**Q 1-1.** Which of the following points should you incorporate in your application design?

1. Create related classes in a single package.
2. Don’t make derived classes overload methods from their base class.
3. Expose the functionality of your classes using public methods.
4. Create private methods to work as helper methods for the public methods.

Explanation: Option (a) is correct. A package enables you to create a namespace to

group related classes and interfaces together.

Option (b) is incorrect. A base class overloads its base class method, as required.

Making derived classes overload their base class methods doesn’t make it an incorrect

or inefficient design.

Options (c) and (d) are also correct. The functionality of your classes should be

exposed using the public methods. The private methods are called within the class in

which they’re defined. They usually work as helper methods.

**Q 1-2.** What is the output of the following code?

class Wood {

public Wood() {

System.out.println("Wood");

}

{

System.out.println("Wood:init");

}

}

class Teak extends Wood {

{

System.out.println("Teak:init");

}

public Teak() {

System.out.println("Teak");

}

public static void main(String args[]) {

new Teak();

}

}

1. Wood:init

Wood

Teak:init

Teak

1. Wood

Wood:init

Teak:init

Teak

1. Wood:init

Teak:init

Wood

Teak

1. Wood

Wood:init

Teak

Teak:init

**A 1-2. a**

Explanation: When a class is compiled, the contents of its initializer block are added

to its constructor, just before its own contents. For example, here’s the decompiled

code for class Wood. As you can see, the contents of its initializer block are added to

its constructor:

class Wood{

public Wood(){

System.out.println("Wood:init");

System.out.println("Wood");

}

}

**=================================================================================**

**Q 1-3.** Examine the following code and select the answer options that are correct

individually.

class Machine {

void start() throws Exception {

System.out.println("start machine");

}

}

class Laptop {

void start() {

System.out.println("Start Laptop");

}

void start(int ms) {

System.out.println("Start Laptop:" + ms);

}

}

1. Class Laptop overloads method start().
2. Class Laptop overrides method start().
3. Class Machine overrides method start().
4. Class Machine won’t compile.
5. Class Laptop won’t compile.

**A 1-3**. a

Explanation: Class Laptop correctly overloads the method start() by defining a different parameter list. Options (b) and (c) are incorrect because classes Laptop and Machine are unrelated.

A derived class can override its base class method. Method start() qualifies as a valid overridden method in class Laptop, if Laptop extends class Machine. It’s acceptable for an overriding method to not throw any checked exception, even if the base class method is throwing a checked exception.

Options (d) and (e) are incorrect because both classes will compile successfully.

**Q 1-4.** Given that classes Class1 and Class2 exist in separate packages and source

code files, examine the code and select the correct options.

package pack1;

public class Class1 {

protected String name = "Base";

}

package pack2;

import pack1.\*;

class Class2 extends Class1{

Class2() {

Class1 cls1 = new Class1(); //line 1

name = "Derived"; //line 2

System.out.println(cls1.name); //line 3

}

}

1. Class2 can extend Class1 but it can’t access the name variable on line 2.
2. Class2 can’t access the name variable on line 3.
3. Class2 can’t access Class1 on line 1.
4. Class2 won’t compile.
5. Line 3 will print Base.
6. Line 3 will print Derived.

**A 1-4. b, d**

Explanation: A derived class can access a protected member of its base class, across

packages, directly. But if the base and derived classes are in separate packages, then

you can’t access protected members of the base class by using reference variables of

class Base in a derived class. So, Class2 doesn’t compile.

Options (e) and (f) are incorrect because Class2 won’t compile.

**Q 1-5.** Select the correct option.

1. The declaration of private variables to store the state of an object is encouraged.
2. The protected members of a class aren’t accessible outside the package in which the class is defined.
3. The public members of a class that’s defined with default access can be accessed outside the package.
4. If you change the signature or implementation of a private method, other classes that use this method cease to compile.

**A 1-5. a**

Explanation:

Option (b) is incorrect because the protected members of a class are accessible by the derived classes, outside the package in which the class is defined.

Option (c) is incorrect because a class with default access isn’t visible outside the

package within which it’s defined. If the class isn’t visible itself, it doesn’t matter

whether its members are accessible or not.

Option (d) is incorrect because a private method can’t be used outside the class in

which it’s defined.

**Q 1-6.** Given the following code

interface Scavenger{}

class Bird{}

class Parrot extends Bird{}

class Vulture extends Bird implements Scavenger{}

class BirdSanctuary {

public static void main(String args[]) {

Bird bird = new Bird();

Parrot parrot = new Parrot();

Vulture vulture = new Vulture();

//INSERT CODE HERE

}

}

In which of the following options will the code, when inserted at

//INSERT CODE HERE,

throw a ClassCastException?

1. Vulture vulture2 = (Vulture)parrot;
2. Parrot parrot2 = (Parrot)bird;
3. Scavenger sc = (Scavenger)vulture;
4. Scavenger sc2 = (Scavenger)bird;

**A 1-6. b, d**

Explanation: ClassCastException is thrown at runtime. So the options that don’t fail to compile are eligible to be considered for the following question: Will they throw a ClassCastException?

Option (a) is incorrect because it fails to compile.

Option (b) is correct because classes Bird and Parrot are in the same hierarchy tree, so an object of base class Bird can be explicitly casted to its derived class Parrot at compilation. But the JVM can determine the type of the objects at runtime. Because an object of a derived class can’t refer to an object of its base class, this line throws a ClassCastException at runtime.

Option (c) is incorrect because class Vulture implements the interface Scavenger, so this code will also execute without the explicit cast.

Option (d) is correct. An instance of a nonfinal class can be casted to any interface type using an explicit cast during the compilation phase. But the exact object types are validated during runtime and a ClassCastException is thrown if the object’s class doesn’t implement that interface. Class Bird doesn’t implement the interface Scavenger and so this code fails during runtime, throwing a ClassCastException.

**Q 1-7.** Assuming that all of the following classes are defined in separate source code

files, select the incorrect statements.

package solarfamily;

public class Sun {

public Sun() {}

}

package stars;

public class Sun {

public Sun() {}

}

package skyies;

import stars.Sun; // line1

import solarfamily.Sun; // line2

class Sky {

Sun sun = new Sun(); // line 3

}

1. Code compilation fails at line 1.
2. Code compilation fails at line 2.
3. Code compilation fails at line 3.
4. The code compiles successfully and class Sky creates an object of class Sun from the stars package.
5. The code compiles successfully and class Sky creates an object of class Sun from the solarfamily package.

**A 1-7. b**

Explanation: Class Sky fails with the following error message:

Sky.java:3: error: stars.Sun is already defined in a single-type import

import solarfamily.Sun;

^

1 error

**Q 1-8.** Select the correct options.

class Color {

String name;

Color(String name) {this.name = name;}

public String toString() {return name;}

public boolean equals(Object obj) {

return (obj.toString().equals(name));

}

}

1. Class Color overrides method toString() correctly.
2. Class Color overrides method equals() correctly.
3. Class Color fails to compile.
4. Class Color throws an exception at runtime.
5. None of the above.

**A 1-8. a**

Explanation: Class Color overrides method toString() correctly, but not method equals(). According to the contract of method equals(), for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true—this rule states that two objects should be comparable to each other in the same way. Class Color doesn’t follow this rule. Here’s the proof:

class TestColor {

public static void main(String args[]) {

Color color = new Color("red");

String string = "red";

System.out.println(color.equals(string)); // prints true

System.out.println(string.equals(color)); // prints false

}

}

**Q 1-9.** Given the following code

class Book {

String isbn;

Book(String isbn) {this.isbn = isbn;}

public int hashCode() {

return 87536;

}

}

Select the correct option.

1. Objects of the class Book can never be used as keys because the corresponding

objects wouldn’t be retrievable.

1. Method hashCode() is inefficient.
2. Class Book will not compile.
3. Though objects of class Book are used as keys, they will throw an exception

when the corresponding values are retrieved.

**A 1-9. b**

Explanation: Method hashCode() returns the same hash code for all the objects of this class. This essentially makes all the values be stored in the same bucket if objects of the preceding classes are used as keys in class HashMap (or similar classes that use hashing), and reduces it to a linked list, drastically reducing its efficiency.

Option (a) in incorrect. Book instances can be used to retrieve corresponding key values but only in limited cases—when you use the same keys (instances) to store and retrieve values. Even though hashCode() will return the same value for different Book instances, equals() will always compare the reference variables and not their values, returning false.

**Q 1-10.** What is the output of the following code?

class Wood {

String wood = "Wood";

public Wood() {

wood = "Wood";

}

{

wood = "init:Wood";

}

}

class Teak extends Wood {

String teak;

{

teak = "init:Teak";

}

public Teak() {

teak = "Teak";

}

public static void main(String args[]) {

Teak teak = new Teak();

System.out.println(teak.wood);

System.out.println(teak.teak);

}

}

1. init:Wood

init:Teak

1. init:Wood

Teak

1. Wood

init:Teak

1. Wood

Teak

**----------------------------------------------------------------------------------------------------------------------------------------------**

**A 1-10. d**

Explanation: When a class is compiled, the contents of its initializer block are added to its constructor just before its own contents. For example, here’s the decompiled code for class Wood. As you can see, the contents of its initializer block are added to its constructor:

class Wood{

public Wood() {

wood = "Wood"; // initial initialization

wood = "init:wood"; // re-assignment by the initializer block

wood = "Wood"; // re-assignment by the constructor

}

String wood;

}

**Q 1-11.** Given the following code

class Cloth {}

class Shirt extends Cloth implements Resizable{}

class Shorts extends Cloth {}

interface Resizable {}

class Factory {

public static void main(String sr[]) {

Shirt s = new Shirt();

//INSERT CODE HERE

System.out.println(res);

}

}

Which options will print true?

1. boolean res = new Cloth() instanceof Shirt;
2. boolean res = new Shirt() instanceof Resizable;
3. boolean res = null instanceof Factory;
4. Cloth cloth = new Cloth();

Shirt shirt = new Shirt();

boolean res = shirt instanceof cloth;

**A 1-11. b**

Explanation:

Option (a) prints false.

Option (c) prints false. It doesn’t fail to compile because null is a valid literal

value that can be used for objects.

Option (d) fails to compile. The instanceof operator must be followed by the

name of an interface, class, or enum.