Homework 7

Q1)(C)

The code does not compile because of two reasons. The `name` variable is declared **`private`** in the superclass, it is unaccessible in the `Movie` class, Movie class cannot directly access the `name` variable. That is the first reason.

If present, a call to a superclass's constructor must be the first statement in a derived class's constructor. Otherwise, a call to **`super();`** (the no-argument constructor) is inserted automatically by the compiler. Since a default constructor is undefined, compiler can not insert a call to no-argument constructor. We must explicitly invoke another constructor.

Q2)(D)

All members of an interface, variables, methods, inner interfaces, and inner classes, are inherently public because that's the only modifier they can accept. Using other access modifiers results in compilation errors. Interface members can be prefixed with public.

But we should also note that since java 9, you will be able to add private methods and private static method in interfaces.

These private methods will improve code re-usability inside interfaces. For example, if two **`default`** methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it's implementing classes. Private interface method cannot be abstract.

Q3)(C)

The code does not compile because there is duplicate methods. That is to say, there is two methods with the same signature, and this is not allowed. Even if the modifiers and the return type is different, method signatures are the same, so they are duplicate.

Q4)(A)

Inheritance allows objects to access commonly used attributes and methods, by allowing classes to inherit code from superclasses. This improves re-usability.

Inheritance does not guarantee to lead to simpler code. Depending on the code, it can be simpler or more complex.

All objects inherit a set of methods form `java.lang.Object`, but that is not true for primitives.

Q5)(A)

`this` keyword refers to the objects itself. Object is the superclass of all classes in java so it can be used. `Canine` is the object's exact type so it can be used, because `Canine` implements `Pet` interface, it can be used too. Class return type can not be used because it is not a superclass of the `Canine` class.

Q6)(B)

It can be done using interfaces. We can use interfaces to develop our code than other team can implement those interfaces. When they finish writing we can easily integrate integrate other team's code. I should note that this can be probably done with abstract classes too, with we are using abstract classes, and other team implementing these abstract classes, but for general case, interfaces are probably the best option.

Q7)(B)

Car class defines a method named drive, same name with its super counterpart, but `drive` method in the `Automobile` class is marked `private` so we are not actually overriding it and because of that `final` keyword does not affect. Private methods does not participate in polymorphism. Instance methods are resolved in runtime.

`ElectricCar` class overrides drive method which resides in its parent class. So in main method when we are invoking the `drive` method, even though we are invoking it through an reference variable which has a `Car` type, the method from the `ElectricCar` class is invoked because of the polymorphism.

Q8)(D)

Java does not allow multiple inheritance through classes, abstract or not, but it allows multiple inheritance through interfaces. We can implement more than one interfaces with a class but a class cannot extend more than one class.

Q9)(C)

Total of three changes must be performed. Final methods cannot be overridden so final keyword must be deleted. Method that overrides another method cannot have lower visibility, so protected or public access modifier should be added. Last but not least, return types must be covariant return types, Object return type should be changed to void.

Q10)(C)

The return type of the overriding method must be covariant. The access modifier of the method in the child class must be the same or broader than the method in the superclass. A checked exception thrown by a method in the child class must be the same or narrower than the exception thrown by the method in the parent class. It is not required that the thrown exception must be the same with the overridden method.

Q11)(C)

The code does not compile. `process()` method in the parent class is declared **`final`**, but extending class `Laptop` defines a method which overrides (well, tries to override at least) it. **`final`** methods can not be overridden. That causes an compilation error.

Q12)(D)

The code compiles fine. `Highschool` class correctly overrides parent class method, exceptions that the method may throw can be covariant types in overridden methods, which is to say overridden method exception should be the same exception or some broader type exception.

Code prints '2' to the console. Regardless of the type of the variable, because of polymorphism, method invoked is decided at runtime depending on the type of the actual object, which is `HighSchool` in this case.

Q13)(B)

The methods of an interface are implicitly public. When we implement an interface, we must implement all its methods by using the access modifier public. Interfaces in Java 8 can also define static methods. Interface methods cannot be declared final in Java which prevents them from being overridden.

But we should note that, with Java 9 onward, you are also allowed to include private methods in interfaces with some constraints.

With java 8 we can have public static and public default methods. With java 9 we can have private methods with constraints that follows:

1. Private interface method cannot be abstract.
2. Private method can be used only inside interface.
3. Private static method can be used inside other static and non-static interface methods.
4. Private non-static methods cannot be used inside private static methods.

For example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it's implementing classes.

Q14)(C)

A class can implement both interfaces which has duplicate default methods as long as it overrides it because otherwise there is an ambiguity on which method will be invoked.

Q15)(B)

An interface can extend another interface. A class can implement two or more interfaces. A class can certainly extend another class. An interface cannot implement another interface, only a class can implement an interface.

Q16)(D)

The code does not compile. Because subclass tries to access private instance variable of the parent class. Also note that `super.getWeight` return the parent classes' instance variable which is 3. If the height variable wasn't private, `super.height` would return the parent class instance variable which is 5.

Q17)(D)

Excluding default and static methods, an abstract class can contain both abstract and concrete methods, while an interface contains only abstract methods. We should also note, again, with java 9 we can define private concrete methods inside interfaces which has neither default nor static keywords.

Q18)(C)

The code does not compile because of line g3. We are trying to instantiate an abstract class. That causes a compilation error. Other than that code would compile just fine. An abstract class can extend a concrete class and vice versa.

Overriding method can have widening access modifiers. Overriding in this question is an example of a correct overriding.

Q19)(D)

The code does not compile so none of the options is the answer. What we are doing is actually overriding the both inherited methods. But there is no type that would comfort the both overridden methods' return types. Maybe it would be more accurate to say that there is no covariant return type with both of these `play()` methods.

Q20)(C)

A class implements an interface, while a class extends an abstract class. Vice versa is not true.

Q21)(A)

The code compiles and runs without any issue. Prints “papyrus” to the screen. Because `super` keyword is used, the returned `material` variable is the parent classes’ static variable not the one in `Encyclopedia` class.

Q22)(B)

If the unknown type reference variable can point to the `Bunny` (it is said it does) object than it can be cast to the `Bunny` type, and with that our reference variable can have access to the same variables and methods that `myBunny` has access to. So, option A and option C are true. Any interface that `Bunny` class implements or any class or abstract class that `Bunny` class extends could be used as a reference type to a `Bunny` object, making Option D correct. Option B is incorrect since `unknowBunny` reference variable cannot be a type of a subclass of `Bunny` class. Because there is no is-a relationship between superclass and subclass. THere is is-a relationship with subclass vs superclass.

Q23)(D)

`default` keyword is used in interfaces and they are not abstract methods, they are implemented methods. Abstract methods cannot declared `final` because that would prevent them to be overridden which is actually against the reason abstract methods exist, this is also true for `private` keyword. But an abstract method in an abstract class can be declared `protected`. But we should note that methods in interfaces which is abstract by default cannot be declared `protected`.

Q24)(D)

The code does not compile. A class cannot extend an interface, and a class cannot implement a class which is what happening here, but the opposite is true. If `Mars` class would extend `Planet` and implement `Sphere` than the code would print “Mars”.

Q25)(B)

We don’t need explicit casting to assign a reference to a class to a superclass reference, since there is a “is-a” relationship between them. But note that, opposite is not true. For example; not every number is an integer, but every integer **is a** number.

A reference to an interface cannot be assigned to a reference of a class that implements the interface without an explicit cast, because interface does not inherit class, class inherits the interface, again the “is-a” relationship. But with the opposite; a reference to a class that implements an interface can be assigned to an interface reference without an explicit cast because that class **is an** instance of that interface, remember the “**is-a**” relationship.

Q26)(B)

All interface variables are implicitly `public`, `static` and `final`, which is to say they are constants.

Q27)(C)

Superclass initialization happens before the subclass initialization. The static initializers are called when the class is first loaded when there is not even an instance. Instance initializers gets executed before the constructors.

The correct order of initialisation is:

1. Static variable initialisers and static initialisation blocks, in textual order, if the class hasn't been previously initialised.
2. The super() call in the constructor, whether explicit or implicit.
3. Instance variable initialisers and instance initialisation blocks, in textual order.
4. Remaining body of constructor after super().

Q28)(C)

Overloaded methods differ in parameters but they have the same same, overridden methods on the other hand have same parameter list and have the same name. So, overloaded and overridden methods always have the same name.

Q29)(A)

The code prints 5 to the screen. Because `SoccerBall` extends `Ball` and implements Equipments, object reference of `SoccerBall` can be explicitly cast to either one of these types. Although get() method return a reference type of Ball the object it points to is an SoccerBall, so that reference can be safely casted to Equipment.

Q30)(C)

A class that defines an instance variable with the same name as a variable in the parent class is referred to as **hiding** a variable. A class that defines a static method with the same signature as a static method in a parent class is referred to as **hiding** a method. Instance variables and static methods do not participate in polymorphism. Overriding members exhibits polymorphic behavior, not others.

Q31)(B)

The code does not compile because `getEqualSides()` method in `Square` class tries to override its counterpart in the parent class `Rectangle` .Instance methods cannot override static methods and static methods cannot hide instance methods.

Q32)()

The code does not compile. `Rotorcraft` class has an abstract method declared in it. A class with an abstract method declaration must be an abstract class. Even if we correct that, code would not compile again because in `main` method `Rotorcraft` is being tried to instantiated. Abstract classes can not be instantiated.

Q33)(B)

A class may be assigned to an superclass reference without an explicit cast, because as we said before there is a “is-a” relationship. A class instance **is an** instance of its superclass too. But the opposite is not true.

Q34)(C)

A concrete class is the first non-abstract subclass that is required to implement all of the inherited abstract methods. Concrete classes must also implement inherited interface abstract interface methods. Basically, they have to implement all the inherited abstract methods.

Q35)(D)

There is a total of three errors in this code. First, interface `CanFly` defines a method with body but interface methods unless declared default are implicitly **`abstract`** and abstract methods cannot have bodies. Second, `Bird` class defines a method with return type **`int`**, but it does not return any value. Last but not least, `Eagle` tries to extend `Bird` class which is declared **`final`**, final classes can not be extended.

Q36)(B)

Only interfaces can contain methods declared with default keyword, default methods are implemented methods which can exhibit polymorphic behavior when the interface is inherited.Interfaces' variables are static by default, abstract classes can define static variables explicitly so they can both have static variables. Interfaces and abstract classes can both contain static methods, and also they can both contain abstract methods which is actually default in interfaces.

Q37)(D)

The code does not compile. The Performance class inherits two defaults methods with same signature. Performance class must override this method or the code won't compile. `talk()` method defined in the Performance class does not override inherited `talk()` methods because their signatures are different. When we don't override this method there will be ambiguity about which polymorphic `talk()` method is going to be invoked? The one from the `SpeakDialogue` interface, or the one from the `SingMonologue` interface?

Q38)(A)

A virtual method is a type of method where the actual method calls depends on the runtime type of the underlying object, which is to say that virtual methods are methods that exhibits polymorphic behavior, and on the contrary a non-virtual method is a type of method where the actual method called depends on the reference type of the object at the point of method invocation. Methods in java are virtual by default, we don't need to explicitly specify that a method is virtual like in C++. But method declared with **`static`**, **`final`** or with **`final`** keywords can not be overridden, therefore they do not exhibit polymorphic behaviour, therefore they are not considered as virtual methods.

Q39)(B)

Interfaces can extend another interface, and classes can extend other classes, classes can also implement interfaces but interfaces cannot implement neither class nor another interface.

Q40)(A)

The code compiles and runs without any issue. Key point here is that, variables in java does not exhibit polymorphic behaviour like methods can do. There is no overriding variables even if it seems like we are overriding inherited variables, but actually we are not. We are just hiding the inherited variable from the parent class. Like **`static`** methods, reference type decides which variable is going to be used, not the actual object type at run-time. So, in this question, even if the actual object type is `InfiniteMath`, variable is resolved not at the runtime, but in the compile-time according to the type of the reference variable of that object which is `Math` type in this case.

Overriding basically supports late binding. Therefore, it's decided at run time which method will be called. It is for non-static methods.

Hiding is for all other members (static methods, instance members, static members). It is based on the early binding. More clearly, the method or member to be called or used is decided during compile time.

Q41)(D)

Overridden methods' visibility can not be reduced in subclasses when overriding. That makes option A and Option C invalid overrides in subclass. Option B is invalid override because overridden methods cannot be declared with a broader exception type in subclass.

Less restrictive access modifiers, more specific exception type, removed checked exception, making method final, and more restrictive return types are allowed in overridden methods.

Q42)(D)

Although the instance variable's reference type is `Canine`, `setAnimal()` method only accepts a more specific type of animal which is `Dog`. But `Wolf` is not compatible with `Dog`. But we should note that we can assign `Wolf` to instance variable `Canine`, we just can't pass it to the `setAnimal()` method because it requires a more specific type. We can also pass **`null`** as an argument to the method because **`null`** can be assigned to any reference type variables.

Q43)(A)

An interface method can be default, static or abstract even though **`abstract`** keyword is redundant because interface methods are **`abstract`** by default. But an interface method can not be defined as **`final`** because it would prevent it from being implemented.

Q44)(A)

The code compiles and runs without any issue. Note that `Room` class does not override inherited `getSpace()` method, it overloads it.This is an abstract overloading method. A class that extends this `Room` class would have to implement both `getSpace()` methods

Q45)(D)

It is true that overloaded methods must have a different list of parameters, but neither overridden nor overloaded methods must have the exact same type. Overridden methods return types must be covariant types. There is no compability rule when overloading methods they can vary.

Q46)(B)

If a parent class does not include a no-argument constructor nor a default one inserted by the compiler, a child class must contain at least one constructor definition, and invoke super constructor in it by itself because compiler cannot insert the default constructor to the subclass (because there is no default no-argument constructor in the parent class).

Q47)(D)

The object type determines which attributes exist in memory, while the reference type determines which attributes are accessible by the caller. Reference type tells us as what type can we use the object that this reference point to as.

Q48)(A)

Since we are overriding both inherited method, the return type type must be compatible (covariant) for both of the inherited `play()` methods. `Integer` is okay for implementing the interface `play()` method since it is covariant with `Number` class but `Long` and `Integer` are not covariant so it doesn't comply with `Long`.

Q49)(B)

Interfaces can have default methods with implementation in Java 8 on later. Interfaces can have static methods as well, similar to static methods in classes. Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

Q50)(C)

The code does not compile, because in overridden methods exceptions must be covariant types too. Overridden method's thrown exception cannot be broader than the method in the parent class.