Bünyamin AKTAŞ
31/05/2020

ASSIGNMENT – 7

Q1 **The answer is C.**

In this code, the class *Movie* extends the class *Cinema* which does not have a no-argument constructor. If the subclass *Movie* does not implement any constructor or put *super()* at the first line of the constructor, Java will try to put it automatically as a call to a no-argument constructor. Since *Movie* does not put any *super()* statement in constructor and the superclass does not have any no-argument constructor, Java adding automatically no-argument *super()* will cause the compilation error.

Other than this, in the main() method, the property *name,* which is private and does not belong to the class main() method belongs to, is attempted to be printed. Since it cannot be reached, this line will also cause a compilation error.

So we have two lines causing compilation error.

Q2 **The answer is D.**

All interface methods are assumed to have *public* access modifier no matter whether *public* is written or not. So *protected* is wrong.

Abstract interface methods do not have any body so that they must be implemented in the classes that implements those interfaces. Since these methods exist to be overridden, they cannot be marked as *final,* which prevents methods from being overridden, or as *static,* which also prevents methods from being overridden and makes them hidden.

So the only valid modifier that can be applied to an abstract interface method is *public.*

Q3 **The answer is C.**

This code contains two methods that have same name. In Java we can have multiple methods sharing same name under one condition: parameter lists must be different. Changing modifiers or return types do not matter in this manner. Since the two methods in this code share the same name and the same parameter list, Java will see these methods as duplicate and will not compile the code.

If we attempt to fix this error by changing parameter list of one of the methods, we have another problem. That is the second method whose return type is int, does not return anything. We must provide a return statement to that method.

Q4 **The answer is A.**

A – Inheritance allows classes to access their parent class's attributes and methods in order to improve code reusability. So this statement is correct.

B – Inheritance does not guarantee simpler code. It can be more complex depending on the structuring.

C – All classes in Java are subclasses of the class *Object* so that all objects in Java inherit a set of methods. But primitives do not inherit anything.

D – Methods referencing themselves are recursive methods and they are not related to inheritance.

Q5 **The answer is A.**

Java has polymorphism which means an object either can be referenced with the same type as the object itself, or as the superclass of the object, or as the interface that the object implements without an explicit cast.

this keyword reference to the current object. So *getDoggy()* method returns an object of *Canine* class. So the return type of the method must be the type of the object, or the type of the superclass or the type of the interface that object implements, which are *Canine, Object,* and *Pet*, respectively. *Object* is okay since all classes inherits from *java.lang.Object* class in Java.

There is no type called *Class* in Java and in this code snippet. **So the option A is wrong.**

Q6 **The answer is B.**

static methods and access modifiers are not helpful to work together with an incomplete code. In order to work properly with a code that is not complete, a class structure must be used.

An abstract class can be implemented to be the superclass of these codes but since each class can extend only one abstract class, this may limit these codes' abilities.

But classes can implement multiple interfaces so that having an interface would not limit any abilities of those codes. So the best facilitate is an interface for this problem.

Q7 **The answer is B.**

This code uses *method overriding* feature of Java. Method overriding means reimplementing a method inherited from the parent class. For method overriding, we have 4 rules:

1) The method in the child class must have the same signature as the method in the parent class.

2) The method in the child class must be at least as accessible as the method in the parent class.

3) The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.

4) If the method returns a value, it must be the same or a subclass of the method in the parent class.

Finally, if the method's access modifier is *private* it cannot be reached by the subclass. So subclass can define a method with the same name as of the parent class without invoking these rules.

In the most upper parent class we have a method called *drive()* and marked as *final* and *private. final* prevents methods from being overridden but this method is *private,* it cannot be seen from subclasses so having same method in subclasses is not forbidden.

In *Car* class *drive()* is implemented again, with *protected* access modifier. And since this method is marked as *private* in the parent class, the current method is not an overridden version.

ElectricCar class reimplements the method *drive()* with a *final* and *public* modifiers. Rule 2 forbids us from reducing accessibility of the methods, not from increasing. So marking the method as *public* is okay. Marking as *final* is okay and prevents the method from overridden any more.

Finally an *ElectricCar* object is created and assigned a *Car* class reference. This is valid as it is explained in **Q5.**

When println() method calls *drive()* method on this *Car* reference, Java will check its object at runtime and will call the appropriate method which is the last overridden version here. **So the code will print "Driving electric car".**

Q8 **The answer is D.**

Java does not allow a class to extend more than one concrete or abstract classes. But **it allows classes to implement multiple interfaces which makes the option D is correct.**

Q9 **The answer is C.**

Three problems are present in this code. First one is that the method *watch()* in the class *Television* is *final* which prevents the methods from being overridden. Second one is that first method has *protected* access modifier while the second has *package-private* access modifier which is forbidden according to the rule 2 of method overriding rules, mentioned in **Q7.** The third one is that the return types of the methods are not related to each other. Since the first method does not return anything, the second one also must not return anything; it must be *void.*

So three changes must be done here: *final* must be removed from first method, *protected* or *public* access modifiers must be provided in the second method and the return type of the second method must be changed to *void.*

Q10 **The answer is C.**

A – This statement is the rule 4, which is explained in **Q7.** So it is true.

B – This statement has the same meaning as the rule 2, which is explained in **Q7.** So this statement is also true.

C – This is not correct since the child method may not throw the exception thrown by the parent method. The child method's only responsibility is

not to throw any new or broader exception than the exception thrown by the parent method which is also the statement of the option D.

D – This statement is the rule 3, which is explained in **Q7** and also in the previous explanation.

Q11 **The answer is C.**

As already explained in **Q7,** methods marked by *final* cannot be overridden in subclasses. But in this code the subclass *Laptop* attempts to override *process()* method which is marked as *final.* **So the code does not compile.**

If we remove *final* modifier from the parent method then the method may be overridden and the output would be 3.

Q12 **The answer is A.**

The parent class has two methods which are overloaded versions of each other. Overloading a method depends on only changing parameter list.

The second method which has no parameters is overridden in HighSchool class with the exception FileNotFoundException which is a subclass of IOException which is thrown by the parent method. This is allowed as explained in **Q7,** rule 3.

Since the code is valid and invokes the overridden method on object *HighSchool* referenced by *School*, it will print 2.

Q13 **The answer is B.**

As explained in **Q2,** interface methods are assumed *public.* And if no other modifier is used, methods are also assumed *abstract.* So *protected* and *private* modifiers cannot be applied to interface methods.

Since interface methods are assumed to be overridden they cannot be prevented from being overridden such as marking as *final* them.

Java has *static* interface methods so that interface methods can have *static* modifier.

Q14 **All options are incorrect.**

Interfaces can have default methods which must be marked with the keyword *default.* These methods must have an implementation. Java includes these methods in interfaces so that even if an interface is implemented by many classes, it may be changed without an issue, without any change requirement in those classes. These default methods are inherited by the classes implementing these interfaces but these classes are not required to reimplement these methods. However they might override the methods. Even interfaces that extends another interface which has a default method, can override the method, or redeclare it as an *abstract* method, or leave it as it is.

The class *Sprint* implements two interfaces which have same method. This will cause a conflict but if the class implements a method with same signature, which means overriding them, the problem goes away.

But we have a main() method which has no parameters. This is valid but it makes this method a regular method rather an entry point for JVM. **So the code compiles but does not run since JVM couldn't find any entry point to the program.**

If we add proper parameters to main() method, the overridden method would be invoked and "Sprinting!" would be printed.

Q15 **The answer is B.**

A – As it is already mentioned in previous question, interfaces can extend another interfaces.

B – Interfaces can only extend another interfaces, but not implement them.

C – Classes can implement multiple interfaces.

D – This statement is not about interfaces but classes can extend another classes.

Q16 **The answer is D.**

In the class *Rocket* the variable inherited from the parent class, *weight,* is hidden by redeclaring it to 2. The second variable named *height* is not a hiding since the parent version is *private* and cannot be reached from *Rocket.* The problem is here that the method *printDetails()* tries to access to parent's *height* variable which is not possible. **So the code does not compile.**

If the parent class's *getHeight()* method was used instead of *height,* then the code would compile and prints 3,5 because polymorphism and overriding rules do not apply to instance variables.

Q17 **The answer is D.**

Java has 3 class structures: class, abstract class and interface.

Classes can only have concrete methods. Abstract classes can have abstract and concrete methods. Interfaces can only have abstract methods if only non-default methods are taken into consideration. **So the answer is abstract class, interface.**

Q18 **The answer is C.**

Abstract classes are defined with the keywords *abstract class.* Abstract methods are defined with the keyword *abstract* and they do not have a body. The abstract classes and the abstract methods cannot be *private* since they must be accessible to be overridden. If *public* modifier is not written they are assumed to be *package-private.* Abstract classes can extend a concrete class and vice versa. Finally, abstract classes cannot be instantiated directly.

Concrete classes that extends an abstract class, must provide implementation for the abstract methods they inherited. The concrete class *RightTriangle* implements the abstract *getDescription()* method it inherited from abstract class *Triangle.* The method is *package-private* in parent class and the subclass makes it *protected* which is broader so that it is allowed.

IsoscelesRightTriangle is an abstract class extending a concrete class. This class overrides *getDescription()* with a broader access modifier again. Then in main() method, an *IsoscelesRightTriangle* object is attempted to be instantiated which is not valid. **So the code does not compile because of g3 line.**

Q19 **The answer is D.**

The concrete class *Saxophone* extends an abstract class and implements an interface which have same methods with a different return types. So these methods are inherited by the *Saxophone* class and they are conflicting. In order for this code to compile, *Saxophone* must override this method in compatible with both methods. But these methods' return types are *Integer* and *Short* which are not related to each other. According to the overriding rule 4, explained in **Q7,** a child method's return type must either be the same as the parent method's return type or the subclass of the parent method's return type. Since none of the *Integer* and *Short* is a subclass of the other **the options A** and **B** are wrong. Finally, we know that *Number* is a not a subclass, but the superclass of *Integer* and *Short.* So it also cannot be applied.

Q20 **The answer is C.**

Classes can implements only interfaces. Concrete and abstract classes cannot be implemented by concrete classes, instead may be extended.

So the answer is A class implements an interface, while a class extends an abstract class.

Q21 **The answer is A.**

Abstract class *Book* has a static variable and two constructors, one of which is no-argument. This is valid that abstract classes can have constructors. Abstract classes cannot be instantiated but the classes extending them will use those constructors.

The concrete class *Encyclopedia* defines the same variable as the parent's which is variable hiding since Java does not support variable overriding. Then a constructor calling super() is implemented. A method *getMaterial()* is also implemented returning the variable of the superclass via *super*. Since the variable is *static,* accessing it via *super* is not recommended but it is legal.

main() method creates a new *Encyclopedia* object and calls *getMaterial()* method on it which returns the parent's *material* field. **So the code will print *papyrus.***

Q22 **The answer is B.**

As already explained in **Q5,** an object can either be referenced with the same type as the object itself, or as the superclass of the object, or as the interface that the object implements without an explicit cast. So the *unknownBunny* must either be of type *Bunny* or of type of superclass of *Bunny.* If it is of type of superclass, then it may not have access to all attributes that exist in the object.

A – If the *unknownBunny* has no access to the same variables and methods that *myBunny* has, this means *unknownBunny* is of a type of superclass as explained above. Then in order to access to all attributes of *Bunny* object it must be cast to

Bunny which is a subclass so that casting must be explicitly. This statement is correct.

B – As explained above *unknownBunny* cannot be a subclass of *Bunny,* it must at least the same as *Bunny.* This statement is incorrect.

C – We know that the type of *myBunny* is *Bunny* so that if the type of *unknownBunny* is *Bunny* then both have access to same methods and variables.

D – As explained in the first paragraph an can be referenced by superclasses and interfaces that object implements.

Q23 **The answer is D.**

An abstract method is a method definition without a body which is written to be overridden in subclasses. So *final* cannot be applied since it prevents methods from being overridden. *private* cannot be applied, too, since the method must be accessible from the child to be overridden. Abstract classes do not a modifier named *default.* Interfaces have *default* modifier but it makes methods regular, not abstract.

Abstract methods of abstract classes can either be *public, protected,* or *package-private.* So the option D is correct.

Q24 **The answer is D.**

As explained in **Q20,** only interfaces can be implemented. And abstract classes can be extended only. But here the concrete class *Mars* attempts to extend an interface while attempting to implement an abstract class which is totally not allowed. **So the does not compile for this reason. The answer is D.**

If we fix the keyword problems above, then the code has no other problems. Since abstract class and the interfaces that are extended and implemented, respectively, have same methods, the concrete class inheriting them must override the methods and it does.

In main() method *getName()* method is invoked on a new *Mars* object which is cast to *Planet*. This is okay since an object can be referenced by its superclass reference. This method will print the result of the overridden method *getName()* which is Mars.

Here we have a new thing; *default* methods of interfaces. If an interface method is marked as *default* this method has a body. And concrete classes implementing this interface are not required to override this method but they might.

Q25 **The answer is B.**

A – A reference to a class cannot be assigned to a subclass without an explicit cast. Event with an explicit cast, if the reference do not contain an object compatible with the cast reference a runtime exception would be thrown out.

B – Since superclass references can contain subclass objects, objects can be referenced superclass types without an explicit cast.

C – Since interfaces cannot be instantiated directly, they must be assigned to classes with an explicit cast to a class that implements that interface.

D – If a class implements an interface, that interface acts as a superclass of it. So that an object of that class can be assigned that interface type without an explicit cast.

Q26 **The answer is B.**

All interface variables are *public, final* and *static* no matter whether they are explicitly marked or not. But variables cannot marked as *abstract* neither in interfaces nor in classes.

Q27 **The answer is C.**

The code has no syntactic error so it compiles. Abstract classes can have static and instance initializers as well as constructors since they will be called when an object extending abstract class is created.

So in main() method a new *BlueCar* object is created. When it is created firstly the superclass is instantiated executing *static initializers, instance initializers* and *constructors,* respectively, which result in 132. Then the current class is begun to be instantiated in same order as the parent which will result in 45. **So the output will be 13245.**

Q28 **The answer is C.**

Overloaded methods must have different parameter lists and same method names. Return types are irrelevant to overloading. Overridden methods must have the same name, same parameter list and covariant return types.

So the only common feature of overloaded and overridden methods is having the same method name.

Q29 **The answer is A.**

The has no syntactic error. when a *SoccerBall* object is created in main() method, that object is cast to *Ball* firstly. This is the superclass of the *SoccerBall* so that it is legal. Then this object is cast again to *Equipment* which is legal, too, since the object implements *Equipment*. So as a result *SoccerBall* object is assigned a reference type *Equipment* interface.

Then in println() method the property *size,* which is inherited from *Ball* abstract class, is attempted to be access by casting this object to *SoccerBall* which is the same type as the object's. **This is legal and the code prints 5, which is the value of the final variable *size*.**

Q30 **The answer is C.**

Java does not allow variable overriding so that defining a variable with same name as a parent variable is hiding variable.

Java allows overriding non-static methods. Static methods can only be hidden in Java.

There is no such a thing called replacing. So that answer is *hiding, hiding.*

Q31 **The answer is B.**

The first abstract class has a method marked as *private.* Since it is *private,* it cannot be seen from subclasses so that defining a method with the same name is not related to this method.

The second abstract class defines a static method with the same name as the method in the parent abstract class but they're not related to each other since the parent method is *private*.

The concrete class attempts to override the method inherited but since the method in parent class is static, it cannot be overridden but hidden. **So the code does not compile because of the line x2.**

Square must mark the method as *static.* If the error is fixed then the *Square* object referenced by *Square* itself would call the hidden method and print 4. If the *Square* object is referenced by *Rectangle* then the method in *Rectangle* class will be called since these methods are hidden and the superclass's reference could not reach the method in the subclass. And the output would be 2.

Q32 **The answer is C.**

Concrete classes cannot define abstract methods. Only interfaces and abstract classes can define abstract methods. **In this code the first class attempts to define an abstract class which will cause a compilation error.**

We can fix this problem by implementing the method as a normal method. But then we have another problem in main() method with casting. A *Rotorcraft* object is attempted to be cast to its subclass *Helicopter* which is not legal. This cast will not cause a compilation error but will throw *ClassCastException* at runtime.

If we create a *Helicopter* object referencing by *Helicopter* the code would compile and print 5 since *fly()* method calls parent's *height* property.

Q33 **The answer is B.**

As already explained in **Q5,** an object can either be referenced with the same type as the object itself, or as the superclass of the object, or as the interface that the object implements without an explicit cast.

But if a reference points to an object that is a child class of its own class, this object can be assigned to reference type of its own class that is a subclass of the current reference type, only with an explicit cast.

So the answer is *superclass, subclass.*

Q34 **The answer is C.**

There is no such a thing named *abstraction* in Java so that **the option B** is wrong.

Interfaces and abstract classes can inherit abstract methods from other interfaces and abstract classes but they are not required to implement them.

Only class structure that is required to implement all of the inherited abstract methods, is *concrete class.*

Q35 **The answer is D.**

Interface methods are assumed to be *public* and *abstract* if no other modifier used. Abstract methods cannot have a body. *fly()* method in interface has a body which is illegal. **(1).**

fly(int) method in the class *Bird* is missing return statement.**(2).**

final modifier on the class *Bird* prevents it from being extended. **(3).**

So this code has 3 compiler errors.

Q36 **The answer is B.**

Interface variables are assumed to be static and abstract classes can have static variables, too.

Interfaces and abstract classes can have abstract and static methods.

But default methods are only available in interfaces.

Q37 **The answer is C.**

Since the class *Performance* inherits two same methods from interfaces *SpeakDialogue* and *SingMonologue* the class is confused about which one to be invoked. **In this way the code does not compile.**

In order for the code to compile we must override the conflicted methods. Then the code does not use this overridden method so that it will print neither 5 nor 7.

Q38 **The answer is A.**

A virtual method is a method in which the specific implementation is not determined until runtime. All non-final, non-static and non-private methods are considered virtual methods, since any of them can be overridden at runtime. If you call a virtual method on an object that overrides a method, you get the overridden method, even if the call to the is on a parent reference or within the parent class.

static methods cannot be overridden but hidden, *private* methods cannot overridden since they cannot be reached, *final* modifier prevents methods from being overridden. **So *protected instance methods* are virtual methods.**

Q39 **The answer is B.**

Interfaces can extend other interfaces. Classes can extend another class. For this matter, only difference here is that interfaces can extend multiple interfaces while classes can only extend one class.

Q40 **The answer is A.**

Java does not allow variable overriding but hiding. So in this code each class hides the variable it inherited. When a hidden variable is attempted to be accessed, the variable that belongs to the reference type used is accessed.

Since in this code, *Math* reference type is used, the variable in *Math* is reached, whose value is 2, although the object the reference points is *InfiniteMath* type. So the code will print 2.0 since the data type of variable is *double.*

Q41 **The answer is D.**

As explained in **Q7 – rules,** the overridden method must be at least as accessible as the parent method. Since the parent method is *protected,* the overridden method must be *protected* or *public.* So **the options A** and **C** is incorrect.

Also as explained in **rule 3,** an overridden method cannot throw a broader exception than its parent method. Since *IOException* is super class of *FileNotFoundException* thrown by the parent class, it cannot be thrown by the overridden method which makes **the option B** incorrect.

Inherited methods can be overridden defining them *final* so the option D is correct.

Q42 **The answer is C.**

setAnimal() method has a parameter of type *Dog.* So the argument to be passed to this method must be in hierarchical relation with *Dog. Husky* is subclass of *Dog* so that a *Husky* object can be passed to this method as well as a *Dog* object since it is the type of *Dog's* itself. *null* pointer can also be passed into the method since it can be assigned to any object.

But *Wolf* has no relationship with *Dog* so that a *Wolf* object cannot be passed into the method.

Q43 **The answer is A.**

Interfaces have *default, static* and *abstract* methods. So that these methods can be applied to interface methods.

Since *abstract* interface methods are assumed to be overridden they cannot be marked as *final. default* and *static* methods are not required to be overridden but they might be so that they cannot be marked as *final.*

Q44 **The answer is A.**

This code has three abstract classes, one of which has a main() method which is valid.

The second abstract class extends the first and defines a method with a same name as the first abstract class's method but since its parameters list is different, this is not considered as overriding. It is a new method.

The last abstract class defines the same method as abstract which is legal. Since all of these are legal the code compiles and prints "Let's start the party!".

Q45 **The answer is D.**

Inherited methods do not have specify any special thing, so **the option B** is incorrect. Overloaded methods must have different parameter lists, return type is not of a concern. Overridden methods must have the exact same parameter list and must have a covariant return type. So none of **the options B** and **C** is true. **The answer is D.**

Q46 **The answer is B.**

If a class didn't define any constructor Java will automatically insert a no-argument constructor. And if this class extends another class Java will insert a call to a no-argument parent constructor via *super()*. If the super class does not have a no-argument constructor (nor a default one), then this will cause a compilation error. So if only the parent class does not have any no-argument constructor the child class must implement at least one constructor calling the parent constructor at the first line.

A – The child class might include a no-argument constructor providing necessary arguments for the parent constructor within its constructor even if the parent constructor does not include a no-argument constructor.

B – As explained above if parent class does not have any no-argument constructor the child must contain at least one constructor definition.

C – If the parent class contains a no-argument constructor, the child might not implement any constructor since Java will automatically insert it.

D – As explained in previous option, the child is not required to implement any constructor if the parent class has a no-argument constructor.

Q47 **The answer is D.**

An object can be referenced by its superclass or by an interface it implements. The attributes of the object is in memory no matter which reference type is assigned to it. But if the reference type is not same as the object type, the reference may not be able to access to all available attributes.

So *the object type* determines which attributes exist in memory, while *the reference type* determines which attributes are accessible by the caller.

There is no feature named *signature* in Java.

Q48 **All options are incorrect.**

The class *Violin* inherits two same methods so that it must override them in order to be able to compile.

As explained in the rule-4 in **Q7,** the return type of the overridden method must be same as the return type of the parent method or a subclass of it. So if we use *Number* it will be incompatible with *Long.* But if we use *Long,* it will be okay with the *Long play()* method of the abstract class and it will be okay with the interface method, too, since it is a subclass of *Number.* So *Long* is the only valid option.

But the overridden method returns a literal number which is assumed to be *int* value which is incompatible with *Long.* **So this code does not compile.**

If we write *return 12L;* or *return 12l;* the code would compile.

Q49 **The answer is B.**

Having *default* methods allow interfaces to be changed without affecting existing classes that implement this interface. Since if a non-default abstract method was implemented then all the existing classes that implement that interface must provide an overridden method for that inherited abstract method. But concrete classes are not required to provide a reimplementation for the *default* interface methods they inherited. **So the best reason to have *default* interface methods is to add backward compatibility to existing interfaces.**

Option A is incorrect since non-default interface methods are also inherited.

While **options C** and **D** are correct they are not the best reasons since *static* methods do already allow to create concrete methods.

Q50 **The answer is C.**

As explained in **rule 3 in Q7,** an overridden method cannot throw a broader exception than its parent method. **Since *IOException,* which is thrown in the subclass *Robot,* is superclass of *EOFException,* which is thrown in the parent class Machine, this code does not compile.**

If we fix this problem, for example by removing the overridden method's exception, then the code would invoke the overridden method because of polymorphism and prints *false.*