Homework 10

Q1)(E)

The compiles but throws an exception at runtime. `sb` variable is initialized with an empty `StringBuilder` object. Then the program arguments are iterated. For every given argument, for loop tries to insert the `String` element being iterated to the `sb` `StringBuilder` at the index of character `'c'` in `sb`. But since the the `sb` `StringBuilder` is empty, `sb.indexof("c")` return `-1`, and hence code throws `StringIndexOutOfBoundsException` at runtime.

Q2)(C, E)

Addition operator has higher precedence over assignment operator, so the option A is wrong. For option B, addition operator is listed last, but it has lower precedence over the previous three operators, so it is wrong too. Option D is wrong because the subtraction operator has lower precedence over the multiplication operator. Option C and option E has the correct order of operators according to increasing or the same level of precedence.

Q3)(C, B, F)

Option A is a getter method but it has a parameter in its method signature, so it is not a valid JavaBean. Option D should should return a boolean, looking at the name, at it is prefixed with 'is', so this option is not a valid bean too. Lastly Option E is wrong because there is no such naming convention. C, B, F are valid beans.

Q4)(A, E)

Array has `length` variable instead of a `size()` method. `size()` method is used for ArrayLists. Upon the correction of this one line, code would compile and run without issue. The nested for loop iterates correctly, and without trying to access out of bounds of the crossword array. It only iterates through the half of the every array in the second dimension though. Because the second dimension has `10` arrays in it with every array having length `20`, we are iterating every `10` array but stopping after the ninth element which corresponds to half of the whole elements. And unassigned half has their values set to default value `0`.

Q5)(B, D)

If a file system resource becomes temporarily unavailable, a checked exception must be used. Error is not a subclass of the class `Exception`, but `Throwable` class. If a user enters invalid input, i think an unchecked exception must be used, because we can consider it a programming error. Although it is possible, it is very unlikely that we would've want to catch it.

Q6)(D, A, C)

`import jungle.tree.*` statement imports all of the class inside tree package, allows us to access them without specifying the full package name. This is also true for the import statement **`import`** `savana.*`. **`import`** `forest.Bird` imports a class, `Bird` class, so again we don't need to use its full package name. `java.lang.Object` package does not need to be imported, since its inherently imported, we can use `Object` class without a full package name. `savana.sand.Wave` class cannot be accessed as asked in the question since the import statement only import classes in the `savana` package. Classes option E and F, should be used with full package name too since only Bird class imported from the forest package. Key point here is that we should not overlook the fact that by using '*', we are not recursively importing all packages.

Q7)(C)

Two of the given four variables represent immutable objects. `Strings` and `LocalDateTime` object are immutable objects, while `Arraylists` and `StringBuilder` classes represent mutable objects.

Q8)(C)

The output of the given code is `"wing"`. Given code iteratively removes characters from the `StringBuilder` unless the length of the `Character` sequence is less than `5`. First iteration leaves the object with the value `"s growing"`, since the length is greater than `5`, iteration runs again, leaving the value with `"wing"`. **`while`** condition check returns **`false`** since the length is now less than `5`. And the last value of the variable which is `"wing"` is printed after the **`do-while`** loop.

Q9)(D)

All of the created `String` objects are new objects, meaning that the all three variables points to different object. **`new`** keyword always instantiates a new object in the heap. Only the `ceiling` string is interned, but the next two variable are instantiated and initialized with **`new`** keyword. The `==` operator checks if the references point to the same object, so since none of the three point to the same object as other, there is no any possibility that any equality check on these three references returns **`true`**. equals method returns **`true`** with every combination of these three variables, one of them as caller other the argument. Because the `equals` method is implemented to check the elements in the `String` rather than comparing references.

Q10)(C)

The code prints true three times. Since `String` objects are mutable, all the methods which return a `String` after an operation which seems to mutate the string, actually returns a new `String` object. Because again, `Strings` are immutable. Last three statements prints **`true`**. Since they are equality checks on the contents of the strings rather than the references, and the `ignoreCase` is used where the contents are different.

Q11)(A, B, C)

Lines 15 and 17 can be removed since we are not referencing to them. Lines 15, 16, 21 describes the outer loop with its label. We can remove those lines safely. Because the operation is inside the inner do-while, and the condition of the outer loop does not affect the output since when the inner loop condition returns **`false`**, outer loop condition would also, always return **`false`**. So it doesn't really affect the outcome, but it would if the condition would be checking against some value bigger than `5`.

Option D describes removing the inner do-while, which would cause the output to change since the condition in the **`while`** statement is different. There would be 4 `'x'` character added and printed instead of `5`. Option E and F is certainly wrong since they suggest a do-while without a while.

Q12)(B, C)

Option B does not compile since since we are assigning a **`double`** literal to a **`long`** type reference, java cannot convert **`double`** to **`long`** since it would not be promotion. But note that a **`long`** value would be promoted to **`double`** value. Option C does not compile too because a floating point number is given with a `'L'` postfix which defines a **`long`**. That is clearly a syntax error. Option D and E does compile, both `'l'` and the uppercase `'L'` are practically the same. Option A compiles, because an **`int`** can be inherently promoted to **`long`**. Option F compiles too because underscores are fine to use between numbers for readability of big numbers.

Q13)(A)

Since `time.getHour()` returns `1`, **`while`** loop is never branched, hence no line is printed. But keep in mind that the `time.plusHours()` is dangerous here since it returns a reference to a copy, not a reference to the time object itself, since the `LocalTime` class is immutable. If the while condition was to evaluate **`true`**, we would have an infinite loop.

Q14)(D)

The code compiles but throws a `NullPointerException` at runtime. Since `game` variable is not initialized, it gets the value **`null`** which is the default value for objects. But hidden with the first thrown exception there is another issue with the code. A reference type variable of one-dimensional Object array is assigned with `game` variable, which is a two-dimensional array itself. Which means that the every element we obtain with `obj` reference (with something like `obj[3]`) is an one dimensional **int** array (`int[]`). A character is assigned to where should be an object of one-dimensional array. This is a problem which would cause an `ArrayStoreException`.

Q15)(C, E)

Option B, D and F does not even compile because no type argument given on the left hand side. Option A gives a warning because of the usage of a raw type when instantiating the ArrayList. Compiler gives warnings because a raw type prevents compiler from doing some type safety checks. It is strongly discouraged to use raw types.

Option C and E does compile without warning because they properly utilize generics. For the option C (`List<String> c = new ArrayList<>();`), it is not mandatory to give a type argument on the right side, because it is inferrible by the compiler from the argument given to the reference type on the left side.

Q16)(B, D)

After the assignments, `shoe1` and `shoe3` references both points to `"flip flop"` and `shoe2` reference points to `"croc"`. `"sandal"` is not referenced by any variable so it is eligible for garbage collection just before the `main()` method ends. After it ends, all off them are eligible anyway. Option D is correct because being eligible to be garbage collected does not mean it is guaranteed that the garbage collector will run.

Q17)(C)

One line of the application does not compile. ”`f.getFish();`” statement does not compile because even though the `Clownfish` class’ `getFish()` method does not declare an exception `Fish` class’ `getFish()` method does and the type of the reference variable `'f'` is `Fish` so we need to declare or handle it in the `main()` method.

Q18)(A)

Only one line printed. static `print()` method expects a `List` and a `Predicate` as parameter. In `print()` method, every item in the list is iterated and is given as argument to Predicate’s `test()` method. If the `test()` returns **true** number is printed.
In the `main()` method, we see that a lambda expression “`e -> e < 0`” is given to the `print()` method. This expression tests the parameter `e` if it is less then `0` or not. The given `list` variable contains three items, one of which is ‘`-5`’. This item makes the `test()` method returns **true**, and hence it is printed.

Q19)(F)

None of the given options are mandatory with a `try` statement. `finalize()` is a `Object` method which is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. `catch` and `finally` blocks are optional as long as one of them is present. `throws` keyword is used to declare exceptions in method signatures.

Q20)(A)

The code outputs `5`. The outer while loop is branched because variable `result` is bigger than `7`. Inner do-while loop decrements the `result` variable until it's equal to `5` and then hands over the control because the contiditon does not meet anymore. The very next statement after the inner do-while loop breaks the loop labeled "loop" which is the outer loop itself. Using a label here is unnecessary actually. After that, we see a print statement which print the value of `result` variable which is `5`.

Q21)(C)

The result of the given application after compiling and executing is "`1 2`". The constructor of the Alligator class increments the class variable, which by the way is initialized with default value `0`, every time when an instance of the class is created. The `teeth` in the `main()` method resolves to the class variable `teeth`. With the first invoke of the `snap()` method, `teeth` is already incremented by `1` so `1` is passed to the `snap()` method. But the `snap()` method's parameter is also named `"teeth"` so it shadows the class variable `teeth` which resides in the outer scope. Referring `teeth` inside `snap()` method does not refers to the class variable but the local variable `teeth`, hence the class variable is not affected by the decrementation. So, the first invoke of the `snap()` method prints 1, and by the second invoke of the `snap()` method, for the reasons explained above, `2` is printed.

Q22)(A)

`concat()` method concatenates the specified string to the end of this string. Key point here is to know that the string objects are immutable. `witch.concat(tail)` does not mutate the object which is referred by the `witch` variable but returns a new one, concatenated with the string value represented by the `tail` variable. Newly created, concatenated `String` is not assigned to `witch` String type reference variable, hence the `witch`, still refers to the same object as before. So, `"b"` is printed.

Q23)(A, C, F)

In java 7 and earlier versions, interfaces could only contain `public abstract` methods, which must be implemented by classes that implements the interface. From java 8, apart from public abstract method, we can also have `public static` and `public default` methods. An interface method can not be `final` since it would have prevented the method to be overridden. **`protected`** keyword can not be applied to interface methods to because all interface methods are inherently **`public`** and **`protected`** methods are intended for sharing implementation with subclasses but interfaces have no implementation at all. Additionally, while being not true for Java 8, with Java 9 an onward, we can also have **`private`** interface methods both as static and non-static methods.

Q24)(D)

The code given in the question does not compile because, in while loop condition does not return a boolean value due to the use of assignment operator. We should use equality operator instead, or some expression which returns a boolean value. Having equality operator (`==`) instead of assignment operator, while would be branched since the tie variable is **`null`** and inside while block it would be assigned to string value `"shoelace"`,hence the code would output `"shoelace"`.

Q25)(B, F)

The option B is true since every class in the `java.lang` package is auto imported. Option C is wrong, `java.util` classes must be imported explicitly. Option D is wrong, java does not require every file to declare package statement, without a package statement it would be in the default package, option A is wrong because of the same reason . Option E is not true, there is no requirement such as to declare at least one import statement. Option F is true, every class is either direct or indirect descendant of the `Object` class.

Q26)(D)

The code does not compile because of line `m1`. Tree class overloads the inherited `grow()` methods, but does not override it. It must override it since there is an ambiguity about which default method would be invoked if it was not overridden. If the `Tree` class would overridden the `grow()` method, then there would be no ambiguity about which implementation is going to be called, since the `Tree` class `grow()` will be called on an instance of `Tree` class whether the reference type is `Living` or Plant, since the `grow()` method is polymorphic.

Q27)(D)

The println statement does not compile. The first three option complains about variable declarations not being able to compile, because of the variable names. This is not true, all of the variable names (`name`, _`number` and `profitt$$$`) have valid syntax. Dollar sign and underscores are allowed to be used in identifiers. The problem with the code is, the `profit$$$` variable not being initialized. `profit$$$` is a local variable and local variables don't get any default values like class or member variables. For this reason `println` statement does not compile.

Q28)(A)

Post-decrement (`x--`) operator decreases the value by one and returns the original value while pre-increment (++x) operator increases the value of `x` by one and returns the new value.

Q29)(B, E, C)

Three methods contain compiler errors. public `Trouble()` does not compile because there is no default `Big()` constructor is defined, so we need to explicitly invoke another constructor. public `Trouble(long deep)` does not compile because the assumed signature Trouble(String, long) is undefined in Trouble class. Last but not least, public `Trouble( int deep )` does not compile because there are two constructor calls, but a constructor call must be a first statement in a constructor.

Q30)(E, F)

Because `main` method is static, we can't refer to instance variables. So, all the A, B, C and D options are wrong. Option E and F is the correct answers. Option E declares two variables but initializes only one which is the `max` variable, `min` variable gets the default value which is `1` for integers. Option F too, declares two variables with one statement but it initializes both variables, but since `min` is assigned to `0`, both option E and option F are practically the same.

Q31)(B, E)

Only one of the right-hand sides of the ternary expression will be evaluated, making the option A wrong. Option B is true since there could only be one default statement in a **`switch`** statement which does makes sense. Option C is wrong since only one else statement if allowed to execute if all other conditions returns false. Option D is wrong because one of them (||) is called short-circuit operator because second operand will not be evaluated if the outcome is guaranteed with the evaluation of the first operand. Option E is true since the logical complement operator (`!`) is undefined for numerical types.

Q32)(C)

The code outputs `"ed"`. StringBuilder `sb` is appended with `"red"` then the first character is deleted leaving the `StringBuilder` with `"ed"`. `sb.delete(1, 1)` does not delete any character since the start index and the end index are the same. So, the `println` statement print `"ed"`.

Q33)(A, D)

Dollar sign and underscore are permitted to use anywhere in identifiers. Although the first option seems absurd, it is valid. All the other special characters used in method identifiers (**%**, \, **#**, **-**) are invalid to use. We should note that even if numbers are permitted to use in identifiers, it can not be used as the first character of an identifier.

Q34)(B, C)

Option B is true and it is describing polymorphism. Option C is true too since inheriting commonly used attributes and methods is one of the main benefits of inheritance. Option D is wrong because every class, even if it does not extend any other class, it inherently extends the object class. Option E is wrong because one of the advantages of inheritance is that it improves code reuse. Lastly, if we are talking about class data rather than instance data, static methods is the way to access them.

Q35)(E)

Statement **I** is true since java does use dot (**.**) to separate packages. Statement **II** is true, with Java 8 and onward java supports functional programming with lambdas. **III** is true, Java is strongly object oriented, even the entry point which is the `main` method has to be in a class for some reason. Since java is strongly object oriented, and one of the main benefits of the object-oriented programming is polymorphism.

Q36)(C)

The output of the following application is `"3 2"`. listing is initialized with an array literal with first dimension's length being `3` since there is `3` array inside the array, and second dimension's length being `2` since all of the arrays inside the array has the `2` elements. Hence the `listing.length` is `3`, and `listing[0].length` is `2`.

Q37)(A, B, E)

Switch statement allows **`byte`**, **`short`**, **`char`** and **`int`** primitive types. Other than these primitive types it also allows enumerated types and the `String` class. Few special classes which wrap primitive types such as `Character`, `Byte`, `Short` and `Integer` are allowed too.

Q38)(D)

The code does not compile because of line **//k2**. The arrow which separates the body of the lambda ant the parameters is typed wrong. It should be **->** instead of **=>**. Having the error corrected, the code would print true since the given lambda checks if the parameter is between the range of values `5` and -`5`, if it is not, it return true, meaning we should correctly aim.

Q39)(B, C, E)

There are three types of comments in Kava.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

The single line comment is used to comment only one line. The multi line comment is used to comment multiple lines of code. The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool. Option C is a valid single line comment, Option B is documentation comment, asterisks after the second one is ignored. Option E does count as a documentation comment too.

Q40)(A, F)

Static imports allows to use **`static`** methods or variables of a class elsewhere in the application without referring its full path with package name. Option A and B correctly imports **`static`** members of the `Grass` class. While A imports the members one by one, option F imports all of the **`static`** members by using asterisk. Option C and E are both have invalid syntaxes since **`static`** keyword is in wrong order. Option B is wrong because, this type of import is used to import classes. `pacakage_name.*` imports all classes in `package_name` and we can't use **`static`** when importing classes. Option D looks like its trying to import members from a class without the **`static`** keyword.

Q41)(D)

Arrays.asList() Returns a fixed-size list backed by the specified array. Note that the returned `List` implementation is not the same everyday `ArrayList` we use. SOme of the methods inside the List interface are optional methods. Returned implementation does not support every `List` method one of which is `remove()`. That throws `UnsupportedOperationException` at runtime.

Q42)(A, D)

Underscores are allowed in identifiers but they are only allowed in numeric values if they are place between numbers. Option A and D is a correct example while other options depicts wrong examples according to the aforementioned rule.

Q43)(B)

Result of the given code when executed is `[0, 01, 1, 10]`. The program arguments `arg` is assigned to an array literal `{ "0", "1", "01", "10" }`, sorted alphabetically and then printed.

Q44)(D)

The blanks must be filled with **`public`** and **`static`**. Static members belong to classes in which they are declared. Hence they do not need instances to be accessed. While **`public`** access modifier makes variables accessible anywhere in the application. Making a member **`public`** and **`static`**, kind of depicts global variables in other languages.

Q45)(A)

Given code prints only one line, the second blank `println` statement is a trick. It is not part of the inner enhanced for loop since no brackets were used. Nested loop iterates the `exams` list as many times the number of elements in the list which is `2`. In first iteration of the outer for loop `e1` is `"OCA"`, in second iteration it is `"OCP"` . So, in the inner loop, `"OCA"` is concatenated and printed with the two elements in the list and then `"OCP"` is concatenated and printed with the two elements in the list, which as a result corresponds to `"OCA OCAOCA OCPOCP OCAOCP OCP"`

Q46)(C, D)

Option C and D is true since the javac command compiles a .java source code text file into a .class file which is a bytecode file. Also for that reason, option F and A is wrong. Option B and E is wrong because, java command does not compile any file, it is used for running already compiled .java files.

Q47)(C)

Only two lines of code compiles. `Integer.parseInt()` method returns an **`int`**. An int can be assigned to `Integer` type, **`int`** type or some primitive type that it can be promoted to, like **`float`**, **`double`** or **`long`**.

Q48)(F, B, D)

The drive method in the `Highway` class is an overloaded method with four different versions. Each version accepts different type of argument. There is two method signatures which returns int `3`. the one which accepts **`double`** as a parameter and the other one accepts **`short`** as a parameter, so the type of the value should be **`double`** or **`short`**. Also, because of the promotion of the primitive types to bigger types, a **`float`**, **`byte`** can be used too. They will be promoted to **`double`** type and **`short`** type respectively when passed to `int drive(double car)` and `public int drive(short car)` as an argument. **`int`** and **`long`** can also be promoted to **`double`** but they are already used as a type in other overloaded `drive()` method which returns `5` and `2` respectively.

Q49)(A)

The `main()` method, which is the entry point in our program has the **`static`** keyword missing. So the code compiles fine, bu it would throw exception at runtime. But if we correct the signature of the `main` method. Then it would print **`true`** two times. The first and the fourth `println` statement prints **`false`** because **`new`** keyword always instantiates new objects. Last `println` statement would print also **`false`** because, `LocalTime.now()` creates a new object and returns its reference and also, a little time passes between execution of `Localtime.now()` on the left hand side and the right handside. The third `println` statement prints **`true`** because the `"bart"` string is interned and reused. Second `println` statement, would obviously returns **`true`**, but note that it could not if there was an operation, maybe an addition and then, result of the addition was being compared to some value that we think it is equal to. That would be because the numbers in base `10` which are not the exponents of 2, can not be represented exactly in binary systems.

Q50)(D)

The code compiles and runs without any issue, printing `"245"`. Try statement throws a `RuntimeException`, more specifically `ArrayIndexOutOfBoundsException` because in the main method two arguments is passed as variable arguments, so trying to access a third element will throw a `RuntimeException`. First block catches the `RuntimeException` and prints `2`, and remember that the **`finally`** block executes after catch block whether or not it throws an exception, in this case printing `4`. Finally, after the `swing` method returns control to the `main` method, there is a `print` line which prints `5`, summing up to printed `"245"` on the screen.

Q51)(E)

The code compiles but throws an exception at runtime. The size of the `drinks` array is `2`. `container` variable of the for loop is initialized with the invoke of `drinks.size()` which is `2` and incremented by `1` as long as it is greater than zero. Problem occurs when we arrive the second iteration, (`container - 1`) given to `drinks.get()` evaluates to `2` and then `get()` throws `IndexOutOfBoundsException` because there is no third element. So, we get the output `"cup,"` with the first iteration since (`container -1`) evaluates to `1`, and the second iteration throws `RuntimeException`.

Q52)(A, F, E)

A main method/entry point to a program must be a method declared as **`public static void main`**`(String[] args)`. Some equivalent signatures are:

```
  public static void main(String a[])
              public static void main(String[] a)
              public static void main(String... a)
```

Also, it is still a valid entry point signature if main is declared **final**. All **final** versions of the above entry point signatures are valid entry points too.

Q53)(C, E, D)

Case statement values must be **final** variables or literals or we get an error, case expressions must be constant expressions. `winter` variable is declared **final** but the type **long** type is an incompatible type for **switch** statements. `fall` variable has a compatible type, but it is not declared as **final**. Line 11 should be removed too, because there can not be two **default** blocks in a **switch** statement.

Q54)(A, B, C)

Lines h1, h2, and h3 does not compile. h1 does compile because abstract interface methods can only be declared **public**. h2 does not compile because we can not reduced the visibility of inherited method from `Friend` interface. h3 does not compile because of the same reason. Line h3 also declares a `RuntimeException` but since it is an `UncheckedException` it is fine to do so. line h4 compiles fine because a `Dog` is also a `Friend`. Line h5 compiles fine too, but throws an exception at runtime because of the casting to `Cat` class. `friend` variable refers to a `Dog` object, not a `Cat` object so casting it to `Cat` will throw a `ClassCastException`. Line h6 compiles fine but casting a **null** variable, or in this case **null** value itself, will throw `NullPointerException`.

Q55)(F, E, B)

Unchecked Exceptions are caused by programming errors, so one could guess from the options. `FileNotFoundException`, `IOException` are certainly exceptions that the program could reasonably recover. Exception is the parent class of all exceptions and is an unchecked exception. `ArithmeticException`, `IllegalArgumentException` and `RuntimeException` can be considered as programming errors, and they are unchecked exceptions.

Q56)(F)

The code compiles and runs without issue. The `ship` variable is assigned to `10` because the ternary operation on the right-hand side returns `10`. space variable is not less than `2` and because of that `10` is returned from the ternary operation.
`printmessage()` member method has two distinct if-then statements. First if block gets executed because the condition (`ship > 1`) returns **true** and prints `"Goodbye"`. With second if-then-else, else block gets executed and prints `"See you again"` due to the fact

that the condition inside the if statement (`ship < 10 && space >= 2`) returns **`false`** because `ship` is not less than `10`.

Q57)(B, C, D)

`clock` variable is accessed in `Coffee` class which is in the same package with the `Alarm` class variable declared in. So the access modifier of the `clock` variable must be at least package-private or above. `getTime()` method is accessed in `Snooze` class which extends `Alarm` class but in a different package, so **`protected`** would suffice, **`public`** would grant more.

Q58)(B)

The code does not compile because of the line q1. `CarbonStructure` class is not an abstract class so it can not declare abstract methods. Aside from that problem, the override is done correctly. Subclass method throws a more specific subclass exception and it returns a `Long` which is covariant with `Number` type.

Q59)(C)

Although the main method's signature is not a proper signature to be an entry point, it is still a valid method. New `LocalDate` objects are acquired through static methods like `of()` or `now()` or `ofYearDay()` etc. There is no visible constructor that we can use to acquire a `LocalDate` object. Line 7 although is used wrongly, since each of the chained methods (`ofYear` and `ofDays`) returns new `Period` objects, but it is a valid code and compiles without any issue. Line 8 does not compile, there is no class such as `DateTimeFormat`, the correct name is `DateTimeFormatter`.

Q60)(A, E)

Option A is true; a catch block can never appear after a finally block. Option E is also true; a try block can have zero or more catch block. Option B is wrong since catch block is optional. Option C is wrong because it the opposite is true. Option D is false since a try block can be followed by a catch. Lastly; option F is wrong because a try statement can have zero or one finally block, not zero or more.

Q61)(B)

The `fish` variable is assigned to `4`, because the ternary operator evaluates to `4`. First the multiplication is done and then the addition, making the ternary operation `11 >= 2 ? 4 : 2` which returns `4`. `mammals` variables evaluation is a little bit tricky. There is another ternary operation inside a ternary operation. `3` is not less than itself, so the right-hand side is going to be returned from the ternary operation which is another ternary operation itself and it evaluates to `9`. The next statement which is a print statement prints the sum of `fish` and `mammals` which is `13`.

Q62)(A, C, E)

Option A is true, an object can be assigned to an inherited interface reference variable without an explicit cast, since then that object is also the implemented interface according to polymorphism. Option B is wrong since compiler can not know such thing, that is probably because dynamic dispatch. Compiler can not guarantee the concrete type of the object that the reference points to. Casting an object does not in any way change the object itself, we kind of just change in what way we access it, making option C true. A parent class object is not its subclass, so option D is wrong. But the vice versa is true and that makes option E true. Lastly, casting an object to one of its inherited type does not throw `ClassCastException`, making the last option F wrong.

Q63)(D)

The output is not guaranteed. `Arrays.binarySearch()` searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order or the result is undefined.

Q64)(B)

`shoe3` variable is a local variable defined in the shopping method, so at the end of the main method `shoe3` is already removed from the stack but before the shopping method hands over the control, it is assigned to `shoe1` class variable. `shoe2` is assigned `shoe1` before that. The only object which is referenceless in this frame is `shoe2` object, or more accurately, the object that was being referenced by `shoe2` variable.

Q65)(E)

The **throws** keyword is used in method declarations, the **finally** keyword is used to guarantee a statement will execute even if an exception is thrown, and the **throw** keyword is used to throw an exception to the surrounding process.

Q66)(B, E)

The code prints two lines before an `ArrayIndexOutOfBoundsException` is thrown. `nycTourLoops[i]`, since the length of the `nycTourLoop` array is based upon in the for loop's condition, does not throw exception in any of the iterations. But the `times[j]` causes `outOfBounds` when the `j` hit the 3. But before exception is thrown this iteration runs `2` times.

Q67)(E)

Because of **virtual methods** , it is possible to **override** a method, which allows Java to support **polymorphism**.

Static type of an object pointed to by a pointer or referenced by a reference is determined by the declaration of this pointer or reference. By its dynamic type we mean its real type. Dynamic type usually can only be determined during execution of the program, while its static type is known during compilation.

Java or in other object-oriented languages, what decides which method is going to be called is the dynamic type of the object pointed to by a pointer. A pointer can be of the base class type, but if we invoke a method for an object pointed to by this pointer and this object happens to be an object of a derived class, the method will be searched for in the scope of the derived class, in some languages via virtual method tables. We say that such a method is virtual. Therefore, in Java all non-private, non-final methods are virtual.

Q68)(E)

The throws an `ArrayIndexOutOfBoundsException` at runtime. `seasons()` method accepts variable arguments which are treated as an array inside the method. Four arguments are given to `seasons()` method so the array length is `4`. int `l` is assigned to the length of the second element (`"fall"`) in the array which is `4`. The next `println` statement tries to print the `names[l]` which evaluates to `names[4]` and throwing an exception because it is out of bounds of the array.

Q69)(D)

The code contains three lines which causes compilation errors. `MakesNoise` interface does not define a method such as `concert`, so even though the actual object referenced by the `mn` variable does define a method named `concert()`, it is an error. We can only call methods defined by the reference type. Secon error is with the line `8`, since the superclass does not define a default constructor, java can not insert a default constructor into the Drum class. Last error is with the line `11`, it calls `play(int)` method on super but the superclass does not contain a method with signature `play(int)`, it has a method `play()` which takes no parameter.

Q70)(B)

The code outputs `"6,LONG"`. Java is pass by value but take care for that because since the object reference copy is passed, we can still change the object in the heap by using the reference in the method. It is a copy but it does refer to the same object as the passed reference to the method as argument. In the `adjustPropellers` method, since the returned value is the incremented value of the first parameter and it is assigned to the `length` variable, length is `6` after the reassignment. The incrementation only increments the local variable, but it is returned at the end of the method and assigned to length variable. The `type` array's element at the index `0` is assigned to `"LONG"`.

Q71)(D)

Three lines contains compilations error. First error occurs when we try to catch a parent exception earlier than the subclass exception, this causes unreachable code because subclass exception will always be catched by the super class exception, hence causing a unreachable block of code. Second error is just a duplicate parameter error, we are declaring a variable in a scope where there is already a variable declare with the same name. The last error is in the main method, `openDrawBridge()` method declares an Checked exception, which must be declared again by the caller of the `openDrawBridge()` method, delegating handling of the exception to someone else or handle it in the place.

Q72)(D)

Object orientation and **encapsulation** are two properties that go hand in hand to improve class design by structuring a class with related attributes and actions while protecting the underlying data from access by other classes.

Q73)(E)

The code does not compile, because the `String` keywords when declaring the type of the variables `bike1` and `bike2` are typed wrong. This is a syntax error. Having this error corrected, code outputs `"false true"`. `bike1 == bike2` evaluates to **`false`** because two objects are different. `bike1.equals(bike2)` evaluates to **`true`** since the contents are the same.

Q74)(A)

The output of the following code when run as "`java EchoFirst seed flower plant`" is `'0'`. This code uses the `binarySearch` method correctly. It first sorts the array and then searches in it. The search statement in the code searches the index of the element at the index `0`, so it does return `0`. This is kind of pointless since we are searching the index of an element with using the index of an element we want to search, but it is valid.

Q75)(B, C, D)

Every for-each loop can be rewritten as a traditional for loop. Every for-each loop can also be rewritten as a while loop. Lastly, every for-each loop can be rewritten as a do-while loop.

The for-each loop, added with Java 5 also called the "enhanced for loop", is equivalent to using a `java.util.Iterator`. It's syntactic sugar for the same thing. Therefore, when reading each element, one by one and in order, a foreach should always be chosen over an iterator, as it is more convenient and concise.

Q76)(E)

The code does not compile because there is no such method with the signature `of(int, int)`. It should be `of(int, Month, int)` representing day, month and year respectively. Also there is no such methods `getHours()` or `plus(Hours)`.

Q77)(C)

Two array objects that were being pointed by the variables balls and scores are eligible for garbage collection before the end of the main method, because their references is assigned to `null`.

Q78)(B)

The while loop iterates `9` times until the `date.getMonth()` is equal to `Month.April`. `count++;` statement is not in the while block while it seems so at first glance. So the count variable is not `9` when printed but `1`. It is incremented only once.

Q79)(D)

Three lines contains compilations errors. There are two `return` statements in the `grunt()` method, which is an error. Also, we cannot call `super.grunt()` since it is unimplemented. And the `sing() += 10;` statement does not compile because the left-hand side of the assignment operator must be a variable, not a method.

Q80)(B, E)

`default` keyword is used in switch statements to specify default execution path. It is also used as a modifier in an interface method with a body.