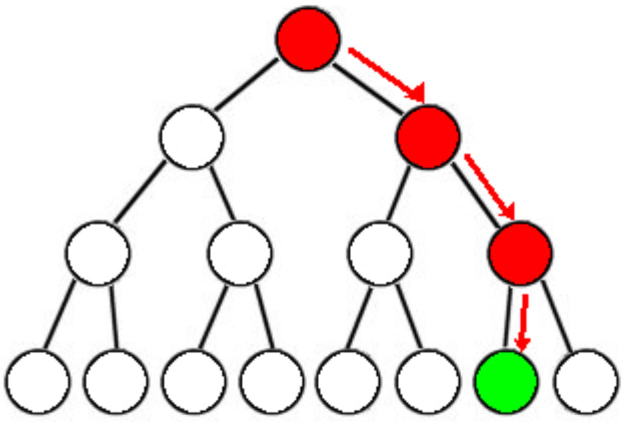
try every solution state:



representation, you might miss some states and potentially miss the correct answer.
- Speedups don't alter the complexity of the solution.
- Try to consider solution states in a sensible order. This way you're likely to find the answer and be able to early exit. Think carefully and look in the most likely places first.

Approach #2: Greedy

Once you've found a correct part of the answer you never need to consider it again:

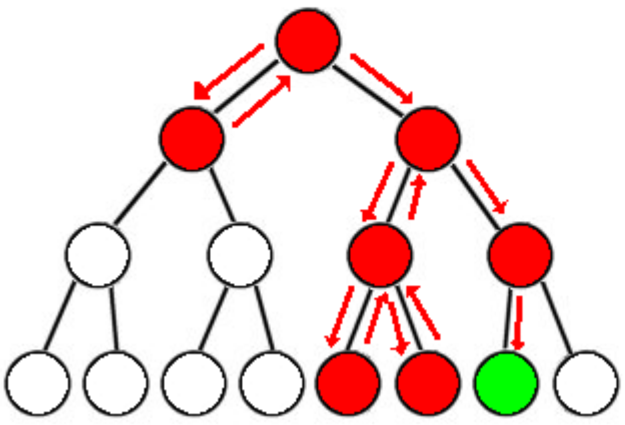


Assume that a lock will produce an audible click if it is unlocked. As soon as you realize that a particular key will not unlock a particular lock, you can safely eliminate an entire subset of solution states. In the same way, if you know that a particular key does unlock a particular lock, you narrow the solution states down to a smaller subset. This is a problem state, a partial solution.

The Greedy method involves constructing the answer state bit by bit rather than testing each permutation. This is possible because unlocking locks A, B and C are tasks independent of one another. This way, to unlock any door it will take at worst (12+12+12) = 36 attempts as opposed to (12*12*12) = 1728 attempts via brute force.

Approach #3: Backtrack

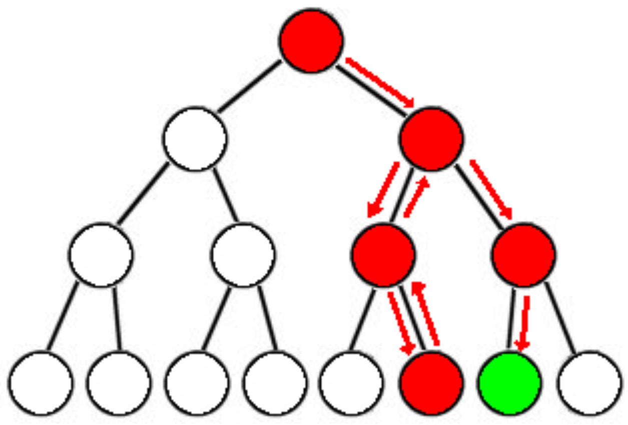
Keep building up a solution in a regular way until you realise you've done something wrong, then go back:



Assume a lock will sometimes click even it hasn't been unlocked. If we finalize our solution via branches if they don't click, leaving only those worth exploring.

Approach #4: Branch and Bound

Use an algorithm to determine how you build up your solution, then back track if it's wrong:



This is similar to backtracking, but instead of visiting every solution state mindlessly, start by just generating its child states (*branching*). After you have these states, you can then look at them and decide which child states could not possibly lead to the solution you want (*bounding*). This is usually accomplished by a boolean function (a *bounding function*) called on the state. For certain problem, this can be much faster than backtracking.

Video 3-1-0 Notes (Backtrack, Branch and Bound)

Backtrack

Breaking the problem into parts by trying to solve it in stages, and evaluating as we go and prunes/binds sub tree of incorrect solutions

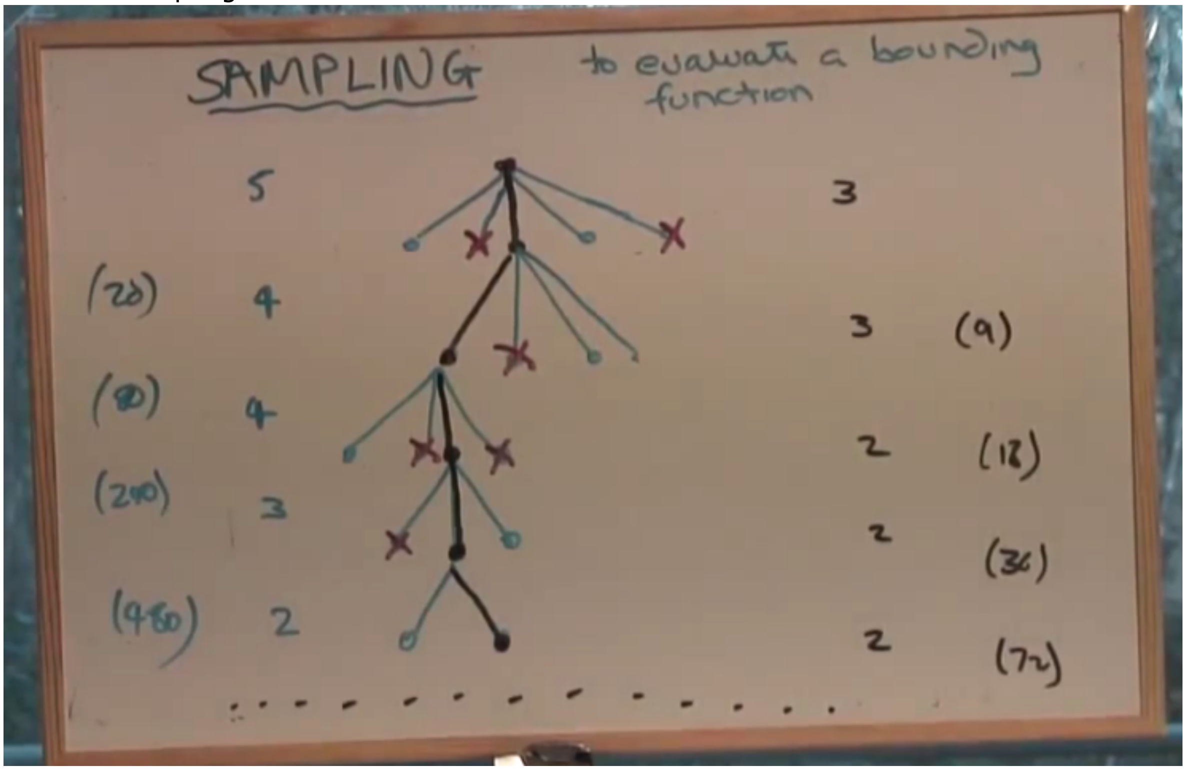
Components of backtrack

- Way of systematically generating problem states to get solution states
- We want to fail as early as possible so that backtracking can take place earlier and more solutions are pruned
- Bounding function that prunes the tree higher up in the tree, something that is:
    - cheap (because it has to run for each component of the solution state)
    - and ruthless (saves a lot of work later)
- It comes down to how much work it is saving us vs. how much it costs us to bound
- Which level in the tree to consider first:
    - Which door to try first? Which keyhole to try first?
    - Evaluating something more likely to fail first will eliminate entire branches quickly, which will increase efficiency

Evaluating bounding functions

Although bounding functions can increase efficiency, the bounding function itself has to be run on every node that it visits. If the bounding function itself is very costly it could decrease efficiency. Thus it is important to determine the costs and the savings of bounding functions.

- Investigate through "try and see", but this is a very expensive approach.
- Analyse a smaller problem to get a feel for the bigger problem
- Random sampling :



    - Generate all the possible children from initial state, randomly pick one of its unbound children to expand, repeat till the bottom of the tree is reached so that a random path is generated.
    - Assume that the branching factor is consistent within each level, e.g. in the first level there's 5 children to each node, second level there's 3 children to each node. (5->4->4->3->2)
    - Estimate the size of the tree (20->80->240->480)
    - Find number of unbound children in random path (3 -> 3 -> 2 -> 2 -> 2)
    - Estimate the number of unbounded children in each level (9->18->36->72)
    - Compare size of the tree to the number of unbound children (480 vs 72)
    - By taking multiple samples, on average the samples will converge to give a rough indication of how effective the bounding function is
- For certain kinds of problems, you may be able to start bounding earlier as you draw closer to the end. For example, you're looking for the best cookie. You find a normal cookie, then maybe a Tim Tam. Then something else, but you put it back down because it isn't better than the Tim Tam. You can now discard options much earlier because you already have the best 'solution' so far.
- Design pattern: Singleton is a class that is set up so that it's only one instance (constructor is hidden, created through a static function, flag to indicate that it has already been created)

3-1-1

Given a vector, compute the max sum found in contiguous subvector

ie |34|-41|**59|26|-53|58|97**|-93|

```
maxSoFar = 0          // (if all numbers are negative the empty set has the max sum
for i = [0,n){        // considering all positions from the start to the end (from
   for j = [i,n){
      sum = 0
      for k = [i,j]{ // sum up from start till end
         sum += x[k]
         maxSoFar = max (maxSoFar, sum) // compare it to maxSoFar and keep updatin
      }
   }
}
```

Complexity: O(n^3) very slow

- Speedups - Keeping track of the sum.

```
maxSoFar = 0
for i = [0,n){
   sum = 0
   for j = [i,n){
      sum += x[j]
      maxSoFar = max (maxSoFar, sum)
   }
}
```

Complexity: O(n^2), ok for small n.

- Divide and conquer: divide and get the best of the two, and the middle bit Complexity: O(nlogn)

15-puzzle

Possible solutions:

- Brute force: can go for ever, we can have loop detection to avoid entering loop, but still the solution state is too big for brute force. Using loop detection can be avoided by simply not using depth first searches.
- Can't make divide and conquer because the problem cannot be divided into subproblems.
- Backtrack, a tree with branching factor 3,2 or 1 (middle, edge, corner). Your pruning function would be a state which you have already seen, or a state in which the next possible move would be a state that you have already seen. However, even with backtrack , you are essentially still using a brute forcer. You never prune on a problem state, only when you have reached a solution state.
- Greedy using montecarlo.
- Breath first search using Manhatan distance.
- The general way of solving this type of puzzle is by using a branch and bound algorithm known as A*. But instead of doing a brute force with breadth first search, every time they have a state they look at the 4 possible children they can generate by moving and at how 'good' they are. How good they are is often determined just by the number of tiles in the right position at the time. You favour the children which have more tiles in the right position.

3-1-2

Decorators:

When thev are most useful:

animalWithClaws class. you might then do a similar thing for animalWithHorns. you can't then make an animalWithHornsAndClaws because you can't extend 2 things, so instead you use decorators.

How to do it:

- Create an interface, eg a shark interface
- Create a 'decorator' which implements the interface (the shark one in this case)
    - This decorator however takes in a 'shark' in its constructor.

```
public SharkWithLazerBeam(Shark shark) {
      this.shark = shark;
}
```

- The decorator can then extend the interface.
- The implementation of the functions is running the functions from the shark that is passed in.

```
public void attack(Object victim) {
      System.out.println("IMMA CHARGIN' MAH LAZER!"); //Added functionality
      System.out.println("IMMA FIRIN' MAH LAZER!"); //Added functionality
      this.shark.attack(victim); // Original function not changed
      System.out.println("SHOOP DA WHOOP!"); //Added functionality
   }

public int getDamagePoints() {
      return this.shark.getDamagePoints() + 9000; // The 9000 is the added functio
   }
```

- This allows you to add features, but still use the original implementations of functions you wrote for your sharks. this way any shark implementation can be decorated with this new class.
    - Essentially, you are able to 'add' to functionality without changing the function itself. In the case of the shark, it is still able to use its functions 'swim' etc, but you can add more features to it (making it swim faster) without changing the swim function.

example code

```
public interface Shark {

   public void swim();
   public void attack(Object victim);
   public int getDamagePoints();

}

public class GreatWhite implements Shark {

   public void swim() {
      System.out.println("SHSHSHSHSHSHSH");
   }

   public void attack(Object victim) {
      System.out.println("OMM NOM NOM " + victim);
   }

   public int getDamagePoints(){
      return 100;
```

```
public class SharkWithLazerBeam implements Shark {

   Shark shark;

   public SharkWithLazerBeam(Shark shark) {
      this.shark = shark;
   }

   public void swim() {
      this.shark.swim();
   }

   public void attack(Object victim) {
      System.out.println("IMMA CHARGIN' MAH LAZER!");
      System.out.println("IMMA FIRIN' MAH LAZER!");
      this.shark.attack(victim);
      System.out.println("SHOOP DA WHOOP!");
   }

   public int getDamagePoints() {
      return this.shark.getDamagePoints() + 9000;
   }
}

public class SharkTank {

   public static void main(String[] args) {
      //normal shark
      Shark greatWhite = new GreatWhite();

      //awesome shark
      Shark awesomeGreatWhite = new SharkWithLazerBeam(new GreatWhite());

      //super awesome shark
      Shark superAwesomeGreatWhite = new SharkWithRacingStripes(new SharkWithLazer
   }
}
```

3-1 live Thursday lecture

Problem solving isn't about knowing things or being clever (although that helps) - follow the process:

1. Brainstorm! Think of as many ideas as possible. Don't rule any out, no matter how dumb they seem.
2. Evaluate the options against the criteria and constraints in the problem. Take some time to do calculations, analyse complexity, etc., to work out what kind of advantages and disadvantages each idea has.
3. Implement ideas, starting with the best idea first. If that idea isn't going to work, move to your next best. Maybe you could implement a maximum of 3 ideas (if you need to) but you probably won't have time for any more.
4. A template rules out all the bad part but may also prevent good parts.

- Though a quantitative assessment often leads to 'restrictions' of the good parts it is still needed simply because you cannot measure qualitatively. (A person can say his doing good work and not do anything)

Richard's Target solution

a single node.

- For every letter in the alphabet, create a trie of all words (with their letters sorted) that contain that letter (in this way, you can easily fulfill the requirement that every solution word must contain the centre letter)
- To count the number of solutions to a nine letter word, do a backtracking search of the prefix tree, making sure that every word in the path generated fits inside the nine letter word
- Remember to account for anagrams in the prefix tree

Dynamic Programming

3.2

3.2.0

Write down the answer after you solved a problem and when you need the answer, you can "cheat" by reading the thing you written down already (needn't any researching / calculating again).
This is the main idea for *Memoisation* and *Dynamic Programing*.

3.2.0.0 Fibonacci Sequence

Fibonacci Sequence: 0 1 1 2 3 5 8 13

f(0) = 0
f(1) = 1
f(n) = f(n-1) + f(n-2) (n>=2)

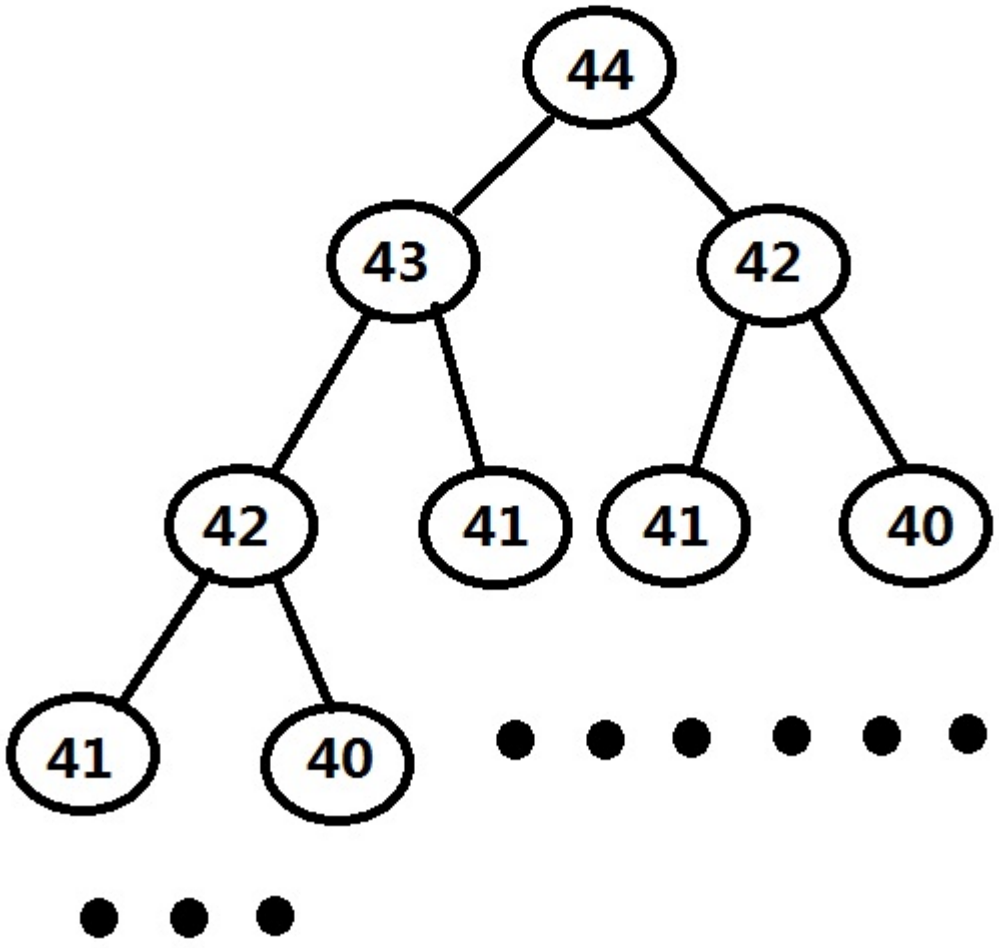
3.2.0.1 Calculation(Basic)

```
public class Fibonacci {
        private int n;

        public Fibonacci(int n){
                assert (n>=0);
                this.n = n;
        }

        public long getValue(){
                long value;
                if (n<2){
                        value = (long) n;
                }
                else{
                        Fibonacci before = new Fibonacci(n-1);
                        Fibonacci beforeBefore = new Fibonacci(n-2);
                        value  = before.getValue() + beforeBefore.getValue();
                }
                return value;
        }
}
```

for calculating fib(42), it take 10.1 seconds
for calculating fib(43), it take 16.3 seconds



3.2.0.2 Calculation (Memoisation)

Be smart : remember solved results -- Memoisation
Solve it at the first time and later on just simply load the result from the memory.
This is a top to bottom approach.

```
public class Fibonacci_Memoisation {
        private int n;
        private long value = -1;

        public Fibonacci_Memoisation(int n) {
                assert (n >= 0);
                this.n = n;
        }

        public long getValue() {
                long ans;
                if (n < 2) {
                        ans = (long) n;
                } else if (this.value == -1) {
                        Fibonacci_Memoisation before =
                                        new Fibonacci_Memoisation(n - 1);
                        Fibonacci_Memoisation beforeBefore =
                                        new Fibonacci_Memoisation(n - 2);
                        ans = before.getValue() + beforeBefore.getValue();
                value = ans;
                return ans;
        }
}
```

This code looks awesome, but still slow, take 26 sec to calc fib(44)

3.2.0.3 Calculation (Memoisation_Advanced)

The previous code example is slow as it contains duplicate objects which are created over and over again. The amount of objects it creates is exponential, so the higher n is, the worse the duplication.
Be smarter : we create so many fibonacci objects although most of them are the same thing, try to avoid creating duplicate objects.
This is still a top to bottom approach.

By using a factory we can access just one object which will store the results for the Fibonacci numbers it is already calculated we only need to store n results.

```
public class Fibonacci_Memoisation_Advanced {
        private int n;
        private long value = -1;
        private static Fibonacci_Memoisation_Advanced[] Memoisation
                = new Fibonacci_Memoisation_Advanced[100];
        public Fibonacci_Memoisation_Advanced(int n) {
                assert (n >= 0);
                this.n = n;
                Fibonacci_Memoisation_Advanced.Memoisation[n] = this;
        }

        public long getValue() {
                long ans;
                if (n < 2) {
                        ans = (long) n;
                } else if (this.value == -1) {
                        Fibonacci_Memoisation_Advanced before =
                                        getObj(n - 1);
                        Fibonacci_Memoisation_Advanced beforeBefore=
                                        getObj(n - 2);
                        ans = before.getValue() + beforeBefore.getValue();
                } else {
                        ans = value;
                }
                value = ans;
                return ans;
        }

        public static Fibonacci_Memoisation_Advanced getObj(int n){
                if ( Memoisation[n]!=null){
                        return  Memoisation[n];
                }
                return new Fibonacci_Memoisation_Advanced(n);
        }
}
```

WOW! Now 0 second for calculating fib(44)

This idea (create same object once only) is called: Singleton Pattern
Here is a brief tutorial for making a Singleton Pattern on your own

3.2.1

```
c(f) = {if breaks k - 1, else c(f-k)} + 1
If we wanna make c(f) small, c(f-k) should be equal to k - 1

f(1) = 1;
f(2) = 3 = f(1) + 2;
f(3) = 6 = f(2) + 3;
f(4) = 10 = f(3) + 4;
f(5) = 15 = f(4) + 5;
f(n) = f(n - 1) + n;
f(n) = 1 + 2 + ... + n = (n - 1) * n / 2 => n ^ 2
```

We will most likely need to know the most optimal way to compute 1+2+3..+n in the exam! (IE using memoisation and dynamic programming)

Dynamic Programming: solving a problem by only using results from previous solved smaller problems.
Keyword: Previous sub-problems only - this problem have to be able to be solved by using previous solved sub-problems (never need to hack to the future), this means the problem have to be an Optimal Problem.
It is important to note that if early decisions effect future problems, doesn't have optimal substructure

There is a subtle difference between divide and conquer and dynamic, divide and conquer is a big problem in half, then half again etc. Dynamic is splitting into problems that are similar to the original but smaller. Sub problems may appear multiple times. We don't want to calculate these multiple times. (THE BIG PAYBACK).

Memo changes complexity of fibonacci from Exponential to order N
We also need a sequence about what we solve first, because all required sub-problems must be solved before solve a problem using Dynamic Programming.

For example, in Richard's Barometer Problem (Though more commonly is given as the egg drop problem, we can find a sequence that result for a floor depends on the floors smaller than it, so we can loop and solve from min floor to the floor number we want to solve.
ie: Drop it at 10, then 19, then 27 etc... It should be 10 drops maximum despite there being 100 floors

ChessBoard example:

```
Someone is at bottom-left corner(0,4) on a 4*4 chessboard(square), how many possib
Shortest way: he can only go left or go up.
possible shortest ways to arrive point(x, y):
f(x, y) = {if (x = 0 and y = 4) 1, else f(x - 1, y) (moved from left) + f(x, y + 1

Easier sub-problems (get Optimal solutions):
1. f(0, 4) = 1;  Easiest
2. f(0, 3) = f(0, 3 + 1); f(0, 2) = f(0, 2 + 1);...;f(0, 0) = f(0, 0 + 1);
2. f(1, 4) = f(1 - 1, 4); f(2, 4) = f(2 - 1, 4);...;f(4, 4) = f(2 - 1, 4);
3. f(x, y) = f(x - 1, y) + f(x, y + 1)
```

Heuristics

3.3

- Simulated annealing (SA) is a generic probabilistic metaheuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more efficient than exhaustive enumeration — provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution.

Girlfriend Analogy

The dilemma: You have a finite amount of time and you have to find the best girlfriend in the world. You also are not sure what you are looking for.

Constraints:

- Finite time - cannot brute force (i.e. date everyone)
- Problem you cannot get the best answer to - must settle for the best you can find.
- Not too sure about what you want.

Solution - Need a change in solution

- Initially be fairly exploratory then...
- Become more fussy (convergence)
- Finding an incorrect solution would give you further hints to the solution (I want someone who is not boring, not self absorbed). Essentially you are learning what to look for. This is also known as 'Structure on Solutions'.

Finding the Highest Point

In the hill climbing problem, there is not enough time to visit **every** point however it is a feasible problem because you have time to visit **any** point. Use this when you cannot reach every point (brute force doesn't like it). Each point you check you should be able to collect information of what the next point may be (for example if you are climbing up and the points in front of you starts going down, you reached a local maxima).

At the start of the search, you will continue to search for higher and higher points (by jumping around quite a bit). Nearing the time limit, you would settle down and try to find the highest spot near the point (hill climbing). Essentially the girlfriend problem.

Simulated Annealing

Inspired by natural processes. Heating up metals may change the property of a metal (for example brass becomes soft) - this is known as annealing. At high temperatures, the molecules will bounce around quite fast trying to find the best spot. As the temperature cools, the molecules will (if they are in a good spot) be resistant to moving. Sounds familiar to the Girlfriend Analogy?

3.3.1 - Simulated Annealing Part 2

The Story So Far

1. Keep track of one solution at a time
2. Evaluate how good the solution is
3. Pick a neighbour which is close (cost functions should be similar)
4. Check if D is a better solution

Components

- Acceptance Function
- Valuing Function
- Neighbourhood Function

Acceptance Function

Probability of acceptance: **$p(A)= e^{(k*\Delta D/T)}$** where k is Boltzmann's constant.

A higher temperature (T) would result in a higher rate of acceptance. If temperature decreases, the rate of acceptance decrease.

Rule of simulated annealing acceptance function:

- If delta D (solution) is good, accept
- If delta D is bad, accept with probability p(A)

(Note: It is not always good to accept only when the solution is better because sometimes in order to get to a better solution, you first have to go through a solution that is not as good as the current solution first.)

Initially the temperature (probability of acceptance) is high. As time passes, you become less and less accepting of taking risks (decrease in temperature).

Valuing Function

Depends on the problem. For example in the knapsack problem: Value of a packing = sum of values * sum of weight.

ONLY IF sum of weights is less than maximum weight. Otherwise scale the value when over weight. e.g. Value of a packing = Sum of values * (Maximum Weight / Weight)

Pseudocode

```
1. Pick start solution
2. Current = start
3. Candidate = random_neighbour(current)
4. if (accepted)
    1. current = candidate
5. Iterate to line 3 for a set period
6. Decrease temperature
7. Iterate to line 2 until T == min
```

Simulated annealing may need tweaking to get it to work well.

1