**Complete Java Masterclass**  
By Tim Buchalka

**UI / Shortcuts**

* Type *sout* and press enter to get *System.out.println()*.
* Type *psvm* and press enter to get ***p****ublic* ***s****tatic* ***v****oid* ***m****ain(String[] args) {}*.
* Type an ANSI colour code when outputting to console to give it colour, e.g *System.out.println (“\u001B[31m” + “Hello World.”)* to output in red.
* Use Code > Reformat Code to fix code formatting automatically.
* Double LMB the tab to make it full screen.
* Highlight code then press Ctrl+/ to comment it out/in.
* Use Code > Generate (alt + insert) to quickly generate boilerplate code for your classes, e.g. getters, setters, constructors, overridden methods, etc.
* Use View > Tool Windows > TODO (alt+6) to see any ‘// … TODO …’ comments in the code.
* Ctrl+LMB a class/method/etc declaration to see it’s definition, or do it on a definition to see uses of it.
* Code completion is when the IDE suggests entities depending on the context. The **v** refers to a variable, the **p** refers to a parameter, the **f** refers to a field, the **m** refers to a method, the **c** refers to a class, the **I** refers to an interface. The **lock** icon refers to a private variable, the **unlocked lock** icon refers to a public variable, the electric symbol refers to a throwable class or subclass.
* When highlighting a regular expression, IntelliJ gives you the option to test the expression on sample text. It doesn’t necessarily agree with *String.replaceAll()* however.

**Section 4: Variables, datatypes and operators**

* To output something to the console: System.out.println(). You can type sout and press enter to generate that quickly.
* A **literal** is a fixed value assigned to a datatype within Java, e.g. *5 (int), 10.0 (double), 244L (long), ‘a’ (char), true/false (boolean), “hello” (String), etc*. An **expression** contains literals, variables, operators or method calls. You ignore the datatype, *e.g. int num = 5; 5* is the literal, and *num = 5* is the expression. Expressions have been highlighted, and literals have been bolded:

int score = 100;  
if (score > 99) {

System.out.println(“You got the high score!”);

score = 0;

}

* **Integers** (*int*) take 4 bytes (32 bits) and can therefore store 232 possible values. Half of these are in the positive range, and half in the negative. 0 is counted as a positive value, hence the positive range is -1 of the negative range. So *int max = 2147483647*, and *int min = -2147483648*.
* Literals can contain **underscores** to make them easier to read, e.g. *int max = 2\_147\_483\_647*. The underscores can be played anywhere in the sequence if it’s not in the beginning or end, e.g. *int max = 2\_\_1\_47\_483\_647*.
* **Byte** (*byte*) is just 1 byte (8 bits) in size. It can only store values from -128 to 127. There isn’t anything smaller than a byte since a byte is the smallest data that can be fit into memory.
* **Short** (*short*) is 2 bytes (16 bits) in size. It can store values from -32768 to 32767.
* If the literal value exceeds these limits, then the compiler will give an error. You can cast the value to ‘wrap around’, e.g. *byte num = (byte) 128* (num will be -128), *byte num = (byte) 129* (num will be -127), etc.
* **Width** is the size of a datatype in bits, e.g. width of short is 16.
* **Long** (long) is 8 bytes (64 bits) in size. To create a long literal you must append L, e.g. *long num = 100L*. You can use lowercase L, but uppercase is easier to read.
* In width order: byte < short < int < long. To convert a larger width datatype to a smaller width datatype you will have to cast. So all other types must be manually casted to byte, whereas none of them need to be casted to long as the compiler will automatically do it. Examples: *int num1 = 5*; byte *num2 = (byte) num1*; *long num3 = num1*.
* **Float** (float) is 4 bytes in size. To create a float literal, you must append f or F. A float has 7 digits of precision. You can use underscores with floating point values. If you attempt to do calculations with the previous types that include decimals, the decimal/remainder will be disregarded, e.g. *int num = 5/2* will return 2 since .5 is ignored. Float and double keep the remainder.
* **Double** (double) is 8 bytes in size. To create a double literal, you must append d or D, or type a decimal value, e.g. *double num = 5d*, *double num = 5.25d*, or *double num = 5.25*. A double has 16 digits of precision. Double calculations tend to be faster than float calculations on modern computers, even though it takes more space. Float is single precision, and double is double precision.
* In width order: float < double. As previously stated, larger types must be manually casted to small types. Smaller types automatically cast to larger types.
* **Char** (char) takes 2 bytes. It’s used to store a single character such as a letter, number, Unicode character, or escape sequence. Example: *char val = ‘A’*, or *char val = ‘\u00A9’* (Unicode for copyright symbol). The apostrophes define a character literal. Unicode character tables may be useful: <https://unicode-table.com/en/>.
* **Boolean** (boolean) takes 1 byte. It stores the state of something, e.g. is it true or false, or *boolean isMale = true*. ‘true’ and ‘false’ are Boolean literals.
* All the datatypes covered up to now are primitive types defined within Java. These basic datatypes can be used to create more sophisticated datatypes, such as string. All other datatypes are referred to as a class.
* **Strings** (String) takes a variable number of bytes. It is a sequence of characters, so it will take some bytes for each character you type (including spaces), plus more bytes since strings are automatically null terminated. Speech marks are used to define a string literal. Example: *String str = “Hello World”*. Strings support direct concatenation, e.g. *String str = “Hello” + “ world!”*. It can also do this with all primitive types, e.g. *int num = 50; String str = “Hello ” + num*, output: “Hello 50”. When adding objects together in a string, *toString()* is automatically called for that object.
* **Operators** can work on 1 (unary), 2 (binary) or 3 (ternary) operands. Unary ops: (post-fix, pre-fix). Binary ops: arithmetic (multiplicative, additive), assignment (=, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=), bitwise (<<, >>, >>>, &, ^, |), logical (&&, ||), relational (comparison, equality). Ternary operator: conditional operator.
* When assigning values, you can chain them, e.g. *int a, b, c; a = b = c = 5;*. This is read from right to left and sets all to 5. First c is set to 5, then b is set to c, then a is set to b.
* **Relational (comparison, equality) and logical operators** check to see if the two operands match a condition, in which case the output is true, otherwise false. All of them are binary operators except for the not operator, which is unary.
* The **conditional operator** checks to see if the condition is true, and returns the first statement, otherwise it returns the other statement, e.g. *(5>4) ? “Yes” : “No”;* returns “Yes”. In general the syntax is *condition ? if\_true : if\_false;*. You can return literals, expressions, statements, etc.
* For an if statement to be valid, the expression must evaluate to a boolean. This is why *boolean bool = false; if (bool = true);* compiles, but *int num = 5; if (num = 5);* doesn’t.

**Section 5: Expressions, statements, code blocks, methods, etc.**

* There are 53 reserved keywords in Java. Keywords are reserved names defined within Java that carry out specific functionality.
* **Scope** refers to the **block** {} in which a variable is defined. Variables are automatically deleted once they fall out of scope - at the end of the block they were defined in. Variables cannot be accessed outside of their scope.
* **Methods** allow you to type code once and reuse it as needed, thus preventing code duplication. This is ideal because code duplication can lead to an increase in errors since the programmer may forget to make identical changes in the duplicated code.
* **Parameters** are datatypes + variable names expected by a method. **Arguments** are the actual values/objects being supplied to a method call.
* **Procedures** don’t return anything. **Functions** return something. **Methods** are procedures or functions defined within a class.
* **Method overloading** is when a new method is created that has the same name as another method, but has different parameters. The compiler decides which method to call based on the arguments provided.

**Section 6: Control flow statements**

* These change the flow of the code depending on various conditions, e.g. **if statement, switch statement, for loop, while loop, do while loop.**
* If statements are like switch statements. Switch statements can only test one variable, can only test for equality (can be OR’ed), and can only equate constant values. Switch statements should only be used for simple test cases. Examples:

|  |  |
| --- | --- |
| int num = 3; if (num == 1)  System.out.println(“1”);  else if (num == 2 || num == 3)  System.out.println(“2 or 3”);  else  System.out.println(“<1 or >3”); | int num = 3;  switch (num) {  case 1:  System.out.println(“1”);  break;  case 2: case 3:  System.out.println(“2 or 3”);  break;  default:  System.out.println(“<1 or >3”);  break;  } |

* An if statement’s ‘else if’ is equivalent to a switch statement’s ‘case’. An if statement’s ‘else’ is equivalent to a switch statement’s ‘default’.
* You must type break after every case or execution will fall through to the next case, this is how you OR different cases together.
* Switch statements can only switch on byte, short, int, char, their wrapper classes, enum, and String.
* Syntax for an indexed for loop is: *for (init; condition; action) {}*.

|  |  |
| --- | --- |
| for (int i = 0; i < 5; i++) {  System.out.println(i);  } | for (int i = 0, j = 5; i < 5; i++, j--) {  System.out.println(i + " " + j);  } |

* By convention, simple variables are used to track the index, e.g. *int i, int j, int k, …*
* *String.format("First: %d Second: %.2f Third: %c", 5, 10.0, 'c')*, will output “First: 5 Second: 10.00 Third: c”. The % notation refers to the corresponding argument, e.g. the first % refers to the first argument etc.
* Nested if statements are more efficient than sequential if statements, if the sequential if statement depends on code executed in the former if statement. Example:

|  |  |
| --- | --- |
| int primes = 0;  for (int i = 1; i < 20; i++) {  if (isPrime(i)) {  System.out.println(i);  primes++;  }    if (primes >= 3)  break;  }  } | int primes = 0;  for (int i = 1; i < 20; i++) {  if (isPrime(i)) {  System.out.println(i);  primes++;  if (primes >= 3)  break;  }  } |

The code on the left checks two conditions every time, while the code on the right checks two conditions sometimes.

* For loops are ideal for looping a certain number of times. While loops are ideal for looping until a certain condition is met. Do while loops are the same as while loops, but run at least once since the condition is checked after the loop.
* You can use the **break** keyword to exit a loop if a condition is met, or you can use the **continue** keyword to start the next iteration of the loop.

**Section 7: OOP Part 1: Classes, constructors, and inheritance**

* It’s a naming convention to start classes with a capital letter (upper CamelCase), and to make it a noun. Methods use lower camelCase and are verbs. Classes should be created in their own files with the same name as the class.
* Each section in the package name refers to another subfolder. If the package name is “com.cjm.ms” and the new class is created in a file called car, then it will be stored as “ProjectPath/com/cjm/ms/car.java”.
* **Public** is an **access modifier**, as is **private**, **default** and **protected**. Public gives unrestricted access to the class. Private means the class can only be accessed from within. Default means the class can only be accessed by classes within the same package. Protected means the class is accessible within the package, or from another package through inheritance. Most of the time you will use public. You can remove the access modifier, but this is equivalent to default.

Top level access modifiers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Can be accessed from: | Inside the class only. | Current package only. | Any package w/ inheritance. | Anywhere. |
| public | x | x | x | x |
| protected | x | x | x |  |
| default/none | x | x |  |  |
| private | x |  |  |  |

* A class is a blueprint for an object you will create. They are used to model real world problems.
* Variables defined at class scope are referred to as member variables or fields. They also require an access modifier. Unlike with class access specifier you will tend to mark these private for encapsulation. The idea is to prevent accidental misuse of what your class was designed for by hiding data from external access. In other words it can only be access from within the class.
* The fields within a class represent the **state** of the class at runtime since they are the only entities which can be changed. The methods represent the **behaviour** of the class.
* Typing “*Car car;*” will create a reference variable. Typing “*new Car();*” will create a Car object in memory. To point the reference to the object, type “*Car car = new Car();*”. To have a reference point to nothing use **null**, e.g. “*Car car = null;*”.
* Variables with primitive datatypes and reference variables exists in **stack memory**, while objects exist in **heap memory**. The stack is cleared when execution reaches the end of a code block. If no references point to an object then the **garbage collector** will delete the object at an undetermined time.
* **Setter methods**, or setters, are used to set fields from outside of the class.
* When referring to a field with a method that has the same name as a local variable, use the **this** keyword to specify the field. Otherwise the compiler will assume you want the local variable.
* **Getter methods**, or getters, are used to retrieve a field from outside of the class.
* You use setters and getters to put logic in between the user of your class and them setting/getting fields. Even if you do not need to put any logic in between you should still follow this practice as in the future you may need to add logic. The benefit of this is that if you decide to add something in the future, users of your class will not need to change all of their code to accommodate your new setters/getters.
* An example of adding logic is input validation. You may want to check what input the user is sending before assigning it to the field. There is also verification, exception handling, logging, etc.
* You can either initialise an object using setters or, more preferably, a constructor. Java creates a default constructor automatically if no custom constructor is defined. A default constructor does not take any arguments because it has no parameters, e.g. “*Car car = new Car();*” will use a default constructor. If none are defined implicitly or explicitly, this will result in a compilation error. A constructor is syntactically similar to a method, except that it has no return type and runs automatically when a class is instantiated.
* Constructors can call one another, or other methods, which may be ideal for code reuse. There is debate, but using other methods/setters is not recommended in the constructor. Other methods cannot call a constructor since they are special methods that only run when the class is instantiated.
* Classes should carry out specific functionality rather than trying to do everything. You can **extends** the functionality of a class by using **inheritance**. The simpler class is called the **superclass** while the more complex class using inheritance is called the **subclass**. Inheritance represents an **is-a** relationship.
* The superclass should only contain functionality that will be common to all subclasses. Example: you can have a superclass called Animal, that may be subclassed by Dog, Cat, etc. Animal should contain state/behaviour that is applicable to all shapes, while Dog and Cat will contain state/behaviour specific to them.
* ‘this.’ is used to access fields and methods within the current class. ‘super.’ is used to access fields and methods within superclasses. These cannot used by static blocks/methods. ‘this()’ is used to call constructors within the current class. ‘super()’ is used to call constructors within the superclass. The constructor calls can only be made from other constructors and must be the first statement.
* Member level access modifier result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Can be accessed from: | Inside the class only. | Current package only. | Any package via subclassing. | Anywhere. |
| public | x | x | x | x |
| protected | x | x | x |  |
| default/none | x | x |  |  |
| private | x |  |  |  |

* The subclass can **override** methods from the superclass. Thus the subclass’ version of the method is called instead of the superclass’ version. You can do this to run specific functionality in the subclass, or to extend functionality from the superclass’ version you can use *super.methodname(…)*. Example: Animal may contain the eat method since it applies to all animals, but a dog might eat differently to a cat and so they can both override Animal.eat(…) to provide their own definition/functionality. If they have some common aspects in the way they eat then that common functionality can be defined in Animal.eat(…) and the dog or cat can use super.eat(…) to run that functionality before running their own.
* When overriding a method you must use the same method name, parameter types, and return type.
* You can call superclass versions of methods without using the super keyword as long as there is nothing else with that name in scope. This is recommended because if you later override a method in your subclass, it will automatically use that version of the method instead.
* All classes subclass the **Object** class. When you create a new class without using the extends keyword, it implicitly extends from Object. You can also type *extends Object* to make it explicit, but it is not required.
* A **class** is a blueprint that defines state and behaviour for a datatype. An **instance** of a class is created using the new keyword, or you can say that using the new keyword instantiates a class. Instances of a class exist in heap memory as **objects**. **Reference** variables point to an object.

**Section 8: OOP Part 2: Composition, encapsulation, and polymorphism**

* While inheritance represents is-a relationships, **composition** represents **has-a** relationships. Example: A dog is an animal, a square is a shape, a car is a vehicle, etc. Example: A dog has a heart, a square has an edge, a car has an engine, etc. To use composition make one class a field in another class.
* Java can only use single inheritance to prevent over-complicating code. Thus composition is another way to extend classes.
* You can instantiate a class directly in a method call if it won’t be used anywhere else.
* **Encapsulation** is used for data hiding and to protect the inner workings of a class. This ensures that a class follows it’s defined behaviour and protects your design. It also allows you to rename fields without affecting dependant classes. It also allows you to introduce logic before returning the field the user of the class sets/gets. Encapsulation is achieved through access modifiers, and getters and setters.
* You can create multiple classes within one file, but it is usually not recommended since it can make your code confusing and unorganised. It is usually only done if the class will not be reused anywhere else in the code. A public class must exist within its own file.
* Use *Math.random()* to return a double literal from 0 <= x < 1.
* **Polymorphism** (many forms) allows entities with the same name to act like something else. There is **static polymorphism** which refers to method overloading, and **dynamic polymorphism** which refers to method overriding. The former is called static because the appropriate method to call is decided at compile time. The latter is called dynamic because it is decided at runtime.
* Dynamic polymorphism allows references of one class to point to an instance of a derived class at runtime. Example: *Animal animal1 = new Animal()*, and *Animal animal2 = new Dog()* are both valid assuming Dog is a subclass of Animal. If you call a method within animal1, e.g. *animal1.eat()* it will call *Animal.eat()*. If you call *animal2.eat()* it will still call *Animal.eat()*. If you first override *Animl.eat()* within Dog, then calling *animal2.eat()* will call *Dog.eat().* This allows you to feed all your different types of animals in one loop rather than having to treat each animal differently.
* The fact to note is that a reference for one datatype can point to an instantiation of any of its subclasses. You will only be able to use methods defined in that type, or methods that have been overridden from that type. For example, an Object reference can point to any object in memory since they all inherit from this class. However, you will only be able to call methods declared in Object.
* Call *getClass().getSimpleName()* to get the name of the class without having to type it. This method is defined within the Object class. It doesn’t display the name of the static class, but rather the dynamic class.

**Section 9: Arrays, Java inbuilt lists, autoboxing and unboxing**

* Regular variables store a single value. **Arrays** can hold multiple values of the same type. You can make an array of any type. Example: *int var = 5* stores a single value, and *int[] values = {5, 7, 9}* stores multiple. Arrays use a zero-based index, e.g. first element (5) is 0, second element (7) is 1, third element (9) is 2, etc. Trying to access the fourth element in this case will throw an ArrayIndexOutOfBoundsException.
* There are a few ways to initialise an array: int[] arr1 = {5, 5, 9}; int[] arr2 = new int[]{5, 5, 9}; int[] arr3 = new int[3]; arr3[0] = arr3[1] = 5; arr3[2] = 9;.
* The length of the array during initialisation can be a literal or an expression.
* You can use a for loop to go through an array, or use a for each loop if you don’t care about the index. Use arr.length to determine how many iterations to run the indexed loop for as opposed to hardcoding it.
* To get **input** from the user, use the Scanner class: *Scanner sc = new Scanner(System.in)*. Then call one of it’s many methods to store a value: *int val = sc.nextInt()*.
* Since arrays have a fixed length it isn’t simple to add another value. You must create a new array that’s larger than the old array, then copy the original array to the new array, and then add the values that you originally intended to add. This is because arrays occupy contiguous memory. So each of the elements appear next to each other in memory as they do in the array. This allows for fast random access but makes it slow at resizing.
* Alternatively, a **list** has a dynamic size and makes it easy to add another value. This is because the list contains nodes that point to the next node in memory. So the data is stored randomly in memory. This means that to get the 5th value you must go through the first four nodes. This makes it slow for random access, but fast for resizing. A **doubly linked list** has a previous node and a next node.
* To create an ArrayList you can either use *ArrayList<String> list = new ArrayList<String>()*, or *List<String> list = new ArrayList<String>()*. The latter is recommended since you’re coding to the interface which makes use of polymorphic concepts. You can remove the second <String> from either since the compiler can use the first one to determine that it must be the same.
* Arrays and lists are examples of **collections**. Collections in general store multiple values, but with different rules. Arrays and lists are **ordered collections**, meaning that the order of insertion is kept. List is an interface that’s implemented by many different types of list subclasses. The most common of which is ArrayList.
* An ArrayList works like a resizable array, while a LinkedList works like list.
* You cannot access collections elements directly and must use getters/setters and other methods for manipulation. Example: *ArrayList.get(), ArrayList.set(), ArrayList.remove(), ArrayList.contains(),* etc.
* When using methods such as *remove()* or *contains()*, you must make sure that the argument you pass is a reference to the same object in memory, or that the class being compared overrides the *equals()* method. Otherwise the operation will fail as the *List* implementation has no way of knowing if the elements match. If you don’t override *equals()*, then *List* will use the default implementation provided by *Object.equals()* which compares memory addresses and returns true if they match.
* To create a copy of an array use *Arrays.copyOf()*. To create a copy of an ArrayList use *ArrayList.addAll()* or just pass an existing ArrayList into the constructor of the new ArrayList. An ArrayList can be converted to a built-in array using *ArrayList.toArray()*.
* Although you can do ‘ArrayList<String>...’ you can’t do it with primitive types ‘ArrayList<int>…’. You must use wrapper classes for the primitive types. For int this is Integer. **Autoboxing** is when you convert a primitive type to its **wrapper class**. **Unboxing** is the opposite.
* **Value type** and **reference type** refers to two types of variables in Java. A value type is a variable with a primitive datatype. A reference type is a variable with a non-primitive datatype.
* Both value types and reference types exist in stack memory and are deleted when execution reaches the end of their code block.
* Reference types point to objects created in heap memory. Objects are created in heap memory every time the new keyword is used. When the number of references pointing to an object in heap memory is 0, the garbage collector will delete that object. This is done non-deterministically.
* When assigning one variable to another or when passing variables as arguments, a copy of the variable is created, so changing the new variable doesn’t affect the old variable. Variables are passed by value in Java. When assigning one reference type to another, they both point to the same location in heap memory, so changing the object through one reference is the same thing is changing it through the other.
* **Iterator** can be used to transverse collections. Methods: hasNext(), next(), remove(). **ListIterator** can do much more. Methods: add(), hasNext(), hasPrevious(), next(), nextIndex(), previous(), previousIndex(), remove(), set(). A for each loop is syntactic sugar for iterators when traversing a collection, but if you’re going to add/remove elements while traversing the list you need iterators.
* An Iterator works like a list, while a ListIterator works like a doubly linked list.
* When you call ListIterator.next() and then switch to ListIterator.previous(), it will actually return the same element because of the way the cursor/position works. You must call it twice to actually return the previous element. This happens anytime you change direction, so calling ListIterator.previous() then ListIterator.next() will have the same result. This is because a ListIterator doesn’t have an index, but just has references to the next and previous nodes. This is why it’s said to sit in between two nodes.
* A **side effect** is when calling a method changes the object in such a way that calling the method again returns a different result, e.g. iter.next(). Side effects should generally be avoided.
* If you try to use a ListIterator after changing add/removing elements from the original list, you will get a ConcurrentModificationException. So either only use ListIterator to set/remove elements to the original list, or create a new ListIterator after making changes to the original list.

**Section 10: Inner classes, abstract classes, and interfaces**

* An **interface** ensures common behaviour between different classes. The classes must implement the interface and provide definitions for declared methods.
* It is convention to start interfaces with an ‘I’, e.g. IAnimal, IVehicle, etc.
* Classes are defined using the class keyword. Interfaces are defined using the interface keyword.
* They can only contain public fields, public method signatures, or private defined methods.
* Interface methods are public by default.
* Use the **implements** keyword after a class declaration to implement an interface. All public methods must be given a definition, or if the class is abstract then it is optional. You can use code generation to speed this up. A class can only ‘extends’ one class, but implement many interfaces.
* Interfaces can’t be instantiated, but a reference variable for an interface can make use of polymorphism to instantiate classes that implement the interface, e.g. *ITelephone timsPhone = new DeskPhone()*.
* This is why you should always code to the interface as the resulting code isn’t specific to a single class, e.g. use *List<String> str = new ArrayList<>()* instead of *ArrayList<String> str = new ArrayList<>()*. This way you could change *str* to other lists and reuse code, e.g. *str = new LinkedList<>()*.
* **Inner class** refers to a class defined within a class. It can also be called a nested class. There are four types. **Outer class** refers to the class that contains the inner class. You can also create an inner interface.
* Inner classes make sense to use when the inner class won’t be used anywhere else in the program, e.g. a Gear class will only need to be used to by a GearBox class so it can be an inner class.
* **Non-static inner class** (or just inner class) are defined within classes, but outside of methods. They can access all fields and methods defined in the outer class including private entities. To access a field defined within the outer class that’s also defined within the inner class use *Gearbox.this.gearNumber*. The inner class field is **shadowing** the declaration of the outer class field. Avoid shadowing declarations as it reduces code quality. To access a public inner class from outside the outer class use dot notation, e.g. *Gearbox mcLaren = new Gearbox(6); Gearbox.Gear first = mcLaren.new Gear(1, 12.3);*.
* **Static inner class** is the same as a non-static inner class except that it uses the static keyword in the declaration. This means that the inner class belongs to the outer class, rather than an instance of the outer class. To access a public static inner class from outside the outer class use dot notation, e.g. *Gearbox.Gear first = new Gearbox.Gear(1, 12.3)*.
* **Local class** is a class that is defined within a method.
* **Anonymous class** is a class with no name, e.g. IOnClickListener action = new IOnClickListener() { @Override public void onClick(String title) { System.out.println(title + “was clicked.”); } }. The class returned in this example is a subclass of the interface IOnClickListener, but since the subclass doesn’t have a name that can be used to instantiate another instance of the class, it’s an anonymous class.
* **Abstract class** is a class that can’t be instantiated without being inherited from. That’s because it defines ‘abstract’ methods which work exactly like interface methods, i.e. they are declared without a definition and the definition is provided by the subclass.
* It’s best to use an abstract class when a class contains common functionality, but has no useful meaning on its own. An example of this is that an Animal class could contain state and behaviour that’s used by every animal, but Animal on its own doesn’t describe any specific animal and as such is meaningless to use directly. You can mark Animal as abstract and then inherit it by a Dog or Cat class and give it meaning, e.g. *public class abstract Animal { … }*.
* Abstract methods, e.g. *public abstract void eat()*, must be implemented in immediate subclasses unless the immediate subclass is also abstract.
* Abstract methods can only go in an abstract class, but an abstract class doesn’t require abstract methods.
* Abstract class vs. interface: Interfaces can only have ‘public static final’ fields, they don’t have constructors, can’t have public implemented methods, many interfaces can be implemented by one class, neither can be instantiated, etc.

**Section 11: Generics**

* Methods can take arguments as parameters. Similarly classes, interfaces, and methods can take **type parameter** arguments, e.g. *List<String> list = new List<>()*. This is referred to as **generics** as code can work with different data types. Type parameter arguments are provided in angled brackets <…>.
* Generics reduces code duplication and increases code reuse.
* To specify one type parameter: *class Test<T> {…}*, to specify two type parameters: *class Test<T, U> {…}*, etc.
* If you omit the type parameter, then Java assumes that the type parameter is Object, e.g. *List items = new ArrayList()* is equivalent to *List<Object> items = new ArrayList<Object>()*. This allows items to accept any non-primitive datatype since they are upcasted to Object. This isn’t recommended because it can break type checking.
* **Bounded type parameters** limit the datatypes that will be accepted by a class. Unbounded example: *class Team<T>*. Bounded example: *class Team<T extends Player>*. Another benefit is that the compiler will know what methods to expect since T subclasses Player. With unbounded type parameters you will need to use casts which can be hard to read.
* To specify multiple upper bounds use the following syntax: *class Team<T extends Player & Coach & Manager>*. Similar to inheritance, you must have one class followed by interfaces.

**Section 12: Naming conventions and packages / static and final keywords**

* **Naming conventions** refer to a set of rules enforced by the programmer to write consistent code so that when other programmers look at the code they can quickly understand it.
* The following entities can be named: packages, classes, interfaces, methods, constants, variables, and type parameters.
* Package names are always lower case, unique, and use the reverse of your Internet domain name as a prefix (e.g. com). Replace invalid characters with an underscore. Examples: ‘java.lang’, ‘java.io’, ‘org.xml.sax.helpers’, ‘com.timbuchalka.autoboxing’.
* Class names use upper CamelCase, are nouns, start with a capital letter, and each word in the name starts with a capital. Examples: ArrayList, LinkedList, String, TopSong, GearBox, Main.
* Interface names use the same rules as class names. Examples: List, Comparable, Serializable.
* Method names use mixedCase (lower camelCase), are verbs, and reflect function performed or result returned. Examples: size(), getName(), addPlayer().
* Constant names are all upper case, separate words with an underscore, declared using the final keyword. Example: static final int MAX\_INT, static final short SEVERITY\_ERROR.
* Variable names are mixedCase, meaningful/indicative, start with lower case letter, and do not use underscores. Examples: I, league, sydneySwans, boxLength.
* Type parameter names use a single capital letter. Example: T, U, K, E.
* Due to number of programmers and common naming conventions, naming conflicts are probably. **Packages** help solve the problem by providing a way to couple code with the developer/library. They allow using classes with the same name within the same project, or even within the same class. They also make it clear which classes are related as classes pertaining to a specific problem could all be contained within one package. Classes within a package can have unrestricted access to each other while restricted access to classes outside package (default access).
* If a class will be used in many points within a file its best to use an **import** statement, e.g. ‘*import java.util.List*’. If a class will only be used in one place you can directly specify it, e.g. ‘*java.util.List<String> list = new java.util.ArrayList<>()*’.
* You can import static methods directly, e.g. *‘import static org.junit.Assert.assertEquals;’*.
* The asterisk imports all classes, interfaces, and static objects from a package – e.g. *‘java.util.\*’*. It won’t automatically import packages within packages though, e.g. *‘java.awt’* is different to *‘java.awt.event’*. Use \* sparingly since it automatically imports all names which could lead to namespace pollution if many common names are being used.
* If two or more imports have the same class defined, then only one import must remain. The other class declarations must use a fully qualified name.
* ‘java.lang’ is implicitly imported and contains fundamental Java classes, e.g. wrapper classes, String, Math, Object, etc.
* The package name must match the directory that the file is contained within since each section of the package name refers to another sub-folder.
* Java files have a **package** statement at the top of the file which indicate which package the file belongs to, e.g. com\cjm\ms\Main.java will have ‘*package com.cjm.ms;*’ as the first line.
* To turn a project into a **library** in IntelliJ IDEA, go to Project Structure > Artifacts > + > JAR > From modules with dependencies. Select a ‘Main class’ to create an executable, otherwise leave it, and press OK. Press Build > Build Artifacts to output the JAR file.
* To use the library in another project File > Project Structure > Libraries > + > Java > Select JAR file > OK. When you import the library, it exist in its original location and isn’t copied over. This allows IntelliJ IDEA to automatically load it when changes are made.
* **Scope** refers the visibility of a class, member, or variable.
* When an entity is used the compiler checks the local code block first, followed by enclosing code blocks to find the name before giving an error.
* Similarly, when a method is called the compiler checks the current class, then the super class, then its super class, etc.
* Local variables take precedence over fields with the same name.
* A method within an outer class can access private fields from within an inner class, or an inner class of an inner class, etc.
* Only classes, interfaces, and enums can exist at the top level. All other entities must be included within one of these.
* A **static field**, also known as a **class variable**, is associated with a class rather than an instance of the class. So there’s only one copy of it in memory regardless of how many class instances you create. You can access the field anywhere by typing *ClassName.staticField* (if it’s public).
* A **static method**, also known as a **class method**, is similar to a static field in the sense that you don’t need to create an instance of a class to access it. You can use the class directly, e.g. *ClassName.staticMethod()* (if it’s public).
* If a method only deals with static fields/class variables, and/or if it uses local variable only, then you should mark it as a static method. If a method deals with instance variables it can’t be marked static.
* The main() method is marked static because the JVM needs an entry point into the program, but until main() is called no class instances are created, hence it must be marked static. It doesn’t matter what main()’s containing class is called, but the convention is to call it Main.
* **Static initialisation blocks** are like constructors, but for static final fields. They are run once when the class is initially loaded. Static final fields must be assigned to during declaration OR in the SIB. You can have many SIBs as required, they will run in the order declared before any other constructor/method.
* The **final** keywords marks entities as constants.
* A final field can be assigned a value either during declaration, or in the constructor, but never again. It’s best to set a field as final if you know its value should never be altered more than once.
* Declare fields as static final if they will be constant for the class rather than each instance.
* Marking a class as final stops it from being subclassed.
* Marking a method as final stops it from being overridden. This is recommended for when a method is crucial to the workings of your class and you don’t want it to be replaced.

**Section 13: Collections**

* The **collections** interface is a superinterface for set, list, queue, and deque. Map is a separate interface so it isn’t a collection.
* *List<String> list = new ArrayList<>(otherList)* just creates a **shallow copy** of the otherList. Shallow copy means that all of the references are copied, so if you use the second list to change an element, it will be reflected in the other list since it’s only the references that were copied. **Deep copy** is when the objects themselves are copied over such that the original objects aren’t affected.
* Methods such as Collections.sort() and Collections.reverse() require classes to override the compareTo() method from the Comparable interface. This is so that the algorithm can compare different objects to see which is greater than the other.
* **Comparator** does the same thing except a class isn’t required to override this interface. You can define an anonymous inner class for *Comparator<T>* and override the *compare()* method. You can then provide it as an argument to *Collections.sort()*. This is ideal when you’re working with a class that you can’t change. You can define as many comparators as you want and they can sort in different ways.
* Comparators can cause problems with sorted sets and maps since they might not be consistent with *equals()*.
* Maps require two type parameters, for the key type and for the value type. The key is a unique index, which is associated with a value. Use *put()* to insert a key-value pair in a map, use *get()* to retrieve a value associated with the given key.
* If you *put()* using a pre-existing key, its value will be overwritten. *put()* returns the previous value associated with the given key. Initially it will return null.
* You can use *contains()* to check if a key exists to prevent accidental overwrites.
* To loop through all key-value pairs, use a for each loop on *keySet()* as it will return a *Set* object that contains all the unique keys within the map. Then in the body of the loop use *get()* with the current key to retrieve its value. Keys aren’t stored in a predictable order so don’t rely on that.
* *remove()* will remove a key-value pair given a key. It can also be provided with a key and value in which case it will only remove the pair if both match. The method returns true is the key-value pair is removed.
* *replace()* will replace a value for a given key.
* To make a class immutable, remove any way of setting fields after they have been initialised in the constructor. Mark fields as private and final. Do not use setters. Mark class as final. Do not allow setting a field directly through the constructor, e.g. instead of *this.myMap = myMap*, write *this.myMap = new HashMap<>(myMap)*. This prevents the calling class from modifying your local copy of the map.
* ***Set*** data structure doesn’t have any defined ordering, and doesn’t contain duplicate data (contains keys).
* There’s no method to get a key at a specific index. You can use a for each loop to iterate through each key. You can use the *contains()* method to see if a key exists within the set. Sets can create unions by using the *addAll()* method, duplicates are ignored. These are called bulk operations, others include *containsAll()* - returns true is the set is a subset of another set*, retainAll()* – performs an intersection and contains common elements from both sets*, and removeAll() –* keeps the asymmetric difference/unique elements. To find symmetric difference, find the union then remove the intersection.
* If a class is used as a key in a *Map* or *Set* it should override the *equals()* and *hashCode()* methods. Otherwise the default implementation defined in *Object* will be used.
* By default *Object.equals()* will just check to see if the objects being compared have the same address in memory. When overriding *equals()* ensure that *subclass.equals(superclass)* is equal to *superclass.equals(subclass)*. This could be a problem is the subclass checks to see if the object argument is an instanceof itself.
* If two objects are equal then they must generate the same hash code otherwise the objects will end up in different ‘buckets’ which will cause problems when trying to find/remove those elements. You can just *return 0* in the overridden hash code method since it satisfies that constraint, but this misses the point of the using hash tables and lowers performance to that of a linked list since the algorithms must go through each element.
* The override annotation isn’t required, but the compiler ensures that you override a method rather than accidentally overloading it, e.g. you might accidentally write *public boolean equals(ClassName o)* rather than *public boolean equal(Object o)*.
* **Enumerators**, or enum, are useful when creating a list of possibilities. A method can accept an integer that uniquely represents many cases, but if the possibilities are limited and have names an enum can be more useful. Example: *public enum HumanLifecycle {FETUS, BABY, TEENAGER ADULT, SENIOR}.* This prevents invalid values from being passed and makes code easier to read.
* While *HashSet* and *HashMap* don’t have any particular order, their linked variants maintain insertion order – *LinkedHashSet* and *LinkedHashMap*, and their tree variants maintain *ClassName.compareTo()* order – *TreeSet* and *TreeMap*. Naturally the additional sorting means that these collections will be slower than the unsorted implementations.
* *Collections* contains static methods that return unmodifiable variants of lists, sets, and maps. This only makes the collection unmodifiable, the underlying data can still be retrieved and altered.
* *Objects.hashCode()* can be used to generate a hash code for a single object. *Objects.hash()* can be used to generate a hash code for an array of objects. When generating a hash code you should use the same variables you use in the *equals()* method to determine which objects are equal. In other words use the variables that together uniquely identify an object. For a bank account this could just be a unique customer ID. For a person it might be their name, birthday, and ethnicity that uniquely identify them since many people can share the same names and birthdays.

**Section 14 – JavaFX**

* JavaFX is the successor to Java Swing. Can create UI for desktop/Internet/mobile applications.
* Designed with MVC in mind - model, view, and controller. It keeps the application/logic code separate to the UI code. Controller handles logic between UI and data.
* JavaFX project automatically creates a *Controller* class - controller, and a *fxml* file – view.
* JavaFX programs must extend the *Application* class. It manages the lifecycle of the program.
* The important methods are *init(), start()* and *stop()*.
* A stage refers to a top-level container. A scene refers to controls within a stage.
* There are 8 layouts which can be used to automatically manage position and resizing behaviour.
* Preferred size refers to how controls decide their size based on their contents.
* GridPane is a table based layout which can be manipulated by row/column indices.
* The hgap and vgap properties refer to horizontal and vertical column/row gaps between each cell.
* Set gridLinesVisible to true to see how the grid pane has been formatted.
* ColumnContraints element ensures that a certain percentage of a column is always visible.
* rowSpan/columnSpan properties allows elements to span multiple rows/columns.
* halignment/valignment properties sets the horizontal/vertical alignments of an element.
* Padding and insets elements are used to pad around the border.
* HBox lays out elements horizontally. VBox lays them out vertically.
* The style property is used to enter CSS commands.
* The spacing property adds a space between elements.
* The prefWidth button property can be used to set the preferred width.
* BorderPane consists of five sections: left, right, top, bottom, and centre. The space remaining after the border sections is assigned to the centre section.
* Layouts can be nested. This provides extra control regarding how elements are positioned.
* Nest the layouts under left/right/top/bottom/centre elements within the BorderPane element.
* The BorderPane.alignment property can be used by a child to align it within it’s section.
* FlowPane is similar to HBox/VBox, but it automatically repositions elements based on available space.
* The orientation property determines whether new elements are placed horizontally or vertically.
* TilePane is similar to FlowPane, but it maintains consistent cell size based on the largest element.
* StackPane places elements on top of each other. It is similar to the concept of layers in Photoshop.
* To add an image to a button use ‘graphic > ImageView > Image’ elements and specify the URL.
* Set wrapText to true to allow a label’s text to wrap.
* RadioButtons are used to force the user to select one option. They must be grouped by being assigned to the same ToggleGroup.
* A dollar symbol is used to specify an fx:id value.
* CheckBoxes are used to allow the user to select multiple options. They usually have two states. The third state can be enabled by setting indeterminate to true.
* ToggleButtons are buttons that behave like checkboxes by default. They can also be put into a ToggleGroup.
* TextField allows users to enter in data.
* Password field is similar to TextField, except that it masks input and you cannot copy its data.
* ComboBox provides a drop down menu from which a user can select a single option. Use ‘items > FXCollections > String’ elements to add options. Use the value element to set the default. Set editable to true to allow users to edit the options.
* ChoiceBox is similar to ComboBox except that it marks the user’s choice with a tick.
* Slider lets the user select a value within a range. Use min and max properties to set the range. Set showTickLabels to true to let the user see what value they’re selecting.
* Spinner is similar to Slider, but it allows the user to be precise with the value that they want.
* ColorPicker opens a dialog box to allow users to select a specific color.
* DatePicker opens a dialog box to allow users to select a specific date.
* TitledPane sounds like a layout, but it’s a control. It allows grouping several elements together which can then be collapsed or opened by clicking on the control.
* Accordion allows grouping multiple TitlePanes such that only one is open at a time.
* When a user interacts with a control it raises an event on the UI thread. Write event handlers/listeners in the Controller class to respond to these. Use the button’s onAction property to specify the event handler’s name with a hashtag.
* Procedural programs are those that mainly run from the entry point to the end of the code, e.g. console applications. Event-driven programs are those that don’t have a pre-determined order and depend on the user’s input to decide what to do next.
* To reference a control from code assign an fx:id to it. Then go to the source file and create an instance variable with the same name as the id. Use the *@FXML* annotation to associate the two.
* Although not required, you should put the *@FXML* annotation above event handlers to make it clear.
* The *initialize()* Controller method is automatically called.
* Event handlers should return as fast as possible because they all run on the UI thread. Running too much code in them will make it look as if the program has hanged and prevent further user input.
* UI controls can only be used on the UI thread to prevent complicated bugs. Use *Platform.runLater()* to make sure that code specific to UI is run on the UI thread. Otherwise the code will throw an exception. This was done to prevent complicated UI bugs.
* The FXCollections methods are just wrappers for different types of Java collections, e.g. *FXCollections.ObservableArrayList()* returns a wrapper for *ArrayList*. It’s more efficient since it only raises an event once rather than multiple times.
* *Path*, *Paths*, and *Files* are from Java NIO which is a new set of classes for dealing with data.
* Use ‘*MenuBar* > *Menu’* elements to create a menu, and ‘*items* > *MenuItem’* to create items in a menu.
* *DialogPane* simplies the process of adding a dialog box. Create a separate fxml file for it since each window requires its own fxml file. The main text goes under the *headerText* element. The body goes under the *content* element.
* *ListView* items exist in a cell. Cell factories determine how the cell appears to the user.
* When the user clicks on, or within, a control it gains focus.
* When the user presses a key, a key event is raised on the control that has focus.
* CSS can be added to a control through the style property. If you need to use the same style for multiple controls it’s faster to save the CSS code to a separate file, and then reference the file using the stylesheets property. Use the @ symbol before specifying the file, since it’s a resource.
* CSS defined in the style property will take precedence over CSS defined in a stylesheet. However, inline/style CSS is not recommended since it makes it difficult to update it later.
* Class selectors allow specific CSS to be used by specific controls, e.g. ‘.button {}’ limits the contained CSS to the button class.
* ID selectors serve the same purpose, but for controls with a specific ID, e.g. ‘#button\_three {}’ limits the contained CSS to a control with the button\_three ID. Use the *id* property to set this ID. The *fx:id* property is used to reference controls from the associated controller class.
* Class selectors are denoted by ‘.’. ID selectors are denoted by ‘#’.
* JavaFX uses the Modina theme by default. It can be switched for other themes, e.g. Caspian. The themes are just different stylesheets. You can change the theme, or set your own, by calling *Application. SetUserAgentStyleSheet()*.
* Nodes can be transformed, e.g. translation, rotation, scaling, etc. You can do this through methods such as *setScaleX()*, *setScaleY()*, etc, or through the fxml file with properties such as *rotate*.
* *FileChooser* and *DirectoryChooser* open a dialog box that allows a user to select files or directories. Call *DC.showDialog()* to select a directory. Call *FC.showOpenDialog(), ‘FC.showOpenMultiple Dialog()’,* or *FC.showSaveDialog()* to open a file, open multiple files, or save a file respectively.
* You can set various aspects of the dialogs including title, supported file extensions, etc.
* The *Hyperlink* control descends from *Label* and can be used create clickable text. Clicking it can direct the user to a webpage, open a dialog box, etc through an event handler.
* A webpage can be opened using either the system’s default browser, or a built-in WebView controller. To do it using the browser call *Desktop.getDesktop().browse()*.
* You can layout JavaFX programs manually using fxml files, or through Scene Builder which accomplishes the same task through a GUI.
* Scene Builder isn’t maintained by Oracle, but the community is keeping it up to date. GluonHQ provides up to date installers: <http://gluonhq.com/products/scene-builder>.
* Link IntelliJ IDEA to Scene Builder through Settings > Languages & Frameworks > JavaFX > Path to SceneBuilder (default: C:\Users\*user*\AppData\Local\SceneBuilder\SceneBuilder.exe). On an fxml tab you can now switch between text mode and Scene Builder mode.
* Scene Builder integration isn’t as good as using the Scene Builder standalone application. It supports automatically reading changes from stylesheets, searching through controls/properties, has fewer bugs, and lists fx:ids in use. It can only be used to build the UI. It’s a recommended workflow to use the standalone Scene Builder to build the UI, and IntelliJ IDEA to work on the code.
* Select multiple controls then use the context menu to conveniently wrap/unwrap them with a layout.

**Section 15 - Basic input and output including java.util**

* Exceptions are used to deal with unexpected situations. Code that may not execute as intended is wrapped in a try block. Potential exceptions are dealt within catch statements.
* An example is dividing by zero which is undefined and will crash the program. The caller of a method may not realise that invalid arguments were provided, but by throwing *ArithmeticException* the caller will realise this mistake.
* Similarly when calling *Scanner.nextInt()*, if the input is not an integer this will crash the program. You can either deal with this by going through each character and confirming it’s an integer and then parsing the input to integer, or you can use a try-catch block to see if *InputMismatchException* is thrown.
* When an exception is caught execution stops immediately in the try block and jumps straight to the catch statement.
* Sometimes checking conditions using an if statement is faster/cleaner, sometimes exception handling is faster/cleaner. It depends on the situation.
* There are multiple types of exceptions which extend from the *Exception* class. All exceptions can be caught by catching exceptions of type *Exception* since exceptions are upcasted if possible, e.g. an *ArithmeticException* can be caught with *catch(Exception e)* since *ArithmeticException* extends *Exception*.
* It's better to be specific with regards to which exception you're trying to handle as it suggests what went wrong with the code.
* When an exception is handled, this leads to a crash in which the stack trace is output to console. The stack trace or call stack lists the chain of methods which were called that lead to the exception. When a new method is called it's pushed/added onto the stack. When the method returns it's popped/removed from the stack.
* When an exception is thrown it propagates up the call stack until a catch statement deals with it. If no suitable catch statement is found and the main method is popped off the stack then the program ultimately crashes. You can’t propagate exceptions out of a static initialization block.
* Exceptions represent recoverable situations, e.g. mismatched input, while errors represent irrecoverable situations, e.g. out of memory. *Exception* and *Error* both extend *Throwable* which represents all potential problems.
* *RuntimeException* extends *Exception*. Subclasses of *RuntimeException* and *Error* represent unchecked exceptions, while other subclasses of *Exception* represent checked exceptions. Checked exceptions must be dealt-with/propagated explicitly in code, while unchecked exceptions can be ignored.
* To propagate an exception use *throw* to create an instance, and *throws* to propagate it out of the method to the caller method.
* It's best to propagate an exception when the problem is better handled by the caller method, or if it doesn't make sense to return data from the current method given that the problem occurred.
* To improve code readability, ensure that code that may throw a *Throwable* is all contained within a single try block. You can then use multiple catch statements one after another to deal with potential problems. The catch statements are compared against the thrown exception in top-bottom order.
* If multiple catch statements contain the same code, then they can be combined into a single catch statement:

|  |  |
| --- | --- |
| try {  ...  } catch(InputMismatchException e) {  …  } catch(ArithmeticException e) {  …  } | try {  ...  } catch(InputMismatchException | ArithmeticException e) {  …  } |

* In addition to *try-catch* blocks, you can also have *try-finally* blocks, and *try-catch-finally* blocks. The *finally* block must be the last block and will run regardless of what happens in the previous *try-catch* blocks leading up to it. This is true even if the previous blocks have a *return* statement. The finally block is ideal for closing resources, logging data, etc.
* You can also have try-with-resources statements. They automatically close a resource whether an exception does or doesn't occur. This means the developer doesn't have to write a separate *finally* block with other *try-catch* blocks in there. Example: '*try(FileWriter locFile = new FileWriter ("locations.txt")) {…}'*. You can list multiple resources separated by ';'.
* To be used in the above statement, classes must implement *AutoCloseable*. Java automatically calls *close()* to close the resource.
* *FileReader* and *FileWriter* are used to read and write from files respectively. *FileReader* can be provided as a source to *Scanner* to read through it.
* *FileReader* can also be provided as a source to *BufferedReader* which is then provided as a source to *Scanner*. The advantage is that the *BufferedReader* loads everything into memory which saves time by preventing unnecessary seeks on the hard drive. *BufferedWriter* provides a similar benefit.
* You can go through the data line-by-line by initially setting the delimiter via *Scanner.useDelimiter()*, and then alternating between *Scanner.next()* and *Scanner.skip(Scanner.delimiter())* to read different sections from each line. Better yet use ‘*String line = scanner.nextLine(); String[] data = line.split(“,”)’* to read the entire line into an array. This way you avoid having to manually deal with the delimiter.
* You can also just use *BufferedReader* directly instead of using it through *Scanner*.
* These classes make use of character streams that input and output data in terms of characters. A **byte stream** does input and output in terms of bytes. As such as byte stream is a more general concept, while a **character stream** is a derived concept. A **stream** just accesses a data source sequentially.
* Usually primitive data types are written to files. However, whole objects can also be stored this way in a process called **serialization**. To do this a class must be serializable, which can be done by implementing the *Serializable* interface.
* It is recommended to define a private field called *serialVersionUID* for this purpose. It’s not required, but different compilers calculate this using different methods, so for consistency you should define it yourself. It represents the version number for the class.
* The runtime uses the *serialVersionUID* field to determine if the saved data is compatible with the executable. If they don’t match then an *InvalidClassException* is thrown.
* Ideally for a class to be serializable, it’s fields are also serializable. This makes the process easier.
* *ObjectInputStream* and *ObjectOutputStream* are used to read and write the objects.
* When serializing to one file, a unique copy of each instance is stored regardless of how many references point to the instance. However, if the references are serialized to different files then copies will be created.
* When dealing with large files that could take up a lot of memory, it may be more efficient to make use of **random access**. So far the techniques covered use sequential access, i.e. you start from byte 0 and then read to the last byte without skipping any bytes in between. Random access allows you to read just the bytes you’re interested in.
* The **file pointer** determines which byte to start reading or writing from. It uses a zero-based index. It increases by the number of bytes read/written. Call *RandomAccessFile.seek()* to set the file pointer.
* **Offset** refers to the position of some data relative to byte 0.
* **Java** **NIO** is an improvement to Java IO and performs operations in a non-blocking manner and fixes some problems developers could run into with the old system.
* NIO isn’t always better than IO, in some cases it’s slower and requires more lines of code.
* NIO classes either deal with the file system, or deal with reading and writing data.
* IO works with byte/character streams in which one byte or character is read at a time and can be buffered. NIO works with blocks of data using **channels, buffers, and selectors**.
* A channel is a data source, e.g. file, socket, etc. To use a class as a channel it must implement *Channel*.
* A buffer is a container for the block of data to be read or written. Buffers can only hold one type of data.
* Selectors allow single threads to manage the I/O for multiple channels.
* IO requires separate classes for reading and writing data. NIO can do this with one class, except for *FileChannel*. The operations are always buffered so you don’t need a separate class for that either.
* The *Path* class in NIO replaces the *File* class from IO. Use *Path* with NIO whenever possible.
* Remember to *flip()* buffers when switching from reading to writing.
* Pipes are used to transfer data from one thread to another. They have two channels: sink and source. Multiple channels can write to the sink channel, multiple threads can read for the source channel.
* The **Path** interface is used to deal with files and folders. Each folder, or directory, is a node in a path. Example: C:\downloads\file.txt contains three nodes: C:\, downloads, and file.txt.
* Each file has a unique path that points to it as you can’t have files with the same name in the same folder.
* The node separating symbol is called a delimiter, and it varies between operating systems. Windows: \, Unix/Mac: /. The \ must be escaped in code \\.
* It’s best not to hard code the separator. Call *File.separator* or *FileSystems.getDefault().getSeparator()* and store the result.
* Absolute path: C:\downloads\file.txt. Relative path: photos\vacation\mountain.jpg.
* Use *Paths.get()* for absolute, and *FileSystem.getDefault().getPath()* for relative.
* It’s best practice to use relative paths as you can’t be sure where the program will be installed.
* It’s best practice to use absolute paths when the user can specify it.
* A ‘.’ node is equivalent to the working directory. A ‘..’ node is equivalent to the parent directory.
* Call *Path.normalize()* to remove redundant nodes in the path.
* Path doesn’t check if a file exists.
* The *File* class in IO was split into multiple classes in NIO. *File* has some disadvantages: 1) Many methods don’t throw exceptions to say why they failed, just a Boolean. 2) Although Java is portable across platforms, *File.rename()* isn’t. 3) No support for symbolic links. 4) Retrieves limited file metadata, and it’s inefficient. 5) Methods can hang if there’s too much data.
* The *Files* class is used to copy, move, delete, etc files and folders. All methods are static.
* Can also used *Files* class to gather various metadata, including operating system specific data.
* *Files.createTempTile()* creates a temporary file in the operating system’s default temporary file location. On Windows this is: ‘C:\Users\*UserName*\AppData\Local\Temp\’.
* The ***Iterator*** interface allows objects to be used in a for-each loop.
* Glob is a simple version of regex. It’s used by Java to pattern match paths, while regex is used to pattern match text. Regex can still be used to pattern match paths.
* Recommended to use NIO when working with file system. For reading or writing files, it varies.

**Section 16 – Concurrency**

* A **process/application** is a unit of execution that has its own memory space, e.g. when you run a console/JavaFX application. They can’t access another process’ memory.
* A **thread** is a unit of execution within a process. Each process has at least one thread – the main thread, but can create as many as it needs. Some threads are automatically created for memory management and I/O. Developers don’t touch these threads. Our code runs on the main thread or sub-threads we create.
* Threads require less resources than processes since all threads share the same memory space as the process they’re created in. This can cause problems.
* Threads can access their thread stack which is an area of memory only they have access to.
* There are two main reasons for creating threads: 1) To perform a task that will take a long time. Otherwise your main thread will suspend until it’s complete. To a user this looks as if the application is stuck. Example: querying database, parsing large files, downloading files from the Internet, etc. 2) An API might require the developer to provide one.
* **Concurrency** refers to an application working on more than one task at a time. In other words, the application doesn’t have to wait for one task to end before starting the next one.
* The output of multithreaded applications varies as there’s no guarantee to when exactly threads will run since it depends on the JVM and operating system. You can give threads priority in relation to other threads, but there’s still no guarantee.
* To create a thread: 1) subclass *Thread* and override ***run()***. Then create an instance of the subclass and call ***start()****.* The thread will automatically be created and call the *run()* method. If the code only needs to be used in one place then you can always create an anonymous class instead.
* 2) You can implement ***Runnable*** and override *run()*. Then pass the runnable subclass as an argument to *new Thread()*. Or again, you can create an anonymous *Runnable* subclass. Most developers work with *Runnable* since it’s more flexible – convenient due to lambda expressions and it can be provided as an argument to more of the API.
* A thread can only be started once otherwise it will throw an ***IllegalThreadStateException***. To do it multiple times create another instance of the subclass.
* A thread terminates when it returns from *run()*. If you call *run()* instead of *start()*, the *run()* method will run on the original thread instead of a new thread since it’s just a regular method call.
* Call myThread.*setName()* to set a name for the thread, and myThread.*getName()* to retrieve it. You can get a reference to the current thread by calling *Thread.currentThread()*.
* Call *Thread.sleep()* to put the thread to sleep for some milliseconds/nanoseconds. It must be used in a try/catch block since it can throw an ***InterruptedException*** if another thread wakes it up prematurely. The timing is also operating system / JVM dependant so don’t depend on it being exact.
* You might want to interrupt another thread if you know it’s no longer required, e.g. a thread that monitors the state of something. Call *myThread.interrupt()* to do so.
* Use *myThead.join()* to make the current thread sleep until *myThread* returns from *run()*. The current thread might be interrupted so *join()* should be called within a try/catch block. To prevent the current thread from waiting forever and becoming stuck you can provide a timeout argument.
* Each thread has its own thread stack so variables created locally are unique to each thread. Threads share instance variables however, so this may provide unpredictable results if each thread reads and writes to them. This is because instance variables exist in the heap which all threads share. This is known as a **race condition**. It would be fine if the threads only read the data, or if each thread has access to a unique class instance. This isn’t always possible in the real world.
* **Synchronization** allows limiting access to resources so that only one thread can execute code at a time, e.g. if a method is synchronized only one thread can access it, the rest will wait until that thread returns.
* To synchronize a method, use the *synchronized* keyword in its signature. If a class has three synchronized methods, only one will run at a time for one thread. Synchronization prevents race conditions. Constructors can’t be synchronized since only one thread can access it anyway.
* Instead of synchronizing a method it’s more efficient to synchronise the portion of the code that actually has the race condition. You can do this using a synchronization block. It requires a variable to be passed in that all threads have access to so that it can lock it when one thread accesses it, and unlock when the thread exits. Use an instance or class variable for this since all threads will share it. You can also use the *this* reference and use the instance of the class as a lock object.
* Synchronized methods also lock on *this* – instance lock. Static method locks are different to regular method locks. So even if a regular method’s lock is obtained, a static method’s lock is still available, and vice versa.
* **Thread safe** refers to code that won’t result in race conditions.
* **Deadlock** refers to two or more threads that become blocked because of each other and can’t progress. This can occur with synchronized methods, since only one thread can obtain the lock at once. That first thread may wait in the method for the second thread to complete its task, but the second thread cannot do so since its can’t obtain a lock since it was obtained by the first thread.
* To solve this problem, synchronized methods/blocks can make use of *wait()* and *notifyAll()* methods. The calling thread gives up its lock when *wait()* is called and sleeps until the lock is reacquired and then continues from where it called *wait()*. Calling the *notifyAll()* method makes the calling thread give up its lock at the end of the synchronized method/block which wakes up a random thread that was sleeping due to *wait()*.
* This pattern is commonly used with the producer-consumer architecture in which some threads produce a resource and some threads consume the resource. All threads share a common object to which they write/read, or produce/consume.
* More tips to prevent deadlocks include: use the minimum required locks, use them in order, e.g. lock 1, then lock 2, etc. You can also use *ReentrantLock*.
* Some collections are not thread safe, e.g. ArrayList. You can still use them, but you will have to make sure to synchronize access.
* Synchronized methods/blocks have many disadvantages. A thread waiting to obtain a lock can’t do anything else until it gets it. It can’t even be interrupted. When the lock is available a thread is chosen at random, it’s not first come first serve. This can be mitigated by giving the thread a higher priority. A thread might be waiting for a while before it gets a lock which is bad for load balancing. You also can’t specify a timeout for how long a thread should wait.
* **Reentrant locks** allow a thread to obtain a lock it already has and continue execution rather than waiting for the lock to become free.
* The disadvantages of synchronized code can be avoided using a *ReentrantLock* object to monitor threads. Call *lock()* first to obtain a lock, and *unlock()* to release the lock.
* A thread must call *unlock()* as many times as *lock()* is called to release the lock. It’s good practice to place the call to *unlock()* within a finally block to ensure that it always runs.
* To prevent blocking a thread call *tryLock()* to see if the lock is obtainable. If it is, the method will return true and the calling thread will obtain the lock. Otherwise you can run different code until it’s available.
* You can use *ReentrantLock* to check how many threads are waiting for the lock, and to provide locks on a first come first serve basis by setting the fair parameter to true in its constructor.
* The **executive service** manages threads for the developer while allowing them to focus on the code that the threads will run. It does this by creating **thread pools**. To create it use a factory method: *ExecutorService srv = Executors.newFixedThreadPool(3)*. To initiate it, call *execute()* and provide a runnable object.
* It can limit the number of threads running at any given time, e.g. three in the above example.
* You must call *shutdown()* when you’re done with the service otherwise the program will keep running. It will wait for all threads to complete before exiting.
* ***Callable*** is like *Runnable*, except it returns an object. Call *submit()* and provide a *Callable* object and store the result in a ***Future***. You can then call *future.get()* to retrieve the value at some point in the future. The calling thread will block until the result is returned.
* **ArrayBlockingQueue** is a queue based on a fixed length array. It’s FIFO just like a regular queue. It’s thread safe and only allows one thread to make changes at a time. This means that you won’t need to keep synchronizing code when working with it.
* If your code depends on the state of the queue, e.g. *isEmpty()*, then you may still need to use synchronization blocks to group method calls together. Otherwise this could result in a slipped condition.
* **Slipped conditions** arise when thread 1 checks the status of a buffer, and reads data if the status is okay. In between that time however thread 2 might change the status of the buffer, making the data not safe for thread 1 to read thereby resulting in an exception or unexpected output. The issue is that thread 1 acted on obsolete information.
* Marking a *ReentrantLock* as a fair lock will make it provide locks on a first come first serve basis.
* A fair lock has a larger performance penalty than a regular lock so use with caution. The more threads the higher the loss in performance.
* *tryLock()* ignores fair lock flag.
* **Livelock** is similar to deadlock, but instead of threads being blocked waiting for locks, they’re waiting for other threads to finish their task. When using locks that block threads indefinitely, you may run into a deadlock. When using locks that block threads with a timeout, you may run into a livelock.
* **Atomic** operations are those that don’t suspend the thread in the middle of execution. Examples: 1) Reading/writing reference variables, e.g. myO*bj2 = myObj1*. 2) Reading/writing primitive variables except long and double, e.g. *myInt = 10*. 3) Reading/writing all variables declared **volatile**.
* Just because a primitive is marked volatile, doesn’t make it thread safe. Incrementing or decrementing it for example, can still lead to memory consistency errors. To prevent this use *AtomicInteger*. It supports atomic operations for increment, decrement, etc. There are atomic versions of many primitive types.
* When multiple threads read/write to one variable we can get **memory consistency** errors. Some variables are saved to the cache on the processor to speed up calculations. Volatile forces the variables to update in the main memory rather than the cache. This means that threads won’t deal with obsolete data.
* To use concurrency specialised for JavaFX programs import the *javafx.concurrent* package.
* A *Task* is similar to *Thread*, but it can dynamically update controls by binding their items property to the task’s value property: e.g. *ListView.itemsProperty().bind(Task.valueProperty())*. The task must return an *ObservableList*.
* A progress bar’s progress can be binded to a tasks progress, e.g. *ProgressBar.progressProperty() .bind(Task.progressProperty())*.
* A label’s text can be binded to a tasks message, e.g. *Label.textProperty() .bind(Task.messageProperty()).*
* Generally, many properties can be binded to other properties as required. This ensures that all binded properties will be the same even if one is changed.
* Rather than working directly with *Task*, it’s recommended to use a *Service* instead. This allows JavaFX to manage the thread. Create a new class that extends from *Service*, override *createTask()*, and call *Service.start()*.
* To start a service multiple times, check to see if *Service.getState() == Service.State.SUCCEEDED* then call *Service.reset()* before calling *Service.start()*.

**Section 17: Lambda expressions**

* Lambda expressions, or Lambdas, were introduced in Java 8 and simplify working with interfaces that declare one method. They replace anonymous classes in this case.
* Lambdas have three parts: argument list, arrow token, and body. Example: *“() -> {}”*. The parenthesis are optional if there’s exactly one argument. The curly brackets are optional if the body contains one statement.
* Example 1: *‘() -> System.out.println(“Hello World”)’* valid for an interface that has one abstract method that requires no inputs and produces no output.
* Example 2: *‘(String string) -> System.out.println(string)’* valid for interface abstract method that takes one String input and produces no output.
* Example 3: *‘string -> System.out.println(string)’* valid for one string input, and no output. Assumes type of parameter can be inferred.
* Example 4: *‘string -> string.toUpperCase()’* valid for one String input, and String output.
* Example 5: ‘*(string1, string2) -> { System.out.println(string1); System.out.println(string2); return string1 + string2; }’* valid for two String inputs, and String output. Assumes parameters can be inferred.
* A **statement lambda** makes use of {}, while an **expression lambda** doesn’t.
* When the lambda body contains a single statement that evaluates to a value, it is automatically returned.
* An interface is a **functional interface** if it declares only one abstract method.
* The argument list must correspond to the abstract method’s parameter list, and the returned data type must correspond to the abstract method’s return type.
* Special case: The *Comparator* interface contains two abstract methods, and many defined methods. The abstract methods are *compare()*, and *equals().* Since the latter is already defined by *Object*, which every class (including anonymous classes) derives from, *equals()* is already defined. Thus, *Comparator* is still a functional interface.
* Anonymous classes and lambdas can capture local variables directly by using them, but the local variable must be final or effectively final.
* Lambda expressions don’t add anything to Java that couldn’t be done with anonymous classes, but they streamline the process by removing boiler plate code.
* Simple statement lambdas that pass arguments to another method can be simplified to **method references**.
* Example 1 constructor method reference: *‘(args) -> new Class(args)’ = ‘Class::new’*.
* Example 2 static method reference: ‘*(args) -> Class.staticMethod(args)’ = ‘Class::staticMethod’.*
* Example 3 instance method reference of object: *‘(obj, args) -> obj.method(args)’ = ‘Class::method’.*
* Example 4 instance method reference of captured object: ‘*(args) -> obj.method(args)’ = ‘obj::method’.*
* java.util.function contains various interfaces for use with lambdas.
* The *Consumer* interface takes one argument and returns nothing. Hence it models a consumer.
* The *Predicate* interface takes one argument and returns a Boolean. It’s used to return true or false based on a given definition.
* There are multiple versions of these interfaces designed to take primitive arguments, e.g. *IntConsumer, IntPredicate, LongConsumer, LongPredicate,* etc. This is also true with other interfaces in function.
* *Predicate* can be chained using *and()*, *or(),* or *negate()* methods, the resulting *Predicate* can be stored in memory and chained further as required. It does this by returning a new *Predicate* that captures the current and other *Predicate* – in the case of *and()* and *or()*. This is also true with other interfaces in function.
* The *Supplier* interface takes no arguments and returns data. Hence it models a supplier.
* The *Function* interface takes one argument and returns data. It can be seen as a combination of *Consumer* and *Supplier*.
* The *BiFunction* interface takes two arguments and returns data. It’s similar in use to *Function*. The prefix ‘bi’ usually means two of something. There are ‘bi’ versions of other interfaces, e.g. *Consumer*.
* The *UnaryOperator* interface uses the same type for the input and output.
* **Java Streams** (*java.util.stream*) allow chaining multiple method calls in succession on a collection.
* This works because many of the methods return a *Stream* object which allows further chaining – intermediate operations. Terminal operations return a non-*Stream* object which ends the sequence. A *Stream* must end in a terminal operation to run the chain of methods.
* *Stream* methods that have a non-*Stream* return type are called terminal methods as you can’t chain further.
* To access the *Stream* object for a collection, start by calling *theCollection.stream()*.
* The *Stream* source can be a collection, array, IO channel, or another *Stream* from *Stream.of()*.
* *stream()* returns a Stream that contains all the items in the list, in the same order. *parallelStream()* is used for concurrent *Stream*s.
* *map()* returns a Stream that maps the input *Stream* to another type of data. Example: *map(String::toUpperCase)* maps each input *String* to an uppercased version of itself, so “Hey” to “HEY”. It always returns same number of items as the input *Stream* since it’s a one-to-one mapping.
* *filter()* returns a Stream in which items match the *Predicate*'s constraints (returns true). It always returns <= items as the input *Stream*. Example: *filter(String str -> str.startsWith(“JA”))* will return a *Stream* in which each item starts with “JA”.
* *sorted()* returns Stream with natural ordering, but there’s also a *Comparator* overload. It always returns the same number of items, but can change ordering.
* *forEach()* performs an action for each item in the input *Stream*. It’s a terminal operation. *peak()* is the non-terminal version of *forEach()*.
* *collect(Collectors.toList())* is a terminal operation that converts the *Stream* results into a *List*. This method can also collect data into other data structures, e.g. *Map*.
* *flatMap()* concatenates multiple sub-*Stream*s within another *Stream*. Subsequent operations will apply to the resulting larger set of data. It returns the same number of items, or more. It’s called *flatMap()* because flattening is a synonym for combining.
* *reduce()* iterates through the whole *Stream,* compares two items at a time, and returns the item that you choose to return. Example: *‘stream.reduce((Integer a, Integer b) -> a + b)’* sums up all the numbers.
* A verbose lambda expression can be preferred over a concise lambda expression if it makes it more Human readable.

**Section 18: Regular expressions**

* **RegEx**, or regular expressions, are used to pattern match strings.
* The simplest form is a string literal, i.e. “hello”, or “hello World234 =+”, etc.
* *String.replaceAll()* uses a RegEx to replace a matching substring, with another String.
* A **character class/set** represents a set of characters, e.g. a dot (.) represents any character.
* To create your own character set, put characters between square brackets.
* Examples: ‘[a]’ is the same as ‘a’ and matches a, ‘[ac]’ matches a OR c in a string, ‘[qom]’ matches q or o or m, ‘[a-d]’ matches a or b or c or d, ‘[a-zA-Z0-9]’ matches any letter or number, etc.
* ‘[^]’ has a special meaning in which characters are excluded from the character set, e.g. ‘[^0-9]’ will match any string that’s not a number.
* A **group** also represents a set of characters, but each character has an AND relationship instead of OR. Example: ‘(hi)’ matches hi, ‘(hi)+’ matches hihihi,, ‘(hi|he)+’ matches hihihehi etc.
* ‘abc’ can be thought of as a&&b&&c. ‘a[bc]’ as a&&(b||c). ‘a[^bc]’ as a&&!(b||c). ‘(hi|he)’ as hi||he.
* A **boundary matcher** pattern matches at specific points in the string, e.g. a caret (^) represents the beginning of a string, and a dollar ($) represents the end of a string.
* *String.matches()* returns true if the regex matches the string. However, *matches()* checks the whole string, while *replaceAll()* looks for substrings. Their results will be inconsistent.
* A **quantifier** repeats a character.
* Optional quantifier ? matches 0 or 1, e.g. e? matches ‘’ or ‘e’.
* \* matches zero or more, e.g. ‘e\*’ matches ‘’, ‘e’, ‘ee’, ‘eee’, etc.
* + matches one or more, e.g. ‘e+’ matches ‘e’, ‘ee’, ‘eee’, etc.
* {n} specifies a range, e.g. ‘e{3}’ matches ‘eee’, ‘e{2,}’ matches 2 or more e’s, ‘e{2,5}’ matches 2 to 5 e‘s.
* Putting ? after \* or + makes it a lazy quantifier, e.g. for string ‘{hi}{bye}{3}’ ‘\\{.\*\\}’ matches the entire string, but ‘\\{.\*?\\}’ will stop after the first set of parenthesis since that’s the first set to match.
* The ***Pattern*** class compiles a regex for efficiency using *Pattern.compile()*. *Pattern.matcher()* can then be used frequently to match the pattern with a string. The result is returned as a ***Matcher*** object.
* The *Matcher* object can find indices for where the pattern starts and ends being true. Call *Matcher.find()* in a while loop, then in the body call *Matcher.start()* and *Matcher.end()* to obtain indices.
* The *Matcher* object can also use groups denoted by paranthesis to extract specific text from a string, e.g. the text between two HTML tags. Use *Matcher.group()* for this.

**Section 19: Debugging and unit testing**

* A **debugger** allows monitoring the state of the application as it runs to identify errors in the source code.
* The compiled program’s state can be halted to monitor variables and to change them at runtime.
* A **breakpoint** suspends execution when the program reaches the chosen statement, but right before the statement is actually executed. Multiple breakpoints can be selected.
* To set a breakpoint, click in the gutter line that you’re interested in.
* The Frames window displays the stack trace and all threads. To see activity in another thread, select it from the dropdown menu.
* Clicking along the stack trace will show the data in variables in that scope – including private fields.
* The show execution point button switches the focus to where execution has halted from the current file.
* The step over button completes execution for the current statement, then waits before the next statement.
* The step into button goes further into the current statement, e.g. through a loop, or inside a method call, etc. However, it behaves the same as step over for JDK methods and for simple statements.
* The force step into button will step into JDK methods.
* The step out button completes execution for the rest of the method then continues where it left off.
* The drop frame button reverses executed code from the current method and goes back up the call stack. However, it doesn’t work as well when networking/database code is involved and only reverses local variable states.
* The run to cursor button keeps executing code until it reaches the statement that the cursor is on.
* The resume program button continues execution until another breakpoint is reached.
* Run > View Breakpoints lists all of the active breakpoints in the code.
* To track specific variables, RMB > Add to **Watches**. You can remove watches when you’re done.
* Variables are marked blue when they are changed by the last statement.
* If you attempt to set a breakpoint for a field it will actually create a **field watchpoint**. This suspends execution any time the field is modified. RMB > watchpoint, will give more options. You can suspend execution if a certain condition is met, or if the field is accessed, etc.
* If multiple methods are executed in one statement, Run > Smart Step Into, allows you to pick which method to step into.
* You can directly set a value for a variable in the variables window – RMB > Set Value.

**Section 20: Databases**

* **Database** refers to a container which contains all the data you store, how its structured, the queries and views.
* SQLite stores databases on a single file. This isn’t done by most database systems.
* A **database dictionary** provides a comprehensive list of the structure and types of data in the database.
* A **table** is a collection of related data held in a database. Example 1: A contacts table can contain name, address and phone number fields. Example 2: An invoice can contain ID, date, item, and cost of item sold fields. These are examples of structured data. Big Data / Hadoop uses NoSQL which uses unstructured data.
* **Fields/columns** store the actual data. They have a type and name. They can store numbers, text, audio, images, dates and times, etc. They are analogous to arrays in Java.
* **Record/row** a collection of all columns for one entry.
* **Flat file database** stores all data in a single table. This results in a lot of duplication. Not used much these days since to update data in one place you have to update it in the duplicated places too. In a relational database, tables can be related to other tables to prevent this.
* Tables are related to other tables on a single column, this is called a **join**. There are different types of joins. One to many is when one row corresponds to many rows – e.g. customer data and orders data.
* **View** is a selection of rows and columns, possibly from more than one table.
* Install SQLite from the official website. Copy the files to a permanent location on your storage device. Add its path to Window’s path environment variable. Open command prompt and type *sqlite3* to confirm that it works. Type *‘.quit’* to exit.
* The notes will focus on the SQL language with command prompt, then get into the Java aspect afterwards.
* *‘sqlite3 test.db’* creates a new database called test if it doesn’t exist, otherwise it opens it. It will be created/opened from the current active directory. Use ‘*cd’* in command prompt to change it. The extension isn’t important, but ‘db’ is typical, avoid ‘sql’ as that’s used for SQL scripts.
* *‘.help’* lists all the possible commands you can input.
* *‘.headers on’* displays the names of columns.
* *‘create table contacts (name text, phone integer, email text);’* creates a table called contacts with the given columns. If SQLite stays quite when you enter a command it’s because there were no errors.
* *‘insert into contacts(name, phone, email) values('Tim', 6545678, 'tim@email.com');’* inserts a row of data into the table. You can use single quotes or double quotes. When working with Java it’s consistent to use single quotes for SQL statements and double quotes for strings.
* Although not required, it’s common to type SQL keywords in capital to help differentiate commands from data. SQL itself is case insensitive however.
* *‘SELECT \* FROM contacts;’* will display (select) all (\*) columns from contacts table.
* *‘SELECT name, phone, email FROM contacts’* will display the selected columns from the table.
* If you omit the semicolon by mistake SQL will assume that you want to enter more clauses/statements for the query. This is often done to make commands more readable.
* SQL commands require the semicolon, SQLite commands (beginning with dot) do not.
* When inserting, you can omit ‘contacts (name, phone, email)’ if you specify data for each column, e.g. ‘INSERT INTO contacts VALUES(“Brian”, 1234, “Brian@email.com”);’.
* Although you specify the type of each column when creating the table, SQLite doesn’t enforce the type of data you insert. A string can easily be inserted into an integer column. Other database systems don’t do this.
* SQLite doesn’t support altering tables to change name/type like other database systems.
* *‘.backup testbackup’* creates a backup of the current database and saves it to the specified file. You can provide an argument for a specific database.
* *‘UPDATE contacts SET email="steve@email.com";’* will set each email row to steve@email.com. *‘.restore testbackup’* will restore the specified backup.
* *‘UPDATE contacts SET email="steve@email.com" WHERE name="Steve";’* sets the email only when the name is Steve. You can use > for more than, < for less than, <> for not equals.
* *‘SELECT name, email FROM contacts WHERE phone > 1’* applies a constraint so that only rows with a phone number > 1 will be shown.
* *‘DELETE FROM contacts’* will delete every row.
* *‘DELETE FROM contacts WHERE phone=1234’* deletes a row when the phone number is 1234.
* *CREATE, INSERT, SELECT, UPDATE,* and *DELETE* are some of the most common commands.
* *‘.tables’* lists all the tables in the current database.
* *‘.schema’* prints out the structure of the tables. You can specify a specific table to check.
* *‘.dump’* gives you all the commands to create the table and to insert the data. This can be directly copy and pasted into code.
* *‘CREATE TABLE songs(\_id INTEGER PRIMARY KEY, track INTEGER, title TEXT NOT NULL, album INTEGER);’* -> The ID column is called **\_id** as some Java classes require it. **Primary key** means that the column must contain unique data on each row. This speeds up searching and joining since the rows are sorted by it. By default the order is undefined similar to Sets and Maps in Java. Not null means that text must be entered.
* Inserting into this table without specifying the ID will automatically pick the next unique ID. Not all databases do this.
* *‘SELECT \* FROM artists ORDER BY name;’* for ascending order by the specified column.
* ‘SELECT \* FROM albums ORDER BY name COLLATE NOCASE;’ will ignore case when ordering.
* ‘SELECT \* FROM albums ORDER BY name DESC;’ or ‘‘SELECT \* FROM albums ORDER BY name COLLATE NOCASE DESC;’ for descending order.
* ‘SELECT \* FROM albums ORDER BY artist, name COLLATE NOCASE’ sorts by artist, then by name.
* *‘SELECT \* FROM songs JOIN albums ON songs.album = albums.\_id;’* will combine the two tables into one to make it easier to see which albums the songs belong to. The tables are combined wherever *songs.album* matches *albums.\_id*. The columns from album are appended to the columns from songs so two columns will have the exact same data.
* ‘SELECT songs.track, songs.title, albums.name FROM songs JOIN albums ON songs.album = albums.\_id;’ specifies which columns you want from the joined tables. You can remove songs and albums and write ‘SELECT track, title, name FROM songs JOIN albums ON songs.album = albums.\_id;’ if the column names are unique, but it’s good practice to fully qualify the name.
* ‘SELECT albums.name, songs.track, songs.title FROM songs INNER JOIN albums ON songs.album = albums.\_id ORDER BY albums.name, songs.track;’ will order the joined table.
* There are different types of joins. JOIN is the same as INNER JOIN.
* ‘SELECT artists.name, albums.name, songs.track, songs.title FROM songs

INNER JOIN albums ON songs.album = albums.\_id

INNER JOIN artists ON albums.artist = artists.\_id

*ORDER BY artists.name, albums.name, songs.track;’* joins multiple tables. It makes it easier to follow if you imagine songs and albums being joined first, then songs-albums being joined with artists to form a songs-albums-artists table.

* ‘SELECT artists.name, albums.name, songs.track, songs.title FROM songs

INNER JOIN albums ON songs.album = albums.\_id

INNER JOIN artists ON albums.artist = artists.\_id

WHERE artists.name=”Pixies”

*ORDER BY artists.name, albums.name, songs.track;’* The order in which the clauses appear is important, e.g. to apply a WHERE constraint it must go after joins, but before ordering.

* *‘WHERE songs.title LIKE “%doctor%”’* uses a wildcard to match strings that contain doctor. LIKE is not case sensitive. If you remove the first wildcard (%) then the string must start with doctor, etc.
* **Stored procedures** are like methods in that they save queries and take parameters to put into the saved queries. It minimises code duplication and maximises code re-use. SQLite doesn’t support them however.
* **Views** save the results of a query. They can’t be modified once created, but further queries can be performed on them. A view is created by prepending *‘CREATE VIEW view\_name AS’* to your query. Once created you can see it using the *.table* and *.schema* commands. To work with a view, just treat it like any other table. Example:

‘CREATE VIEW artist\_list AS

SELECT artists.name, albums.name, songs.track, songs.title FROM songs

INNER JOIN albums ON songs.album = albums.\_id

INNER JOIN artists ON albums.artist = artists.\_id

ORDER BY artists.name, albums.name, songs.track;’

* ‘DROP VIEW album\_list;’ deletes the view. ‘DROP TABLE album’ deletes the table.
* ‘SELECT artists.name AS artist, albums.name AS album, songs.track, songs.title FROM…’ uses an alias so that the first two columns appear as artist and album.
* ‘SELECT ar.name AS Artist, al.name AS Album, s.title AS song

FROM songs AS s

INNER JOIN albums AS al ON s.album = al.\_id

INNER JOIN artists AS ar ON al.artist = ar.\_id

*ORDER BY ar.name, al.name, s.track;*’ You can also replace the name of a table using an alias to make it quicker to type.

* ‘SELECT ar.name Artist, al.name Album, s.title song FROM songs s

INNER JOIN albums al ON s.album = al.\_id

INNER JOIN artists ar ON al.artist = ar.\_id

*ORDER BY ar.name, al.name, s.track;’*. Actually typing *AS* is optional, SQL assumes that the next string is an alias anyway.

* Challenge 1: ‘SELECT s.title FROM songs s INNER JOIN albums a ON s.album = a.\_id WHERE a.name = "Forbidden”;’.
* Challenge 2: ‘SELECT s.track, s.title FROM songs s INNER JOIN albums a ON s.album = a.\_id WHERE a.name = "Forbidden" ORDER BY s.track;’.
* Challenge 3: ‘SELECT s.title FROM songs s INNER JOIN albums al ON s.album = al.\_id INNER JOIN artists ar ON al.artist = ar.\_id WHERE ar.name = "Deep Purple";’.
* Challenge 4: ‘UPDATE artists SET name="One Kitten" WHERE name="Mehitabel";’.
* Challenge 5: ‘SELECT count(\*) FROM artists WHERE name = "Mehitable";’ (output 0).

‘SELECT count(\*) FROM artists WHERE name = "One Kitten";’ (output 1).

* Challenge 6: ‘SELECT s.title FROM songs s INNER JOIN albums al ON s.album = al.\_id INNER JOIN artists ar ON al.artist = ar.\_id WHERE ar.name = "Aerosmith" ORDER BY s.title;’.
* Challenge 7: ‘SELECT count(s.title) FROM songs s INNER JOIN albums al ON s.album = al.\_id INNER JOIN artists ar ON al.artist = ar.\_id WHERE ar.name = "Aerosmith" ORDER BY s.title;’ (output 151).
* Challenge 8: ‘SELECT DISTINCT s.title FROM songs s INNER JOIN albums al ON s.album = al.\_id INNER JOIN artists ar ON al.artist = ar.\_id WHERE ar.name = "Aerosmith" ORDER BY s.title;’.
* Challenge 9: ‘SELECT count(DISTINCT s.title) FROM songs s INNER JOIN albums al ON s.album = al.\_id INNER JOIN artists ar ON al.artist = ar.\_id WHERE ar.name = "Aerosmith" ORDER BY s.title;’.
* Challenge 10: ‘SELECT count(DISTINCT ar.name), count(DISTINCT al.name) FROM songs s INNER JOIN albums al ON s.album = al.\_id INNER JOIN artists ar ON al.artist = ar.\_id WHERE ar.name = "Aerosmith" ORDER BY s.title;’.
* The **JDBC API** or Java Database Connectivity allows Java to work with databases, spreadsheets and flat files. Specifically, we need the SQLite JDBC driver to work with SQLite. Since all data sources implement the same interfaces from JDBC it’s trivial to switch between them. So to switch between SQLite and MySQL mainly requires switching the driver rather than changing the code.
* Download the JDBC driver from SQLite. Download DB Browser for SQLite work with databases via a UI rather than console.
* Opening the database in DB Browser might make it unavailable for the Java application. Close the database if you have this issue.
* When using SQLite, connecting to a database that doesn’t exist will automatically create it: *‘Connection conn = DriverManager.getConnection("jdbc:sqlite:data\\testjava.db");’.* Conn now refers to the database ‘testjava.db’.
* Use statements to execute SQL commands: ‘Statement statement = conn.createStatement(); statement.execute("CREATE TABLE IF NOT EXISTS contacts(name TEXT, phone INTEGER, email TEXT)");’. A semicolon was not needed at the end since SQL expects execute to contain a full statement. Creating a table that already exists results in an error.
* When you’re done, make sure to call *close()* on any statements first, then the database. It’s good practice to close resources as soon as your done with them.
* JDBC connection automatically commits database changes. This can be disabled by called *setAutoCommit(false)*. You must then call *commit()* to manually commit changes before closing the connection.
* If you use *execute()* to perform a query the results can be obtained by calling *getResultSet()*. It’s a buffer which contains all the date. You can use *ResultSet.next()* to see if more data exists, then use *ResultSet.getString(), ResultSet.getInt(), etc* to cast it. Calling *next()* moves the cursor to the next row in the query results. Call *ResultSet.close()* to free resources.
* Statement.*executeQuery()* is equivalent to calling *execute()* and then calling *getResultSet()*. Calling *ResultSet.getString(), etc* with an actual column name is not recommended since it’s slow compared to using an index, it’s open to SQL injection attacks, and it’s susceptible to column name changes.
* Use try-with-resources to avoid having to write messy finally clauses everywhere. A resource class implements *AutoCloseable*.
* You can run SQL statements within DB Browser. This is recommended before typing an advanced statement into your code to see if the results are as expected.
* Call ResultSet.getMetaData() to find out information about table columns, e.g. number of columns, column names, column SQL types, table name, if it’s nullable, etc.
* SQL injection attacks are possible when blindly accepting input from a user and appending it to a SQL string to be executed.
* *PreparedStatement*protects against this since you don’t just keep appending to a string. They allow use of placeholders which makes them easier to work with. They build the query only once (precompiled / stored procedure) which makes them faster than regular statements – only when the query is used more than once otherwise it’s slower.
* Placeholders are represented by a ‘?’. Call *setString(), setInt(), etc* to specify a placeholder by index, and its value. Placeholders are treated as literals which prevents SQL injection. Placeholders can’t be provided for tables or columns.
* A *PreparedStatement* should only be instantiated once for efficiency.
* PreparedStatement.close() also closes any associated ResultSet.
* Creating a *PreparedStatement* and passing *Statement.RETURN\_GENERATED\_KEYS* will automatically return any generated keys, e.g. ID. Call *PreparedStatement.getGeneratedKeys()* to retrieve them after executing.
* Call *PreparedStatement.executeUpdate()* to return how many rows were affected by SQL code.
* **Transactions** allow running multiple SQL statements as a single unit. This is to prevent compromising a databases’ integrity. Either all of the statements will complete, or none of them will.
* Example: When transferring money from one bank account to another you first withdraw it from an account, then deposit it to another account. If the second step fails then the money is missing. Transactions make sure that both steps will complete.
* Database transactions must be **ACID** compliant:
* **Atomicity**: Either all of the statements complete, or none of them do.
* **Consistency**: The database must be in a valid state before and after a transaction.
* **Isolation**: Until a transaction completes, other connections won’t see the commits.
* **Durability**: Changes committed by a transaction are permanent. If the application or database goes down then the changes made will still be there when the database or application comes back up.
* To execute transaction commands: 1) turn off auto-commit *Connection.setAutoCommit(false)*. 2) Perform SQL operations that form the transaction. 3) If there are no errors call *Connection.commit()* to commit the changes, otherwise call *Connection.rollback()* to undo changes. 4) Turn auto-commit back on.
* You can use Connection.setSavePoint() to rollback to a specific point.

**Section 21: Networking**

* >

**Section 22: Java 9 module system**

* >

**Section 23: Migrating Java projects to Java 9**

* >

**Section 24: Bonus material**

* >