Bünyamin AKTAŞ
05/06/2020

ASSIGNMENT - 8

Q1 **The answer is D.**

Java has exception feature which declares that there is something that is illegal and that Java cannot handle. Java has three types of exceptions; unchecked exceptions (runtime exceptions), checked exceptions and errors.

Unchecked exceptions extend *RuntimeException* class and do not require to be declared or handled.

Checked exceptions extend *Exception* class but not *RuntimeException* and require to be declared or handled.

Errors extend *Error* class and should not be handled or declared. *Error* means something went so horribly wrong that your program should not attempt to recover from it.

These exceptions can be handled by *try/catch/finally* blocks. A *try* block must be followed by a *catch* block and/or a *finally* block. Since none follows *try* statement in this code, it does not compile. **So the answer is D.**

Line *k1* is okay since declaring an exception in method definition is okay and it just declares that the method could throw an exception.

Q2 **The answer is B.**

try/catch block must always be in order try and catch. finally block comes at the end and always run whether catch block had caught an exception or not. So try block must come before catch block so that if an exception is thrown in the try block catch block could catch and handle it. **So the only correct answer is the option B.**

Q3 **The answer is D.**

Java has a *java.lang.Throwable* class for all exceptions and errors. *java.lang.Error* class extend *java.lang.Throwable* class and is not related to *java.lang.Exception* or *java.lang.RuntimeException* class. *java.lang.Exception* class is extended by *java.lang.RuntimeException*. **So the correct diagram is in the option D.**

Q4 **The answer is A.**

There is no class called CheckedException in Java so the option D is incorrect.

Exception must be handled or declared so that it must be caught.

RuntimeException might be caught and it is okay.

But *Error* should never be caught since it means a fatal failure such as going out of memory that recovering cannot be guaranteed. **So the answer is A.**

Q5 **The answer is D.**

This code contains a try/catch/finally block. In each block and outside of the blocks a variable score is printed and incremented then by 1. Since there is no mistake in definition of score and in print() method, try block will not throw any exception so that catch block will never run. finally block always runs so that it will execute its code.

However this code has a problem that a local variable that belongs to try block is attempted to be used outside of its scope. This will cause the code not to compile so **the answer is D.**

If we declare the variable outside of *try/catch/finally* statement then the code would compile since the variable now becomes present in all blocks. In this case try block would print 1 and increment the variable. catch block would be skipped and finally block would print the new value, 2, of score and increment it. At the end, outside of *try/catch/finally* blocks, another print() method would print 3 and increment the variable again.

Q6 **The answer is B.**

A - *ClassCastException* is a *RuntimeException* and occurs when an attempt is made to cast a class to a subclass of which it is not an instance.

B - *IOException* is a checked exception which occurs when there is a problem about reading and writing a file.

C - *ArrayIndexOutOfBoundsException* is a RuntimeException that occurs when an illegal index is used to access an array.

D - *IllegalArgumentException* is a *RuntimeException* that occurs when an inappropriate argument is passed into a method.

Q7 **The answer is A.**

“throws” keyword is used in method declarations and it means that the method may throw an exception.

“throw” keyword is used to throw an exception. When this keyword is used an exception will be thrown.

So the answer is the option A.

Q8 **The answer is B.**

try blocks can have multiple catch blocks in order to be able to catch multiple exceptions. Exceptions to be caught in these blocks are not required to be compatible, which makes the option D wrong.. Java checks catch blocks in order they appear. So if a broader exception appears before than a narrower one Java will never visit the check block for the narrower one because even if the narrower exception is thrown, it would be handled in the previous catch block for the broader exception. This means order of catch blocks matters so that the option C is also wrong.

IOException is a subclass of Exception so that it must appear before Exception class. If not, the code would be able to handle caught IOException in the

catch block for Exception. This creates an unreachable code, IOException catch block, which causes the code not to compile. So the only correct statement is the option B.

Q9 **The answer is D.**

This code has a statement, throw t, in the catch block. There is no variable or object named t so this statement will cause a compilation error **which makes the option D is correct.**

If this problem is fixed, for example by removing it, then the code would compile. The code would enter in try block and print A. Then it would throw an exception and check catch block to catch the exception but the catch block is not adequate for catching this exception since it is wider than the exception in catch definition. Finally, finally block would run since it runs whether an exception is thrown or not, which prints C.

When the try/catch/finally block ends, a stack trace for RuntimeException would be printed, which could have made the option C correct.

Q10 **The answer is C.**

The method openDrawbridge() declares that it could throw an exception. Then it implements a try/catch/finally block. try block throws an Exception with a message. catch block catches this exception and prints “Opening!”. Since catch block caught the exception, its message is not going to be printed. Finally, finally block runs and prints “Walls”. There is no compilation error until now.

In main() method openDrawbridge() method is invoked on a new Fortress object. But the exception which might be thrown by openDrawbridge() is not handled or declared. Since openDrawbridge declared that it might throw an exception, the caller must declare or handle this exception. **So the line p3 causes compilation error.**

If we fix the problem the output would be “Opening!Walls”.

Q11 **The answer is B.**

Runtime exceptions might be declared and/or handled but they are not required to be since that the methods are free to throw any runtime exception without mentioning them.

But checked exceptions which have Exception class in their hierarchy must be declared or handled.

“NullPointerException”, “RuntimeException” and “ArithmeticException” are runtime exceptions so they are not required to be declared or handled. “Exception” is a checked exception that does need to be declared or handled. The option b is correct.

Q12 **The answer is A.**

This code has try/catch/finally block which has two catch blocks. try block throws an exception which will be caught in the second catch block. The first catch block’s exception type is not related to the exception thrown by the try

block so that it will be skipped. As already explained in Q2, finally block runs whatever happens. So the code will print 1345.

Q13 **The answer is C.**

finally block can throw an exception as any block/statement in Java can. This could cause some lines of the finally block be skipped. So the options A and D is wrong.

finally block is executed whether an exception is caught or not, so the option B is also incorrect.

finally block needs brackets as well as the try and catch blocks.

Q14 **The answer is C.**

As explained in Q8, if a catch block is implemented for a broader exception before a narrower exception the catch block for the broader exception handles the narrower one's exceptions, too. This makes the catch block for the narrower exception unreachable so that the code does not compile.

Here, in this code "FileNotFoundException" which is a subclass of "IOException" is used after "IOException" so that this code does not compile.

If we change the order of the catch blocks then the code would compile and print "XY"

Q15 **The answer is C.**

There is no "finalize" structure in try/catch blocks.

As explained in Q1, a try block must be followed by a catch block and/or by a finally block. So the answer is the option C.

Q16 **The answer is B.**

A – This statement is true since exceptions appear and are used when thing go wrong.

B – If an application uses exception handling via try/catch blocks it may not terminate. So this statement is incorrect.

C – For example, ArithmeticException can be avoided by checking if the denominator is zero. This statement is correct.

D – Having try/catch structure in Java is for being able to recover from unexpected problems. This statement is also true.

Q17 **The answer is D.**

This code defines an exception class that extends Exception. This custom exception class is used in the first class "Transport" in the travel() method definition. Then the superclass of this custom exception is used in the second class in the travel() method definition. These two methods are not related

to each other since the classes to which they belong are not related. So the first class does not interest us.

The second travel() method block has a semicolon after the brackets, at the line j1, which is considered as an empty statement so this semicolon does not cause a compilation error.

main() method both declares and handles the Exception that travel() might throw. Then it prints the return value of the travel() method in a try block. But we have a problem here that parentheses are used for catch block instead of brackets which is a syntax error causing compilation error. So the does not compile making the option D correct.

If we change parentheses with brackets the code would compile and print 4 since it would never go into catch block because try block does not throw any exception.

Q18 **The answer is B.**

As it is explained in HW07-Q7, an overridden method cannot throw a new or a broader exception than the declared exception in the parent method. Since the printData() method throws “PrintException” the overridden methods either throws “PrintException”, or throws “PaperPrintException” since it extends “PrintException”, or throws nothing at all. Option A and C is valid according to this rule but option B attempts to throw “Exception” which is a superclass of ”PrintException” which is declared in the parent method. This is illegal so the signature in the option B is not allowed in a class implementing “Printer” interface.

Q19 **The answer is D.**

“Exception”, “RuntimeException” and “Throwable” classes are in the package “java.lang” which is imported automatically in every class. So they do not require a package import statement.

Q20 **The answer is C.**

A custom exception named GasException is defined.

getSymbol() method in “Element” class declares that it might throw a GasException.

“Oxygen” class, which extends “Element” class, overrides the parent method getSymbol() declaring a GasException again, which is legal since it does not declare a new or broader exception.

try block is also okay. But “catch” block’s definition is missing an exception reference definition. catch block has to have an exception reference in order to be able to catch exceptions. So the line g3 causes a compilation error.

Q21 **The answer is B.**

As explained in Q1, a program must handle or declare “checked exceptions” but should never handle “java.lang.Error”, which makes the option B correct.

A - “java.lang.Error” must not be handled or declared and “unchecked exceptions” might be handled and declared so this option is incorrect.

B - “checked exceptions” must be handled or declared and “java.lang.Error” should never be handled so this option is correct.

C - “java.lang.Throwable” may reference to Exceptions that could be handled and declared but may also reference to Error objects which should not be handled or declared. So this option is incorrect.

D - “unchecked exceptions” might be handled or declared but “java.lang.Exception” must be handled or declared so that this option is also incorrect.

Q22 The answer is B.

This code creates two custom checked exceptions called “CastleUnderSiegeException” and “KnightAttackingException”.

Then a runtime exception is declared on a method in class “Citadel”. In this method a try/catch block is used. One of custom exceptions is thrown in try block and caught in catch block which throws “ClassCastException” which is a runtime exception that does not need to be handled. In “finally” block a new checked exception is attempted to be thrown without handling it. This is not allowed. “So the code does not compile because of line q2.”

If we enclose the exception in “finally” block with try/catch block then the code would compile. Calling “openDrawbridge()” method does not require a exception declaring or handling since the method declares a runtime exception. Methods are free about declaring and handling runtime exceptions. In this case the program would compile and print a stack trace for the exception thrown in “catch” block. Since in this case the exception in “finally” block is handled, it may not print a stack trace depending on how it is handled.

Q23 The answer is A.

Java allows multiple catch blocks following a try block so the “option D” is wrong.

When there is an exception that is thrown, “catch” blocks are looked in order they appear. Whenever a match is found, that block is executed and the rest is skipped. So **the answer is A.**

Q24 The answer is C.

“compute()” method declares that it could throw an Exception which is a checked exception. And it throws a RuntimeException which is not required to be handled or declared.

main() method calls compute() method in try/catch block. Since the runtime exception is not required to be handled there is no need to write a catch block that exception. But this method declared that it could throw an Exception so that either this exception must be handled in the try/catch block or must be declared in main() method’s definition. “catch” block catches a “NullPointerException” which is a runtime exception, not a checked exception. So this code cannot catch Exception that might be thrown, which means this code does not compile.

If we declare Exception in main() method's definition, the code would compile and throw a runtime exception in compute() method printing a stack trace and terminate. It would not visit catch block since the exception thrown is not a "NullPointerException".

Q25 **The answer is D.**

An exception could be thrown depending on the value of "list".

If the array "list" is assigned to a "null" object then the "if-then" statement in the "for" loop would cause a NullPointerException which makes the option A correct.

If "list" is assigned an array object of size less than 10 it could cause an "ArrayIndexOutOfBoundsException" since the "for" loop attempts to reach an array of size, at least, 10.

"ClassCastException" would be thrown if "list" is assigned an Object which is not of type "Boolean".

So the answer is the option D.

Q26 **The answer is B.**

If a recursive method do not have a termination condition then it would call itself forever or until the memory is full. When the stack memory is full, the program throws "StackOverflowError".

A nonexistent object is called "null object" in Java which can cause a "NullPointerException" if it is attempted to be reached. "So **the answer is B.**"

"NoClassDefFoundError" occurs if a class that exists in compile time, could not be found in runtime.

"ClassCastException" occurs when a reference is attempted to be cast into a subclass of which it is not an instance.

"IllegalArgumentException" occurs when a method has been passed an illegal argument.

Q27 **The answer is C.**

A - When there is a checked exception in a method's signature, the caller must handle or declare this exception so this statement is correct.

B - When there is a declared checked exception in a method's signature, the caller will know that there is a potential problem that must be handled.

C - Declaring a checked exception in a method's signature ensures that a specific problem will never cause the application to terminate, not any problem. So this statement is wrong.

D - Declaring a checked exception in a method's signature gives the caller a chance to recover from that single problem as understood in previous statements.

Q28 **The answer is D.**

"try/catch-finally" statement must be in order a try block, one or more catch block(s) and an optional finally block. In this code the "finally" block comes before the "catch" block "which will cause a compilation error. So the answer is the option D."

If we change the order of "catch" and "finally" blocks the code would compile. In this case the method would firstly check if "try" block throws an exception and would execute the "catch" block if so. Then the "finally" block would be executed and finally the "return" statement would be executed exiting the method.

Since main() method does not pass a null object to the "getFullName()" method the method would not execute the "catch" block. "finally" block is executed and prints "Finished!". After that the print() method in main() method would print the returned value which is itself the arguments given by the main() method.

Q29 **The answer is A.**

As explained in Q1, a try block must be followed by a catch block and/or a finally block. This means a try statement can have a catch block and not a finally block, and vice versa.

So try statement can have zero or more of both blocks. **The answer is A.**

Q30 **The answer is D.**

An abstract class is defined with a protected instance variable which is not initialized explicitly. Java will initialize it to a default value which is zero for int.

"Ducklings" class extends "Duck" class and implements "getDuckies()" method returning the value of dividing "age" by "count".

main() method creates a "Ducklings" object, passing 5 to constructor which will be the value of "age", and assigns this object to a "Duck" reference which is okay since it is superclass of "Ducklings". Then main() method invokes "getDuckies()" method which will divide "age" by "count", whose value is default to zero and is not changed in "Ducklings" class. So it attempts to divide an integer by a zero which will throw an "ArithmeticException" at runtime. **So the answer is D.**

Q31 **The answer is B.**

If both "catch" and "finally" blocks throw exceptions only the last one will be thrown which is of "finally" block. The exception thrown in "catch" block is forgotten. This problem can be resolved by handling the exception thrown in "finally" block. But this question does not mention handling the exception in "finally" block so that in this case the caller sees only the exception from the "finally" block.

Java forgets the exception thrown from the catch block since finally block is executed no matter what happens in catch block and if finally block throws an exception Java forgets about everything else and goes on with this new exception.

Q32 **The answer is A.**

"BigCat" class defines a method, "roar(int)", which attempts to throw an exception in its signature via "throw" keyword, which is illegal since methods cannot throw exception in their signature. So the code does not compile because of the line m1.

The problem can be resolved by replacing "throw" keyword with "throws" keyword which makes the statement an exception declaration which is okay. Line m2 would be okay and it would overload the parent method changing the parameter list.

In main() method a "Lion" object would be created and assigned to a "BigCat" reference which is valid since "BigCat" is superclass of the "Lion". Then roar(int) method would be invoked on "kitty" reference and the code would print a stack trace.

Q33 **The answer is A.**

The second statement attempts to cast "exception" reference to RuntimeException. This statement would work if the "exception" pointed to an instance of RuntimeException or to an instance of child classes of RuntimeException. But "exception" points to an "Exception" object which is superclass of "RuntimeException" so this statement will throw a "ClassCastException".

"ClassCastException" is a runtime exception type but since we have an exact option for "ClassCastException", the option B cannot be the answer.

Q34 **The answer is C.**

As explained in Q3, "java.lang.Throwable" class is the parent class of all exceptions and errors. When a Throwable reference is used to catch an exception, any exception and any error will be caught since subclasses can be referenced by their parent class.

Q35 **There is no correct answer.**

A "finally" block is executed no matter what happens in "try" block. So since it is executed before the method exits, "Posted:" will be printed at first. Then the method will return the street and city names and a colon between names..

Only the second statement contains "Posted:" But none of the statements contain a colon in the middle of them. So none of the statements are correct.

Q36 **The answer is A.**

If a catch block for a broader exception comes before a catch block for a narrower one, the second catch block never executes since all exception compatible with the second block will be handled in the first block. This will even cause a compilation error for that the second block will never be reached.

"ClassCastException" is a subclass of RuntimeException so that it must appear before RuntimeException. So the option A is correct.

There is no rule such as exceptions must be compatible with each other

in catch blocks. Any type of exception must appear in catch blocks.

Q37 **The answer is C.**

A - If the computer caught fire, the computer itself does not work so that exceptions is not the concern.

B - Exceptions are thrown at runtime so a code not compiling is not an exception use case.

C - If a caller passes an invalid argument to a method it may terminate the program and also it can be avoided using exceptions. So this is a good scenario for using exceptions.

D - A method finishing sooner is also not a good case for using exception. It may not be finishing because of an error, but may be finishing because of such as input size.

Q38 **The answer is C.**

This code attempts to override a method declaring a new exception type which is illegal so that the code does not compile.

If we remove exception declaration or change it to RuntimeException on the overridden method, the code compiles. And it would invoke the overridden method in main() method printing “beat”.

Q39 **The answer is D.**

As explained in Q1, runtime exceptions are not required to be handled or declared while checked exceptions are required to be.

“NullPointerException” is a runtime exception so that it is not required to be handled or declared so the options A and B are incorrect.

Option C is also incorrect since although runtime exceptions are not required to be handled or declared, they might be, which makes the option D correct.

Q40 **The answer is D.**

The code does not compile since the statement in the *catch* block in the *zipper()* method is missing “new” keyword. Since exceptions are in fact objects they must be created by calling their constructor so that they need *new* keyword when getting thrown via *throw* keyword. **So the answer is D.**

If we fix the problem by adding “new” to the statement the code would compile. In that case, *zipper()* method tries to assign a String reference to a new Object type object via casting which is illegal since Object is wider than String. This attempt throws a *ClassCastException*, which is a runtime exception, which will be caught in *catch* block throwing a runtime exception. The *catch* block for *Exception* can catch *RuntimeException*, too, since it is a child of *Exception*. So this method ends without returning any value since the program flow never comes to the return statement.

main() method invokes *zipper()* method in *try/catch* block as it must do so since the method declared that it could throw an exception. When *zipper()* method is invoked it will throw a runtime exception at the end and the code will jump to *catch* block again skipping print() method.

The *catch* block in main() method catches all exceptions and error since it has *Throwable* reference which is the superclass of all exceptions and errors.

Since the last exception would also be handled, the program would terminate without producing any output.

Q41 **The answer is C.**

try block throws a "ClassCastException" which is a subclass of "RuntimeException". Then the code begins to check "catch" blocks for a match. The second "catch" block matches since it catches "RuntimeException". This "catch" block is executed and throws a new NullPointerException. Finally, "finally" block is executed throwing RuntimeException. Since "finally" throws an exception, too, the exception thrown in previous "catch" block is forgotten and a stack trace is printed for the RuntimeException thrown at the end. So **the answer is C.**

Q42 **The answer is D.**

Since an overridden method cannot declare a new exception or a broader exception than the parent method this method can only declare "OutOfBoundsException" since this is the most subclass of custom exceptions defined in this code. Only the option A declares appropriate exception.

But even the option A, as well as the others, uses int return type while the parent method uses void. This is not allowed in method overriding, return types must be same. So **the answer is D.**

Q43 **The answer is D.**

Since the "catch" block is missing exception identifier the code does not compile.

If we add a reference name, exception identifier, the code would compile. But since "IllegalArgumentException" is a runtime exception and not an "Error", the exception cannot be caught. And a stack trace for "IllegalArgumentException" would be printed.

Q44 **The answer is D.**

DragonException and *Exception* are checked exceptions that does need to be handled or declared. *openDrawbridge()* method declares *Exception* so that these do not need a *try/catch block*.

In *try* block, a checked exception is thrown and is not caught since *catch* block catches *RuntimeException.* But this is okay for compilation since the method declared *Exception. finally* block throws a *RuntimeException* which causes the previous exception thrown from *try* block to be forgotten.

main() method invokes *openDrawbridge()* method on a *Lair* object throwing the exception from *finally* block. This invoke is okay since main() method also declares *Exception.*

So the code compiles and prints a stack trace.

Q45 **The answer is C.**

catch blocks' order matter when there are exceptions related to each other as explained in **Q8**. *IllegalArgumentException* and *ClassCastException* are subclasses of *RuntimeException* but they have no relationship other than that.

These exceptions are incompatible but this does not prevent to have multiple *catch* blocks for these exceptions.

As a result, order of *catch* blocks for these exception types are not important.

Q46 **The answer is D.**

This code attempts to define a class that implements *RuntimeException* class which is not legal since *RuntimeException* is not an interface. S**o the code does not compile.**

If *implements* was replaced with *extends* the code would compile. In that case, *try* block throws *BiggerProblem* which extends *RuntimeException* and it gets caught in the first *catch* block printing "Problem?". The second *catch* block is skipped since the exception thrown from *try* block is handled already. If it didn't, it can get caught here, since *Problem* is a superclass of the exception thrown by *try* block.

Then, *finally* block would be executed and print "Fixed!". So the final output would be "Problem?Fixed!".

Q47 **The answer is D.**

throws keyword is used to declare exceptions in method signatures. So the statement in *try* block is not valid and causes a compilation error. **The code does not compile.**

If we fix the problem by replacing *throws* by *throw* then the code would compile and throw a *RuntimeException* with a message since the exception is not handled. The output would be throwing a *RuntimeException* with a message "Circuit Break!". But since *finally* must be executed no matter what happens, it would be executed and print "Flipped!" before throwing exception.

Overridden *flipSwitch()* definition is okay since overridden methods can have a narrower exception, or can not have any exception at all.

main() method declares *Throwable* but it is not necessary since *flipSwitch()* method throws a *RuntimeException* which is not required to be handled or declared.

Q48 **The answer is D.**

java.lang.Error is thrown when there is a problem that cannot be recovered from.

A – This problem can be handled easily by blocking one user while

allowing the other one. When the first one has done his/her job, the other can be allowed.

B – This problem can also be handled by waiting for internet connection since it says the connection is lost temporarily.

C – If a user enters password incorrectly, it can be asked again.

D – If the application runs out of memory the program cannot handle this problem since even solving the problem requires to use memory which is not available.

Q49 **The answer is C.**

For exceptions that have a relationship, the *catch* block for the narrower one must appear before the broader one. It is so in this code that *IOException* is caught by a *catch* block for an *Exception,* which has another *try/catch block* inside. This new *try/catch block* is also okay since it the *catch* block is for the same exception that is thrown from *try* block. If the code compiles this code would print "Failed" since until that print() method every exception is handled.

However, we have a problem that the inner *catch* block uses "e" identifier for the second time which is not allowed and causes the code not to compile. So the line z1 causes the problem.

Q50 **The answer is B.**

try block attempts to access to fourth element of the array of size 3, which will throw *ArrayIndexOutOfBoundsException.* This is a *RuntimeException* which will be caught by the *catch* block printing "Awake". But *finally* block is always executed and it throws an unhandled checked exception which is not allowed, in line x1. **So the code does not compile because of line x1.**

If we surround that line with a *try/catch* block then the problem would be solved.