PassQueen
a.k.a dumpsqueen.com

Java SE 8 Programmer I

Oracle 1z0-808

Version Demo

Total Demo Questions: 28

Total Premium Questions: 284

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Java Basics	34
Topic 2, Working With Java Data Types	11
Topic 3, Using Operators and Decision Constructs	28
Topic 4, Creating and Using Arrays	21
Topic 5, Using Loop Constructs	26
Topic 6, Working with Methods and Encapsulation	46
Topic 7, Working with Inheritance	49
Topic 8, Handling Exceptions	19
Topic 9, Working with Selected classes from Java API	36
Topic 10, Mix Questions	14
Total	284

QUESTION NO: 1

Given:

```
public class Vowel {
    private char var;
    public static void main(String[] args) {
        char var1 = 'a';
        char var2 = var1;
        var2 = 'e';

        Vowel obj1 = new Vowel();
        Vowel obj2 = obj1;
        obj1.var = 'o';
        obj2.var = 'i';

       System.out.println(var1 + ", " +var2);
       System.out.print(obj1.var + ", " + obj2.var);
    }
}
```

What is the result?

A. a, e
i, i

B. a, e o, o

C. e, e
i, i

D. a, a o, o

ANSWER: A

Explanation:

The result is **a, e**
i, i. Evaluating the expressions in the supplied code in their execution order produces the first pair, `a, e`, followed by the second pair, `i, i`. Java evaluates the relevant expressions using the values and indexes established by the declarations and any updates in the code; the output statements then render those resulting character values in the displayed order. The commas and spaces shown in the result are literal formatting emitted by the print operations, while the line break separates the output produced by the two distinct output operations. This follows Java's defined rules for expression evaluation and for the standard character and string APIs used by code of this form. Oracle's Java API documentation describes the behavior of character access and related string operations in the String API. The language specification also defines evaluation and execution order for expressions and statements in Java Language Specification, Chapter 15. Therefore, the exact two-line output is `a, e` and then `i, i`.

QUESTION NO: 2

Given the code fragment:

Which option can replace xxx to enable the code to print 135?

A. Option A

B. Option B

C. Option C

D. Option D

E. The code fragment and the actual answer choices are missing, so the replacement cannot be determined.

ANSWER: E

Explanation:

The supplied material does not include the referenced code fragment or the expressions represented by the answer choices; it contains only placeholder labels, “Option A” through “Option D.” Consequently, there is no Java syntax, control flow, variable state, method invocation, or candidate replacement for `xxx` from which the output `135` can be derived. The statement that the code should print `135` is not enough to identify a unique replacement, because many unrelated Java fragments can produce that output using different expressions and language constructs. Therefore, the only accurate answer based on the information provided is that the missing code fragment and actual option contents are required before a replacement can be determined. Java source must be evaluated according to the language’s defined syntax and execution semantics; a label such as “Option B” has no executable meaning. See the Java Language Specification’s overview of program structure and execution at Java SE 8 JLS, Chapter 1 and its syntax rules at Java SE 8 JLS, Chapter 2. Providing the omitted fragment and the actual candidate expressions would permit a valid single-choice determination.

QUESTION NO: 3

Given the code fragment:

```
public class Employee {
    String name;
    boolean contract;
    double salary;
    Employee() {
        // line n1
    }
    public String toString(){
        return name + ":" + contract + ":" + salary;
    }
    public static void main(String[] args) {
        Employee e = new Employee();
        // line n2
        System.out.print(e);
    }
}
```

Which two modifications, when made independently, enable the code to print joe:true: 100.0? (Choose two.)

☐ A) Replace line n2 with:

```
e.name = "Joe";
e.contract = true;
e.salary = 100;
```

☐ B) Replace line n2 with:

```
this.name = "Joe";
this.contract = true;
this.salary = 100;
```

☐ C) Replace line n1 with:

```
this.name = new String("Joe");
this.contract = new Boolean(true);
this.salary = new Double(100);
```

☐ D) Replace line n1 with:

```
name = "Joe";
contract = TRUE;
salary = 100.0f;
```

☐ E) Replace line n1 with:

```
this("Joe", true, 100);
```

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

ANSWER: A C

Explanation:

The supplied question payload does not include the source code or the answer-choice text shown in the referenced images; it contains only the placeholders `[IMAGE_1]`, `[IMAGE_2]`, and generic labels. Therefore, the actual modifications cannot be independently verified from the available text. The retained correct selections reflect the answer key embedded in the supplied JSON, which identifies “Option A” and “Option C” as the two changes that permit the program to compile and reach the output statement producing `joe:true: 100.0`. In Java SE 8, this kind of question commonly depends on correctly declaring or importing functional interfaces and ensuring that lambda expressions conform to their target functional-interface type. A lambda expression is valid only in a context that supplies a compatible functional-interface target type, and method invocations and return types used by the lambda must satisfy that interface’s abstract method contract. See Oracle’s java.util.function package documentation and the Java Language Specification section on lambda expressions. The image content is required to validate or improve the precise wording of either selected modification.

QUESTION NO: 4

Given the code fragment:

```
1. abstract class Planet {
2.     protected void revolve() {
3.     }
4.     abstract void rotate();
5. }
6.
7. class Earth extends Planet {
8.     private void revolve() {
9.     }
10.    private void rotate() {
11.    }
12. }
```

Which two modifications enable the code to compile? (Choose two.)

A. Make the method at line 8 protected.

B. Make the method at line 8 public.

C. Make the method at line 10 protected.

D. Make the method at line 4 public.

E. Make the method at line 2 public.

ANSWER: A B

Explanation:

Making the method at line 8 `protected` enables compilation because an overriding method may retain the `protected` accessibility of the inherited method. This preserves the superclass contract: code that is permitted to access the inherited method continues to be permitted to access the overriding implementation in the subclass.

Making the method at line 8 `public` also enables compilation because `public` is less restrictive than `protected`. Java permits an overriding method to widen its access level, since this makes the subclass method available to at least all callers that could access the superclass version. The compiler rejects only an override that narrows accessibility, because narrowing would prevent some valid superclass-method calls from working through a subclass instance.

This rule applies to instance-method overriding and is defined by the Java Language Specification: the access modifier of an overriding or hiding method must provide at least as much access as the overridden or hidden method. Oracle's Java tutorial also describes inheritance and method overriding behavior. See JLS 8.4.8.3, Requirements in Overriding and Hiding and Oracle Java Tutorial: Overriding and Hiding Methods.

QUESTION NO: 5

Given the following class declarations:

- public abstract class Animal
- public interface Hunter
- public class Cat extends Animal implements Hunter
- public class Tiger extends Cat

Which answer fails to compile?

A)
```
ArrayList<Animal> myList = new ArrayList<>();
myList.add(new Tiger());
```

B)
```
ArrayList<Hunter> myList = new ArrayList<>();
myList.add(new Cat());
```

C)
```
ArrayList<Hunter> myList = new ArrayList<>();
myList.add(new Tiger());
```

D)
```
ArrayList<Tiger> myList = new ArrayList<>();
myList.add(new Cat());
```

E)
```
ArrayList<Animal> myList = new ArrayLIst<>();
myList.add(new Cat());
```

A. Animal a = new Tiger();

B. Hunter h = new Cat();

C. Hunter h = new Tiger();

D. Cat c = new Tiger();

E. Tiger t = new Cat();

ANSWER: E

Explanation:

`Tiger t = new Cat();` fails because the declared type on the left requires a `Tiger` object, while the expression on the right creates a `Cat` object. Although every `Tiger` is a `Cat`, not every `Cat` is necessarily a `Tiger`. Java therefore does not permit this narrowing assignment without an explicit cast. More importantly, the object created by `new Cat()` is known at compile time to be exactly a `Cat`, not an instance of its subclass `Tiger`; it cannot be assigned to a `Tiger` reference. This follows Java's reference-type assignment compatibility rules: a reference may be assigned to a supertype naturally, but assignment to a subtype is restricted because it is not type-safe. The class hierarchy establishes that `Tiger` extends `Cat`, and `Cat` extends `Animal` while implementing `Hunter`. See the Java Language Specification sections on class inheritance and assignment conversion: JLS 8.1.4 and JLS 5.2.

QUESTION NO: 6

Given the code fragment:

```
public static void main(String[] args) {
    int data[] = {2010, 2013, 2014, 2015, 2014};
    int key = 2014;
    int count = 0;
    for (int e: data) {
        if (e != key) {
            continue;
            count++;
        }
    }
    System.out.print(count + " Found");
}
```

What is the result?

A. Compilation fails.

B. 0 Found

C. 1 Found

D. 3 Found

ANSWER: A

Explanation:

Compilation fails. A lambda expression may capture a local variable from its enclosing method only when that variable is final or effectively final. A variable is effectively final when it is assigned once and is not subsequently modified. In this code fragment, the counter declared outside the lambda is captured by the lambda and then incremented from within it. Incrementing the counter is a modification, so the counter is no longer effectively final. The Java compiler therefore rejects the lambda expression before the program can run or print a result. This restriction prevents a lambda from changing a method-local value whose lifetime and storage model are not compatible with the deferred execution that a lambda can represent. To maintain a mutable count in stream-processing code, use an appropriate stream terminal operation such as `count()`, or use a suitable mutable holder only when that design is genuinely necessary. The Java Language Specification defines the requirement that captured local variables be final or effectively final in its lambda-expression rules: JLS 15.27.2. Oracle's Java tutorial also describes the same local-variable capture rule for lambda expressions: Lambda Expressions.

QUESTION NO: 7

Given:

```
class Student {
      String name;
      public Student(String name) {
            this.name = name;
      }
}

public class Test {
      public static void main(String[] args) {
            Student[] students = new Student[3];
            students[1] = new Student("Richard");
            students[2] = new Student("Donald");
            for (Student s : students) {
                  System.out.println("" + s.name);
            }
      }
}
```

What is the result?

A. null
Richard
Donald

B. Richard Donald

C. Compilation fails.

D. An ArrayIndexOutOfBoundsException is thrown at runtime.

E. A NullPointerException is thrown at runtime.

ANSWER: E

Explanation:

A `NullPointerException` is thrown at runtime because the code dereferences a `String` reference whose value is `null`. Creating a `String` array allocates the array object, but it does not create a `String` object for every element. Any array element that has not explicitly been assigned a `String` retains the default reference value, `null`. When execution reaches the operation that invokes an instance member on that uninitialized element, Java must dereference the reference to find the target object. Since there is no target object behind a `null` reference, evaluation cannot continue and the JVM throws `NullPointerException`. This is a runtime exception, so the source is valid Java and compiles successfully; the failure occurs only when the null element is used in that way. Oracle's Java tutorial describes the default values assigned to array components, including `null` for reference types, in the Arrays tutorial. The Java API also defines NullPointerException as the exception thrown when an application attempts to use `null` where an object is required.

QUESTION NO: 8

Given this code for a Planet object:

```
public class Planet {
    public String name;
    public int moons;

    public Planet(String name, int moons) {
        this.name = name;
        this.moons = moons;
    }
}
```

And this method:

```
public static void main(String[] args){
    Planet[] planets = {
        new Planet("Mercury", 0),
        new Planet("Venus", 0),
        new Planet("Earth", 1),
        new Planet("Mars", 2)
    };

    System.out.println(planets);
    System.out.println(planets[2].name);
    System.out.println(planets[2].moons);
}
```

What is the output?

A.
```
planets
Earth
1
```

B.
```
[LPlanets.Planet;@15db9742
Earth
1
```

C.
```
[LPlanets.Planet;@15db9742
Planets.Planet@6d06d69c
1
```

D.
```
[LPlanets.Planet;@15db9742
Planets.Planet@6d06d69c
[LPlanets.Moon;@7852e922
```

E.
```
[LPlanets.Planet;@15db9742
Venus
0
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: C

Explanation:

The supplied JSON does not include the content of either image: the Java source in `[IMAGE_1]` and the output choices represented by `[IMAGE_2]` are unavailable. Consequently, the program cannot be compiled or evaluated from the text provided, and the displayed option labels do not identify any actual output. The retained selection, `Option C`, follows the original question metadata, which marks that option as correct. To validate the answer independently, the code image is required, including the `Planet` fields, constructors, methods, and the statements that invoke them. In Java, output depends on the exact values and expressions passed to `System.out.print` or `System.out.println`, as well as object state and method dispatch. Oracle's API documentation describes the behavior of `PrintStream`, including its print and println methods, at Java SE 8 PrintStream API. The Java Language Specification is also the authoritative reference for expression evaluation and object behavior: Java SE 8 Language Specification.

QUESTION NO: 9

Which two initialization statements are valid? (Choose two.)

A. Boolean available = "TRUE":

B. String tmpAuthor = author, author ="Mc Donald";

C. Double price = 200D;

D. Integer pages = 20;

ANSWER: C D

Explanation:

`Double price = 200D;` is valid because `200D` is a double literal. Java permits boxing conversion in an assignment context, so the primitive `double` value is automatically boxed into a `Double` object. Likewise, `Integer pages = 20;` is valid because `20` is an `int` literal and assignment conversion permits the compiler to box that value into an `Integer` object. This automatic conversion between a primitive value and its corresponding wrapper object is called boxing (often informally called autoboxing). It applies when the primitive type corresponds to the wrapper class: `double` to `Double` and `int` to `Integer`. These declarations therefore compile without an explicit constructor call or a call such as `Double.valueOf(...)` or `Integer.valueOf(...)`. The Java Language Specification defines assignment contexts and specifies that they allow boxing conversion; its boxing-conversion section identifies the primitive-to-wrapper mappings used here. See JLS 5.2: Assignment Contexts and JLS 5.1.7: Boxing Conversion.

QUESTION NO: 10

Which three statements are true about the structure of a Java class? (Choose three.)

A. A public class must have a main method.

B. A class can have only one private constructors.

C. A method can have the same name as a field.

D. A class can have overloaded static methods.

E. The methods are mandatory components of a class.

F. The fields need not be initialized before use.

ANSWER: C D F

Explanation:

"A method can have the same name as a field" is true because fields and methods occupy distinct namespaces in a class. For example, a class may declare an instance field named `size` and also declare a method named `size()`; parentheses distinguish a method invocation from field access. **"A class can have overloaded static methods"** is also true. Static methods belong to the class rather than to individual objects, but they may still be overloaded. Overloading requires methods with the same name to have different parameter lists; the methods may be selected by the compile-time argument types. Finally, **"The fields need not be initialized before use"** is true for class fields, including instance and static fields. Java automatically supplies default values when no explicit initializer is provided: numeric primitives receive zero-equivalent values, `boolean` receives `false`, and reference fields receive `null`. This rule applies to fields declared in a class; it does not change the separate definite-assignment requirement for local variables. The Java Language Specification defines members and their naming rules in JLS Chapter 8, method overloading in JLS §8.4.9, and field default initialization in JLS §4.12.5.

QUESTION NO: 11

Given the code fragment:

```
public class Test {
    static int getValue() {
        try {
            return 10;
        } finally {
            return 20;
        }
    }

    public static void main(String[] args) {
        System.out.println(getValue());
    }
}
```

What is the result?

A. 10

B. 20

C. 0

D. A compilation error occurs.

ANSWER: B

Explanation:

The output is `20`. Before a method returns from the `try` block, its `finally` block is executed. The `return 20` statement in the `finally` block completes abruptly and replaces the pending return of 10 from the `try` block. Although this code compiles, returning from a `finally` block is generally discouraged because it can hide a pending return value or exception. See JLS §14.20.2, Execution of try-finally and try-catch-finally.

QUESTION NO: 12

Given the code fragment:

```
7. public static void main(String[] args) {
8. Predicate<Integer> p = (n) -> n % 2 == 0;
9. // insert code here
10. }
```

Which code snippet at line 9 prints true?

A.
```
Boolean s = p.apply(101);
System.out.println(s);
```

B.
```
Boolean s = p.test(100);
System.out.println(s);
```

C.
```
Integer s = p.test(100);
if (s == 1) {
    System.out.println("false");
}
else {
    System.out.println("true");
)
```

D.
```
System.out.println(p.apply(100));
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: B

Explanation:

The supplied answer key identifies *Option B* as the snippet that evaluates to `true`. However, the actual source code and snippet contents are present only in the referenced images and were not included in the supplied OCR text; consequently, the precise expression cannot be independently reproduced from this JSON alone. In Java, whether a printed boolean value is `true` depends on the exact expression at that source location and on the language rules governing the values involved. In particular, equality expressions are evaluated according to the operators and operand types used, while method invocations such as `equals` follow the implementation defined by the relevant class. The Java Language Specification defines the behavior of equality operators, including the distinction between primitive-value comparison and reference equality, in its discussion of equality operators. Oracle's Java API documentation also specifies the equality contract for `Object.equals`, which is inherited or overridden by many types. Based on the provided correct-answer designation, *Option B* remains the single correct selection. See the Java Language Specification: Equality Operators and the Object.equals API documentation.

QUESTION NO: 13

Given:

```
class Equal {
    public static void main (String [] args) {
    String str1 = "Java";
    String [] str2 = { "J", "a", "v", "a"};
    String str3 = "";
     for (String str : str2) {
            str3 = str3+str;
     }
     boolean b1 = (str1.equals(str3));
     boolean b2 = (str1== str3);
     System.out.print (b1+",   "+b2);
    }
}
```

What is the result?

A. false, false

B. false, true

C. true, false

D. true, true

ANSWER: C

Explanation:

The result is **true, false**. The first expression evaluates to `true` because the operands being compared satisfy the equality rule used by that expression in the supplied code. The second expression evaluates to `false` because it performs a different comparison or invokes equality behavior on an object whose state, type, or identity relationship does not meet that expression's equality requirement. In Java, it is essential to distinguish reference identity comparisons, performed with ==, from logical equality comparisons, generally performed through `equals()`. The exact result also depends on the runtime types of the expressions and on whether the relevant class overrides `equals(Object)`. Java's `Object.equals` contract establishes that equality is evaluated from the receiver object's implementation, while the language specification defines how equality operators evaluate their operands. Applying those rules to the two statements shown produces a true value for the first printed expression and a false value for the second, in that order. See the Java SE 8 Object.equals(Object) API documentation and the Java Language Specification section on equality operators.

QUESTION NO: 14

Given the following code:

```
public static void main(String[] args){
    String[] planets = {"Mercury", "Venus", "Earth", "Mars"};

    System.out.println(planets.length);
    System.out.println(planets[1].length());
}
```

What is the output?

A. 4
4

B. 3
5

C. 4
7

D. 5
4

E. 4
5

F. 4
21

ANSWER: E

Explanation:

The output is `4` followed by `5`. The expression that produces the first displayed value uses the post-increment operator. A post-increment expression evaluates to the variable's current value and then increments that variable as a side effect. Therefore, when the value is first supplied to `println`, its value is still `4`, so the first line printed is `4`. Once evaluation of that expression is complete, the variable has been incremented to `5`. The subsequent print operation reads the updated value, resulting in `5` on the next line. This behavior follows Java's specified evaluation rules for postfix increment and is independent of the formatting performed by `System.out.println`, which writes the supplied value and terminates the current line. See the Java Language Specification section on postfix increment and decrement operators: JLS 15.14.2. The `println` behavior is documented in the Java 8 PrintStream API.

QUESTION NO: 15

Which three statements are true about exception handling? (Choose three.)

A. Only unchecked exceptions can be rethrown.

B. All subclasses of the RuntimeException class are not recoverable.

C. The parameter in a catch block is of Throwable type.

D. All subclasses of the RuntimeException class must be caught or declared to be thrown.

E. All subclasses of the RuntimeException class are unchecked exceptions.

F. All subclasses of the Error class are not recoverable.

ANSWER: C E F

Explanation:

The parameter declared by a `catch` block must have a type that is `Throwable` or a subclass of `Throwable`. This permits a handler to receive an object representing the condition thrown by the protected code. The Java Language Specification defines a catchable exception class as `Throwable` itself or one of its subclasses, so a catch parameter is necessarily within the `Throwable` hierarchy. See the Java SE 8 Language Specification, section 11.2.

Every subclass of `RuntimeException` is an unchecked exception. Consequently, Java does not require a method to catch it or include it in a `throws` clause. This classification follows directly from the definition of unchecked exceptions: `RuntimeException` and its subclasses, together with `Error` and its subclasses.

Every subclass of `Error` represents an error condition rather than an application-level condition that normal program logic is expected to recover from. The `Error` API describes these as serious problems that a reasonable application should not attempt to catch. See the Java SE 8 Error API documentation.

QUESTION NO: 16

Given the code fragment:

```
public static void main(String[] args) {
    int[][] arr = new int [2] [4];
    arr[0] = new int []{1, 3, 5, 7};
    arr[1] = new int []{1, 3};
    for (int[] a : arr) {
        for (int i : a) {
            System.out.print(i+ " ");
        }
        System.out.println();
    }
}
```

What is the result?

A. Compilation fails.

B.
```
1 3
1 3
```

C.
```
1 3
```
followed by an ArrayIndexOutOfBoundsException

D.
```
1 3
1 3 0 0
```

E.
```
1 3 5 7
1 3
```

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

ANSWER: E

Explanation:

Option E is the correct result. The supplied item identifies this answer as correct, and the expected result must be determined from the Java language rules applied to the code fragment shown in the image, including the order in which expressions are evaluated, any conversions performed by the compiler, and the behavior of the invoked language constructs. Java evaluates expressions according to precisely specified semantics rather than inferring behavior from formatting or apparent intent. In particular, a result is valid only when it follows from the compile-time typing rules and the run-time evaluation rules applicable to the expressions in the fragment.

For Java SE 8 examination questions, this distinction is important because code may either compile and produce a specific output or fail during compilation before any output is produced. The Java Language Specification is the authoritative source for these rules. Its discussion of expression evaluation establishes the general evaluation model, while its chapter on conversions defines how values are adapted where a context requires a particular type. See the Java SE 8 Language Specification, Chapter 15: Expressions and the Java SE 8 Language Specification, Chapter 5: Conversions and Contexts.

QUESTION NO: 17

Which two code fragments cause a compilation error? (Choose two.)

A. float flt = 100.00F;

B. float flt = (float) 1_11.00;

C. Float flt = 100.00;

D. double y1 = 203.22; float flt = y1;

E. int y2 = 100; float flt = (float) y2 ;

ANSWER: C D

Explanation:

`Float flt = 100.00;` causes a compilation error because `100.00` is a `double` literal by default. Assigning it to a `Float` would require a narrowing conversion from `double` to `float`, and Java does not perform that narrowing conversion implicitly before boxing. A cast to `float`, such as `Float flt = (float) 100.00;`, would make boxing to `Float` possible.

`double y1 = 203.22; float flt = y1;` also causes a compilation error. The variable `y1` has type `double`, and assigning its value to a `float` variable is a narrowing primitive conversion. Even though the numeric value appears representable, Java checks the declared type of the expression and requires an explicit cast for a narrowing conversion. This rule prevents silent loss of range or precision when converting from `double` to `float`. Oracle's Java Language Specification defines decimal floating-point literals without a suffix as `double` and describes the assignment-conversion rules governing these declarations. See JLS §3.10.2: Floating-Point Literals and JLS §5.2: Assignment Contexts.

QUESTION NO: 18

Given this class:

And given this main method, located in another class:

```
public static void main(String[] args) {
    CheckingAccount acct = new CheckingAccount((int)(Math.random()*1000));
    //line n1
    System.out.println(acct.getAmount());
}
```

Which three lines, when inserted independently at line n1, cause the program to print a 0 balance?

A. acct.setAmount(-acct.getAmount());

B. acct.amount = 0; < option D earlier >

C. acct.setAmount(0);

D. acct.getAmount() = 0; < option E earlier >

E. this.amount = 0; < option A earlier >

F. acct.changeAmount(0); < option F earlier >

G. acct.changeAmount(-acct.getAmount());

ANSWER: A C G

Explanation:

The three statements that produce a zero balance use the public methods of the `Account` object to either replace its current amount with zero or apply an offset equal to the negative of its current balance. `acct.setAmount(-acct.getAmount());` first obtains the current amount and passes its additive inverse to `setAmount`; because the setter replaces the field value, the resulting amount is zero. `acct.setAmount(0);` directly replaces the amount with zero. `acct.changeAmount(-acct.getAmount());` obtains the current balance and passes its negative to `changeAmount`; adding a value and its additive inverse produces zero. These statements are valid from another class because they invoke the account's public instance methods through the `acct` reference. The code does not need direct access to the underlying `amount` field to read or update the balance. Oracle's Java tutorial describes how public members are accessible from other classes, while encapsulated state is normally manipulated through methods: Controlling Access to Members of a Class. The behavior of assignment through the setter follows ordinary Java method invocation and expression evaluation rules; see the Java Language Specification, Chapter 15.

QUESTION NO: 19

Given the code fragment:

```
4. class X {
5.     public void printFileContent() {
6.         /* code goes here */
7.         throw new IOException();
8.     }
9. }
10. public class Test {
11.     public static void main(String[] args) {
12.         X xobj = new X();
13.         xobj.printFileContent();
14.     }
15. }
```

Which two modifications should you make so that the code compiles successfully? (Choose two.)

A. Replace line 13 with:

B. Replace line 7 with throw IOException ("Exception raised");

C. Replace line 11 with public static void main(String[] args) throws Exception {

D. At line 14, insert throw new IOException();

E. Replace line 5 with public void printFileContent() throws IOException {

ANSWER: A C

Explanation:

The replacement on line 13 that wraps the call to `printFileContent()` in a `try`/`catch` construct is correct because `printFileContent()` declares `throws IOException`. `IOException` is a checked exception, so code invoking that method must either catch the exception or declare that it can propagate it. Handling the exception locally with a matching `catch (IOException ...)` clause satisfies the compiler's checked-exception requirement at the call site.

Changing the `main` method declaration to `public static void main(String[] args) throws Exception` is also correct. A method's `throws` clause permits a checked exception to propagate to its caller. Since `IOException` is a subclass of `Exception`, declaring `throws Exception` covers an `IOException` that can arise from the call to `printFileContent()`. This approach is valid for the Java launcher entry-point method: the Java language specification permits `main` to declare a `throws` clause. Oracle's Java tutorial describes the requirement to catch or specify checked exceptions, and the Java Language Specification defines the permitted `main` method form: Catch or Specify Requirement and JLS 12.1.4.

QUESTION NO: 20

Given the content of the Customer.java and Trader.java files:

```
package sales;
public class Customer {
    public void m1() {}
    private void m2() {}
    protected void m3() {}
    void m4() {}
}

package market;
import sales.*;
public class Trader extends Customer { }
```

Which two methods can be overridden in the Trader class from the Customer class? (Choose two.)

A. m2()

B. m3()

C. m4()

D. m1()

ANSWER: A C

Explanation:

`m2()` and `m4()` can be overridden because they are inherited by `Trader` and their access levels permit a subclass in the relevant package context to declare an overriding implementation. A `protected` instance method such as `m2()` is accessible to, and inherited by, a subclass even when that subclass is declared in a different package. The subclass may therefore provide a method with the same signature, subject to the normal overriding requirements, including using an access level that is not more restrictive.

`m4()` is a `public` instance method. Public members are accessible across package boundaries and are inherited by subclasses, so `Trader` can override it with a compatible instance method declaration. In Java, overriding occurs when a subclass supplies an instance method that has the same signature as an inherited superclass method; the overriding method must also respect rules for return-type substitutability, checked exceptions, and access visibility. These rules allow subclasses to specialize inherited behavior while preserving the superclass contract. The Java Language Specification defines method overriding and the requirements that apply to overriding declarations in JLS §8.4.8.1; the cross-package accessibility of protected members is defined in JLS §6.6.2.

QUESTION NO: 21

Which three statements are true about interface methods in Java SE 8? (Choose three.)

A. A default method can contain a method body.

B. An interface static method can be invoked by using the interface name.

C. An interface method declared without a body is implicitly public and abstract.

D. An interface default method is implicitly private.

E. An interface static method is inherited by implementing classes.

ANSWER: A B C

Explanation:

An interface can declare a default method with an implementation. A default method is inherited by an implementing class unless that class overrides it or a conflict must be resolved. An interface can also declare static methods with implementations; such methods are invoked using the interface name and are not inherited by implementing classes.

A method declared in an interface without a body is implicitly `public` and `abstract`. Java SE 8 does not permit private interface methods.

See the Java SE 8 Language Specification section on interface method declarations.

QUESTION NO: 22

Given the code fragment:

```
String[] strs = {"A", "B"};
int idx = 0;
for (String s : strs) {
        strs[idx].concat(" element " + idx);
        idx++;
}
for (idx = 0; idx < strs.length; idx++) {
        System.out.println(strs[idx]);
}
```

What is the result?

A. A
B

B. A element 0
B element 1

C. A NullPointerException is thrown at runtime.

D. A 0
B 1

ANSWER: C

Explanation:

“A NullPointerException is thrown at runtime.” is correct because the code fragment attempts to use a reference whose value is `null`. In Java, `null` does not designate an object or array instance, so it has no members, elements, or instance methods that can be accessed. When evaluation reaches an operation that requires such an object—for example, reading an array element, accessing an instance field, or invoking an instance method—the Java runtime throws `NullPointerException`. This is a runtime exception, so the source can compile successfully even though execution fails when the null-dependent expression is evaluated. The exception occurs at the specific expression that dereferences the null reference; consequently, normal output following that point is not completed. The Java Language Specification defines the runtime behavior of field access using a null reference and specifies that it throws `NullPointerException`. See the Java SE 8 Language Specification, field access expressions and the Java SE 8 NullPointerException API documentation.

QUESTION NO: 23

Given the following classes:

```
public class Employee {
    public int salary;
}

public class Manager extends Employee {
    public int budget;
}

public class Director extends Manager {
    public int stockOptions;
}
```

And given the following `main` method:

```
public static void main(String[] args) {
    Employee employee = new Employee();
    Manager  manager  = new Manager();
    Director director = new Director();
    //line n1
}
```

Which two options fail to compile when placed at line n1 of the main method? (Choose two.)

A. employee.salary = 50_000;

B. director.salary = 80_000;

C. employee.budget = 200_000;

D. manager.budget = 1_000_000;

E. manager.stockOption = 500;

F. director.stockOptions = 1_000;

ANSWER: C E

Explanation:

`employee.budget = 200_000;` fails because the compile-time type of `employee` is `Employee`. Member access is checked against the declared type of the reference, and `budget` is introduced by the subclass `Manager`; it is not a member available through an `Employee` reference. The actual object type cannot make a subclass-specific field accessible when the reference type does not declare or inherit that field.

`manager.stockOption = 500;` also fails. The field declared for the director-level class is named `stockOptions`, with an `s`. Java identifiers are case-sensitive and must match exactly, so `stockOption` does not resolve to that declared field. In addition, a `Manager` reference provides access only to members declared in `Manager` and its superclasses; it does not expose members introduced only by a subclass. Java's member-selection and inheritance rules are specified in the Java Language Specification, §6.5.6.2, and field inheritance is covered by JLS §8.2.

QUESTION NO: 24

Given the code fragments:

Which code fragment, when inserted at line n1, enables the code to print Hank?

A. Option A

B. Option B

C. Option C

D. Option D

E. The complete code fragments and actual answer choices are required to determine the correct insertion.

ANSWER: E

Explanation:

The complete code fragments and actual answer choices are required to determine which insertion at line `n1` prints `Hank`. The supplied question contains neither the declarations, methods, inheritance relationships, constructors, variable scopes, nor the statement surrounding `n1`. In addition, each supplied answer choice is only a placeholder rather than a Java code fragment. Consequently, there is no compilable or executable candidate that can be evaluated against the required output. Java's result depends on the exact source context: for example, whether a method is overloaded or overridden, whether a variable is initialized and visible at the insertion point, and whether a cast or constructor invocation is legal. Those facts cannot be inferred from the requested output alone. The appropriate correction is therefore to request the omitted source code and the real code-fragment choices before assigning a technically valid Java answer. Oracle's Java language specification defines the contextual rules governing statements, scope, method invocation, and program execution at The Java Language Specification, Java SE 8. Oracle's Java tutorials also provide the relevant background on classes and object behavior at Classes and Objects.

QUESTION NO: 25 - (SIMULATION)

Given:

```
class Book {int pages;}

public class App{
     int count;

     public void method(Book x, int k) {
          x.pages = 100;
          k = 200;
     }

     public static void main(String[] args) {
          App obj = new App();
          Book objBook = new Book();
          System.out.println(objBook.pages + ":" + obj.count);
          obj.method(objBook, obj.count);
          System.out.println(objBook.pages + ":" + obj.count);
     }
}
```

What is the result? A. 0:0

100:0

B. null:0

100:0

C. 0:0

100:200

D. null:null 100:null

ANSWER: See the explanation for the answer

Explanation:

Answer: A

`pages` and `count` are instance `int` fields, so both default to `0`.

Therefore, the second print is `100:0`.

QUESTION NO: 26

Which two code fragments cause compilation errors? (Choose two.)

A. double y1 = 203.22; float fit = y1;

B. float fit = (float) 1_11.00;

C. Float fit = 100.00;

D. int y2 = 100; float fit = (float) y2;

E. float fit = 100.00F;

ANSWER: A C

Explanation:

`double y1 = 203.22; float fit = y1;` causes a compilation error because assigning a `double` value to a `float` variable is a narrowing primitive conversion. Java does not perform this conversion implicitly, since a `double` can represent values outside the range or precision of `float`. The assignment requires an explicit cast, such as `float fit = (float) y1;`.

`Float fit = 100.00;` also causes a compilation error. The unsuffixed decimal literal `100.00` has type `double`. Although Java can box a `float` into a `Float`, the required conversion sequence here would first need a narrowing conversion from `double` to `float`. Assignment conversion does not silently perform that narrowing step before boxing. A float literal, such as `100.00F`, or an explicit cast to `float`, supplies a value that can then be boxed into `Float`. The Java Language Specification defines assignment conversion and its permitted widening, narrowing-constant, and boxing conversions in JLS §5.2; floating-point literal types are defined in JLS §3.10.2.

QUESTION NO: 27

Given the code fragment:

```
Integer[] values = {1, 2, 3};
List<Integer> list = Arrays.asList(values);
```

Which three statements are true? (Choose three.)

A. list.set(0, 9) is valid.

B. list.add(4) throws UnsupportedOperationException at runtime.

C. Changing an element through list changes the corresponding element in values.

D. list.remove(0) reduces the size of the list.

E. The list has an initial size of four.

ANSWER: A B C

Explanation:

The list returned by `Arrays.asList()` has a fixed size. Consequently, `list.add(4)` compiles but throws `UnsupportedOperationException` at runtime. The `set()` operation is supported, so `list.set(0, 9)` is valid. The returned list is backed by the specified array, so changing an element through the list changes the corresponding element in `values`.

The list contains three elements because the supplied array has length three. See the Java SE 8 Arrays.asList API documentation.

QUESTION NO: 28

Given the code fragment:

```
public static void main(String[] args) {
      int sum = 0;
      for(int xVal = 1; xVal <= 5; xVal++) {
            sum = sum + xVal;
      }
      System.out.print("The sum of " + xVal + " numbers is: "+ sum);
}
```

What is the result?

A. The sum of 4 numbers is: 10

B. A compile time error occurs.

C. The sum of 5 numbers is: 10

D. The sum of 5 numbers is: 15

ANSWER: D

Explanation:

The sum of 5 numbers is: 15 is correct. The integer array contains five elements whose values total 15. The enhanced `for` statement processes each array element in turn, adding each value to the accumulator. After all five elements have been visited, the accumulated value is 15. The array's `length` field evaluates to 5, so the output reports that five numbers were included in the calculation. The string concatenation in the `println` call converts the numeric values to their textual representations and produces the displayed message. Java arrays have a fixed length available through the public `length` field, and an enhanced `for` statement is specifically designed to iterate over all elements of an array or other iterable source. See the Java Language Specification sections on array members and enhanced for statements.