Bünyamin AKTAŞ
22/05/2020

ASSIGNMENT – 6

Q1 **The answer is C.**

Java has four different access modifiers. These are as below:

public : allows access from everywhere.

protected : allows access from classes in the same package and from the subclasses.

(default) : allows access from the classes in the same package. This access modifier doesn't have any keyword to be used. If the member of the class doesn't have any access modifier keyword in its declaration, then it has default access. This access type is also called ***package-private.***

private : allows access only from inside the class to which the member belongs.

The question requires two access modifiers in descending order.

A – Since *package-private* is narrower than *protected* this option is wrong.

B – *protected* is narrower than *public.*

C – *protected* wider than *package-private* since *protected* allows subclasses to access in addition to *package-protected.*

D – *private* is the most restrict modifier.

Q2 **The answer is B.**

There is no command called *construct()* or *that()* so that **the options C** and **D** are wrong.

super() command is used to call the constructor of the parent class.

this() command is used to call another constructor. Java knows which constructor is meant by looking at the parameter list.

Q3 **The answer is D.**

Since the *sell()* method that has a *boolean* return type is missing a default case situation that would prevent the method from terminating without returning a value, **the code would not compile.** We need either an *else* statement in this method to solve the problem or a *return* statement outside of the *if-then-else* statement.

If we add an *else* statement then the code would compile. Since in main() method three objects are created calling *sell()* method, the static variable *price* is incremented at each time. And since *price* is static there is only one copy of it and it is shared among all the instances of the *Bond* class.

The initializing value of *price* is 5. *sell()* is invoked three times so the last value of it would be 8.

Q4 **The answer is D.**

This code snippet has a classical method structure that methods calling other methods.

print() method takes *nested()* method as argument. *print()* method takes methods as argument and prints the values returned by these methods. But here we have a problem. *Nested()* method has *void* return type which means this method returns nothing. So *print()* method has no value to print. This will cause a compiler error. **So the answer is the option D.**

If we fix the above problem as;

public int nested() { return nested(2,true); } // g1. Then we would have no problem any more. The semicolon at the end of the line *g2* is valid. Java allows stand-alone semicolons since it means a null statement. And the semicolon here is outside of the method block so it is valid.

Since the fixed *nested()* method would return the value of *nested(int,boolean)* method, in which the *boolean* parameter is not used, which returns the value of *nested(int)* method which returns the incremented value of the parameter. So, since the first int argument passed is 2, the return value would be 3.

Q5 **The answer is B.**

There are two types of parameter passing approach among programming languages: *pass-by-value* and *pass-by-reference.* So **the option A** is wrong since there is no parameter passing approach called *pass-by-null.*

Pass-by-Value : In this approach, only the value of the variable is passed into the method so that the changes made to the parameter does not affect the original variable’s value. **Java uses this approach.**

Pass-by-Reference : In this approach, the reference to the value of the variable is passed to the method so the changed made to the argument affects the original variable, too. **Java does not use this approach.**

Q6 **The answer is C.**

JavaBean naming conventions are:

1 – Properties are private: *private int numEggs;*

2 – Getter methods begin with *is* if the property is a *boolean*: *public boolean isHappy(){ return happy; }*

3 – Getter methods begin with *get* if the property is not a *boolean*: *public int getNumEggs() { return numEggs; }*

4 – Setter methods begin with *set*: *public void setHappy(boolean happy) { this.happy = happy; }*

5 – The method name must have a prefix of *set/get/is,* followed by the first letter of the property in uppercase, followed by the rest of the property name: *public void setNumEggs(int num) { numEggs = num; }*

A – This method's name is valid according to the JavaBean convention but since this method is named as a getter method, it must return a value. But this method has void return type. So this option is not correct.

B – This method's name is valid as a setter method but it must have a parameter to set *bow.* Since it does not have any parameter this option is also incorrect.

C – This method's name is valid as a setter method and this method also has a parameter to be assigned to a property. **So this option is correct.**

D – This method is neither a setter nor a getter method.

Q7 **The answer is B.**

A – this() command must be the first line of the constructor in order to prevent conflicts between constructors.

B – super() calls the constructor of the parent class. The parent class must be set correctly before setting the current class's object to ensure the object is set correctly. **So this statement is wrong.**

C – If this() takes arguments then there must be a constructor that has a parameter list matching these arguments.

D – If there is a constructor calling *this()* then Java would not create a default constructor so that the no-argument constructor must be implemented explicitly.

Q8 **All options are incorrect.**

A – Since Java is case sensitive and *public* access modifier begins with a lower p, this signature would cause a compiler error.

B – Java can autobox an int value to an Integer object or assign an int value to a wider type, for example to *long.* But it cannot convert *int* to *long* and then to *Long.* So this option is also incorrect.

C – void methods cannot return any value so that this keyword would cause a compiler error.

D – This signature creates a method that has a String return type which is incompatible since the statement returns an int value.

Q9 **The answer is C.**

A – A public variable in an instance of a class can be accessed from everywhere if the reference to the instance is maintained in all instances. So this does not guarantee that the variable can always be accessible.

B – There is no keyword names *local* in Java.

C – *static* variables are class variables that they can be accessed from all instances.

D – instance variables can only be accessed by the members of the instance to which the variable belongs.

Q10 **The answer is A.**

This code has a private property and a protected property but since it is never used outside of the class, it means nothing.

This code has two constructors, one is no-argument and the other one has one parameter.

A – If we insert *this(4)* at line p1 then this statement will call the other constructor that takes one int parameter. That other parameter assigns a value to the *rope* property according to the value of boolean variable *outside.* Since *outside* is still not initialized explicitly at this point, Java will assign default value *false* to this property. Since *outside* is false, *rope* will be one more than the value passed to the constructor, which is 5. So this statement provides what we need.

B – *new Jump(4)* will create a new instance calling the one-argument constructor which will cause the *rope* to be 5. But since this constructor does not print anything and also does not affect the *rope* in the current instance, this will not make the application print 5.

C – This is the same statement as **the option A** with one difference that the argument passed is 5, so that the *rope* will be 6.

D – If we simply put *rope = 4,* this will make *rope* 4. The application prints 4 rather than 5.

Q11 **The answer is B.**

A – If members of a class are declared public then they can be accessed from anywhere. So that any instance can access to this instance's members.

B – If parent class's members are declared package-private then they are accessible only from the classes in the same package. Even subclasses cannot reach them if they are not in the same package.

C – Since we can compare two instances of a class by equality operator or *equals()* method, these two instances can see their corresponding private data.

D – *protected* access modifier provides access to the classes in the same package as *package-private* access modifier does. So this statement is also true that an instance of a class can access to an instance of another class's attributes if both classes are in the same package.

Q12 **The answer is D.**

In order to ensure that data of a class is properly encapsulated the property variables which represent the data of a class must be declared private. But in this class the property *stuff* is declared public which makes it accessible and modifiable from and by everywhere. So nothing that could be placed in the blanks can make the data encapsulated. **So the correct option is D.**

If we change the access modifier of the *stuff* to private then all the options would be correct since the name of the method is not related to encapsulation. Also the method may be public.

Q13 **The answer is C.**

A – Java compiler will insert a no-argument constructor if there is no constructor declared explicitly.

B – If there is no implemented constructor in the parent class Java will automatically create a default no-argument constructor in the parent class as it does in all classes so that the parent class could be not implementing a constructor.

C – If a class extends another class and does not implement a constructor explicitly Java will attempt to create a no-argument constructor calling super() which requires a no-argument constructor in the parent class. So if the parent class has only one-argument constructor the child class must implement a constructor to prevent this situation. **So this statement is correct.**

D – Constructor can be overloaded by only changing parameter lists. So that if there is more than one no-argument constructor Java will see these constructors as same so that the code will not compile.

Q14 **The answer is A.**

All options have a vararg parameter which is a special parameter type in Java that is considered as an array parameter. This parameter has to be at the end of the parameter list since it can take individual values as argument and then turn them into an array. In **options B, C** and **D** are vararg parameters whose places are not the end of the parameter list. **Only the option A has a vararg parameter that is at the end of the parameter list.**

Q15 **The answer is C.**

Java uses *pass-by-value* approach for method parameters. So that references of objects and copies of primitive data are sent to the method. This means changes made to the primitives and to the references do not affect the primitives and references in original method. But the data in objects pointed by the references passed to the method can be modified.

A copy of *mySkier* reference is sent to the *slalom* method. Since the method attempts to change the value of *age* using this reference it affects the original object. But then when the code sets the reference to the *mySkier's* object to null, it does not affect the original reference. So the final state of *mySkier* is pointing to a *Ski* object whose property named *age* is 18.

Then a copy reference of to the *mySpeed* array is sent to the *slalom* method. Then this copy reference is reassigned to another object. Since it is reassigned to another object, operations performed on this reference do not affect the original array any more. And since the members of the original array *mySpeed are* not initialized explicitly, its members are default to 0.

Since String is an immutable object, changes made to this object's reference in the method do not affect the original object which means the final state of the *myName* is Rosie.

So the only diagram representing the situation is in the option C.

The thing that should also be noted is that the keyword *final* makes variables not changeable but as explained above even if the *final* keyword is not used, the final situation would not change.

Q16 **The answer is B.**

Method overloading means having multiple methods having same name with a different parameter list. Access modifiers and optional specifiers are not related to overloading methods. So in order to have overloaded *findAverage,* methods we need methods that are named *findAverage* and have parameter lists except (int sum). Since the method signatures in **the options A** and **D** have the *findAverage* name with a parameter list (int sum), these methods are considered same method as the original one so that the code does not compile.

The method in **option C** has a different name than the original one so that this is not overloading.

Finally the method signature in **the option B** has the same name as the original one and a different parameter list so that this signature is a overloading.

Q17 **The answer is D.**

Encapsulating data of a class prevents uncontrolled intervention to and modification of the properties of the class so that the data integrity can be maintained and the usability of the class could be improved due to that well encapsulated classes are easier to use.

Encapsulation does not ensure concurrency and well performance so that **the option D** is wrong.

Q18 **The answer is A.**

The original value of primitives and String variables cannot be modified in methods since only the copies of them are passed into the method.

For objects, a copy of reference to the object is passed into the method. So the original reference cannot be changed but the object pointed by the reference can be modified.

Since arrays are objects, members of arrays can be modified.

Q19 **The answer is B.**

Since the class is in the same package there is no need to explicitly specify the package name. In order to be able to call the method specifying only the name of the method it should be *static imported* which is not. So we can invoke this method by stating with its class name.

Java uses dots (.) to direct packages and folders. Colons has no use to direct to classes and methods.

So the best way to call this method is: *MathHelper.roundValue(3.1).*

Q20 **The answer is D.**

Only return type for methods that does not return anything is *void.* But even this one does not prevent a return statement from being used. A void return statement, *"return;"* , can be used in *void* methods.

byte and *String* methods must return a value in these types so that these methods cannot prevent return statements from being used.

Q21 **The answer is C.**

final keyword prevent variables from being changed, methods from being overloaded, classes from being extended. So the *final*s in class and method signature is okay for this code.

But the final variables *score* and *result* are attempted to be reassigned in the *finish()* method. This is illegal. So *final* keyword must be removed from these variables.

Q22 **The answer is D.**

In Java we use *super()* to call parent class's constructor and *super* followed by a dot, to access a member of the parent class. For example if the parent class has a method named *calculate()* we can reach it by *super.calculate().*

So the answer is D.

Q23 **The answer is B.**

As explained in **Q1** if a method does not have any access modifier then it has default access which allows only the class in the same package to access to this class's members. Not even the subclasses can reach to members of this class if they are not in the same package.

Q24 **The answer is A.**

Encapsulation is about protecting data of the class. This class has two properties called *material* and *strength.* Former is already private but the latter is *protected.* It must be *private* to prevent access from outside of the class. So **the statement I** is true.

Adding getter and setter methods improves usability of the class but they are not related to encapsulation.

So only the statement I is true.

Q25 **The answer is A.**

Method names can have letters, numbers, $ and _. Numbers cannot be at the beginning of the name. So **the option D** is not a valid name.

new is a reserved keyword so that it cannot be used to specify a name. **The option C** is also wrong. **The option B** contains an illegal character (-) so that this name is not valid.

Only valid method name is option A.

Q26 **The answer is D.**

A method signature can have optional specifiers and access modifiers that must be before everything else and return types must be right before the method names but here *static* keyword is put before the method name. This method's declaration is illegal so that the code does not compile. **The answer is D.**

Q27 **The answer is B.**

When passing values to methods, Java passes copies of them. When this value is an object, Java passes a copy reference to the object. This means changes made to primitives such as *boolean* values, does not affect the variable in the calling method.

If an object reference is reassigned in the method, it only changes the copy reference, not affects the reference in the calling method. However if data in the object pointed by the copy reference is altered then this change is reflected in the calling method, too. **So the answer is option B.**

Q28 **The answer is C.**

final properties of classes must be initialized either in their declaration statement or in static initializers or in instance initializers or in constructors. If they are not initialized in these structures **the code does not compile.**

If this problem is solved then we have another problem which is the setter method that sets the *final contents*. Since *contents* is final, it cannot be reassigned but this method attempts to reassign it. **This will also cause the code not to compile.** The other parts of the code is okay.

Q29 **The answer is A.**

As explained in **Q6,** valid JavaBean method prefixes are *is, get* and *set.* There is no JavaBean method prefix called *gimme, request* or *put.* **So the answer is A.**

Q30 **The answer is C.**

The second class attempts to call the static method that belongs to the first class. Since the second class only uses the name of the method, the method must be imported to the class. Static methods can be imported with "import static" keyword. So the valid importing statement for this method is:

import static clothes.Store.getClothes;

since the class *Store* which has *getClothes* method, is in the package *clothes.*

Q31 The answer is D.

As explained in **Q1,** package-private means that only the classes in the same package can access to that member of the class. This access type is called default access in Java and does not have any keyword.

private makes members available only in the class.

default does not exist for defining an access type.

protected makes members available in the same package and in the subclasses.

Since there is no required (or optional) keyword to define package-private the answer is the option D.

Q32 The answer is B.

The method *Stars()* has a constructor name but this is allowed since Java will know which one is referred when *Stars()* is used. For example if the constructor is referred then there will be *new* keyword. When the method is referred then it will be used with a reference to an instance of that class. Since references cannot call constructors of the class of the object they point, Java will know that this statement refers to the method.

The method *Stars()* attempts to call the constructor of parent class. This is not legal since the constructor calls must be the first line of classes. Methods is not the first structures to be run. So this code will cause a compiler error.

The constructor *Stars(int)* assigns the property *inThe* to itself. This is a meaningless statement but it is not illegal.

main() method prints the *inThe* property by specifying the whole path of it, which is legal.

So only one line contains a compilation error.

Q33 The answer is A.

Static members cannot refer to instance members without using an instance reference. Instance members can refer to static members. This makes **options B** and **C** incorrect. **The option A is correct.**

final static variables are non-changeable and are shared among instances. Since *final* variables cannot be reassigned they must be initialized in the structures that are possibly to be run firstly. Constructors are one of the first structures to be run but they belong to instances. So that they can cause the final variable to be reassigned or not to be assigned at all. So that **option D** is incorrect, too.

Q34 **The answer is B.**

The method has a *short* return type so that it must return a *short* or a narrower value.

In **option A** an *Integer* value is attempted to be returned but it wider than *short.* So this is not valid.

In **options C** and **D** a *long* and a *Long* value is attempted to be returned but they are also valid since they are larger than *short.*

In option B a value cast to *byte* is attempted to be returned and it is valid since *byte* is narrower than *short.*

Q35 **The answer is C.**

About method overloading there are 2 rules:

1 – Methods must have same name.

2 – Methods must have a different parameter list.

Else is irrelevant to method overloading. So they can have either same return type or a different return type. **The correct statement are I and III.**

Q36 **The answer is B.**

As it is explained in **Q33,** *final static* variables must be initialized in their declaration line or in static initializers. This class has no static initializer so that *monday* must be initialized in its declaration statement. Since it does not, the code does not compile.**(1)**

final static wednesday = 3; is missing data type so that this line causes compiler error. **(2)**

Access modifiers and optional specifiers can be placed in any order as long as optional specifiers are put together. So the last statement is correct.

So that two lines must be removed in order for the code to compile.

Q37 **The answer is D.**

print() method in main() method attempts to call a constructor of Puppy class that has one parameter which does not exist. So this line cause a compilation error.

If the method *Puppy(int)* is desired to be called then the syntax must be so: *new Puppy().wag.*

Q38 **The answer is A.**

As explained in **Q1,** access modifiers are *public, protected, package-private* and *private,* in descending order from larger scope to the most restrict one.

The statement in the question asks for 2 access modifiers in descending order. **Only the option A, *public, private* has what is requested.**

Q39 **The answer is A.**

The code does not have any syntax error. A *final* Phone object with reference name phone is created and sent to the *sendHome()* method. But it is useless since the method reassigns the copy reference passed to it, to a new object. Since our main object is not affected from any modification in *sendHome()* method, print() method will print 3 as it is the size of the property named *size.*

Q40 **The answer is B.**

water() method is a static method so that it can be called with its class name which is Drink. So **the option D** is true.

water() can also be called with using any path specifier since it is attempted to be called within the class already it belongs to. So **the option A** is also true.

this keyword is used to refer to class members so that if we use *this* followed by dot, with class methods or variables, it is valid. So **the option C** is also true.

As explained above *this* is used to refer to members of the current class. So in order for **the option B** to be valid there has to be a member named Drink in the class Drink other than a constructor. So this statement is illegal.

Q41 **The answer is C.**

This method has 3 parameters. First one is an *int* parameter that must be specified when calling this method. Second one is a *String* parameter that must be specified when calling this method. Third one is a vararg parameter whose data type is *String.* A vararg parameter can take an array or, zero or more individual values as argument since it will convert these individual arguments to an array.

So consequently when this method is called, at least 2 arguments must be passed; one for *int* parameter and one for *String* parameter. If argument(s) is wanted to be passed for the vararg parameter then they all must be of type *String.*

A – This call has valid arguments for the first two parameters. But the third argument which is of type *int* is not legal for *String* vararg parameter.

B – This method has only one argument so that it is not valid.

C – This method has two legit arguments and an individual argument for the vararg parameter.

D – This method has two arguments but first one is of type *String* which is not legit since it must be of type *int.*

Q42 **The answer is D.**

A – static variables does not have to be initialized explicitly since Java will assign a default value to them.

B – If static final variables are not private or a proper getter method is implemented then they can be read outside of the class.

C – If we import *static* methods with *import static* keywords then we can reference to them.

D – A static variable belongs to the class's itself rather than to its instances so that it is available in all instances of that class.

Q43 **The answer is A.**

A – If parent constructor is not called explicitly then Java will try call a no-argument parent constructor. As long as parent class has a no-argument constructor, the first line of the constructor of the subclass does not have to be super(). If parent class does not have a no-argument constructor then an explicit call to parent constructor must be implemented.

B – If a class does not implement a constructor explicitly then Java will create a no-argument constructor automatically so that yes, a class does not have to have a constructor explicitly defined.

C – If the parent constructor has a parameter list then the subclass's constructor must pass arguments to it while calling with *super().*

D – A final instance variable must be initialized explicitly. So that it must be set in a structure that is certain to be run whenever the class is used. These are the variable's own declaration statement, static initializers, instance initializers and constructors. Since static members of a class cannot refer to instance members, a final instance variable must be set either in its declaration statement or in an instance initializer or in a constructor.

Q44 **The answer is D.**

Static members of a class cannot refer to the instance members. But in this code an instance variable named *height,* is attempted to be used in static initializers which is illegal. So no matter we remove final modifiers or not, the code does not compile. And actually, final modifiers do not cause any error here.

Q45 **The answer is D.**

Since class Forest has only one constructor that takes one argument, subclasses of this class must explicitly call this constructor in their constructor. If not, Java will attempt to call a no-argument parent constructor and since it does not exist this will cause a problem.

In this code the constructor of RainForest does not implement an explicit call for parent constructor so that the code does not compile.

If the super constructor is called, then the code does compile. RainForest inherits *treeCount* attribute from its superclass and can access to it since it is defined *public*. One more of the argument passed into its constructor, which is 5, is assigned to this *treeCount.* So if the error was fixed by calling parent constructor the code would print 6.

Q46 **The answer is A.**

This code has a constructor calling a parent constructor although it does not extend any class explicitly. In Java, all classes are subclasses of the Object class, so that every class calls super() either explicitly or implicitly. So this structure is valid.

print() method creates a new instance of ChooseWisely class calling *choose()* method by passing a value. The value passed to this method is an addition of a *byte* value by casting and an *int* literal value. Since one of the values is *int* the sum will be *int.* There 3 overloaded methods of *choose()* takes parameters *int, short* and *long,* respectively. When there are multiple overloaded methods Java will select the most proper one. In this example Java shall choose a method that has a *int* parameter or a wider type than *int.* Since there is a method having *int* parameter Java will choose it, which prints 5.

Q47 **The answer is C.**

The method *getScore* takes a Long parameter and returns its two times multiplied value. This is okay.

print() method sends a final int variable as an argument to this method. Being final is not a problem here. But Java cannot autobox *int* to *Long.* Java can autobox *int* to *Integer* or convert *int* to *long.* But it cannot handle two steps converting. **So the code does not compile because of line m2.**

Q48 **The answer is A.**

As explained in **Q25,** method names can have letters, numbers, $ and _ and numbers cannot be at the beginning of the name. **Option B** has \, **option C** has # and **option D** has % characters which are not allowed in method names.

Only the option A's method is valid.

Q49 **The answer is B.**

As already explained in **Q1,** *protected* members of a class can be accessed from the classes in the same package as the class of the member and from subclasses.

Subclasses can reach members of parent classes but parent classes cannot reach the members of their subclasses. So **the option D** is incorrect.

Q50 **The answer is D.**

import static keywords imports static members of a class. But the class *Bank* has no static member. So none of the members of this class can be imported to another class providing the ability to use them without using an instance reference name.

If the methods in class Bank were static then the statement to import those methods would be: *import static commerce.Bank.*;* (*) is a wildcard to refer to all members to be imported.

Option B is not valid since it puts the *static* before *import* which is not allowed.

Option C attempts to import the class Bank which is also not valid since classes cannot be imported via *import static.*