CAB302

OOP

- Object orientation extends procedural programming with various abstraction mechanisms
  - Encapsulation
    - Expose the specification; keep the implementation and data private
  - Inheritance
    - The child has all the features of the parent with a few specific differences
    - Promotes code reuse
  - Polymorphism
    - Many versions of the one operation
    - Let the system choose the specific one at run time

Inheritance is when a 'class' derives from an existing 'class'. So if you have a `Person` class, then you have a `Student` class that extends `Person`, `Student` *inherits* all the things that `Person` has. There are some details around the access modifiers you put on the fields/methods in Person, but that's the basic idea. For example, if you have a private field on `Person`, `Student` won't see it because its private, and private fields are not visible to subclasses.

Polymorphism deals with how the program decides which methods it should use, depending on what type of thing it has. If you have a `Person`, which has a `read` method, and you have a `Student` which extends `Person`, which has its own implementation of `read`, which method gets called is determined for you by the runtime, depending if you have a `Person` or a `Student`. It gets a bit tricky, but if you do something like

Encapsulation

- A design principle that seperates the spefication from the implementation
    - Hiding implementation from others
- Specification - what the class/method does and how to interact with it (e.g. method signature)
- Advantage
    - Makes programs easier to understand because unimportant detail is hidden away
    - Makes programs easier to modify because the implementation can be changed without affecting the interface
- Supported in Java by
    - Method signatures
        - Specify how methods should be called and what they return
        - Applied without worrying about how they work but they do
    - Visibility
        - Classes, which restrict the visibility of fields and methods
        - Packages, which restrict the visibility of classes

- Make fields *private*
- Make accessors (getters) and mutators (setters) *public*
- Make helper (utility) methods *private* (unless you have a utilities class)

Inheritance

- A way of defining type hierachies
- Subclasses of objects
  - More specialised or complete than superclasses but which share common traits
- In general, objects instantiated from subclasses can do everything that superclass objects can, and sometimes more
  - They may also change (override) some supercalss characteristics
  - Superclass = parent/ancestor
  - subclass = child/descendant
- Car is a vehicle
- Bird is an animal
- Inheritance in Jav is via the **extends** keyword
- All classes extend root class java.lang.object
- A superclass constructor is called with super(), and its methods with super.method()
- super() not needed if implementing a no parameter constructor

Finality

- Prevents
  - Variables value from being altered
  - A class from being extended
  - A method from being overrid

Concrete Classes

- Have their fields and methods implemented

Interfaces

- Dont have any fields
- Methods are just specifications
- Contains abstract methods only
- Methods have no body
- Interfaces cannot be instantiated as objects
- Use keyword implements for **inheriting** from interfaces
- All methods in interface are public by default

Abstract Classes

- May have fields
- Have at least one method with is just specification
- Clas that implements some member fields and methods but leaves others abstract

Why use interfaces?

1. Each implementation is very different
    - E.g. interface with makesound implemented by dog and cat class
2. Acts as contract
    - Each implementing class HAS to provide implementation
    - Calling class knows that method is being implemented
3. Multiple inheritnace
    - Java class can only extend one class, while you can implement any number of interfacs

Why use Abstract Classes?

- Some shared traits (fields or methods) amongst classes but some non-shared features
- Benefits of OO
    - Code reuse
    - Works conceptually
    - Benefits of polymorphism
- Not as constrained as an interfac
    - No need for subclasses to implement non-abstract methods

When to Use

Interface	Abstract	Concrete
Don't have fields	Have fields	Have fields
Methods have specification & no implementation	Methods have some specification & some implementation	Methods always have implementation
Want to enforce (usually unique) method implementation	Allow class to use ancestor implementation (of non-abstract methods)	Allow class to use ancestor implementation
When you need to inherit from more than one class		

Polymorphism

- Polymorphism makes life easier for programmers by making common operations available in an indentical from in otherwise different classes
- In practice polymorphism is supported by method overriding and overloading

Method Overriding

- To determine which method to apply, the Java Virtual Machine begins at the bottom of the type hierarchy and searches upwards until a match is found

Dynamic Binding

- Subclasses inherit all superclass features
  - state and behaviour
- Subclass may add new state or behaviour
- Subclasses may *override* a superclass' behaviour
  - *polymorphism* means a variable of a superclass type can refer to a subclass' object
  - *dynamic binding* means the subclass' behaviour will occur, even though the client does not know it is using the subclass' object
    - because it is the subclass' object and it has its own behaviour

Enum types in Java

- Enums implicitly have a '`values`' method – can be used in **`for`** loops

```
for(Day d : Day.values()){
   printDayGreeting(d);
}
```

```
Some other day
Some other day
Some other day
Some other day
Some other day
TGIF
Some other day
```

Test Driven Development and Source Control

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

- Embrace change
- Deliver early and deliver often
- Customers make business decisions
  - Featrures
  - Priority (must have, nice to have)
  - Acceptance tests
  - And the customer is always available
- Developers make technical decisions
  - Estimation
  - Refactoring
  - Unit Tests

Test Driven development

- Automated unit testing
- Tests are developed before code is written
- Once code passes the tests, new tests are added
- Code is then extended to pass new tests
- Tests determine what code to write
  - Think of tests as detailed design specifications

- The iterative Test-Driven Development process
  1. Red — Add a small test that doesn't pass, and may not even compile
  2. Green — Make the test pass, in the quickest and easiest way, without breaking any previous tests
  3. Refactor — Rewrite the code to eliminate 'duplication' that was introduced merely to make the new test pass

Unit Test Guidance

- Limited in scope
  - Test one thing only
  - Usually one assertion per test
- Work in isolation
  - Without relying on other tests
- Clearly reveal their intention or purpose
- Test boundary conditions early
- Avoid testing against real resources ie GUIs or databases, to support testing determinism
  - This is where you use mock objects and services

- Only add tests that fail
    - Otherwise they dont help us at all
- Follow the usual unit-testing principles
    - Have normal cast tests for each equivalence clas
    - Have test for all boundary and exceptional cases
- Include both black-box and glass-box testing
- As the programmer, add glass box tests for
    - Conditionals
    - Loops
    - Operations
    - Polymorphism

Advantages to TDD

- Confidence to make quick improvements to the program
- Feedback on state of program
- Changes that break the program are detected immediately
- Testing takes the place of debugging
- Test cases take the place of documentation
- Forces programmers to think about the clients needs first
- Only necessary code is written
- Short development cycles mean there is always a usable product available
- Encourages better detailed design
- Automated refactoring tools can speed TDD and help avoid trivial coding errors

Disadvantages of TDD

- Difficult to use for programs or modules that have complex interactions with their environment
  - GUIs
  - Database interface
  - Embedded systems
- Bad tests will produce bad code
- If programmer write their own tests then blind spots may exist
- Emphasis on bottom up unit testing may cause integration problems to be overlooked

Mocking

- Fakes - Test objects whose methods work but have only limited functionality
  - for example, a fake database that stores all records in-memory
  - Or a fake GUI form that tracks components and handles events but draws nothing to screen

Version Control

- It is okay to have a local source control system on your machine, but you really must have a repository or copy elsewhere
- It doesn’t help much if your code is complete and beautifully versioned if your hard drive crashes
- It is better to have too much stuff under version control than to have too little

Branches

- Can be created using the 'git branch <branch name>' command (or 'git checkout -b <branch name>')
- Switch to a branch using 'git checkout <branch name>'. Commits will then go onto this branch.
- You can also 'git checkout <commit hash>' to detach the HEAD node and make it point to that commit. This can be useful for looking at older versions of the codebase, e.g. when trying to find when a bug was introduced.
- Can be deleted with 'git branch -d <branch name>'
- Can be merged by switching to the branch you want to merge this branch into, then issuing 'git merge <branch name>'.

Documentation

Javadoc

- javadoc command to generate
- HTML documentation from source code
  - Can be applied to individual classes or whole packages
  - Can be called directly from within IDEA

Javadoc Example

```java
/**
 * Given two integers representing hours and minutes, sets the
 * clock to the specified time, provided it is a valid 12-hour time.
 *
 * @param hours number of elapsed hours in the range 0 to 11
 * @param minutes number of elapsed minutes in the range 0 to 59
 * @return true if the parameters provided were valid and the
 * time was updated accordingly, false otherwise
 */
public boolean setTime(int hours, int minutes) {
//...
}
```

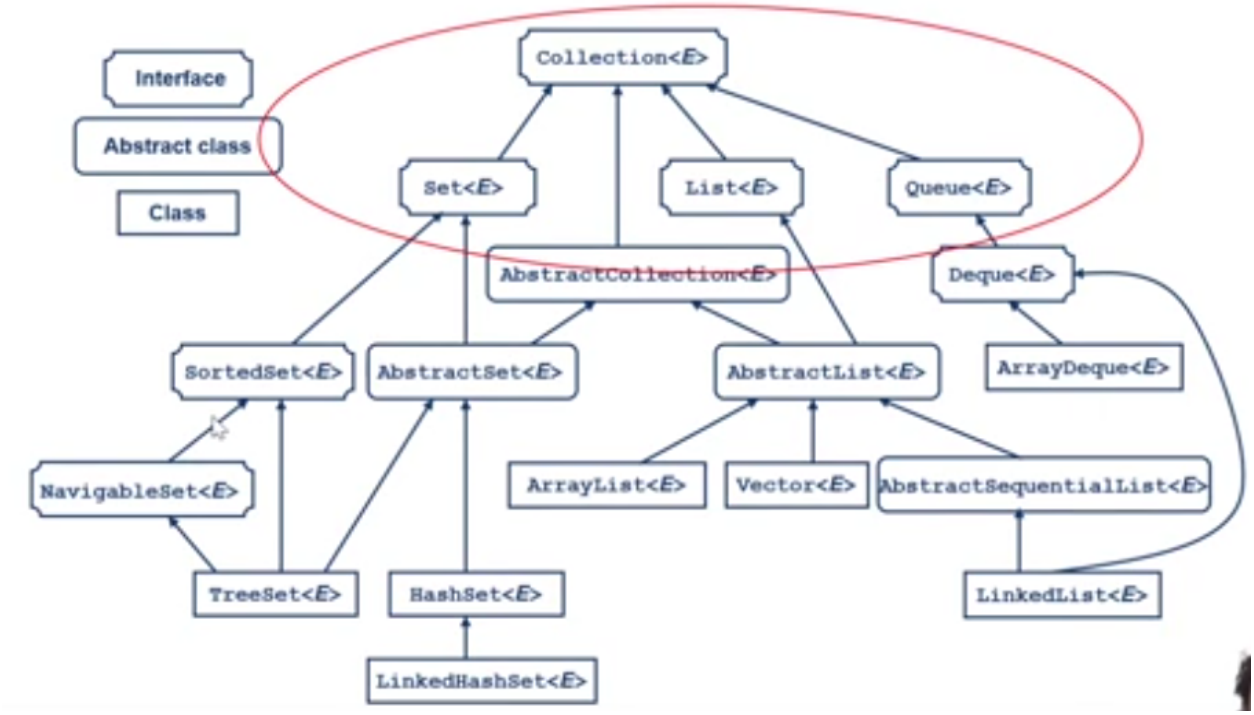
Collections

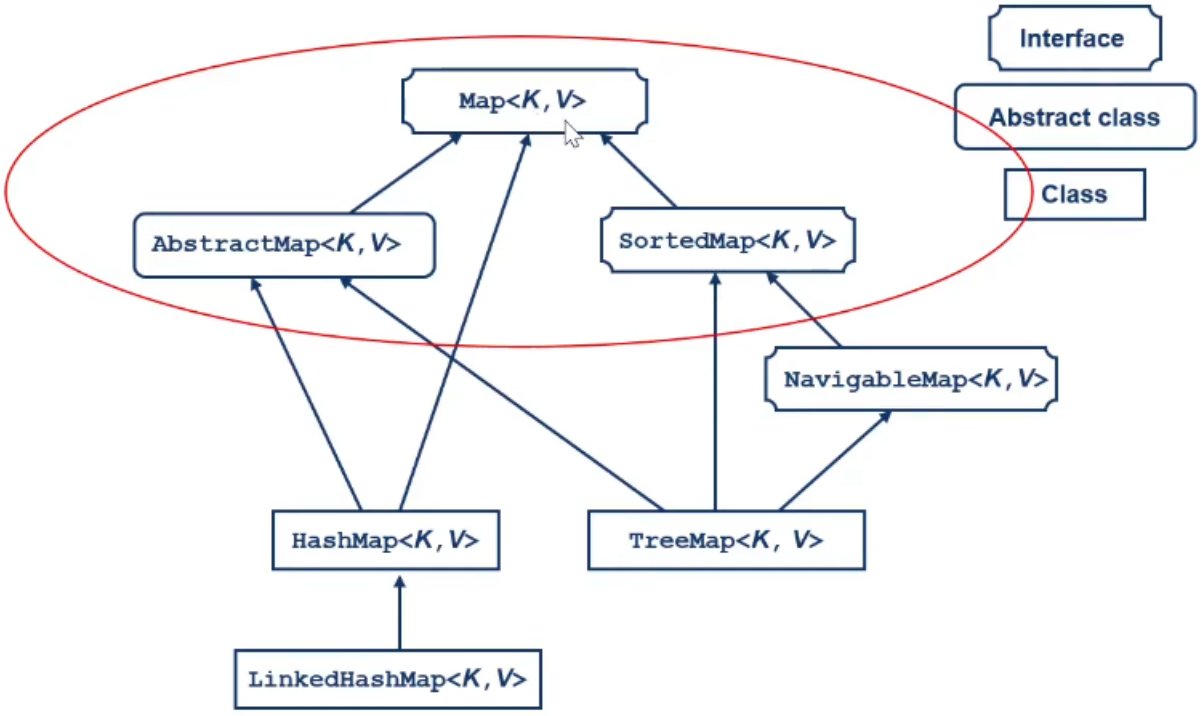
- List
- Queue
- Set
- Map

Multiple versions with different implementations.

- Linked List
- Resizable Array
- Balanced Tree
- Hashtable

Javas collection framework


Interface
Abstract class
Class
Collection<E>
Set<E>
List<E>
Queue<E>
AbstractCollection<E>
Deque<E>
SortedSet<E>
AbstractSet<E>
AbstractList<E>
ArrayDeque<E>
NavigableSet<E>
ArrayList<E>
Vector<E>
AbstractSequentialList<E>
TreeSet<E>
HashSet<E>
LinkedList<E>
LinkedHashSet<E>


Interface
Abstract class
Class
Map<K, V>
AbstractMap<K, V>
SortedMap<K, V>
NavigableMap<K, V>
HashMap<K, V>
TreeMap<K, V>
LinkedHashMap<K, V>

Collection operations:

- isEmpty(),
- contains()
- size()
- toArray()

Arrays vs ArrayLists

- ArrayLists support iterators, arrays do not
- ArrayLists can grow in size at runtime. arrays cannot
- ArrayLists only store objects. There is more overhead with storing objects as opposed to primitive types (ints, doubles, etc)

Sets

- Groups of elements with no duplicates
- Usually unordered
- Tree set is ordered vs hashset is not

Lists

- Ordered sequence of elements
- Indexable by individual elements or a range
- Can have elements inserted at or removed from any position
- Examples include ArrayList, LinkedList, Stack and Vector
- Methods include get(), set(), add(), remove(), indexOf(), sort(), reverse(), rotate(), and fill()

- Lists dont need a size to be specified in advance like arrays to

Queues

- Ordered sequences of elements
- Accessed via their endpoints (head and tail or front and end)
- Structures for FIFO and LIFO
- Implemented by classes ArrayDeque, ArrayBlockingQueue and PriorityQueue
- Methods include add(), peek(), remove()

Maps

- Map is dictionary with key value pair
- HashMap, TreeMMap, LinkedHashMap
- Methods include put(), get(), containsKey(), containsValue(), and remove()

Resizable Array

- Sotres data in indexable structure
- Complexity
  - Access, delete, insert - Direct O(1)
  - Search - Linear(N)

LinkedList

- Each element links to the next element
- Average complexity
  - Access, search, delete, insert - Linear O(N)

    - This is because to access the third element, you need to start from first element, and take the link to the 2nd etc..
    - Slower in most cases than an array list

Hashtable

- Use a hash function to uniquely assocaite keys with values
- Complexity
    - Access, delete, insert - direct O(1)
    - Search - Linear O(N)
- Complexity of hash table is really good

Iterating over collections in Java

- For loop can be used for indexable collection (ArrayList)

```
ArrayList<String> names;
... add some names
for (int i=0;i < names.size(); i++) {
    System.out.println(names.get(i));
}
... and backwards
for (int i=names.size()-1;i >=0; i--) {
    System.out.println(names.get(i));
}
... and for each
for (String name : names) {
    System.out.println(name);
}
```

- What about not indexable collections
- Java's 'for each' style **`for`** loop works for **`Collection`** types and arrays

```
HashSet<String> names;
... add some names
for (String name : names) {
    System.out.println(name);
}
```

```java
TreeSet<String> ts = new TreeSet<>();

ts.add("Test1");
ts.add("Test2");

Iterator<String> iter = ts.iterator();

while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

- Can also use the iterator object alongside the hasNext() to loop through collections

Convert Array into ArrayList

Sorting Collections

```
public void StringSorts(){
   String[] strings = {"do","re","mi","fal","sol","la","te"};
   ArrayList<String> list = new ArrayList(Arrays.asList(strings));

   System.out.println("Unsorted");
   for(String s : list){
      System.out.println(s);
   }

   Collections.sort(list);

   System.out.println("Sorted");
   for(String s : list){
      System.out.println(s);
   }

}
```

Generics

- Types that accept another type as a parameter
- Buy using generics, it is possible to create classes that work with different data types

Designing Subtypes

- Well designed subtypes are supposed to obey the substitution principle: subtypes should be usuable by just following the supertypes specification

- Signature rule - subtypes must have signature compatible methods for all of the supertypes methods
- Methods rule - Calls to the subtype method must behave like corresponding subtype methods
- Properties rule - The subtype must preserve all of the tprovable properties (e.g. invariants) of the supertype)

Software Patterns and Refactoring

- Gang of Four - Design patterns
  - Creational
    - Managing the creation of objects; flexibility, constraints
    - Abstract Factory, Builder, Factory Method, Singleton
  - Structural
    - Frequently providing a more workable interface for client code
    - Adapter, Bridge, Composite, Decorator, Facade, Proxy
  - Behaivoural
    - Managing common interactions
    - Chain of responsibility, command, interpreter, mediator, observer, state, strategy, template method, visitor

3 Important Patterns

- Factory method

    - Define an interface for creation of multiple similar objects
    - Allows creation of objects without exposing the creation logic
    - Creating classes of objects that are similar but different (share interface or superclass)
- Singleton
    - Class only has one instance (restricts the instantiation of a class to one instance)
    - Provide a global point of access it
    - E.g. window managers, print spoolers, filesystems, logging systems
- Observer
    - One-to-many dependency between objects
    - When one object changes state all dependents are notified and updated automatically

Factory Method - Example

```java
public enum Coin {
    HEAD,
    TAILS;
}

public class CoinFactory {
    public static Coin getCoin(char c) throws CoinException{
        switch(Character.toUpperCase(c)) {
            case('H') :
                return Coin.HEAD;
            case('T') :
                return Coin.TAILS;
            default :
                 throw new CoinException();
         }
     }
}
```

```java
public class PetFactory {
   public static final String DOG = "DOG";
   public static final String CAT = "CAT";
    public static Pet getPet(String petType) throws PetException{
        switch (petType) {
            case DOG: return new Dog();
            case CAT: return new Cat();
            default:
              throw new PetException("Invalid pet type specified");
        }
    }
}
```

Singleton - Example

```
public class Singleton {
   // Protected constructor is sufficient to suppress unauthorized
   // calls to the constructor
   protected Singleton() {}

   /**
    * SingletonHolder is loaded on the first execution of
    * Singleton.getInstance() or the first access to
    * SingletonHolder.INSTANCE, not before.
    */
   private static class SingletonHolder {
     private final static Singleton INSTANCE = new Singleton();
   }

   public static Singleton getInstance() {
     return SingletonHolder.INSTANCE;
   }
 }
```

```
public static void main(String[] args) {
     Singleton s1 = Singleton.getInstance();
     Singleton s2 = Singleton.getInstance();
}
```

Singleton Pattern:

- Ensure only one instance of class is created
- Provide global point of access to the object
- Allow multiple future clients to access the instance without affecting previous clients
- The constructor is made protected

    - Protected means that the member can be accessed by any class in the same package and by any subclasses even if they are in other packages
- For example, if you have a database, having multiple connections to it could be problematic. This is why you would want to use the singleton design pattern.
- Instead of using global variable in class, we want to keep details out of the specification incase things change in the future.

Observer Pattern:

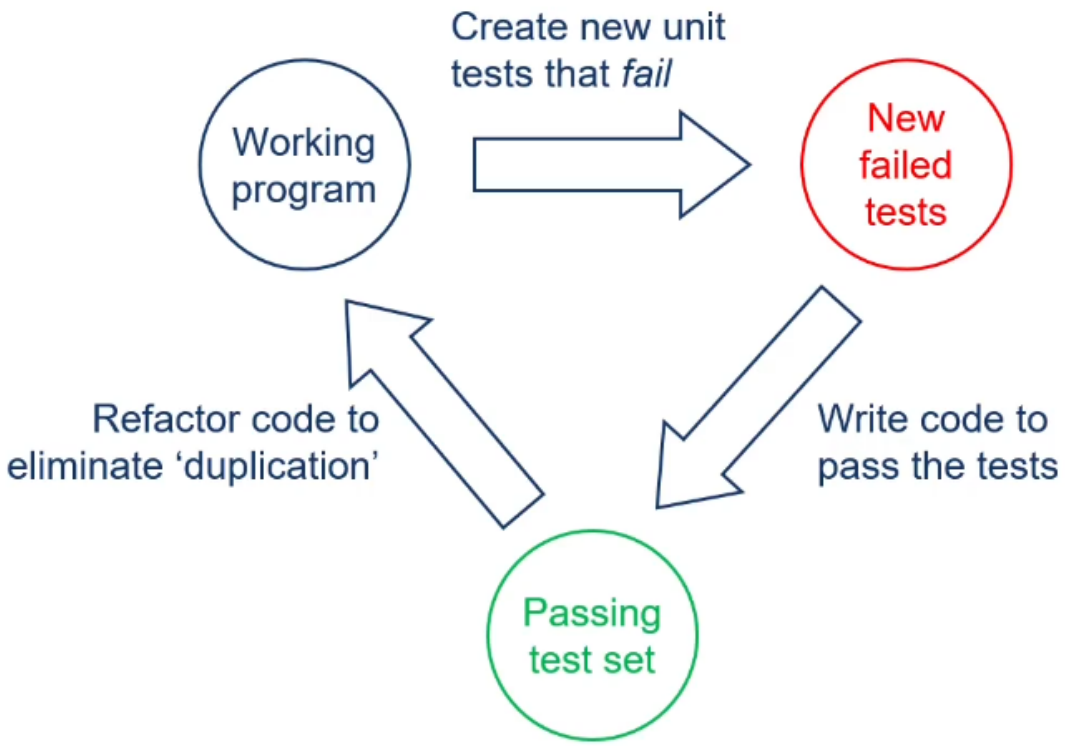
- Common problem: synchronising state and/or responses across a collection of objects in response to an event
- When there are changes in the state of the Subject, we keep the observers up to date
- Scenario 1: Weather monitoring system
    - Sensor notes change in temperature (subject)
    - Observers: Client GUIs displaying temp
    - Observers: Weather modelling systems
- A subject has one or more observers
    - Subject notifies the observers when things change
    - This is the magic behind GUI events and responses
        - When button has mutiple add event listeners
        - Has a list of all the listeners and performs some action when needed

Pros and Cons with Observer Pattern:

- Very loose coupling between Subject and Observers
    - But very tight coupling between observers and subject
- New observers can be added withotu changing the subject

- Objects which change the state of the subject may be completly unaware of the observer collection

Test Driven Development Mantra:



What is refactoring?

- Improving code while preserving behaivour
- Small changes that preserve behaivour
- Each transformation does little but together they do a lot
- Less likely to cause system breaks

Decompose Conditional

You have a complicated conditional (if-then-else) statement.

Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
   charge = quantity * _winterRate + _winterServiceCharge;
else
   charge = quantity * _summerRate;
```

http://www.refactoring.com/catalog/decomposeConditional.html

```
if (notSummer(date))
        charge = winterCharge(quantity);
else
        charge = summerCharge (quantity);
```

Extract Method:

You have a code fragment that can be grouped together.

Turn the fragment into a method with a self-explanatory name.

```
void printOwing() {
   printBanner();
   //print details
   System.out.println ("name: " + _name);
   System.out.println ("amount " + getOutstanding());
}
```

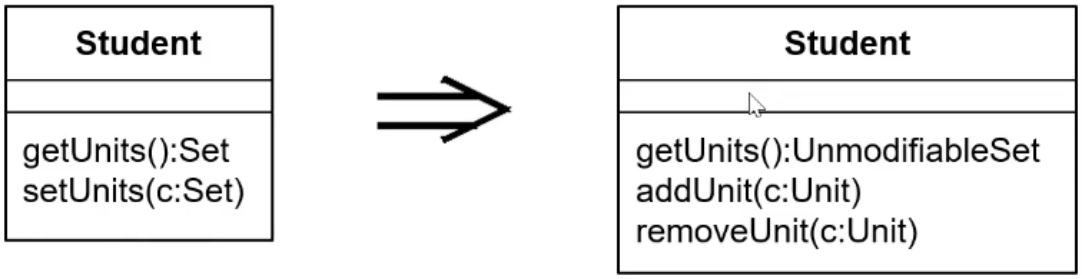
```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails(double outstanding) {
        System.out.println ("name: " + _name);
        System.out.println ("amount " + outstanding);
}
```

Encapsulate Collection:

A method returns a collection.

Make it return a read-only view and provide add/remove methods.



http://www.refactoring.com/catalog/encapsulateCollection.html

Replace Error Code with Exception:

A method returns a special code to indicate an error.

Throw an exception instead.

```
int withdraw(int amount) {
    if (amount > _balance)
        return -1;
    else {
        balance -= amount;
        return 0;
}
```

```
void withdraw(int amount) throws OverdrawException {
    if (amount > _balance)
        throw new OverdrawException();
    _balance -= amount;
}
```

Split Loop:

You have a loop that is doing two things

Duplicate the loop

```
void printValues() {
   double averageAge = 0;
   for (int i = 0; i < people.length; i++) {
      averageAge += people[i].age;
   }
   averageAge = averageAge / people.length;

   double totalSalary = 0;
   for (int i = 0; i < people.length; i++) {
      totalSalary += people[i].salary;
   }

   System.out.println(averageAge);
   System.out.println(totalSalary);
}
```

- When you have unit tests, you are able to refactor confidently because you can see if the change has broken your existing code or not.

Patterns from GoF

Abstract Factory

- Provide an interface for creating families of releted or dependent objects without specifying their concrete classes

Builder

- Seperate the construction of a complex object from its representation so that the same construction process can create different representations

Factory Method

- Define an interface for creating an object, but let subclasses decide which class to instantiate.

Prototype

- Specify the kinds of objects to create using a protoypical instance, cand create new objects by copying this prototype

Singleton

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

Adapter

- Convert the interface of a class into another interface clients expect

Bridge

- Decouple an abstraction from its implementation so that the two can vary independently

Facade

- Provide a unified interface to a set of interface in a subsystem

... many more

File Access

Java.io

- API for reading and writing data.
- Contains classes for eading and writing to various things, including external files
- java.nio contains more classes of this nature

java.io classes

- File - represents an abstract pathname
- FileInputStream - subclass of InputStream, used for reading data from a file

Java's char type

- A char in Java is a 16 bit value containing a Unicode character in the basic multilingual plane
- Essentially a design flaw

Garbage Collection

- Java runs a garbage collector to free up objects that are no longer reference anywhere.
    - This means that as a programmer, you do not need to worry about destroying an object you are finished with
- However the garbage colelctor runs on its own time, not yours. Which means that an object can stay around for some time after it is no longer needed without being destroyed
- If the object holds resources, this is a problem, hence the need to either close() these objcts, or allocate them within a try with resources statment

Serialization

Converting data into binary so it is smaller size for storage or transmission

Important of file format specifications

- The amount fo care that needs to go into defining a file format depends on what the files are used for:
  - Interoperability - When files need to be used by other applications, there is usually an existing file format specification that needs to be carefully adhered to.
  - Future proofing - Is it important that files created by this application continue to work in it when future versions come out
  - Human readability - Is it necessary or useful for others to be able to look at or modify your files in e.g. a text editor?
  - Performance / space efficiency

- Serializable tuypes are indicated by implementing the java.io.Serializable interface
- A class that implements this interface can have objects of it serialised into a stream of bytes, suitable for storing as a BLOB in a database, in a file or sending over a network connection. The bytes can later be deserialised into an object and used again.
- Also known as marshalling

When not to use automatic serialization

- Your class contains explicity references that are not serialisable (e.g. open database conenction, file stream)

    - Java will still atempt to serialise the class but not serialisable fields will not be stored, and will be default/noarg-constructed during serialization
- Your class contains implicit references to external things that reflection cannot detect
    - E.g. an int field that references an entry in a HashMap object that is necessary for your object to function, but the hashmap is not explicity referenced
- Your class contains references to external objects that should not be duplicated for every instance

Benefits of Serialization

- No need to define a file format, just implement serializable
- Less chance of making mistkaes; omitting fields or having particular objects not be stored correctly because an insufficiently expressive underlying type was used to store them
- Javas Serialisation API is well established and may do a better job than your own code

Drawbacks of serialization

- Serialization is rarely the most efficient way of representing an objects contents in byte form; large amounts of overhead
- Does not allow you to follow a precise file format, which is bad for interoperability with other programs
- Serialised data is not human readable or human editable
- Serialisation can have security concerns when deserialising untrusted data

Network Connectivity

Sockets

- When direct communication between programs is required, the most popular solution is sockets
- Sockets are an abstraction over some kind of communication medium (e.g. TCP IP) to allow processes to talk to each other
- Its like reading/writing from/to a file except instead of writng to disk you are writing data to be read by another process
- Java provides sockets API for allowing communication through TCP
    - Key classes
        - Socket
        - ServerSocket
        - DatagramSocket

Closing sockets

- Like files, sockets are close()able
- Make sure your close() them
- ServerSockets in particular need to be closed properly; server ports are exclusively held by one application.
- You can use a try with resources block to ensure that the connection is closed

Programs, Processes and Threads

Programs

- Executables, not yet runnng

Processes

- The executing program
- Has access to memory and file resource
- Runs on CPU

Threads

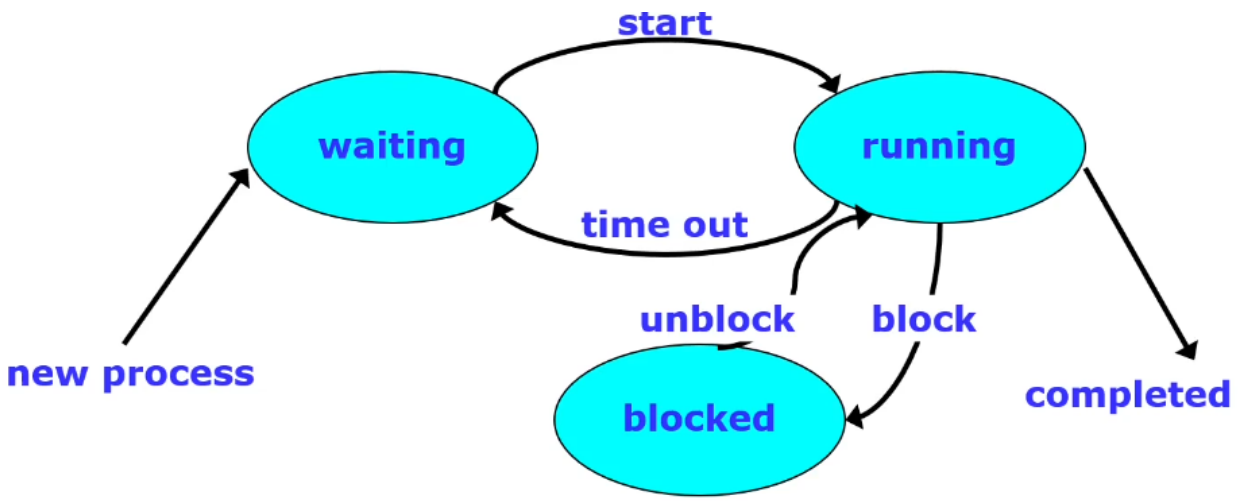
- Seperate units of a parent process
- Share resources of parent process

Why Threads?

Two main reasons

- Asynchronicity
  - Threads allow you do perform multiple tasks at the same time, so tasks that take a long tiem do not interrupt the usual execution of the program
- Parallelism
  - Bu putting multiple threads to work on a problem (or many threads to work on many problems), those problem(s) can be finished faster

Thread Lifecycle



The Thread Life Cycle – threads make transitions between these states

- Threads are dangerous
- Threads introduce asynchronous behaivour
  - Could alter data at same time
- Java uses the monitor approach
- Control mechanism for thread safety
  - Due to C.A.R Hoare
  - When a thread enters the monitor, all other threads must wait until the occupying thread exists
  - This type of thread safety is provided by java through the synchronized keyword

Threads in Java

- java.lang.Thread
  - Class for creating and running threads
- java.lang.Runnable
  - interface implemented by classes that run on threads
- Remember that java does not support multiple class inheritance
- The Runnable interface is thus the only real approach
- Can choose to extend the Thread class
  - Overide the run method

- **`void run()`**
  - subclass must override this method
  - if **`Thread`** created with **`Runnable`** object that object's **`run()`** is called
- **`void start()`**
  - causes thread to begin execution
- **`void yield()`**
  - makes thread pause to allow another thread to execute
- **`void sleep(long milliseconds)`**
  - makes thread pause for specified time
- **`void join()`**
  - waits for thread to terminate
- **`boolean isAlive()`**
  - tests if thread is still running

- Do not call run(),
- Call start()
- This activates the thread and invokes the run() method

- **`stop()`**, **`suspend()`** and **`resume()`** are deprecated
  - Do not use them
  - `suspend`ing a locked thread means the lock won't be released
  - can leave program in an inconsistent state (see later)
- Allow threads to terminate naturally or use a wrapper with a guard pattern
  - That way we are sure that the thread has completed its 'current work'
  - We can then safely stop

- To stop a thread safely, embed the action in a loop
- Carefully control the loop with a `stopFlag`
  - Make sure that this variable is independent of other threads
  - Check the value frequently
- Have a method that sets the flag
  - Then we can control the termination
  - But don't call it **`stop()`**
  - **`end()`** is a reasonable choice

Service Threads

- Service threads of daemon threads
- Contain never ending loop
  - Receive and handle service requests
- Useful in client/server programming
  - Server listens using a daemon thread for each port

- When request comes, daemon thread responds
- Creates/calls another thread to do the work
- Worker thread terminates naturally
- Daemon thread continues monitoring port

- The JVM has a number of these

- `void setDaemon(boolean on)`
- `boolean isDaemon()`
- Terminates only when the program terminates
  - JVM detects that all non-daemons have finished
  - This means the program has finished
  - So the JVM kills the daemons as well.
  - Allows program to terminate

Scheduling Tasks

- The thread based approach
  - Create a thread and put it to sleep for desired time
  - Then do the tasks
- Or the time based approach
  - Use TimerTask and Timer from java.util package
  - TimerTask implements Runnable
  - Timer runs a TimerTask object on a JVM background thread

Thread Safety

- Thread safe variables belong to one thread alone
- Different threads have their own copy
- Local variables are inherently thread safe
- Instance and static variables are not inherently thread safe

- Instance variables
  - Specific to each instance of a class
  - Shared by all threads that access that instance
  - Thread safety depends on how program is structured
  - Synchronization often provides the necessary safety
- Class variables
  - Declared static
  - Shared by all instances of the class on all threads
  - Only one copy
  - Not thread safe in your wildest dreams.
  - Use `ThreadLocal` or `InheritableThreadLocal` type for declaration

- Programs must be written to be thread safe
- If we mismanage threads we can end up with:
  - Race conditions (processes racing against each other)
  - Deadlock
  - Livelock

- Starvation

Race condition

- Threads race to modify the value
- The outcome depends on whcih thread gets to the value first

- One way is to qualify shared variables with **`volatile`** keyword
  - The volatile keyword in Java is very different to the volatile keyword in C/C++ it was based on
  - Volatile in Java means:
    - Reads and writes involving this variable will be **atomic** (that is, one thread won't see a variable that's only been half-written by another thread)
    - Accesses to the variable will be surrounded with memory barriers ensuring the value appears the same in every thread

Thread 1	Thread 2
`bankAccount += 5000;`	`bankAccount -= 5000;`

- `volatile` doesn't help here as 'bankAccount += 5000' is effectively at least two operations- the addition and storing the result back in bankAccount.
- If bankAccount was originally 5000, after these two threads finish it could be any of:
  - 0
  - 5000
  - 10000

Atomics

- The **`java.util.concurrent.atomic`** package contains atomic classes and operations carried out on these will happen instantaneously (from the perspective of other threads)
- If 'bankAccount' is an AtomicInteger, for example, the threads can carry out these operations:

Thread 1	Thread 2
`bankAccount.addAndGet(5000)`	`bankAccount.addAndGet(-5000)`

- After these two threads finish, the value of bankAccount will be the same as its initial value, no matter what order they complete in

Thread Safety

Deadlock

- Blocked waiting for a resource held by another blocked thread

Livelock

- Thread acts in response to another thread. makes no progress but is not blocked
- Note that thread states can and do change - we just dont make any progress with any of them
- Corridor dance for the excessively polite

Starvation

- Some threads allowed to be greedy, others are starved of resources making no progress.

Synchronization

- Synchronization is about allowing only one thread access at a time...
- Sets a lock on a method or object
- Dont over use it
    - Lock only code requiring exclusive access to the object or class
    - Synchronizing code places an overhead on execution time
- Request is automatic and satisfied if lock is available
    - Lock is always available unless some other thread has requested and been granted the lock
    - When a synchronized method terminates the lock is released
- Levels of synchronization
    - None
    - Class methods only
    - Class and instance methods

Thread Communication

- Multiple threads need to communicate
- Threads can notify each other when conditions change
- Tell each toher to wait or stop waiting
- Object provides

- `void wait(),`
  `wait(long time),`
  `wait(long time, int nano)`
- `void notify()`
- `void notifyAll()`

- wait() puts currrent thread into a blocked state
- Can only call wait from a method which has a lock
  - ie. it is synchronized
- Calling wait releases the lock
- If not time period is specified, it waits until notified by another thread.
- notify() sends wake up call to a single blocked thread
- notifyAll() sends wake up call to all blocked threads
- These methods are final in Object
  - cannot be overridden
- If called when thread does not have a lock
  - IllegalMonitorStateException

- using **`wait()`**
  - call wait when code must wait for a certain condition to be reached
    - waiting for input
    - waiting for numeric threshold to be reached
  - this saves CPU cycles allowing other threads or processes do some work while blocked
- **`notify()`** could be called for many reasons,
- Putting wait in a loop and testing a condition is safest

```
while (condition) {
   wait()
}
```

- Using **`notify()`**
  - Cannot specify which thread to wake
  - JVM decides which thread to wake
    - a single thread
  - Useful when all threads are waiting for same condition
  - More efficient than **`notifyAll()`**
  - Deadlock is possible

- using **`notifyAll()`**
  - Wakes up all waiting threads
  - Use when threads are waiting on different conditions
  - Only one unblocked thread can obtain the lock

Threads and Swing

- Swing (& other GUI libraries) are not thread-safe
- Events are processed on the EDT or event-dispatch thread
- Swing provides methods in the SwingUtilities class to ensure that this takes place
- Time consuming tasks should be on background thread handled by a SwingWorker

java.util.concurrent.ForkJoinPool

- JDK 1.7 introduced a fork and join model for recursive threading – especially for multiprocessing environments
  - Fork the process into threads for lightweight asynchronous multithreading
  - The threads can in turn fork to create additional threads
  - Wait for them to terminate
  - Continue with the merged results
- This model makes use of thread pools to minimise overhead from creating additional threads

java.util.stream.Stream

- JDK 1.8 introduced **streams**, designed for efficiently and flexibly applying operations to objects, including in parallel
- Has helper methods in java.util.Collection (stream() and parallelStream())
- Easiest way to parallelise independent operations over a collection in Java
- Parallel convenience methods also available in Arrays
  - `Arrays.parallelSort` etc.

Devops

Software Developers

- Gather requirements, design, program and test software in consultation with clients and stakeholders to ensure the correct software is being created

IT Operations

- Administrate the servers on which the software runs, keeping operating systems software, web server software etc. up to date, are usually the people that get the phone call at 3am if the site goes down

Agile is about bridging the gap between customers and developers

DevOps is about bridging the gap between developers and operations

Devops Pillars of Success

- Reduce organisational silos
- Accept failure as normal
- Implement gradual changes
- Leverage tooling and automation
- Measure everything

Reduce organisational silos

- Organisational silos are described as the phenomenon in which different parts of the organisation are isolated and do not colloborate or share information
- The traditional adversial relationship between developers and operations that DevOps attempst to break down is one example of organisational silos
- Organisational silos means less efficiency as the organisation is effectively working against itself

Accept failure as normal

- Normal part of devops

Implement Gradual Changes

- Agile promotes continuous integration
- DevOps promotes continuous delivery
- Small changes carry less risk and can be implemented and deployed faster

- Contrast with traditional versioned software deployment where many bug fixes and additional features tend to be batched into version releases, a DevOps project may receive a constant steady stream of tiny updates, most invisible to end users

Leverage Tooling and Automation

- Menial tasks anywhere along the development-deployment pipleine are automated, even when they are simple or fast enough to be performed manually.
- Automation reduces the risk of human error and ensures devops engineers spend their time on knowledge work instead

Measure Everything

- Another agile practise that devops extends
- keep track of progress and bugs, but also areas like site performance, uptime, revenue coming in and so forth.
- Devops engineers use this infomration to get an unprecedented picture of how to most effectively target their develoment efforts

Continuous Integration

- Varies in implementation but the essential point is that a build is generated whenever
    - There is a change to a defined repository
    - At fixed intervals (not just everyday)
- Jenkins

Continous Delivery

- Means the software is always in a state where it can be deployed
- Key differences with Continuous Integration
  - Continuous Integration is about making sure the software can build and run at any time, not making sure it can be released to users at any time. Actual releases still happen based on a given deadline with additional release engineering work taking place
  - Continuous Delivery requires that the software be deployable to users at any time. This makes the timing of deployment a business decision, instead of both a business and technical decision
- Continuous delivery requires developers to prioritise working software over developing new features
- Continuous delivery is impossible with organisational silos in place
- Heavy emphasis on automation- everything up until the final deployment to production should be fully automatic, and the final deployment should be push-button
- Maintenance and testing of CD scripts becomes vitally important:
  - If build scripts fail, produce corrupt outputs or let bugs through in a CI environment, it is an inconvenience
  - If build scripts fail in a CD environment, it is a showstopper

Build Systems

- Ant
- Maven
- Gradle

Continuous Integration

- Jenkins

- Bamboo
- Github Actions
- Travis CI

Automated Builds

- A build means putting the system together
- Builds are based on a series of steps
    1. Editing source code
    2. Successful compilation of source code
    3. Successful unit tests of the source code
    4. Integration of the object code into an application
    5. Successful (system) integration testing
    6. Deployment preparation

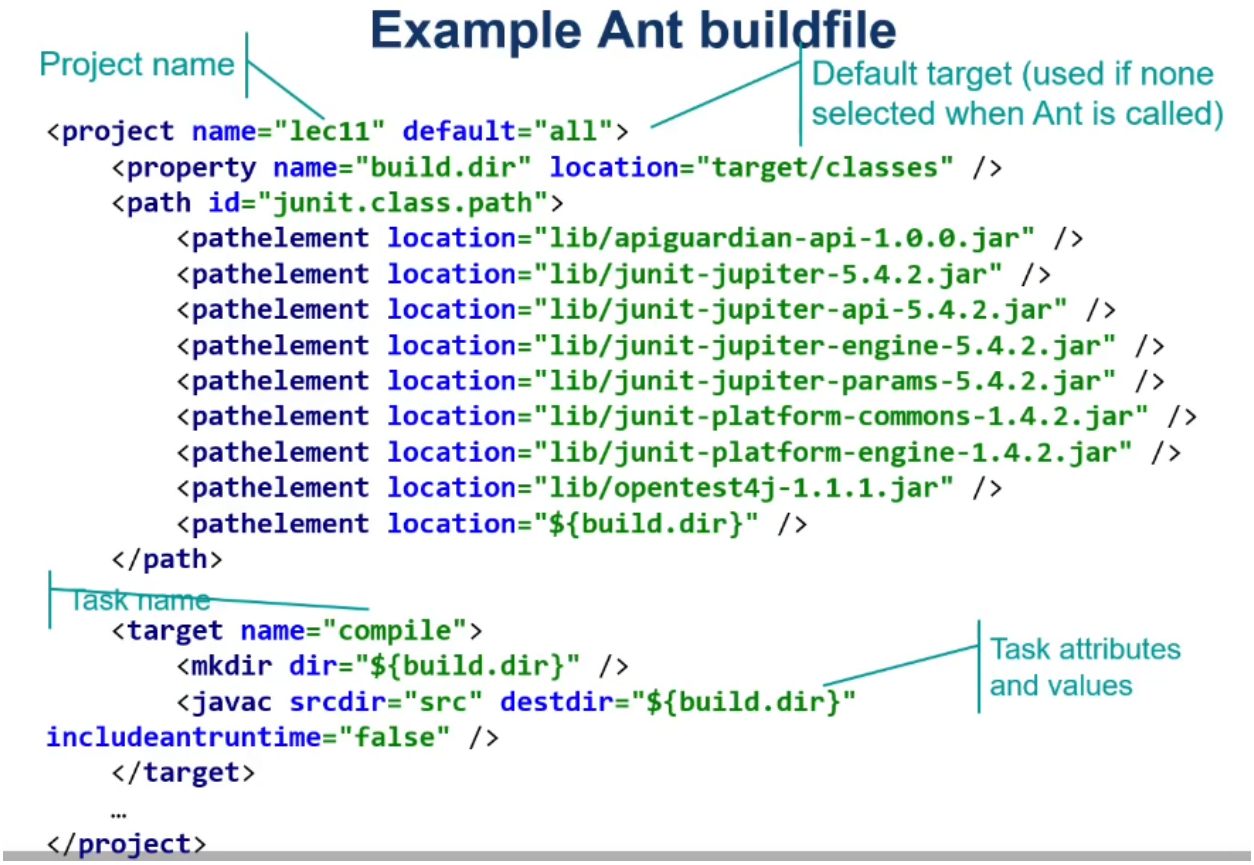
Apache Ant

- Very widley used, pure java, genuinely portable and extensible
- Based on a build file specification
    - build.xml
- Others exist but notable Maven and Gradle

- Ant files

```
<project>
        <properties/>
        <path>
        </path>
        <target>
            <!-- set of tasks -->
        </target>
</project>
```

Example Ant buildfile




```
<project name="lec11" default="all">
    <property name="build.dir" location="target/classes" />
    <path id="junit.class.path">
        <pathelement location="lib/apiguardian-api-1.0.0.jar" />
        <pathelement location="lib/junit-jupiter-5.4.2.jar" />
        <pathelement location="lib/junit-jupiter-api-5.4.2.jar" />
        <pathelement location="lib/junit-jupiter-engine-5.4.2.jar" />
        <pathelement location="lib/junit-jupiter-params-5.4.2.jar" />
        <pathelement location="lib/junit-platform-commons-1.4.2.jar" />
        <pathelement location="lib/junit-platform-engine-1.4.2.jar" />
        <pathelement location="lib/opentest4j-1.1.1.jar" />
        <pathelement location="${build.dir}" />
    </path>
    <target name="compile">
        <mkdir dir="${build.dir}" />
        <javac srcdir="src" destdir="${build.dir}"
includeantruntime="false" />
    </target>
    …
</project>
```

Maven

- Not as much configuration as Ant
- Still configured via an XML file

Continous Integration

- Continuous integration (CI) is thus a scheduler for the extreme version of he daily build

- Varies in implementation, but the essential point is that a build is generated whenever
- CruiseControl was the original, other tools like Jenkins followed but most CI tools are in house scripts
    - CruiseControl gives build history of project via web browser

Jenkins

- Modern CI suite still used today
- Has a GUI interface
- Support for many plugins
- Runs as a service, configured via browser

Git Hooks, Github Actions etc.

- Git can be configured to run scripts in response to events
- Github and Bitbucket also provide functionality for performing actions in response to changes to a repository
- Highly configurable, but specific to SCM provider (compared to Jenkins which is more generic)

Travis CI

- Another online service, popular with open source projects
- Also configurable via .yml files
- Can be used with a variety of repository providers
    - Github
    - Bitbucket

- Gitlab
- Assembla

Summary

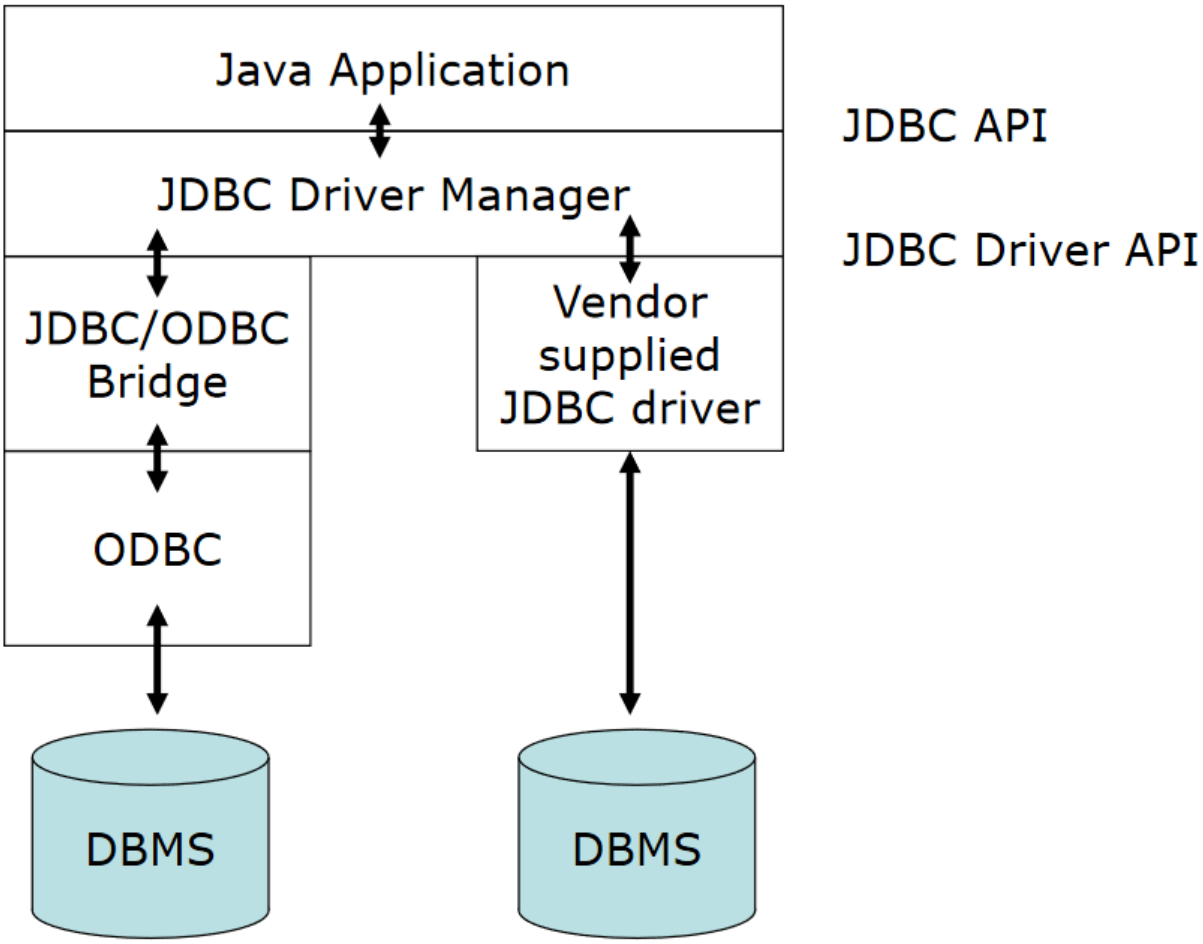
- Build process
  - Needed to handle changes in large systems (esp. with teams)
- What to build
  - Ant – Relies on dependencies
  - Ant can also be used for testing, documentation and deployment
  - Maven - Does much of the same, less configuration
- When to build
  - CI scripts and tools (Jenkins, Git hooks etc.)
  - Continuous integration = integrate when there has been a successful check in

---

JDBC

- Java database connectivity (JDBC) is an application programming interface (API) which defines how a client may access a database
- The whole idea is to have a vendor-independent data API

    - Learn one API instead of half a dozen
- Need a driver for each DBMS
    - Driver handles details of connecting to DBMS
- API provided in java.sql package
- Vendor specific drivers are generally faster and better



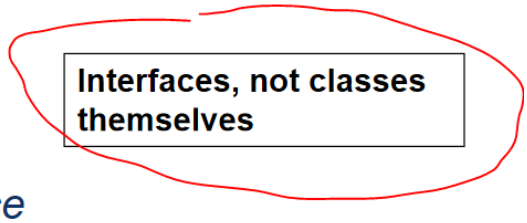
Connections

- Server side apps can benefit from connection pools
    - Creating connections is expensive
    - Keeping connections open reduces overhead

- If a pool of connections are kept open, when one is requested, the application returns one not in use from the pool
- When the conneciton is finished, it is returned to the pool

Statements

- When a **`Connection`** is established
  - use it to execute **`Statements`**
- **`Statement`** *interface*
  - represents basic SQL statement
- **`PreparedStatement`** *subinterface*
  - represents a precompiled SQL statement which can offer improved performance, but mainly convenience
- **`CallableStatement`** *subinterface*
  - allows JDBC programs access to stored procedures within database itself, offers more improved performance



Statements

- **Statement**

  Factory method

  – use `Connection.createStatement()` to get one

```
Statement st = connection.createStatement();
```

- For SQL commands
  - `boolean execute(String sql)`
  - `st.execute("DROP TABLE names");`
- For SQL queries
  - `ResultSet executeQuery(String sql)`
  - `rs = st.executeQuery("SELECT * FROM names");`
- For SQL update queries

  Database Integrity? ☺
  - `int executeUpdate(String sql)`
  - `rows = st.executeUpdate("UPDATE names SET age=21 WHERE name='Colin Fidge'");`

- **PreparedStatement**
  – Pre-compile SQL statements at object creation time rather than at execution time
  – Use `?` symbol to indicate parameters for the statement and fill the values in later
  – see NamePlayPrepared.java

```
PreparedStatement pstmt = connection.prepareStatement(
      "INSERT INTO people (name, age) VALUES (?, ?) ");

pstmt.clearParameters();
pstmt.setString(1, "Kevin Rudd");
pstmt.setInt(2, 56);
pstmt.executeUpdate();
```

Statement – Used to execute string-based SQL queries. **PreparedStatement** – Used to execute parameterized SQL queries. 10 Aug 2020

- **`CallableStatement`**
  - If the DBMS supports stored procedures (MariaDB does, SQLite3 does not), the **`CallableStatement`** interface provides the means to use them

```
CallableStatement cstmt = connection.prepareCall("call proc(?,?)");
```

  - The SQL statement can contain **`?`** placeholders for parameters within the statement like **`PreparedStatement`**
  - See NamePlayCallable.java

- Batch processing
  - While **`PreparedStatements`** can improve database access times, if there are 10,000 insert statements executed, there is a lock and release carried out in the database for each insert
  - **`stmt.addBatch(String sql)`** can be used to add the SQL statement in the parameter to a batch of statements, then
  - **`stmt.executeBatch()`** carries out all SQL statements
  - batches spanning multiple tables are not allowed
  - a failure in the middle of a batch will cause all remaining SQL statements to fail
  - see Batch.java

- It's all about Commitment…
  - Use **`setAutoCommit(false)`** to prevent part of the transaction from being automatically committed to the database
    - by default, auto commit is enabled
  - Use the **`commit()`** method to explicitly carry out any outstanding statements
  - if a problem occurred, use the **`rollback()`** method to restore the database table to its previous state
  - See savepoints from JDBC 3.0 – like breakpoints for transactions

```
Savepoint save = connection.setSavepoint("half done");
.........
// if problem detected
connection.rollback(save);
```

ResultSets

- **`ResultSet`** is used to store the result of an SQL query
- A two-dimensional table of all rows in the database that matched the criteria of the query
- Use get methods to get data from the ResultSet
  - **`getXXX(yyy)`** where **`XXX`** indicates the data type and **`yyy`** and is either the column number (indexed from 1) or the column name
- Navigation analogous to list iterator over a collection
- JDBC 1.0 allowed forward movement only and the **`ResultSet`** could not be manipulated
- JDBC 2.0 introduced scrollable and updateable **`ResultSets`**

- Scrollable and updateable **`ResultSet`**s require extra parameters in the **`createStatement()`** method

```
Statement st = connection.createStatement(
               ResultSet.TYPE_SCROLL_INSENSITIVE,
               ResultSet.CONCUR_UPDATABLE);
```

- Can use **`first()`**, **`last()`**, **`next()`**, **`previous()`**, **`absolute(int)`**, **`relative(int)`** etc. methods to navigate through the **`ResultSet`**

TYPE_SCROLL_INSENSITIVE means that the resultset can be navigated (scrolled) both forward and backwrads. Insensitive to changes when resultset is open

- Use update methods to update the ResultSet
  - **`updateXXX()`** where **`XXX`** indicates the data type, followed by
  - **`updateRow()`** – writes updates back into the database
- BUT these change the data in the database!!!
  - Same with **`deleteRow()`** and **`insertRow()`** methods
  - Updating and scrolling are useful features of JDBC 2.0 but the spec does not require driver vendors to support them
- Information about the underlying driver can be obtained using **`DatabaseMetaData`** interface methods

Large Objects

- Some data does not fit in standard data types
  - Pictures, media files, archives
  - BLOBs offer an alternative to serialization to a file
- Use the appropriate large object
  - BLOB = binary large object
  - CLOB = Character large object

- JDBC provides BLOB and CLOB support
  - specific vendors provide a number of BLOB and CLOB types for different sized objects
  - MySQL provides:
    - A16 bit BLOB field can store a 65,536 byte object
    - A 32 bit BLOB allows 4 GB of storage
  - BLOBs are converted to byte arrays before being stored
  - BLOBs increasingly important in cloud services

Summary

- Databases are used to manage large structured data
- So Java needs to interact with DB either through:
  - Vendor specific JDBC
  - ODBC:JDBC Bridge
- Java interacts with DB via statements
  - Statement interface (basic)
  - PreparedStatement subinterface (produced at compile time)
  - CallableStatement subinterface (calls statements in the DB)
- Statements return ResultSets
  - Can iterate or update
- Large objects handled by BLOBs and CLOBs