Java Notes

Naming Coventions

Camel Naming convention, second words must be capitals
Final modifier means the value of that variable cannot be changed.

Four Categories for Primitive Types

Integer Types

Each memory slot holds 8 bits (1 byte) so in order to store an interger we need to allocate 4 slots of memory. If we wanted to store an array of 4 integers we would need to allocate 16 slots of memory in continuous slots.

- **byte** (-128 <> 127) – 8 Bits
- **short** (-32768 <> 32767) – 16Bits
- **int** (-2,147,483,648 <> 2,147,483,647) – 32Bits
- **long** (……. You get it 2^63) – 64Bits

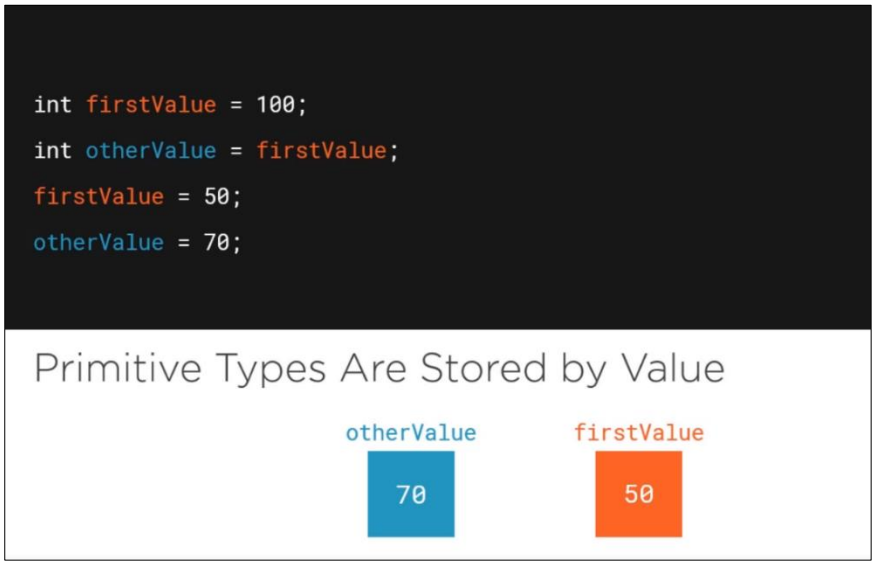
Floating Point Types

- **Float** () – 32 Bits
- **Double** ()- 64 Bits

Chat Types

Stores a single Unicode character
Literal value placed between quotes (Can use Unicode etc)

Memory Allocation



Method Overloading

- Can have same name as another method, just needs a different parentheses

Control Flow Statements

- While
  - Will continue unless you add in a statement
  - **Continue** skips rest of code
  - **Break** will stop the loop

Reading User Input

- Scanner (java built in class) allows you to read in user input
- To handle enter key issue add an empty nextline() call

Object Orientated Programming

Classes

- Enable you to have a powerful user defined type
- Public (Unrestricted access to that class)
- Private (No other classes can access it)
- Protected (Allows classes in that package to access it)
- Allow us to create variables that are accessible anywhere in the class (Member Fields)
  - Specify access modifiers
  - General rule is to use private (Encapsulation (hides fields/methods from public access))
- To distinguish between fields in method type “This” to refer to a field in a class
- **Contructors**
  - You can set all the parameters to you want in one hit
  - Created automatically by java
  - You can have multiple contructors that contain a diff number of params.
  - The other constructors will call the main one.

Objects

- State
  - Age, Number of eggs etc
  - Software = fields
- Behaviour
  - Printing something, outputting sound etc
  - Methods

This vs super

- **Super** is used to access/call the parent class members
- **This** is used to call the current class members
- **Constructor chaining**
  - Don’t repeat the same code
  - Make constructors call each other

Method Overriding vs Overloading

Method overloading

- Means proving two or more separate methods in the class with the same name but different parameters
- Can also be treated as overloaded in a subclass of that class
- Follow the rules:
  - Same name
  - Different parameters

Method Overriding

- Means defining a method in a child class that already exists in the parent class with the same signature
- Known as **Runtime Polymorphism**
  - Because that method that is going to be called is decided at runtime by the JVM
- Follow the rules:
  - Same name and arguments
  - Return type can be a subclass of the return type in the parent class
  - It cant have a lower access modifier
  - Constructors and private methods and final methods cannot be overridden
  - Only inherited methods can
  - A subclass can use super.MethodName(). To call the superclass version of an overridden method.

Static Methods

- Declared using a static modifier
- Static methods can access instance methods and instance variables directly.
- The are usually used for operations that don't require any data from an instance of the class (from 'this')
  - Cant us **this** keyword
- Whenever you see a method that does not use instance variables (Objects) that method should be declared as static

Instance Methods

- Belong to an instance of a class
- To use an instance method we have to instantiate the class first by using the "new" keyword
  - Can access directly:
    - Instance variables
    - Static methods
    - Static variables

Should a method be static?

- Does it use any fields (Instance variables or instance methods)
  - **Yes**
    - Be an instance method
  - **No**
    - Probably a static method

Static Variables

- Also known as static member variables
- Every instance of that class shares the same static variable
- If changes are made to it, it will effect all other instances
- Not used very often but can be useful
    - Declare a scanner as a static variable, that way all methods can access it directly.

Instance Variables

- Don't use the **static** keyword
- Instance variables are also known as fields/ member variables
- Instance variables belong to an instance of a class

Composition – Has a relationship with it (Parts of the greater model)

- Doesn't extend but does require.
- The monitor has a resolution class as part of its object
- A new class that has classes in its constructor
    - E.g. a PC that has a monitor, case, motherboard classes
    - Multiple inheritance (only extend one)

Encapsulation

- Preventing class/code from accessing the innerworkings of a class
- Give you more control/ change things without breaking code
- Fields within a class aren't accessible
- Making things less abusable

Polymorphism

- Allows an object to take on many forms.
- Only useful making classes within one java file if they are small and compact/not reused
- Assigning different functionally depending on what method is generated
- Movie (Loop through different movies)
- Complex functionality can be built into it.

4 (5) Pillars of OOP

1. Inheritance
    a. (**extends** keyword) allows you to use the state behaviour of the other
    b. Super – call the constructor from the class we're extending from
        i. **Super**.**function**(will call the main classes method)
    c. You can add extra parameters to the inherited constructor
    d. Overide – will override a method in the super class
2. Polymorphism
3. Encapsulation
4. Composition
5. Abstraction

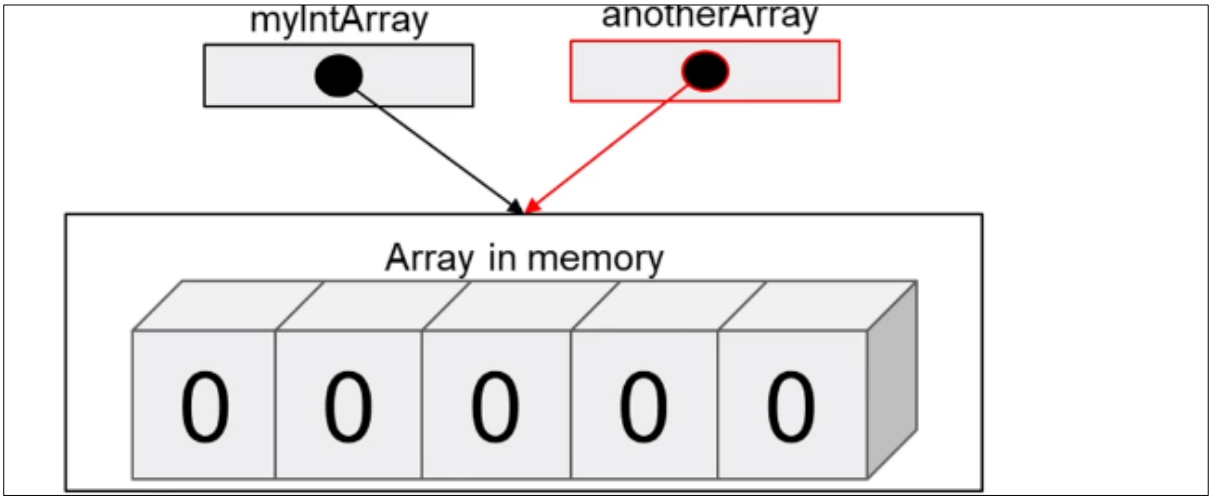
Arrays

- An array is a data structure that allows us to store multiple values of the same type into a single variable
- Indexed from 0.. n-1

Reference Types vs Value Types

- If both variables point to the same object they'll but update the same value

```
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;
```



Autoboxing

- Converting a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:
    - Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
    - Assigned to a variable of the corresponding **wrapper class**.

Unboxing:

- Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example conversion of Integer to int. The Java compiler applies unboxing when an object of a wrapper class is:
    - Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
    - Assigned to a variable of the corresponding **primitive type**.
    -

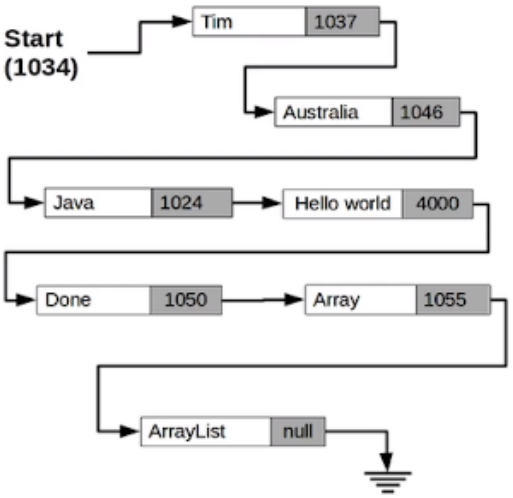
LinkedList

- Java can calculate the address of other integers
- Not efficient to add /remove items from an array List if it has lots of members as other members need to be moved. (Lots of manipulation)
- Each element in the list holds a link to the item that follows it.

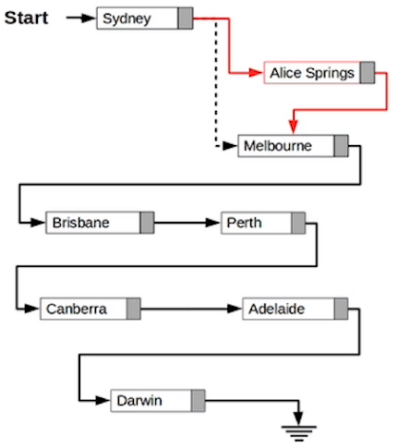
LinkedList

Index	Address	Value
0	100	34
1	104	18
2	108	91
3	112	57
4	116	453
5	120	68
6	124	6

LinkedList



LinkedList



- Make Alice Spring point to Melbourne, Sydney points to Springs.
- If you remove an object java will collect it during garbage collection.

Interfaces

- Doesn't fill methods, just declares them with their name and parameters.
- In general, interfaces facilitate the key concepts in OOP
  like abstraction, inheritance and polymorphism. In addition, interfaces add flexibility and re-usability for software components.
- Interfaces allow you to fix this issue by editing methods within specific classes
- RULES:
  - Fields in an interface are public, static and final implicitly**:**
  - Methods in an interface are public implicitly
  - A class can implement multiple interfaces:
  - The overriding methods cannot have more restrict access specifiers
  - Abstract classes are not forced to override all methods from their super interfaces. The first concrete class in the inheritance tree must override all methods:
  - An interface cannot extend another class
  - An interface can extend from multiple interfaces:
  - An interface can be nested within a class:
  - An interface can be nested within another interface:
  - Methods in an interface cannot be static and final
  - Since Java 8, an interface can have default methods and static methods
  - Functional interface is an interface that has only one method:

Inner Classes

- Refrain from using the same variable names
- This. Refers to inside that specific class
- Must specific the outer class first
- ```
  Gearbox mc = new Gearbox( maxGears: 6);
  Gearbox.Gear first = mc.new Gear( gearNumber: 1, ratio: 12.3);
  ```
- Inner classes can be private if you don't need to access them directly

Abstract Classes

- The **abstract** class states the class characteristics and methods for implementation, thus defining a whole interface.
- Interfaces are purely abstract
- You must determine how the classes are inheriting etc
- Can have constructs/variables
- Methods can have any visibility
- Methods can be defined/contain code
- If an abstract class is subclassed and you don't want to implement all the methods, you must declare that class as abstract
- The purpose of an abstract class is to provide a common definition of a bass class that multiple derived classes can share
  ```

Interfaces vs Abstract Classes

- Use abstract classes when you want to share code among several closely related classes (Animals with fields name, age etc.)
- You want non static or non-final fields
- You want to use public/protected/private modifiers
- Declaration of methods of a class
- Interfaces form a contract between a class and the outside world which is enforced at build time by the compiler

Generics

- Generics add stability to your code by making more of your bugs detectable at compile time.
- Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there.
- Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.
- Arraylist<Integer> etc…..
- Finish off section
- Team <T extends player> Means it will accept any class the extends the class player

Name Conventions

- **Packages**
  - Always lower case
  - Should be unique
  - Use your internet domain name reversed
  - Co.uk.jackghawkins
- **Interface**
  - Capitalised
  - Consider what objects implementing it will become
- **Methods**
  - mixedCased
  - often verbs
- **Constants**
  - All Upper case
  - Separate words with underscored _
  - Use final keyword
- **Variable names**
  - mixedCase
  - meaningful
  - don’t used underscores

Packages

- Programmers can easily determine that classes are related
- It is easy to know where to find the classes and interfaces that can provide the functions provided by the package
- Because the package creates a new namespace, class and interface name conflicts are avoided
- Classes within the package can have unrestricted access to one another
- Java.awt.***** brings everything from that package (*)

Access Modifiers

- Only classes, interfaces and enums can exist at the top level, everything else must be included with one of these
- Public access everywhere
- **Package-private**
    - When you don't specific a keyword
- Protected is visible anywhere within that package

Collections Overview

Binary Search - O(Log n).

- Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found, or the interval is empty.

Collections – List

- Has a variety of useful methods
    - Collections.Max
    - Collections.Reverse
    - Collections.Min
    - Collections.Shuffle

```
Theatre theatre = new Theatre("Hero",8,12);
List<Theatre.Seat> seatCopy = new ArrayList<>(theatre.seats); // shallow copy
```

```
    /* Uses CompareTo Method*/
    Theatre.Seat minSeat = Collections.min(seatCopy);
    Theatre.Seat maxSeat = Collections.max(seatCopy);
    System.out.println(minSeat.getSeatNumber());
    System.out.println(maxSeat.getSeatNumber());
}
```

- You can't use the compare method without doing more work for price and seats etc. Example to follow

Maps

- .put (adds the reference)
- (Key ,Value)
- You can use .containsKey("Method") to prevent duplicates
- If you call sout(map.put("")) it will print null as its just been done
- Loop through map by doing

```
for(String key: languages.keySet()){
   System.out.println(key + " : " + languages.get(key));
}
```

- Standard methods Remove, Replace etc

Immutable Classes

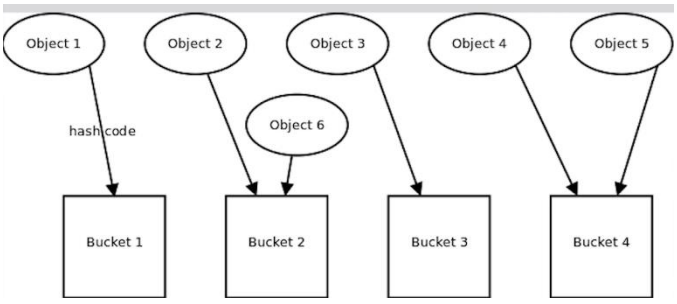
- Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable
    - The class must be declared as final (So that child classes can't be created)
    - Data members in the class must be declared as final (So that we can't change the value of it after object creation)
    - A parameterized constructor
    - Getter method for all the variables in it
    - No setters(To not have the option to change the value of the instance variables

Sets

- The set interface present in the java.util package and extends the Collection interface is an unordered collection of objects in which duplicate values cannot be stored

HashSet

- Java **HashSet** class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java **HashSet** class are
    - **HashSet** stores the elements by using a mechanism called hashing.
    - **HashSet** contains unique elements only.
- Hashsets can contain values that are the same at face value but their java objects do not match. This is where the comparison between Equals() and HashCode() come in.
- When we add an object, its hashcode is compared to the other objects in that bucket. If our new object breaks the rules and has a different hashcode to an object it is equal to it will not be spotted as a duplicate



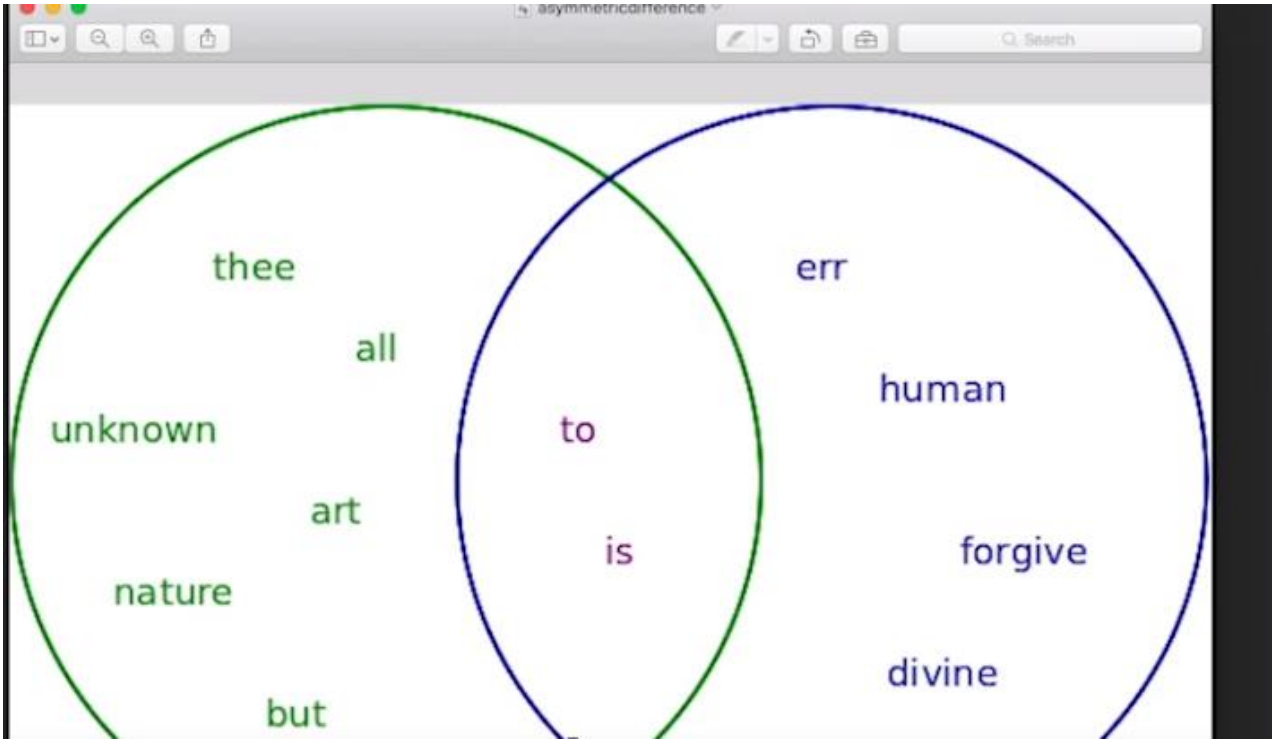
- If two objects are equal then they must have the same hashcode. We need to override the methods in most cases.
- This contains the rules which must be met in order to return true https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object
- If you have subclasses you need to think about how you're going to treat comparisons.
- If there going to be the same object that mark your equals method as final.
- Symmertic & Asymmetric

Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

- `s1.containsAll(s2)` — returns `true` if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
- `s1.addAll(s2)` — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
- `s1.retainAll(s2)` — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
- `s1.removeAll(s2)` — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

- This allows you to manipulate sets like you would unions (imagine as venn diagrams)



- Containsall is a check to see if one set is a subset of another.
- Issue with using maps is planets can exist with the same name,to overcome this you can add an inner class called "key" which is essentially like a key in a database (see heavenlybody example"
- You then call this within your main class and override the equals/hashcode methods again.
- You need to ensure that the same object will have the same hashcode. (Overrdiing hashcode). Adding a number to the hashcode will cause it to be different.

Sorted Sets/Collections

Not too much to say here.

LinkedHashMaps allow you to order your list of items alphabetically which can definitely be useful in some scenarios

Unmodifiable maps provide the user with a read only view of the map, Individual objects can still be accessed and modified.

TreeMap sorts its list. It goes through its list and compares each item (there's a performance cost for this when deciding if you want to use it).

JavaFX

JavaFX was designed with the MVC (Model-View-Controller) design pattern in mind. In a nutshell, this pattern keeps the code that handles an application's data separate from the UI code. Because of this, when we're using the MVC pattern, we wouldn't mix the code that deals with the UI and the code that manipulates the application data in the same class. The controller is sort of the middle man between the UI and the Data.

When working with javaFX, the model corresponds to the applications data model, the view is the FXML, and the controller is the code that determines what happens when a user interacts with the UI. Essentially the controller handles events.

```
@Override
public void start(Stage primaryStage) throws Exception{
    Parent root = FXMLLoader.load(getClass().getResource( name: "sample.fx
    primaryStage.setTitle("Hello World");
    primaryStage.setScene(new Scene(root,  v: 300,  v1: 275));
    primaryStage.show();
}


public static void main(String[] args) {
    launch(args);
}
```

Very similar to android development, The start method defines what the application wil launch. In this example we're creating a simple rectangle object. The stop method will be called when the user closes the application.

Transition between this and android is very similar, mix of layouts, buttons, panes events and event handlers etc.

Feels slightly unuseful as there's better API's used for building applications (Angular etc)

Exceptions

Theres two different methods to approaching exceptions. The first is to deal with it before you execture code if/else statements (Look before you leap) and the second approach is to try it and catch the exception if it is thrown (Easy to ask for forgiveness permission).

```java
// look before you leap (checking before)
private static int divideBYL(int x, int y) {
    if(y != 0 ){
        return x/y;
    }
    return 0;
}
// Easy to ask for forgiveness
private static int divideEAFP(int x, int y) {
    try {
        return x/y;
    } catch (ArithmeticException e){
        return 0;
    }
}

private static int divide( int x, int y ){
    return x/y;
}
```

This isn't the best example as they're the same amount of code. However if you look at the screenshot below you can see a scenario where the try catch statement is far more efficient.

```java
private static int getInt() {
    Scanner s = new Scanner(System.in);
    boolean isValid = true;
    String input = s.next();
    for (int i = 0; i < input.length(); i++) {
        if (!Character.isDigit(input.charAt(i))) {
            isValid = false;
            break;
        }
    }
    if (isValid) {
        return Integer.parseInt(input);
    }
    return 0;
}

private static int getIntEAFP() {
    Scanner s = new Scanner(System.in);
    try {
        return s.nextInt();
    } catch (InputMismatchException e) {
        return 0;
    }
}
```

Stack & Call Trace

The call stack is the methods that are called during execution. When a method throws an exception it will stop running and the callstack will be shown. The best approach when looking at these stacks is to star from the bottom and work your way up.

```
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at com.timbuchalka.Example.getInt(Example.java:25)
    at com.timbuchalka.Example.divide(Example.java:16)
    at com.timbuchalka.Example.main(Example.java:11) <5 internal calls

Process finished with exit code 1
```

You can click through the existing java methods and find out what exceptions those methods are throwing.

In this program we first initially check the the input was entered correctly. If it gets passed then we have suitable checks in the divide method to ensure that we catch the exception.

```
private static int divide(){
    int x ,y;
    try{
        x = getInt();
        y = getInt();
    } catch (NoSuchElementException e){
       throw new ArithmeticException("No suitable input");
    }
    try{
        return x/y;
    } catch (ArithmeticException e){
        throw new ArithmeticException("Attempt to divide by 0");
    }
}

private static int getInt(){
    Scanner s = new Scanner(System.in);
    while(true){
        try{
            return s.nextInt();
        }catch (InputMismatchException e){
            s.nextLine();
            System.out.println("Please enter only a number from 0-9");
        }
    }
}
```

A cleaner way to do this is to introduce multiple catch statements and have all the code execute in the try block. You can also add try/catch methods to the main method executing the code. You could do it using "|" statements if you wished (added in java 7) (bitwise inclusive or)

```
private static int divide(){
    int x ,y;
    try{
        x = getInt();
        y = getInt();
        return x/y;
    } catch (NoSuchElementException e){
       throw new ArithmeticException("No suitable input");
    } catch (ArithmeticException e){
        throw new ArithmeticException("Attempt to divide by 0");
    }
}
```

Introduction to I/O

It really depends on what you want to do/your file type when deciding on what code to write (what does the data represent?). Sequential access can be thought of as a string of data that arrives at your program. Random access applies to files (Database indexes etc).

Writing contents to a file can be useful when wanting to read in/out multiple times. In the screenshot below we're writing all of the location data to a text file. Its imperative that we surround this this try/catch statements to ensure everything is being checked. It is also **extremely** important to ensure that you have closed your input stream. This is where we introduce **finally** which is another part to try/catch statements that always runs after the checks have been complete. In this case after everything has run we ensure that the stream is closed in our finally statement.

```
public static void main(String[] args)  {
    FileWriter locFile = null;
    try {
        locFile = new FileWriter( fileName: "locations.txt");
        for (Location location : locations.values()) {
            locFile.write( str: location.getLocationID() + "," + location.getDescription() + "\n");
        }
    }catch (IOException e){
        System.out.println("In Catch block");
        e.printStackTrace();
    } finally {
        System.out.println("In finally block");
        try{
            if(locFile != null){
                System.out.println("Successfully closed lockfile");
                locFile.close();
            }
        } catch (IOException e){
            System.out.println("issue closing lockfile");
            e.printStackTrace();
        }

    }
    // very important to close streams.
}
```

Throwing you own exceptions can be a good way to simulate stuff during testing (max sure you delete it before it goes into production). Try with resources was introduced in a lter version of java that automatically closes reader objects (very cool! https://docs.oracle.com/javase/7/docs/technotes/guides/language/try-with-resources.html).

```
public static void main(String[] args) throws IOException {
        try(FileWriter locFile = new FileWriter( fileName: "locations.txt")){
            for (Location location : locations.values()) {
            locFile.write( str: location.getLocationID() + "," + location.getDescription() + "\n");
            }
        }
```

```
Scanner scanner = null;
try {
    scanner = new Scanner(new FileReader("locations.txt"));
    scanner.useDelimiter(",");
    while(scanner.hasNextLine()) {
        int loc = scanner.nextInt();
        scanner.skip(scanner.delimiter());
        String description = scanner.nextLine();
        System.out.println("Imported loc: " + loc + ": " + description);
        Map<String, Integer> tempExit = new HashMap<>();
        locations.put(loc, new Location(loc, description, tempExit));
    }

} catch(IOException e) {
    e.printStackTrace();
} finally {
    if(scanner != null) {
        scanner.close();
    }
}
```

This code reads in a text file and gets the loca and descriptions and stores them in our map object. Very simple and very cool. The .skip functionality in this case allows use to skip over the "," delimeter in our strings. A more streamline way to do this would be to use a bufferedReader. **Scanner** and **BufferReader** both classes are used to read input from external system. **Scanner** is normally used when we know input is of type string **or** of primitive types and BufferReader is used to read text from character streams while buffering the characters for efficient reading of characters.

Byte Stream

One of advantage of dealing with binay data in a byte stream is that you don't have to parse the data in the various datatypes that we were recently using. Any of the type can be written to a byte, even strings! The follow the same pattern as the bufferreader/scanner but instead use fileinputstream and fileoutputstream classes instead.

```
public static void main(String[] args) throws IOException {
    try (DataOutputStream locFile = new DataOutputStream(new BufferedOutputStream(new FileOutputStream("locations.dat")))) {
        for (Location location : locations.values()) {
            locFile.writeInt(location.getLocationID());
            locFile.writeUTF(location.getDescription());
            System.out.println("Writing location " + location.getLocationID() + " : " + location.getDescription());
            System.out.println("Writing " + (location.getExits().size() - 1) + " exits.");
            locFile.writeInt(location.getExits().size() - 1);
            for (String direction : location.getExits().keySet()) {
                if (!direction.equalsIgnoreCase("Q")) {
                    System.out.println("\t\t" + direction + "," + location.getExits().get(direction));
                    locFile.writeUTF(direction);
                    locFile.writeInt(location.getExits().get(direction));
                }
            }
        }
    }

}
```

Instead of txt file we create a dat file. We could've done it with the other methods but this is just for an example.

Bitwise Operations (Shifting)

Essentially this is saying to shift our number by 24 bits to the right. This example is for the number “922342959” which in binary is “00110110111110011101011000101111”. Essentially “>>>24” means to shift the binary 24 slots to the right. So we would get (00110110) which is equal to 54. The next one means to shift it 16 to the right etc “00000000000000000011011011111001” = “140734” and so on.

```
private static void writeInt(int v) {
    int x;
    display(v >>> 24);
    display(v >>> 16);
    display(v >>> 8);
    display(v >>> 0);
}
```

Now that we have converted our text input into a byte stream we can edit the static class to read it in appropriately! The dataInputstreams through an exception when they hit the end of the stream so we need to catch this exception. Its very easy to cause an exception but difficult to catch them.

```
 54 and 0xFF is    110110              54 is 00000000000000000000000000110110
249 and 0xFF is 11111001           14073 is 00000000000000000011011011111001
214 and 0xFF is 11010110         3602902 is 00000000001101101111100111010110
 47 and 0xFF is    101111       922342959 is 00110110111110011101011000101111
```

Instead we add a Boolean and add an inner try within the while loop. Now we’re looking for the EOF exception instead of just a general IOException. This not only allows us to perform better debugging but ensure the whgile loop is closed appropriately.

```
try(DataInputStream locFile = new DataInputStream(new BufferedInputStream(new FileInputStream("location.dat")))) {
    boolean eof = false;
    while(!eof) {
        try {
            Map<String, Integer> exits = new LinkedHashMap<>();
            int locID = locFile.readInt();
            String description = locFile.readUTF();
            int numExits = locFile.readInt();
            System.out.println("Read location " + locID + " : " + description);
            System.out.println("Found " + numExits + " exits");
            for(int i=0; i<numExits; i++) {
                String direction = locFile.readUTF();
                int destination = locFile.readInt();
                exits.put(direction, destination);
                System.out.println("\t\t" + direction + "," + destination);
            }
            locations.put(locID, new Location(locID, description, exits));

        } catch(EOFException e) {
            eof = true;
        }

    }
} catch(IOException io) {
    System.out.println("IO Exception");
}
```

Serialisable

When we want to write objects objects to file we need to implement the serializable interface (it has no methods but gives the JVM a headstart). If a class we serialise has objects from other classes we need to make sure that they are serializable as well. We need to declare and assign the serial version when new versions come out later down the line.

```
try (ObjectOutputStream locFile = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream("locations.dat")))) {
    for(Location location : locations.values()) {
        locFile.writeObject(location);
    }
}
```

Very similar to the filewriter format, in this case we're using linkedhashmaps to store our location objects, luckily these designed to be serializable.

```
try(ObjectInputStream locFile = new ObjectInputStream(new BufferedInputStream(new FileInputStream("locations.dat")))) {
    boolean eof = false;
    while(!eof) {
        try {
            Location location = (Location) locFile.readObject();
            System.out.println("Read location " + location.getLocationID() + " : " + location.getDescription());
            System.out.println("Found " + location.getExits().size() + " exits");

            locations.put(location.getLocationID(), location);
        } catch(EOFException e) {
            eof = true;
        }
    }
```

The code has been greatly simplified here! By reading the location object it takes care of all its associated fields (We need to make sure to case the object to the correct type.

RAF (Random Access Files)

Random access files permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file. The RandomAccessFile is designed for random access of *binary* data. i.e. you can access anywhere in the file by index.

Java NIO

This package was described as an improvement to Java I/O because the classes in the package perform I/O in a non-lbokcing manner (fix problems with other java.io classes). It falls into one of two buckets: those that deal with the file system, and those that deal with reading and writing data.

When using classes in the java.io package, a thread will block while its waiting to read or write from a stream buffer. However, threads using the java.nio classes will not block. They are free to continue executing, so java.nio was introduced as a performance improvement. However, many developers have argued that it was a backwards step (its more complex but important to understand).

Where as previously we have delt with single characters at a time, data will be read in blocks.

Filesytem

So far we know how to create, read, and write files and datasources using java.nio. Sometime swe don't want to read from a file or write to a file, we want to copy files, delete files and move them. Java.nio.file is dedicate for this! But what is a path, we're familiar with the concept of a file path because you use them all the time (note they are unique to a specific file)

File paths can be absolute or relative. The path C:\\downloads\file.txt is an absolute path because it starts at the root node. The path photos\mountain.jpg is a relative path as it does not specify the root node. When using relative parths In applications, there's usually a the concept of a current **working directory** that you can combine with relative paths. For example, when you were running applications that used Path, you didn't specifiy the entire path. You did the following:

Relative path

```
Path dataPath = FileSystems.getDefault().getPath("data.txt");
```

The string "data.txt" doesn't give enough information about where the file is located. But you'll notice that you're calling **FileSystems.getDefault()** first, and then **getPath()**. What's happening is that the getDefault() call returns a FileSystem object with a working directory set to the current user directory.

Instead, you could have done something like the following:

Use absolute path

```
Path dataPath = Paths.get("C:\\MyIdeaProjects\\Project1\\data.txt");
```

However, in practice this is not a good method. When the user downloads your application you do not know where they will save it, hence its always best to use the working directory.

```
filePath = Paths.get(".");
System.out.println(filePath.toAbsolutePath());
```

This command will give you the absolute file path of your project.

```
filePath = FileSystems.getDefault().getPath("files");
System.out.println("Exists = " + Files.exists(filePath));

System.out.println("Exists = " + Files.exists(path4));
```

Good practice to check that a file path exists before using it.

```
try {
    Path sourceFile = FileSystems.getDefault().getPath("Examples", "file1.txt");
    Path copyFile = FileSystems.getDefault().getPath("Examples", "file1copy.txt");
    Files.copy(sourceFile, copyFile);
} catch(IOException e) {
    e.printStackTrace();
}
```

You can copy a file by doing this. You can also copy directories using this method.

- Files.move() method
- Files.delete() method

```
Path fileToCreate = FileSystems.getDefault().getPath("Examples", "file2.txt");
Files.createFile(fileToCreate);
```

Run this to create a file

```
Path dirToCreate = FileSystems.getDefault().getPath("Examples", "Dir4");
Files.createDirectory(dirToCreate);
```

Create a directory. You can create multiple directores using this method.

```
public static void main(String[] args) {
    Path directory = FileSystems.getDefault().getPath("FileTree/Dir2");  // FileTree\\Dir2 (windows)
    try (DirectoryStream<Path> contents = Files.newDirectoryStream(directory)) {
        for(Path file: contents) {
            System.out.println(file.getFileName());
        }

    } catch(IOException || DirectoryIteratorException e) {
        System.out.println(e.getMessage());
    }
}
```

Reading in contents from an existing directory.

```
DirectoryStream.Filter<Path> filter =
        new DirectoryStream.Filter<Path>() {
            public boolean accept(Path path) throws IOException {
                return (Files.isRegularFile(path));
            }
        };
```

This is a better way to do it. This file stuff is incredibly boring.

Concurrency in Java

A process is a unit of execution that has its own memory space. Each instance of a JVM runs as a process (this isn't true for all JVM implementations but is for most of them). When we run a java console application, we're kicking off a process.

Many people will use the terms process and application interchangeably. If one java application is running and we run another one, each will have its own memory space of **heap**. The first java application can't access the heap that belongs to the second application.

A **thread** is a unit of execution within a process. Each process can have multiple threads. In java, every process has at least one thread, the **main thread** (For ui applications, this is called the JavaFX application thread). In factg, just about every process also has multiple system threads that handle tasks like memory management and I/O. We, the developers, don't explicity create and code those threads. Our code runs on the main thread, or in other threads that we explicity create.

Creating a thread doesn't require as many resources as ceating a process. Every thread ccreated by a process shares the process's memory and files. This can create problems that we will discuss. In addition to the process's memory, or heap, each thread has what's called a thread stack, which is the memory that only that thread can access. We'll look at what stored in the heap vs the thread stack.

<u>Two main reasons why we would want more threads.</u>

Firstly, we sometimes want to perform a task that's going to take a long time. For example, we might want to query a database, or we might want to fetch data from somewhere on the internet. We could do this on the main thread, but the code within each main thread executes in a linear fashion, thus I won't be able to do anything until we finish this particular task. To the user, this could look like that application has died or frozen. The second reason we would want touse threads is because an API requires us to provide one.

```java
public class AnotherThread extends Thread {

    @Override
    public void run() {
        System.out.println("Hello from another thread.");
    }
}
```

```java
        Thread anotherThread = new AnotherThread();
        anotherThread.start();
```

We can extend the thread class and then call the class from inside the main method. You can never assume that the threads will run In a specific order.

You're not allowed to start the same thread multiple times, you would have to create a new instance of that class.

```java
    new Thread() {
        public void run() {
            System.out.println("Hello from the anonymous class thread");
        }
    }.start();
```

You can also initialise it like so, if you wanted to perform something In a specific class.

Runnable and Sleep

The advantage of using the runnable method is that we only have to do it once. This is how you implement it. Most of the time developers use this version of running threads as its move convenient and many methods of the java API allow a runnable instance to be parsed to them. The thread will terminate if it finishes all the code or its explicitly terminated.

```java
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(ANSI_RED + "Hello from MyRunnable's implementation of run()");
    }
}
```

```java
        Thread myRunnableThread = new Thread(new MyRunnable());
        myRunnableThread.start();
```

You want to make sure that you don't call the run method directly from inside the main thread, you should always call the start method. Otherwise the run method will excute inside that thread in which it was instantiated.

```java
    try {
        Thread.sleep(3000);
    } catch(InterruptedException e) {
        System.out.println(ANSI_BLUE +  "Another thread woke me up");
    }

    System.out.println(ANSI_BLUE + "Three seconds have passed and I'm awake");
}
```

We can tell threads to sleep if we don't want them running between tasks (see code above). We interrupt the thread if we want to stop it and do something else.

- Use the thread.interrupt() method.

We could have a situation where we know another thread can't execute until another thread has finished executing. In this scenario we could join the thread to this thread, the first thread will wait until that other thread has terminated before executing.

```
Thread myRunnableThread = new Thread(new MyRunnable() {
    @Override
    public void run() {
        System.out.println(ANSI_RED + "Hello from the anonymous class's implementation of run()");
        try {
            anotherThread.join();
        } catch(InterruptedException e) {
            System.out.println(ANSI_RED + "I couldn't wait after all. I was interrupted");
        }
    }
});
```

However, what happens if we join thread b to thread a but thread a never terminates (the application would look as if it has frozen). To prevent this we can add a timeout method to the join method.

```
anotherThread.join(2000);
```

The primary function of multithreading is to simultaneously run or execute multiple tasks. These tasks are represented as threads in a Java program and have a separate execution path. Also, handling of multithreaded Java programs is easy because you can decide the sequence in which execution of Java threads take place.

Following are some of the common advantages of multithreading:

- Enhanced performance by decreased development time
- Simplified and streamlined program coding
- Improvised GUI responsiveness
- Simultaneous and parallelized occurrence of tasks
- Better use of cache storage by utilization of resources
- Decreased cost of maintenance
- Better use of CPU resource

Multithreading does not only provide you with benefits, it has its disadvantages too. Let us go through some common disadvantages:

- Complex debugging and testing processes
- Overhead switching of context
- Increased potential for deadlock occurrence
- Increased difficulty level in writing a program
- Unpredictable results

This is just an overview of what multithreading provides the users with; however you can learn more about this concept in detail by getting trained in Core Java Online Training. Multisoft Virtual Academy offers Core Java Course Online for the developers, who want to prove their worth as a Java Application Developer. According to the requirement of the student, training can be provided in either Classroom or Online Training mode.

Multiple Threads

Look at the example where we change the for loop variable to an instance variable.

```
Thread 1: i =10
Thread 1: i =9
Thread 1: i =8
Thread 1: i =7
Thread 1: i =6
Thread 1: i =5
Thread 1: i =4
Thread 1: i =3
Thread 1: i =2
Thread 1: i =1
Thread 2: i =10
Thread 2: i =9
Thread 2: i =8
Thread 2: i =7
Thread 2: i =6
Thread 2: i =5
Thread 2: i =4
Thread 2: i =3
Thread 2: i =2
Thread 2: i =1
```

As you can see, in the second example the threads are no sharing the same instance variable, and the number of countdowns is effectively halved, why you might ask? When multiple threads are working with the same object they share that same object. In this case, when thread 1 changes the instance variable, thread 2 will get the new version of the variable. (Threads share the heap)

```
Thread 1: i =10
Thread 2: i =10
Thread 2: i =8
Thread 1: i =9
Thread 2: i =7
Thread 2: i =5
Thread 1: i =6
Thread 2: i =4
Thread 2: i =2
Thread 1: i =3
Thread 2: i =1
```

When we used the local variable it is instantiated on each of the threads stacks, hence we get 20 calls instead of 10.

Lets remember that a for loop is a number of steps. In this example a thread could execute all the steps but the print line before being suspended (hence the ordering of the console output). For example, thread 1 was suspended before it decremented the value from 10, hence thread 2 still thought the value was 10. This is known as thread interference, clearly not a good way of using multiple threads (race condition).

Synchronisation

Synchronisation solves this issue by only allowing one thread to execute that specific method at a time. If a class has multiple synchronous methods only one of them will run at a time. We need to sync all methods where we think thread interference might happen

```
public synchronized void doCountdown() {
```

We add the synchronized key word to the method that is executing the code, in this example I'm not sure on the relevance however as we're executing the count down twice again which seems counter productive.

Every java has what is known as an intrinsic lock, this is another way to synchronise a block of statements (threads must acquire the objects lock before they can execute the code) (**Primitive types do not have intrinsic locks)**.

```
synchronized(color) {
    for(i=10; i > 0; i--) {
        System.out.println(color + Thread.currentThread().getName() + ": i =" + i);
    }
}
```

You could do it like such, however in this example we still get issues. The problem is that we're using

a local variable and in this case each thread will have their own variable, **we need to use an object that they both share**. *Don't use local variables to synchronise threads!*

```
synchronized(this) {
    for(i=10; i > 0; i--) {
        System.out.println(color + Thread.currentThread().getName() + ": i =" + i);
    }
}
```

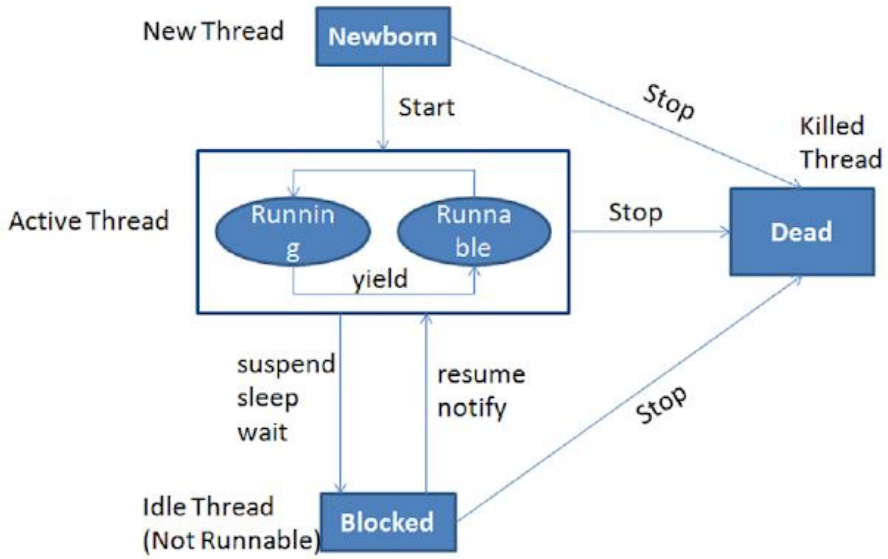
In this case, if we synchronise it on the countdown object that they both share it will work as expected.

Producer and Consumer (These methods are essential to understand!)

We can also synchronise static methods and objects, and when we do this the lock is owned by that thread. If a thread acquires that objects lock and it calls a method within that object it can keep executing as it already owns that lock. If a class is thread safe, it means a developer has synchronised all the critical sections in the code so we don't need to worry about thread interference. **We want to keep the code we syncrhonise down to an absolute minimum.**

Methods only available to sysnchronised code.

When a thread starts looping and the other thread cant change the value because it is blocked, this is known as a **Deadlock.** To get passed this we need to make use of the wait, notify and notifyall methods. We always want to call wait within a loop so that when it return it goes back to the start of the loop and checks the condition we're looking for. When we use notifyAll it wakes up all other threads, obviously if we have lots and lots of threads running the same code this may not be a good idea in terms of performance. https://www.java67.com/2016/04/10-points-about-wait-notify-and-notifyAll-in-java-multithreading.html (this is a really good read).



How to Stop a Thread in Java? Use volatile

It's important to note that a thread cannot be suspending during the middle of an atomic statement, for example, x object 1 = y object 2, or if its reading or writing primitive variables (except long and double). (y = 10 no, y = 1.23 yes). Some collections are not threadsafe, for example, arraylists are not sysnchronised (multiple threads can access it at the same time (we're responsible to preventin this)). The vectore class however is synchronised so you wouldn't have to worry about it there.

Java Concurrent Package

By using synchronous blocks we can run into lock issues where the thread remains locked and there's no way to get out of it. To overcome this we will look at classes that have concurrency built into them.

Reetrant lock and Unlock

```
import java.util.concurrent.locks.ReentrantLock;
```

We need to remember that all the threads need to be competing for the same lock to prevent thread interference

```
    ReentrantLock bufferLock = new ReentrantLock();
    MyProducer producer = new MyProducer(buffer, ThreadColor.ANSI_YELLOW, bufferLock);
    MyConsumer consumer1 = new MyConsumer(buffer, ThreadColor.ANSI_PURPLE, bufferLock);
    MyConsumer consumer2 = new MyConsumer(buffer, ThreadColor.ANSI_CYAN, bufferLock);
```

```
for(String num: nums) {
    try {
        System.out.println(color + "Adding..." + num);
        bufferLock.lock();
        buffer.add(num);
        bufferLock.unlock();

        Thread.sleep(random.nextInt(1000));
    } catch(InterruptedException e) {
        System.out.println("Producer was interrupted");
    }
}
```

We need to make sure that we always unlock the lock once we've locked it, otherways threads will lock forever.

This not the best way of implementing lock code, we want to encase the entire critical process in a try finally block as seen below. In this version of the code, if there is an issue adding a number to the buffer no matter what happens the finally statement will execute.

```
        bufferLock.lock();
        try{
            buffer.add(num);
        } finally {
            bufferLock.unlock();
        }
```

Thread Pools

A thread pool is a managed set of threads and reduces the overhead of thread creation, it may also prevent the number of threads that are active at a given time. We can do this using the build in class **ExecutorService.** Note, we need to manually decide how to shut this down.

```
ExecutorService executorService = Executors.newFixedThreadPool(3);

MyProducer producer = new MyProducer(buffer, ThreadColor.ANSI_YELLOW, bufferLock);
MyConsumer consumer1 = new MyConsumer(buffer, ThreadColor.ANSI_PURPLE, bufferLock);
MyConsumer consumer2 = new MyConsumer(buffer, ThreadColor.ANSI_CYAN, bufferLock);

executorService.execute(producer);
executorService.execute(consumer1);
executorService.execute(consumer2);

executorService.shutdown();
```

BlockingQueue Interface

One way we could simplify our code is to use an array blocking queue. Which operates on a first in first out basis. You may still need to add syncrhonised were necessary but it allows you to considerably cut code. (you can remove the Reetrant lock for example).

Deadlocks

When all threads that are active are locked, for example thread one is locked waiting for thread 2 but thread 2 is locked waiting for thread 1. **One way to prevent this is by having the threads request the locks in the same order.**

Deadlock when using synchronized methods

```
private static class Data {

    private Display display;

    public void setDisplay(Display display) {
        this.display = display;
    }

    public synchronized void updateData() {
        System.out.println("Updating data...");
        display.dataChanged();
    }
    public synchronized Object getData() {
        return new Object();
    }
}

public static class Display {
    private Data data;

    public void setData(Data data) {
        this.data = data;
    }

    public synchronized void dataChanged() {
        System.out.println("I'm doing something because the data changed...");
    }
    public synchronized void updateDisplay() {
        System.out.println("Updating display...");
        Object o = data.getData();
    }
}
```

Imagine we have two classes that contain synchronised methods, and each class calls a method in the other class. We'll recall that when a thread is running an objects synchronised method, no other thread can run a synchronised method using the same objet until the first thread exists the method it is running.

Lets assume that we instatiate the two classes in the following way

Construct the objects

```
Data data = new Data();
Display display = new Display();
data.setDisplay(display);
display.setData(data);
```

If one thread (thread 1) calls Data.updateData() while another thread(thread 2) calls Display.updateDisplay(), the following could happen depending on timing:

1. Thread 1 enters data.updateData() and writes to the console, then suspends
2. Thread 2 enters display.updatedisplay() and writes to the console, then suspends

3. Thread 1 runs and tries to call display.dataChanged(), but thread 2 is still running display.updateDispaly(), so its holding the lock on the display object. Thread 1 blocks
4. Thread 2 wakes up and tries to run data.getData(), but thread 1 is still running data.updateData(), so thread 2 blocks

We have to rewrite the code so that the two threads try to obtain the lcoks in the same order. You wouldn't mix Ui and Model code like this in a real world application!

```
public synchronized void sayHello(PolitePerson person){
    System.out.format("%s: %s" + "has said hello to me!%n" +, this.name, person.getName());
}
```

Call little format, %s's get replaced by strings getting passed in

Starvation: *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

```
private static ReentrantLock lock = new ReentrantLock(true);
```

By setting the parameter to true, we instantiate it as a fairlock. This means first come first served. Its important to note that its no fairness in scheduling, a thread could still wait a long time for another thread to execute.

Livelock: A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Thread Issues

In this lecture we'll cover a few loose ends related to threads that didn't fit into the previous stuiff we've covered. First, lets remind ourselves what an **atomic action** means. As we've seen, when a thread is running, it can be suspended when it's in the middle of doing something.

For example, If a thread calls the sout method, it can be suspending I the middle of executing that method. It may have evaluated the argumented that's being passed, but it's suspended before it can print. Or it may be partway though evaluating the argument when its suspened. Essentially, it isn't an atomic action.

An atomic action ncan't be suspened in the middle of being executed. It either completes or doesn't happen at all. Once a thread starts to run an atomic action, we can be confident it won't suspend until it has completed the action.

1. Reading and writing reference variables. For example, the statement **myObject1 = myObject2** would be atomic. A thread can't be suspended in the middle of executing this statement.
2. Reading and writing primitive variables, except those of type long and double. The JVM may require two operations to read and write longs and doubles, and a thread can be suspended between each operation. For example, a thread can't be suspended in the middle of executing **myInt = 10**. But it can be suspended in the middle of executing **myDouble = 1.234**.
3. Reading and writing all variables declared **volatile**.

Let's take a look at the third item: volatile variables. You may think that since we don't have to worry about thread interference with atomic actions, that we don't have to synchronise them, but that isn't true.

Because of the way java manages memory, its possible to get memory inconsistency issues when multiple threads read and write the same variable. Java's memory management model is outside the scope of this for now. All we need to know is that each thread has a CPU cache, which can optain copies of values that are in memory.

Since it's faster to read from the cache, this can improve the performance of an application. There wouldn't be a problem if there was only one CPU, but these days, most computers have more than one CPU.

When running an application, each thread may be running on a different CPU, and each CPU has its own cache. It's possible for the values in the caches to become out of synch with each other, and with the value in main memory—a memory consistency error.

Suppose we have two threads that use the same int counter. Thread1 reads and writes the counter. Thread2 only reads the counter. As we know, reading and writing to an int is an atomic action. A thread won't be suspended in the middle of reading or writing the value to memory. But let's suppose that Thread1 is running on CPU1, and Thread2 is running on CPU2. Because of CPU caching, the following can happen:

1. The value of the counter is 0 in main memory
2. Thread1 reads the value of 0 from main memory.
3. Thread1 adds 1 to the value.
4. Thread1 writes the value of 1 to its CPU cache.
5. Thread 2 reads the value of counter from main memory and gets 0, rather than the latest value, which is 1.

Volatile Variables

Thread Issues

This is where volatile variables come in. When we use a non-volatile variable the JVM doesn't guarantee when it writes an updated value back to main memory. But when we use a volatile variable, the JVM writes the value back to main memory immediately after a thread updates the value in its CPU cache. It also guarantees that every time a variable reads from a volatile variable, it will get the latest value.

To make a variable volatile, we use the volatile keyword.

Volatile variables

```
public volatile int counter;
```

Thread Issues

You might be thinking that we don't have to synchronize code that uses volatile variables. Unfortunately, that's not entirely true. In our example, only one thread is updating the variable. In that case, we wouldn't need synchronization. But if more than one thread can update the value of a volatile variable, we can still get a race condition.

Let's assume that we have two threads that share a volatile int counter, and each thread can run the following code:

Increment counter

```
counter++
```

Note, this isn't an atomic operation. A thread would read in the value, then add one to it then write it back to memory. It could be suspended after any of these steps.

A thread can be suspended after any of these steps. Because of that, the following can happen:

1. The value of the counter is 1 in main memory and in Thread1 and Thread2's CPU caches.
2. Thread1 reads the value of counter and gets 1
3. Thread2 reads the value of counter and gets 1
4. Thread1 increments the value and gets 2. It writes 2 to its cache. The JVM immediately writes 2 to main memory.
5. Thread2 increments the value and gets 2. It writes 2 to its cache. The JVM immediately writes 2 to main memory.
6. Oops! The counter has been incremented twice, so its value should now be 3

A memory consistency error like this can occur when a thread can update the value of the variable in a way that depends on the existing value of the variable. In the counter++ case, the result of the increment depends on the existing value of the variable.

In other words, a thread has to read the value of the counter variable in order to generate a new value for counter. By the time the thread operates on the value it has read, the value could be stale, as it is in this example.

However, when only one thread can change the value of a shared variable, or none of the threads update the value of a shared variable in a way that depends on its existing value, using the volatile keyword does mean that we don't need to synchronize the code. We can be confident that the value in main memory is always the latest value.

That's great, and we can see that there will be times when using the volatile keyword will eliminate the need for synchronization. But it would be nice if we could read and write variables without having to worry about thread interference or memory consistency errors. Fortunately Java provides classes that allows us to do just that, in specific cases.

In an earlier lecture, we took a look at a few of the classes in the java.util.concurrency package, which was introduced with Java 1.5. Another sub-package is the **java.util.concurrent.atomic** package. This package provides us with classes that we can use to ensure that reading and writing variables is atomic.

It's difficult to create an example that illustrates how the classes in the package are useful. Like most thread concepts, the benefits are seen when there are multiple threads performing non-trivial tasks. So we'll have to take a look at some code, but not run anything.

---

*N*e declare the counter as type AtomicInteger and pass 0 as the initial value. In the inc() method, we use **ncrementAndGet()**. This atomically increases the value by 1. The method **decrementAndGet()** decrease the value by 1. To get the value, we call the **get()** method. We don't have to synchronize the ncrement or decrement operations in any way.

Use AtomicInteger

```
private AtomicInteger counter = new AtomicInteger(0);

public void inc() {
    counter.incrementAndGet();
}

public void dec() {
    counter.decrementAndGet();
}

public int value() {
    return counter.get();
}
```

Hint: double-click to select code

We can't use AtomicInteger as a substitute for an Integer object. AtomicIntegers are meant to be used in specific situations like the one above: when thread interference can take place because a thread is changing the value of the variable in a way that depends on the existing value.

There are Atomic classes for the following types: boolean, integer, integer array, long, long array, object reference, and double. Remember that we said reading and writing long and double variables isn't atomic? Well, we could use the AtomicLong and AtomicDouble classes to make these operations atomic.

The Atomic classes have set() and get() methods that allow us to set a value, and get the current value. But the Atomic classes are really meant to be used in situations when a value is being incremented or decremented. They're intended to be used when the code is using a loop counter, or generating a sequence of numbers for some other reason.

Thread Challenges 1-9

Things to remember:

1. It threads are only reading data out they do not need to worry about thread interference, hence you do not need to synchronise those methods
2. If we want both threads to compete for the same lock we need to create a single lock for that class and have them compete for it. Put all critical code within a try/finally block
3. You can use trylock with timeout values to prevent the code getting hung up.

```
public synchronized void deposit(double amount) {
    try{
        if(lock.tryLock( time: 1000,TimeUnit.MILLISECONDS))
            {
                try{
                    balance += amount;
                }
                finally {
                    lock.unlock();
                }
        }else{
            System.out.println("Could not get lock");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

4. If you create local variables within a method you do not need to make them thread safe as each thread will have its own variable to manage.
5. Look for deadlock/live lock issues. For example, threads might prevent eachtoehr from accessing locks.

Lambda Expressions

Every Lambda expression conists of three parts, the first is the argument list, the second is an arrow token and the third one is the body (the code we want to run).

```
new Thread(()-> System.out.println("Printing from the Runnable")).start();
```

In this example, it knows that that runnable interface only has one method that has no arguments. It needs to match the lambda expression. You can only use lambda expressions on interfaces that contain one method (functional interfaces).

```
new Thread(()-> {
    System.out.println("Printing from runnable");
    System.out.println("line 2");
    System.out.println("This is line 3");
}).start();
```

It can have multiple lines within the body.

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
Collections.sort(employees, new Comparator<Employee>() {
    @Override
    public int compare(Employee employee1, Employee employee2) {
        return employee1.getName().compareTo(employee2.getName());
    }
});
```

Implementing lambdas, we can convert the function to this

```
Collections.sort(employees, (employee1, employee2) -> employee1.getName().compareTo(employee2.getName()));
```

```
String sillyString = doStringStuff(new UpperConcat() {
    @Override
    public String upperAndConcat(String s1, String s2) {
        return s1.toUpperCase() + s2.toUpperCase();
    }
},employees.get(0).getName(), employees.get(1).getName());

UpperConcat uc = (s1,  s2) -> s1.toUpperCase() + s2.toUpperCase();
String sillyString = doStringStuff(uc,employees.get(0).getName(), employees.get(1).getName());
System.out.println(sillyString);
```

Here is a good example ofd how lambdas simplify code. Instead of heaving to override the method within the interface, we can simply define the UC object and supply the function directly. A lambda function is treated like a nested block of code, that being it has the same scope as the class it is defined in.

```
    public String doSomething() {
//        UpperConcat uc = (s1, s2) -> {
//            System.out.println("The lambda expression's class is " + getClass().getSimpleName());
//            String result = s1.toUpperCase() + s2.toUpperCase();
//            return result;
//        };
        final int i = 0;
        {
            UpperConcat uc = new UpperConcat() {
                @Override
                public String upperAndConcat(String s1, String s2) {
                    System.out.println("i (within anonymous class) = " + i);
                    return s1.toUpperCase() + s2.toUpperCase();
                }
            };

            System.out.println("The AnotherClass class's name is " + getClass().getSimpleName());

            i++;
            System.out.println("i = " + i);
            return Main.doStringStuff(uc,"String1","String2");
        }
```

For example, in this case we have to define our variable as final so that we can reference it from within our anonymous class. This is because the local variable doesn't belong to the class instance.

```
public void printValue() {

    int number = 25;

    Runnable r = () -> {
        try {
            Thread.sleep(5000);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("The value is " + number);
    };

    new Thread(r).start();
}
```

Because lambdas may not be immediately evaluated, any variables we use inside the lambda from outside must be declared as final.

Functional Interfaces & predicates

```
    printEmployeesByAge(employees, ageText: "Employees over 30", employee -> employee.getAge() > 30);
    printEmployeesByAge(employees, ageText: "\nEmployees 30 and under", employee ->employee.getAge() <= 30);

}

private static void printEmployeesByAge(List<com.timbuchalka.Employee> employees,
                                        String ageText,
                                        Predicate<com.timbuchalka.Employee> ageCondition) {

    System.out.println(ageText);
    System.out.println("==================");
    for(com.timbuchalka.Employee employee : employees) {
        if (ageCondition.test(employee)) {
            System.out.println(employee.getName());
        }
    }
}
```

The Predicate Parameter allows us to use the test function which checks the age of the employee.

```
@FunctionalInterface
public interface Predicate<T> {

    Evaluates this predicate on the given argument.
    Params: t – the input argument
    Returns: true if the input argument matches the predicate, otherwise false
    boolean test(T t);
```

So in this case it will return true if the employees age being passed in is greater than 30 or less than etc.

```
printEmployeesByAge(employees, "\nEmployees younger than 25", new Predicate<Employee>() {
    @Override
    public boolean test(Employee employee) {
        return employee.getAge() < 25;
    }
});
```

You can also do the same using anonymous classes.

```
IntPredicate intp = i -> i > 15;
System.out.println(intp.test(10));
int a = 20;
System.out.println(intp.test(a + 5));
```

You can also use predicates in this dynamic

```
IntPredicate greaterThan15 = i -> i > 15;
IntPredicate lessThan100 = i -> i < 100;

System.out.println(greaterThan15.test(10));
int a = 20;
System.out.println(greaterThan15.test(a + 5));

System.out.println(greaterThan15.and(lessThan100).test(50));
System.out.println(greaterThan15.and(lessThan100).test(15));
```

You can also chain predicates together.

```
Function<Employee, String> getLastName = (Employee employee) -> {
    return employee.getName().substring(employee.getName().indexOf(' ') + 1);
};
```

You can also right functions using lambdas.

```
Function<Employee, String> upperCase = employee -> employee.getName().toUpperCase();
Function<String, String> firstName = name -> name.substring(0, name.indexOf(' '));
Function chainedFunction = upperCase.andThen(firstName);
```

Upper case will run first and then pass as an argument to firstName.

Streams

```
someBingoNumbers.forEach(number -> {
    if(number.startsWith("N")){
        chosenNumbers.add(number);
        System.out.println(number);
    }
});
chosenNumbers.sort((s1,s2) -> s1.compareTo(s2));
chosenNumbers.forEach(s -> System.out.println(s));
```

We can conver the above to the below!

Any Stream operations need to have to operations. That is that they are not interfering and they are stateless (doesn't depend on any state/variable values in a previous field).

```
someBingoNumbers.stream().map(String::toUpperCase).filter(s->s.startsWith("G")).sorted().forEach(System.out::println);
```

The :: notation is called a method reference to call methods.

Let's recap what our stream chain does and look at the results of each step:

Input	Method/Operation	Output
The ArrayList someBingoNumbers	stream()	A Stream that contains all the items in the someBingoNumbers list, in the same order
Stream containing all bingo numbers	map(String::toUpperCase)	A Stream that contains all the bingo numbers uppercased
Uppercased stream	filter(s→s.startsWith("G")	A Stream containing all items beginning with "G" ("G53", "G49", "G60", "G50", "G64")
"G" items stream	sorted()	A stream containing the sorted items ("G49", "G50", "G53", "G60", "G64")
Sorted "G" items stream	forEach(System.out::println)	Each "G" item is printed to the console. Void result. The chain ends.

When a chain is evaluated, a stream pipeline is created. The stream pipeline consists of a source, zero or more intermediate operations, and a terminal operation. In our example, we used a collection as the source, but we could also be an array or an I/O channel, and we can build streams from scratch.

Best Practices

Before we close off our discussion of lambda expressions, let's talk about best practices. Throughout this section, we've intentionally written lambda expressions in different ways. Here are the variations we've used:

1. Specified the types of parameters vs. letting the compiler infer them
2. Used a return statement with curly braces for one-statement lambda expressions vs. not using return because it's implied (and hence not requiring curly braces)
3. Used lambda expressions that contain one statement vs. Lambda expressions that have more than one statement
4. Using parenthesis when a lambda expression only has one argument vs. not using parenthesis, since they're optional when there's only one argument

If you look at the four variations, the two alternatives offer the choice between verbosity vs. conciseness, which in turn, often comes down to the choice between readability and conciseness. Not all the time. Short lambda expressions are usually readable no matter how concise they are. But when striving for conciseness, we can sometimes write lambda expressions that are difficult to decipher because we've left out too much information.

Best Practices

There are a number of articles and tutorials on the internet that discuss best practices when using lambda expressions. Many of them fall on the side of conciseness. Keep in mind that best practices are guidelines, not rules. When reading code I haven't written (or going back to code I wrote months (years) previous), it's not unusual for me to come across a lambda expression that's cryptic and takes a lot of work to figure out.

Was it concise? Yes. Was it readable to the human? No. Yes, we can optionally leave out the parameter types when the compiler can infer them, but will it be easy for a developer reading the code to do the same? We don't have to include the return keyword when a one-statement lambda returns a value, but will a developer be able to tell at a glance that the lambda expression returns a value?

(My opinion) When it comes to practices that are syntactic and don't make a difference to the code that's generated, the key thing is to be consistent. Don't use different styles within the same file. If you're working on the code someone else wrote, and they always leave out the parameter types, then you should do the same. If they always use return keyword with one-statement lambdas, do the same. If you're writing code from scratch, do what's clearer and the easiest for you, and also what you think will be clearer for other developers who may work on the code in the future.

I'd rather see a verbose lambda than a concise one with a comment, because I know from experience that comments aren't always updated when the code is, which can lead to a lot of confusion and wasted time.

Regular Expressions

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.
Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:

- Pattern Class - Defines a pattern (to be used in a search)
- Matcher Class - Used to search for the pattern
- PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

```
String string = "I am a string, Yes I am.";
System.out.println(string);
String yourString = string.replaceAll( regex: "I", replacement: "You");
System.out.println(yourString);
```

Very simple early example of a Regex.

```
System.out.println(alphaNumeric.replaceAll( regex: ".", replacement: "Y")); // "." is a wildcard for any character
System.out.println(alphaNumeric.replaceAll( regex: "^abcDeee", replacement: "YYY")); // "allows you find an inner sequence
```

Only replacesd the first occurrence though if the string starts with it

```
System.out.println(alphanumeric.matches("^hello"));
System.out.println(alphanumeric.matches("^abcDeee"));
```

When using the begins with character the entire string must match

```
System.out.println(alphanumeric.replaceAll("[aei]", "X"));
```

Replaces all laters with a e or I with X.

Don't need to show ever single option here, pretty self explanatory.
https://www.w3schools.com/java/java_regex.asp

Challenge 1&2

Create a regex that matches the expressions

```
String challenge1 =  "I want a bike.";
String challenge2 = "I want a ball.";

// matches characters
String regex = "I want a \\w+.";
System.out.println(challenge1.matches(regex));
System.out.println(challenge2.matches(regex));
```

Challenge 3

Create a Pattern/Matcher object

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(challenge1);
System.out.println(matcher.matches());
```

Challenge 4

```
String challenge4 = "Replace all blanks with undersores.";
System.out.println(challenge4.replaceAll("\\s", "_"));
```

Replace all blanks with underscores, you could also just use " ".

Challenge 5

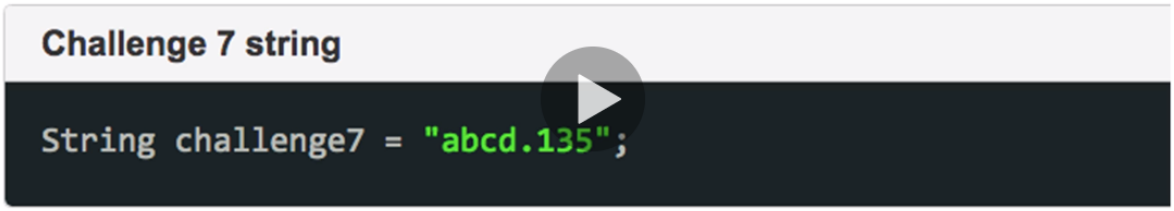
Write a regular expression that will match the following string in its entirety.

```
String challenge5 = "aaabccccccccdddefffg";
System.out.println(challenge5.matches("[a-g]+"));
```

```
System.out.println(challenge5.matches("^a{3}bc{8}d{3}ef{3}g$"));
System.out.println(challenge5.replaceAll("^a{3}bc{8}d{3}ef{3}g$", "REPLACED"));
```

Challenge 6

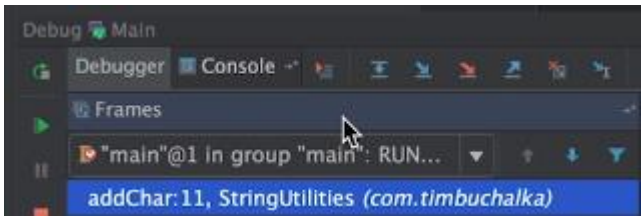
Write a regular expression that will match a string that starts with a series of letters. The letters must be followed by a period. After the period, there must be a series of digits. The string "kjisl.22" would match. The string "f5.12a" would not. Use this string to test your regular expression.




```
String challenge7 = "abcd.135";
System.out.println(challenge7.matches("^[A-z][a-z]+\\.\\d+$"));
```

Debugging

We can add breakpoints so that the debugger will suspend running at the specific point. On the debugging tool bar you have a selection of buttons



- The **show execution button** will return the editor to where the application was suspended.
- The **step over will** move onto the next line of code that is to be executed.#
- The **Step into method** allows us to see whats happening inside that method (this will normally skip over JDK methods as its unlikely to be the source of your problems
- The **Force step into** will force it to step into any method.
- We can use the **Step out button** when we want the debugger to run the rest of the method.]
- The **Drop frame button** will let us rewind the application by one frame (cant rewind everything)
- The **Run to cursor button will** run to where our cursor location is.

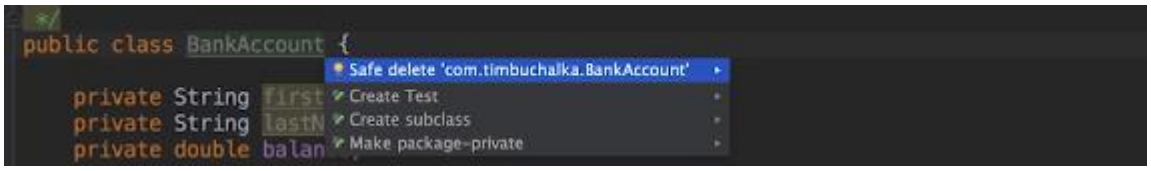
There's a button on the run menu that allows you to view all your breakpoints and remove them if necessary.

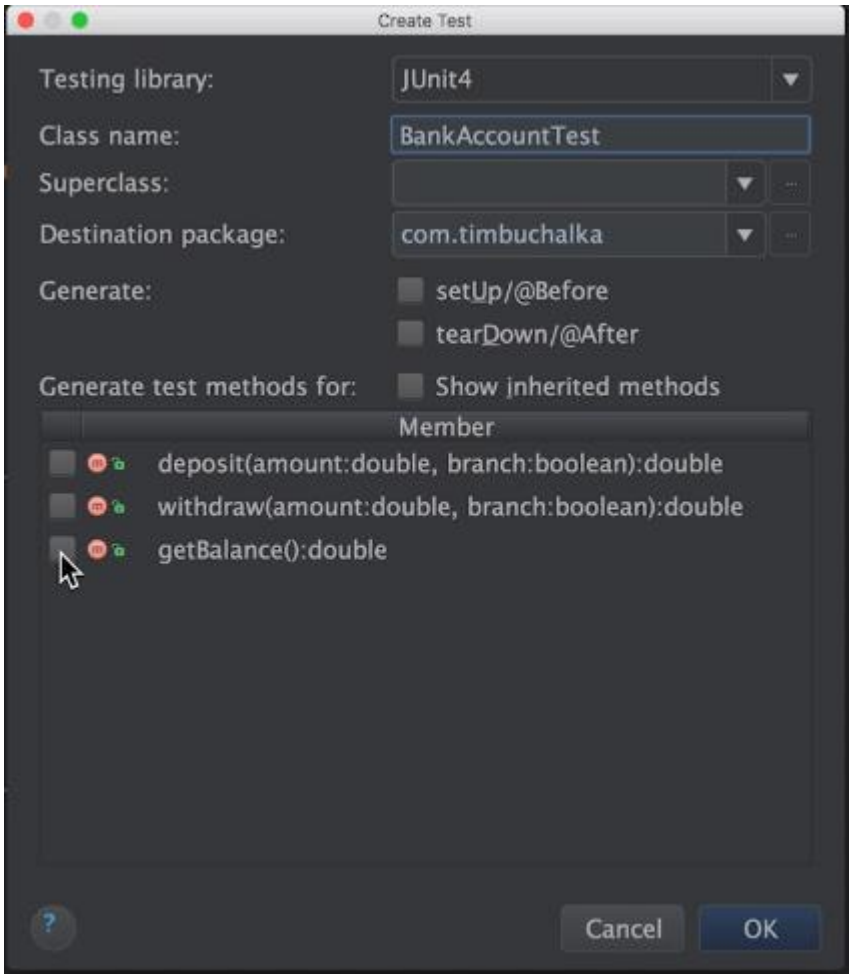
Watchers allow us to set watchers on specific field instance variables that we want to watch. This essentially allows us to declutter our debugging tab. There is also a separate watchers pane available!

Field watch points allow us to set a watch point on a variable. Anytime the field is modified the program will automatically break.

Junit Testing

The build process in a large team will typically run a test unit suite, which will allow developers to work out what code is breaking the tests. As easy way to implement this in a class is to press alt-enter at the top and select test class.





You can create equivalent tests for each of the methods, a unit test typically tests on one method. Go to project structure, modules and change to to compile. Standard asserts methods. Create objects and run them through assert equals etc. (just alt-enter on the @test)

There are other Junit assertion methods

1. assertNotEquals() – we can use this instead of assertEquals() when we don't want the actual value to be equal to a specific value.

2. assertArrayEquals() - when we want to verify the value of an array, we have to use the assertArrayEquals() method. The assertEquals() method won't work, because it will only consider two arrays equal if they are the same instance. The assertArrayEquals() method considers two arrays equal when their lengths are the same, and every element in both arrays is the same (and in the same order).

3. assertNull() and assertNotNull() – we can use this method to check for null (and non-null) values. We can use assertEquals() to check for null, but as with assertTrue() and assertFalse() methods, using assertNull() and assertNotNull() makes the intention clearer, and we only have to pass the actual value to the method.

4. assertSame() and assertNotSame() – we use this when we want to check wether two instances are the exact same instance. Remember that the assertEquals() method uses the equals() method to test for equality. The assertSame() method compares the object references.

5. assertThat() – this method compares the actual value against a matcher (not the Matcher in the JDK, but a JUnit matcher class). This is more powerful than the other assert methods, since we can compare the actual value against a range of values. Note that this method only became available in JUnit 4.4.

@before annotation is run before every test and allows us to use an instance variable etc. Instead of creating a new instance variable for each test!

```
private BankAccount account;

@org.junit.Before
public void setup() {
    account = new BankAccount("Tim", "Buchalka", 1000.00, BankAccount.CHECKING);
    System.out.println("Running a test...");
}
```

```
@org.junit.BeforeClass
public static void beforeClass() {
    System.out.println("This executes before any test cases");
}
```

Ececutes before all the tests

```java
@org.junit.AfterClass
public static void afterClass() {
    System.out.println("This executes after any test cases");
}
```

Executes after all the tests

```java
@org.junit.After
public void teardown() {
    System.out.println("Count = " + count++);
}
```

Executes after everytest

If we tell a method to accept an illagualargument or another type of exception it will pass. Challengees were easy!

SQLite

```
sqlite> create table contacts (name text, phone integer, email text);
sqlite> insert into contacts (name,  phone, email) values('Tim', 6545678, 'tim@email.com');
sqlite> SELECT * FROM contacts;
name|phone|email
Tim|6545678|tim@email.com
sqlite> SELECT name, phone, email FROM contacts;
name|phone|email
Tim|6545678|tim@email.com
sqlite> SELECT email FROM contacts;
```

Basic commands to instantiate database

```
sqlite> INSERT INTO contacts VALUES("Avril", "+61 (0)87654321", "avril@email.com.au")
   ...> ;
sqlite> SELECT * from contacts;
Tim|6545678|tim@email.com
Brian|1234|brian@myemail.com
Steve|87654|
Avril|+61 (0)87654321|avril@email.com.au
sqlite> .backup testbackup
sqlite> UPDATE contacts SET email="steve@hisemail.com";
sqlite> SELECT * FROM contacts;
Tim|6545678|steve@hisemail.com
Brian|1234|steve@hisemail.com
Steve|87654|steve@hisemail.com
Avril|+61 (0)87654321|steve@hisemail.com
sqlite>
```

You can backup an entire table using the .backup command

You can also update every row in a table (careful)

Covered all this stuff before!

JDBC

Overview

We've seen how to work with an sqlite database from the command line. Now we'll learn how to use a database from a Java application. We do so using the JDBC API. JDBC stands for **Java Database Connectivity**. Using JDBC, we can not only work with databases, but also spreadsheets and flat files.

JDBC acts as a middleman between a Java application and a data source. To use a particular data source from an application, we need the JDBC driver for the data source. For example, to access an sqlite database from an application, we need an sqlite JDBC driver.

Overview

The driver is simply a Java library containing classes that implement the JDBC API. Because all JDBC drivers have to implement the same interfaces, it's not difficult to change the data source an application uses.

For example, if an application uses an sqlite database, and then we decide later that we want to use a MySQL database, all we have to do is use the MySQL JDBC driver, instead of the sqlite one (in addition to migrating the data to a MySQL DB, of course).

This is very similar to android stuff in terms of querying etc, maybe look through video to get relevant syntax is necessary. Very similar to music project etc!

Modules

One of the most important and exciting features of JDK 9 is the module system, which was developed under a project codename of **jigsaw.**

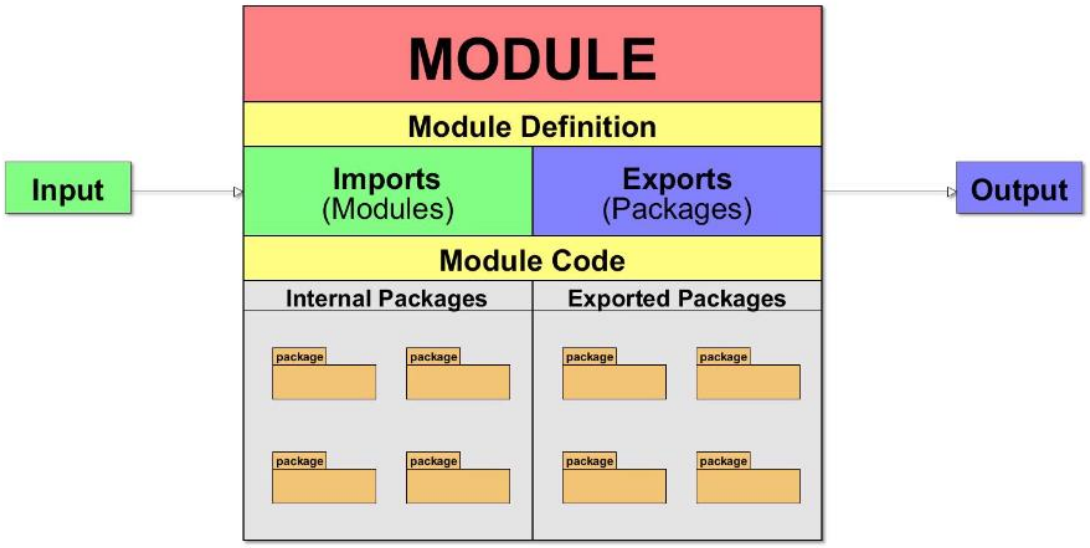
Java 9 introduced a new program compenent known as a module. You can think of a java application as a colletion on modules.

The module system was designed to have a reliable configuration, strong encapsulation and a modular JDK/JRE. The purpose is to solve the problems typically involved with developing and deploying java applications before java 9. These modularity features in java 9 are collectively known as the java platform module system, or JPMS.

Every Module needs to have some configuration.

- Name: unique name of the module
- Inputs: what the module needs to use and what is required for the module to be compiled and run
- Outputs: what the mdouel outputs or exports to other modules

This way you have to specifiy every package that your module needs, only the name of the module you depend on.



Every module comes with a module descriptor file that describes the module and contains metadata ebout the module

The Module descriptor file is always located directly at the module root folder, and always has the name module-info.java

Lets see a minimalistic module descriptor example.

```
module academy.learnprogramming.common {

}
```

- There is a new keyword **module** and this is followed by the module name and curly braces.

Inside the curly braces you can optionally add metadata about the module. In other words inputs and outputs, which we wil see later.

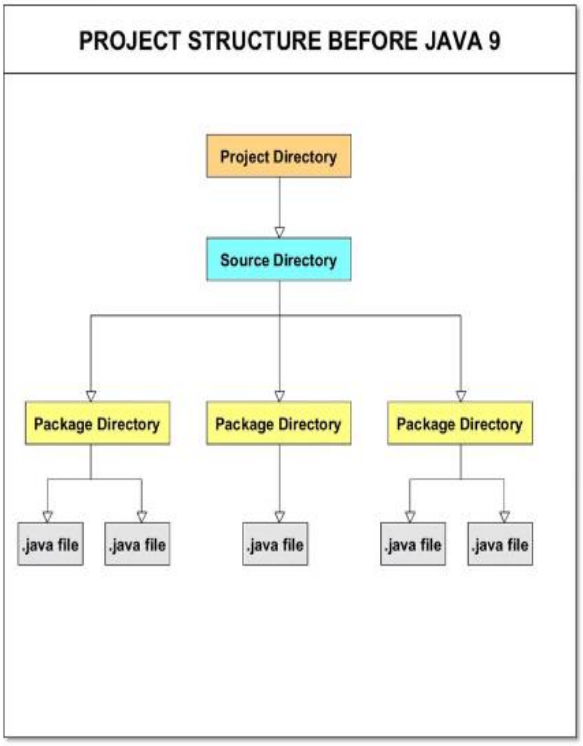
A module declaration introduces a module name that can be used in other module declarations to express relationships between modules.

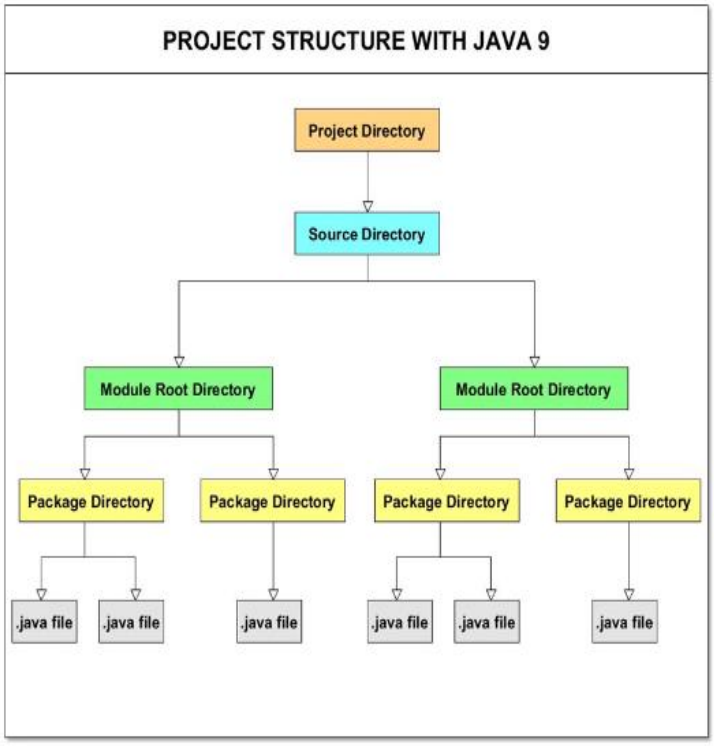
A module name consists of one or mor java identifiers separated by "." Tokens.

There are two kinds of modules, **normal** and open modules.

The kind of module determins access to the modules types, and the members of those types, for code outside the module.

- A **normal module**, without the open modifier, grants access at compile time and run time to types in only those packages which are explicitly exported.
- An **open module**, with the open modifier, grants access at compile time to types in only those packages which are explicitly exported, but grants access at run time to types in all its packages, as if all packages had been exported.





- Project Jigsaw has the following primary goals:

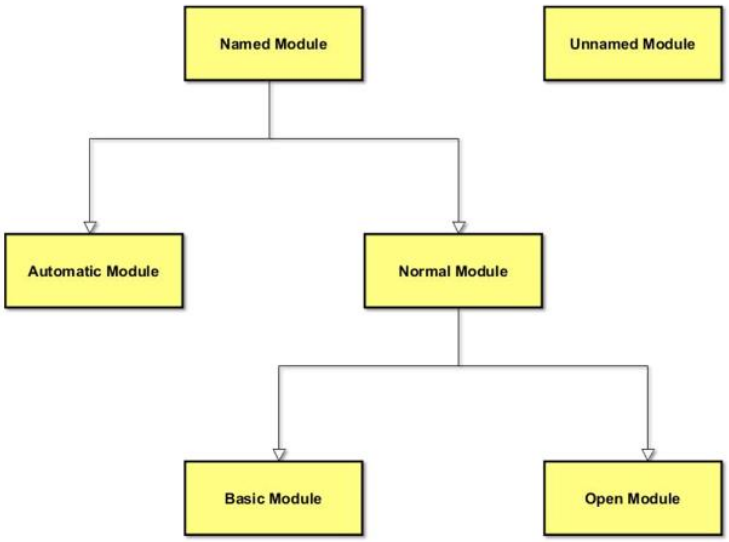
1. **Scalable platform**
   - The ability to scale the platform down to smaller computing devices is achieved by moving from a monolithic runtime.
2. **Security and maintainability**
   - To make a more maintainable platform code base that has better organization, and has it's internal API hidden, so we have better modular interfaces to improve platform security.
3. **Improved Application performance**
   - A platform that is smaller with only the necessary runtimes, resulting in faster performance.
4. **Easier developer experience**
   - As a result of the Module system and the modular platform combination to make it easier for developers to create applications and libraries.

Module declaration and statements

- <moduleName> is the name of the defined **module**.
- The module name is mandatory.
- We can't have two modules inside the same code base with the same name.
- It's good practice to name modules with the same name as packages.

- Inside the curly braces we can have zero or more module statements. Those statements can be:
  - exports statement
  - opens statement
  - requires statement
  - uses statement
  - provides statement

- The ***requires*** statement is used to specify the module that is required by the current module, in other words if our module depends on some other module we have to specify it using the requires statement.
- We can use multiple ***requires*** statements depending on how many modules are needed in our module.
- The ***exports*** statement is used to specify the packages that are exported by the current module.
- The ***provides*** statement is used to specify the service implementations that the current modules provides.
- The ***uses*** statement is used to specify the service that the current module consumes.



Each Named module has a name, can be normal or automatic. Named modules are modules declared in the module-info.java file (the descriptor file). All platform modules are named modules

Normal modules don't officially exist in JDK9, we just used the term normal for a named module that is not an automatic module.

- A normal module has a module descriptor file while a automatic module does not have a module descriptor file.
- A normal moduile is declared using the keyword module.
- A normal module does not export any of its packages by default.
- A normal module is divided into basic and open modules.

An **automatic** module is created after adding a JAR file ot the module path.

- An automatic module is not explicitly declared by the developer inside the module descriptor, it is automatically created when a Jar file is placed into the module path.
- It requires by default all platform modules, all our own modules and all other automatic modules
- It exports all its packages by default
- They are use for third-party code.

Basic modules don't officially exist, we just use the term basic for a nmodule hat is not an open module

These are modules that are not automatic and not open. It has the same characteristsics as a namoraml module except it is not an open module.

- Many third-party libraries like Spring and Hibernate use reflection to access the internals of JDK at runtime.
- These libraries will not work unless we have an **open** module.
- An open module is defined using the **open** keyword.
- An open module makes all packages inside the module accessible for deep reflection.
- The keyword open can be used to declare an open module or to declare specific packages as open.
- We will see this in action later in the course with a Java FX application. We will see how to open some packages for the FXML loader and Application class.

- An **unnamed** module does not have a name and it is not declared.
- It exports all of its packages.
- It reads all modules in the JDK and on the module path.
- An unnamed module is a module made up of all JAR files from the classpath. All these jar files form the unnamed module.
- We will discuss the difference between the module path and classpath later.

- NOTE: A named module can't require an unnamed module.

- These exist for convenience.
- Usually they have no code of their own, they just have a module descriptor.
- They collect and export the contents of other modules, this is the reason why they are named as an **aggregator**.
- For example: When a few modules depend on three modules, we can create an aggregator module for those three modules and that way our modules car depend on a single module (the aggregator module).
- In the JDK there are several aggregator modules. For example java.se module

- JDK 9 introduced a module path.
- It can represent
  - A path to a sequence of folders that contain modules.
  - A path to a modular JAR file.
  - A path to a JMOD file (extended version of a JAR).
  - We will discuss JMOD files later in the course.
- A module path is used by the compiler to find and resolve modules.
- Every module from a module path needs to have a module declaration (module-info.java).
- A classpath represents a sequence of JAR files.

Networking

These days networking is usually discussed in the context ofr the internet, but computers may also communicate via private networks called **intranets**. In fact, thats where networking began are they're still common today, for example sending requests to an officer printer etc.

Most software teams use version control, meaning that developers have to check files int oa central repository, and other developers can check out files and work on them (for example github), but others use version control that's running on the business's intranet.

When discussing networking the machine is usually referred to as the host. A common networking configuration is **client/server**. For example, the client would be the browser and the server would handle and send the files held at that address.

```
public class echoer extends Thread{
    private Socket socket;

    public echoer(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try{
            BufferedReader input = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
            PrintWriter output =  new PrintWriter(socket.getOutputStream(), autoFlush: true);
            while (true) {
                String echoString = input.readLine();
                System.out.println("Received Client");
                if(echoString.equals("exit")){
                    break;
                }
                output.println("Echo from Server " + echoString);
            }
        }catch (IOException e){
            System.out.println("Oops" + e.getMessage());
        } finally {
            try{
                socket.close();
            }catch (IOException e){

            }
        }
    }
}
```

Standard server class, waiting for client socket to ping it. Using a new thread for each client allows us to make multiple connections to the server.

```java
public static void main(String[] args) {
// write your code here
    try(Socket socket = new Socket( host: "localhost",  port: 5000)){
        BufferedReader echoes = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
        PrintWriter stringToEcho = new PrintWriter(socket.getOutputStream(), autoFlush: true);

        Scanner scanner = new Scanner(System.in);
        String echoString;
        String response;
        // use do while loop as we don't know how many times this will execute.
        do{
            System.out.println("Enter string to be echoed");
            echoString = scanner.nextLine();
            stringToEcho.println(echoString);
            if(!echoString.equals("exit")){
                response = echoes.readLine();
                System.out.println(response);
            }
        } while (!echoString.equals("exit"));
    }catch (IOException e){
        System.out.println("Client Error " + e.getMessage());
    }
}
```

Simple client code. Reads in messages from scanner.

URI – Universal Resource Identifier

URL – Universal resource locator

Let's stick to what we need to know to write Java networking code. When working with the java.net package, a URI is an identifier that might not provide enough information to access the resource it identifies. A URL is an identifier that includes information about how to access the resource it identifies.

Another way to state this is that a URI can specify a relative path, but a URL has to be an absolute path, because when we use the URL, it has to contain enough information to locate and access the resource it identifies.

It's easy to convert between URLs and URIs, so all you have to do is provide what the method you want to use requires (a URL or a URI), and you'll be fine. The recommended practice when working with the java.net classes is to use a URI until you actually want to access a resource, and then to convert the URI to a URL.

Having said that, sometimes there's no need to start with a URI because the method you'll use to open or access a resource accepts a URL right off the bat.

According to some developers and documentation, the term URL is now outdated and shouldn't be used, but many developers and APIs still use it. So don't worry about this terminology. Understanding the sometimes razor-thin difference between a URI and a URL isn't necessary to get your feet wet with writing networking code.

When working with the low-level API, we used the following classes: Socket, ServerSocket, and DatagramSocket. When working with the high-level API, we'll use the following classes: **URI**, **URL**, **URLConnection**, and **HttpURLConnection**.

Lets start with URIs, A URI can contain nine components:

1. Scheme
2. Scheme-specific part
3. Authority
4. User-info
5. Host
6. Port
7. Path
8. Query
9. Fragment

The generic form of a URI is as follows (taken from Wikipedia here: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier):

scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]

You'll see examples of this as we progress through the lecture.

URIs that specify a scheme are called absolute URIs. When a URI doesn't specify the scheme, it's called a relative URI.

```
URI uri = new URI("db://username:password@myserver.com:5000/catalogue/phones?os=android#samsung")

System.out.println("Scheme = " + uri.getScheme());
System.out.println("Scheme-specific part = " + uri.getSchemeSpecificPart());
System.out.println("Authority = " + uri.getAuthority());
System.out.println("User info = " + uri.getUserInfo());
System.out.println("Host = " + uri.getHost());
System.out.println("Port = " + uri.getPort());
System.out.println("Path = " + uri.getPath());
System.out.println("Query = " + uri.getQuery());
System.out.println("Fragment = " + uri.getFragment());
```

```
Scheme-specific part = //username:password@myserver.com:5000/catalogue/phones?os=android
Authority = username:password@myserver.com:5000
User info = username:password
Host = myserver.com
Port = 5000
Path = /catalogue/phones
Query = os=android
Fragment = samsung
```