## 1. Java Building Blocks

public class Zoo {

public static void main(String[] args) {

System.out.println(args[0]);

System.out.println(args[1]);

} }

The program correctly identifies the first two “words” as the arguments. Spaces are used to separate the arguments. If you want spaces inside an argument, you need to use quotes as in this example:

$ javac Zoo.java

$ java Zoo "San Diego" Zoo

All command-line arguments are treated as String objects, even if they represent another data type:

$ javac Zoo.java

$ java Zoo Zoo 2

Finally, what happens if you don’t pass in enough arguments?

$ javac Zoo.java

$ java Zoo Zoo

Zoo

**Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1 at mainmethod.Zoo.main(Zoo.java:7)**

public class Conflicts {

Date date;

// some more code

}

The answer should be easy by now. You can write either import java.util.\*; or import java.util.Date;. The tricky cases come about when other imports are present:

import java.util.\*;

import java.sql.\*; // DOES NOT COMPILE

import java.util.Date;

import java.sql.\*;

If you explicitly import a class name, it takes precedence over any wildcards present.

■ octal (digits 0–7), which uses the number 0 as a prefix—for example, 017

■ hexadecimal (digits 0–9 and letters A–F), which uses the number 0 followed by x or X as a prefix—for example, 0xFF

■ binary (digits 0–1), which uses the number 0 followed by b or B as a prefix—for example, 0b10

System.out.println(56); // 56

System.out.println(0b11); // 3

System.out.println(017); // 15

System.out.println(0x1F); // 31

added in Java 7. You can have underscores in numbers to make them easier to read:

int million1 = 1000000;

int million2 = 1\_000\_000;

double notAtStart = \_1000.00; // DOES NOT COMPILE

double notAtEnd = 1000.00\_; // DOES NOT COMPILE

double notByDecimal = 1000\_.00; // DOES NOT COMPILE

double annoyingButLegal = 1\_00\_0.0\_0; // this one compiles

**Declaring Multiple Variables**

int i1, i2, i3 = 0;

As you should expect, three variables were declared: i1, i2, and i3. However, only one of those values was initialized: i3. The other two remain declared but not yet initialized.

int num, String value; // DOES NOT COMPILE

This code doesn’t compile because it tries to declare multiple variables of different types in the same statement.

double d1, double d2; // DOES NOT COMPILE

If you want to declare multiple variables in the same statement, they must share the same type declaration and not repeat it. double d1, d2; would have been legal.

boolean b1, b2;

String s1 = "1", s2;

double d1, double d2;

int i1; int i2;

int i3; i4;

The first statement is legal. It declares two variables without initializing them. second statement is also legal. It declares two variables and initializes only one of them. The third statement is not legal. Variables d1 and d2 are the same type & breaks between them.

The fourth statement is legal.The fifth statement is not legal.The second one is not a valid declaration because it omits the type.

**Garbage Collection**

1: public class Scope {

2: public static void main(String[] args) {

3: String one, two;

4: one = new String("a");

5: two = new String("b");

6: one = two;

7: String three = one;

8: one = null;

9: } }

Questions

6. Given the following classes, what is the maximum number of imports that can be removed and have the code still compile?

package aquarium; public class Water { }

package aquarium;

import java.lang.\*;

import java.lang.System;

import aquarium.Water;

import aquarium.\*;

public class Tank {

public void print(Water water) {

System.out.println(water); } }

A. 0

B. 1

C. 2

D. 3

E. 4

F. Does not compile.

E. The first two imports can be removed because java.lang is automatically imported.The second two imports can be removed because Tank and Water are in the same package,making the correct answer E. If Tank and Water were in different packages, one of

these two imports could be removed. In that case, the answer would be option D.

7. Given the following classes, which of the following snippets can be inserted in place of INSERT IMPORTS HERE and have the code compile? (Choose all that apply)

package aquarium;

public class Water {

boolean salty = false;

}

package aquarium.jellies;

public class Water {

boolean salty = true;

}

package employee;

INSERT IMPORTS HERE

public class WaterFiller {

Water water;

}

A. import aquarium.\*;

B. import aquarium.Water;

import aquarium.jellies.\*;

C. import aquarium.\*;

import aquarium.jellies.Water;

D. import aquarium.\*;

import aquarium.jellies.\*;

E. import aquarium.Water;

import aquarium.jellies.Water;

F. None of these imports can make the code compile

A, B, C. Option A is correct because it imports all the classes in the aquarium package including aquarium.Water. Options B and C are correct because they import Water by classname. Since importing by classname takes precedence over wildcards, these compile.

Option D is incorrect because Java doesn’t know which of the two wildcard Water classes to use. Option E is incorrect because you cannot specify the same classname in two imports.

10. Which of the following are legal entry point methods that can be run from the command line? (Choose all that apply)

A. private static void main(String[] args)

B. public static final main(String[] args)

C. public void main(String[] args)

D. public static void test(String[] args)

E. public static void main(String[] args)

F. public static main(String[] args)

G. None of the above

E. Option E is the canonical main() method signature. You need to memorize it.Option A is incorrect because the main() method must be public. Options B and F are incorrect because the main() method must have a void return type. Option C is

incorrect because the main() method must be static. Option D is incorrect because the main() method must be named main.

12. Which of the following are true? (Choose all that apply)

A. A local variable of type boolean defaults to null.

B. A local variable of type float defaults to 0.

C. A local variable of type Object defaults to null.

D. A local variable of type boolean defaults to false.

E. A local variable of type boolean defaults to true.

F. A local variable of type float defaults to 0.0.

G. None of the above.

G. Option G is correct because local variables do not get assigned default values. The code fails to compile if a local variable is not explicitly initialized. If this question were about instance variables, options D and F would be correct. A boolean primitive

defaults to false and a float primitive defaults to 0.0

19. Suppose we have a class named Rabbit. Which of the following statements are true?(Choose all that apply)

1: public class Rabbit {

2: public static void main(String[] args) {

3: Rabbit one = new Rabbit();

4: Rabbit two = new Rabbit();

5: Rabbit three = one;

6: one = null;

7: Rabbit four = one;

8: three = null;

9: two = null;

10: two = new Rabbit();

11: System.gc();

12: } }

A. The Rabbit object from line 3 is first eligible for garbage collection immediately

following line 6.

B. The Rabbit object from line 3 is first eligible for garbage collection immediately

following line 8.

C. The Rabbit object from line 3 is first eligible for garbage collection immediately

following line 12.

D. The Rabbit object from line 4 is first eligible for garbage collection immediately

following line 9.

E. The Rabbit object from line 4 is first eligible for garbage collection immediately

following line 11.

F. The Rabbit object from line 4 is first eligible for garbage collection immediately

following line 12.

B, D. The Rabbit object from line 3 has two references to it: one and three. The references are nulled out on lines 6 and 8, respectively. Option B is correct because this makes the object eligible for garbage collection after line 8. Line 7 sets the reference

four to the now null one, which means it has no effect on garbage collection. The Rabbit object from line 4 only has a single reference to it: two. Option D is correct because this single reference becomes null on line 9. The Rabbit object declared on line 10

becomes eligible for garbage collection at the end of the method on line 12. Calling System.gc() has no effect on eligibility for garbage collection.

## 2.Operators and Statements

System.out.print(9 / 3); // Outputs 3

System.out.print(9 % 3); // Outputs 0

System.out.print(10 / 3); // Outputs 3

System.out.print(10 % 3); // Outputs 1

System.out.print(11 / 3); // Outputs 3

System.out.print(11 % 3); // Outputs 2

System.out.print(12 / 3); // Outputs 4

System.out.print(12 % 3); // Outputs 0

**Numeric Promotion Rules**

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.

2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value’s data type.

3. Smaller data types, namely byte, short, and char, are first promoted to int any time they’re used with a Java binary arithmetic operator, even if neither of the operands is int.

4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

What is the data type of x / y?

short x = 10;

short y = 3;

In this case, we must apply the third rule, namely that x and y will both be promoted to int before the operation, resulting in an output of type int.

What is the data type of x \* y / z?

short x = 14;

float y = 13;

double z = 30;

In this case, we must apply all of the rules. First, x will automatically be promoted to int solely because it is a short and it is being used in an arithmetic binary operation. The promoted x value will then be automatically promoted to a float so that it can be

multiplied with y. The result of x \* y will then be automatically.

int x = 3;

int y = ++x \* 5 / x-- + --x;

System.out.println("x is " + x);

System.out.println("y is " + y);

int y = 4 \* 5 / x-- + --x; // x assigned value of 4

int y = 4 \* 5 / 4 + --x; // x assigned value of 3

int y = 4 \* 5 / 4 + 2; // x assigned value of 2

we evaluate the multiple and division from left-to-right, and fi nish with the addition.The result is then printed:

x is 2

y is 7

long t = 192301398193810323; // DOES NOT COMPILE

It does not compile because Java interprets the literal as an int and notices that the value is larger than int allows. The literal would need a postfi x L to be considered a long

short x = 10;

short y = 3;

short z = x \* y; // DOES NOT COMPILE

short x = 10;

short y = 3;

short z = (short)(x \* y);

long x = 10;

int y = 5;

y = y \* x; // DOES NOT COMPILE

Based on the last two sections, you should be able to spot the problem in the last line.This last line could be fi xed with an explicit cast to (int), but there’s a better way using the compound assignment operator:

long x = 10;

int y = 5;

y \*= x;

The compound operator will first cast x to a long, apply the multiplication of two long values, and then cast the result to an int.

long x = 5;

long y = (x=3);

System.out.println(x); // Outputs 3

System.out.println(y); // Also, outputs 3

**Equality Operators (==)**

The comparisons for equality are limited to these three cases, so you cannot mix and match types. For example, each of the following would result in a compiler error:

boolean x = true == 3; // DOES NOT COMPILE

boolean y = false != "Giraffe"; // DOES NOT COMPILE

boolean z = 3 == "Kangaroo"; // DOES NOT COMPILE

**Conditional Statements**

int x = 1;

if(x) { // DOES NOT COMPILE

...

}

int x = 1;

if(x = 5) { // DOES NOT COMPILE

...

}

System.out.println((y > 5) ? 21 : "Zebra");

int animal = (y < 91) ? 9 : "Horse"; // DOES NOT COMPILE

^ (X-OR) - Homogenious are FALSE(T,T F,F), Hertrogenious are TRUE(T,F F,T)

QUESTIONS

9. How many times will the following code print "Hello World"?

3: for(int i=0; i<10 ; ) {

4: i = i++;

5: System.out.println("Hello World");

6: }

A. 9

B. 10

C. 11

D. The code will not compile because of line 3.

E. The code will not compile because of line 5.

F. The code contains an infinite loop and does not terminate.

F. In this example, the update statement of the for loop is missing, which is fine as the statement is optional, so option D is incorrect. The expression inside the loop increments i but then assigns i the old value. Therefore, i ends the loop with the same value

that it starts with: 0. The loop will repeat infinitely, outputting the same statement over and over again because i remains 0 after every iteration of the loop.

10. What is the output of the following code?

3: byte a = 40, b = 50;

4: byte sum = (byte) a + b;

5: System.out.println(sum);

A. 40

B. 50

C. 90

D. The code will not compile because of line 4.

E. An undefined value.

D. Line 4 generates a possible loss of precision compiler error. The cast operator has the highest precedence, so it is evaluated first, casting a to a byte. Then, the addition is evaluated, causing both a and b to be promoted to int values. The value 90 is an int

and cannot be assigned to the byte sum without an explicit cast, so the code does not compile. The code could be corrected with parentheses around (a + b), in which case option C would be the correct answer.

15. What is the output of the following code snippet?

3: int x = 1, y = 15;

4: while x < 10

5: y––;

6: x++;

7: System.out.println(x+", "+y);

A. 10, 5

B. 10, 6

C. 11, 5

D. The code will not compile because of line 3.

E. The code will not compile because of line 4.

F. The code contains an infinite loop and does not terminate

E. This is actually a much simpler problem than it appears to be. The while statement on line 4 is missing parentheses, so the code will not compile, and option E is the correct answer. If the parentheses were added, though, option F would be the correct

answer since the loop does not use curly braces to include x++ and the boolean expression never changes. Finally, if curly braces were added around both expressions, the output would be 10, 6 and option B would be correct.

2. What data type (or types) will allow the following code snippet to compile? (Choose all that apply)

byte x = 5;

byte y = 10;

\_\_\_\_\_ z = x + y;

A. int

B. long

C. boolean

D. double

E. short

F. byte

A, B, D. The value x + y is automatically promoted to int, so int and data types that can be promoted automatically from int will work. Options A, B, D are such data types. Option C will not work because boolean is not a numeric data type. Options E and F will not work without an explicit cast to a smaller data type.

## 3. APIs-Arrays-String-Date

1. If both operands are numeric, + means numeric addition.

2. If either operand is a String, + means concatenation.

3. The expression is evaluated left to right.

Now let’s look at some examples:

System.out.println(1 + 2); // 3

System.out.println("a" + "b"); // ab

System.out.println("a" + "b" + 3); // ab3

System.out.println(1 + 2 + "c"); // 3c

System.out.println(string.substring(3, 3)); // empty string

System.out.println(string.substring(3, 2)); // throws exception

System.out.println(string.substring(3, 8)); // throws exception

3: StringBuilder sb = new StringBuilder("animals");

4: sb.insert(7, "-"); // sb = animals-

5: sb.insert(0, "-"); // sb = -animals-

6: sb.insert(4, "-"); // sb = -ani-mals

7: System.out.println(sb);

**Understanding Equality**

StringBuilder one = new StringBuilder();

StringBuilder two = new StringBuilder();

StringBuilder three = one.append("a");

System.out.println(one == two); // false

System.out.println(one == three); // true

StringBuilder methods like to return the current reference for chaining? This means one and three both point to the same object and

the second print statement gives us true

String x = "Hello World";

String y = "Hello World";

System.out.println(x == y); // true

Remember that Strings are immutable and literals are pooled. The JVM created only one literal in memory. x and y both point to the same location in memory; therefore, the statement outputs true. It gets even trickier. Consider this code:

String x = "Hello World";

String z = " Hello World".trim();

System.out.println(x == z); // false

In this example, we don’t have two of the same String literal. Although x and z happen to evaluate to the same string, one is computed at runtime. Since it isn’t the same at compile-time, a new String object is created.

**Arrays**

you can type the [] before or after the name, and adding a space is optional.This means that all four of these statements do the exact same thing:

int[] numAnimals;

int [] numAnimals2;

int numAnimals3[];

int numAnimals4 [];

**Sorting**

Arrays is the fi rst class provided by Java we have used that requires an import. To use it, you must have either of the following two statements in your class:

import java.util.\* // import whole package including Arrays

import java.util.Arrays; // import just Arrays

This simple example sorts three numbers:

int[] numbers = { 6, 9, 1 };

Arrays.sort(numbers);

for (int i = 0; i < numbers.length; i++)

System.out.print (numbers[i] + " "); //1 6 9,

String[] strings = { "10", "9", "100" };

Arrays.sort(strings);

for (String string : strings)

System.out.print(string + " ");//10 100 9.

String sorts in alphabetic order, and 1 sorts before 9. (Numbers sort before letters and uppercase sorts before lowercase, in case you were wondering.)

**Searching**

Java also provides a convenient way to search—but only if the array is already sorted.

**Target element found in sorted array** -Index of match

**Target element not found in sorted array** -Negative value where a match needs to be inserted to preserve sorted order

**Unsorted array A surprise** —this result isn’t predictable

3: int[] numbers = {2,4,6,8};

4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0

5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1

6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1

7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2

8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5

5: int numbers = new int[] {3,2,1};

6: System.out.println(Arrays.binarySearch(numbers, 2));

7: System.out.println(Arrays.binarySearch(numbers, 3));

Note that on line 5, the array isn’t sorted. This means the output will not be predictable.

**ArrayList**

ArrayList is an ordered sequence that allows duplicates.

import java.util.\* // import whole package including ArrayList

import java.util.ArrayList; // import just ArrayList

**Creating an ArrayList:**

As with StringBuilder, there are three ways to create an ArrayList:

ArrayList list1 = new ArrayList();

ArrayList list2 = new ArrayList(10);

ArrayList list3 = new ArrayList(list2);

Java 5 introduced generics, which allow you to specify the type of class that the ArrayList will contain.

ArrayList<String> list4 = new ArrayList<String>();

ArrayList<String> list5 = new ArrayList<>();

Java 5 allows you to tell the compiler what the type would be by specifying it between <and >. **in Java 7, you can omit that type from the right side. The < and >** are still required, though. This is called the diamond operator because <> looks like a diamond.

List<String> list6 = new ArrayList<>();

ArrayList<String> list7 = new List<>(); // DOES NOT COMPILE

**boolean add(E element)**

**void add(int index, E element)**

ArrayList list = new ArrayList();

list.add("hawk"); // [hawk]

list.add(Boolean.TRUE); // [hawk, true]

System.out.println(list); // [hawk, true]

ArrayList<String> safer = new ArrayList<>();

safer.add("sparrow");

safer.add(Boolean.TRUE); // DOES NOT COMPILE

4: List<String> birds = new ArrayList<>();

5: birds.add("hawk"); // [hawk]

6: birds.add(1, "robin"); // [hawk, robin]

7: birds.add(0, "blue jay"); // [blue jay, hawk, robin]

8: birds.add(1, "cardinal"); // [blue jay, cardinal, hawk, robin]

9: System.out.println(birds); // [blue jay, cardinal, hawk, robin]

**boolean remove(Object object)**

**E<Object> remove(int index)**

3: List<String> birds = new ArrayList<>();

4: birds.add("hawk"); // [hawk]

5: birds.add("hawk"); // [hawk, hawk]

6: System.out.println(birds.remove("cardinal")); // prints false

7: System.out.println(birds.remove("hawk")); // prints true

8: System.out.println(birds.remove(0)); // prints hawk

9: System.out.println(birds); // []

The set() method changes one of the elements of the ArrayList without changing the size.

**E set(int index, E newElement)**

The E return type is the element that got replaced.

15: List<String> birds = new ArrayList<>();

16: birds.add("hawk"); // [hawk]

17: System.out.println(birds.size()); // 1

18: birds.set(0, "robin"); // [robin]

19: System.out.println(birds.size()); // 1

20: birds.set(1, "robin"); // IndexOutOfBoundsException

**boolean isEmpty()**

**int size()**

System.out.println(birds.isEmpty()); // true

System.out.println(birds.size()); // 0

birds.add("hawk"); // [hawk]

birds.add("hawk"); // [hawk, hawk]

System.out.println(birds.isEmpty()); // false

System.out.println(birds.size()); // 2

**void clear()**

The clear() method provides an easy way to discard all elements of the ArrayList. The

List<String> birds = new ArrayList<>();

birds.add("hawk"); // [hawk]

birds.add("hawk"); // [hawk, hawk]

System.out.println(birds.isEmpty()); // false

System.out.println(birds.size()); // 2

birds.clear(); // []

System.out.println(birds.isEmpty()); // true

System.out.println(birds.size()); // 0

**boolean contains(Object object)**

List<String> birds = new ArrayList<>();

birds.add("hawk"); // [hawk]

System.out.println(birds.contains("hawk")); // true

System.out.println(birds.contains("robin")); // false

**equals()**

Finally, ArrayList has a custom implementation of equals() so you can compare two lists to see if they contain the same elements in the same order.

**boolean equals(Object object)**

31: List<String> one = new ArrayList<>();

32: List<String> two = new ArrayList<>();

33: System.out.println(one.equals(two)); // true

34: one.add("a"); // [a]

35: System.out.println(one.equals(two)); // false

36: two.add("a"); // [a]

37: System.out.println(one.equals(two)); // true

38: one.add("b"); // [a,b]

39: two.add(0, "b"); // [b,a]

40: System.out.println(one.equals(two)); // false

Converting Between array and List

**ArrayList into an array:(toArray())**

3: List<String> list = new ArrayList<>();

4: list.add("hawk");

5: list.add("robin");

6: Object[] objectArray = list.toArray();

7: System.out.println(objectArray.length); // 2

8: String[] stringArray = list.toArray(new String[0]);

9: System.out.println(stringArray.length); // 2

Line 6 shows that an ArrayList knows how to convert itself to an array. The only problem is that it defaults to an array of class Object.

Line 8 specifies the type of the array as String, size of 0 for the parameter is that Java will create a new array of the proper size for the return value.

**Array to a List:(asList())**

The original array and created array backed List(A List elements can Replace , not Remove)

20: String[] array = { "hawk", "robin" }; // [hawk, robin]

21: List<String> list = Arrays.asList(array); // returns fixed size list

22: System.out.println(list.size()); // 2

23: list.set(1, "test"); // [hawk, test]

24: array[0] = "new"; // [new, test]

25: for (String b : array) System.out.print(b + " "); // new test

26: list.remove(1); // throws UnsupportedOperation Exception

Line 21 converts the array to a List. Note that it isn’t the java.util.ArrayList we’ve grown used to. It is a fi xed-size, backed version of a List. Line 23 is okay because set() merely replaces an existing value. It updates both array and list because they point to the

same data store. Line 24 also changes both array and list. Line 25 shows the array has changed to new test. Line 26 throws an exception because we are not allowed to change the size of the list.

**List<String> list = Arrays.asList("one", "two");**

**asList()** takes varargs, which let you pass in an array or just type out the String values. This is handy when testing because you can easily create and populate a List on one line.

**Sorting**

Sorting an ArrayList is very similar to sorting an array. You just use a different helper class:

List<Integer> numbers = new ArrayList<>();

numbers.add(99);

numbers.add(5);

numbers.add(81);

Collections.sort(numbers);

System.out.println(numbers); [5, 81, 99]

**Date & Time API (java.time.\*;)**

\*\*\*\*\* Dates are immutable

\*\*\*\*\* Period does not allow chaining.But No Compile & Runtime Errors. Only the last Period method called counts.

LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);

LocalTime time1 = LocalTime.of(6, 15);

Finally, we can combine dates and times:

LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);

LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);

LocalDate d = new LocalDate(); // DOES NOT COMPILE

You are not allowed to construct a date or time object directly.

LocalDate.of(2015, Month.JANUARY, 32) // throws DateTimeException

java.time.DateTimeException: Invalid value for DayOfMonth:(valid values 1 - 28/31): 32

LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);

date = date.plusMinutes(1); // DOES NOT COMPILE

LocalDate does not contain time. This means you cannot add minutes to it.

**Periods**

LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);

LocalDate end = LocalDate.of(2015, Month.MARCH, 30);

public static void main(String[] args) {

LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);

LocalDate end = LocalDate.of(2015, Month.MARCH, 30);

Period period = Period.ofMonths(1); // create a period

performAnimalEnrichment(start, end, period);

}

private static void performAnimalEnrichment(LocalDate start, LocalDate end,

Period period) { // uses the generic period

LocalDate upTo = start;

while (upTo.isBefore(end)) {

System.out.println("give new toy: " + upTo);

upTo = upTo.plus(period); // adds the period

}}

The method can add an arbitrary period of time that gets passed in. This allows us to reuse the same method for different periods of time as our zookeeper changes her mind.

**There are five ways to create a Period class:**

Period annually = Period.ofYears(1); // every 1 year

Period quarterly = Period.ofMonths(3); // every 3 months

Period everyThreeWeeks = Period.ofWeeks(3); // every 3 weeks

Period everyOtherDay = Period.ofDays(2); // every 2 days

Period everyYearAndAWeek = Period.of(1, 0, 7); // every year and 7 days

You cannot chain methods when creating a Period. Only the last method is used because the Period.ofXXX methods are static methods.

Period wrong = Period.ofYears(1).ofWeeks(1); // every week

3: LocalDate date = LocalDate.of(2015, 1, 20);

4: LocalTime time = LocalTime.of(6, 15);

5: LocalDateTime dateTime = LocalDateTime.of(date, time);

6: Period period = Period.ofMonths(1);

7: System.out.println(date.plus(period)); // 2015-02-20

8: System.out.println(dateTime.plus(period)); // 2015-02-20T06:15

9: System.out.println(time.plus(period)); // UnsupportedTemporalTypeException

Lines 7 and 8 work as expected. They add a month to January 20, 2015, giving us February 20, 2015. The fi rst has only the date, and the second has both the date and time. Line 9 attempts to add a month to an object that only has a time.

**Formatting Dates and Times**

**DateTimeFormatter** can be used to format any type of date and/or time object. DateTimeFormatter is in the package java.time.format.

LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);

LocalTime time = LocalTime.of(11, 12, 34);

LocalDateTime dateTime = LocalDateTime.of(date, time);

System.out.println(date.format(DateTimeFormatter.ISO\_LOCAL\_DATE));// 2020-01-20

System.out.println(time.format(DateTimeFormatter.ISO\_LOCAL\_TIME));// 11:12:34

System.out.println(dateTime.format(DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME));// 2020-01-20T11:12:34

DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

System.out.println(shortDateTime.format(dateTime)); // 1/20/20

System.out.println(shortDateTime.format(date)); // 1/20/20

System.out.println(shortDateTime.format(time)); // UnsupportedTemporalTypeException

DateTimeFormatter shortF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);

DateTimeFormatter mediumF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);

System.out.println(shortF.format(dateTime)); // 1/20/20 11:12 AM

System.out.println(mediumF.format(dateTime)); // Jan 20, 2020 11:12:34// AM

// If you don’t want to use one of the predefined formats, you can create your own.

**//M outputs 1, MM outputs 01, MMM outputs Jan, and MMMM outputs January.**

DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");

System.out.println(dateTime.format(f)); // January 20, 2020, 11:12

DateTimeFormatter df = DateTimeFormatter.ofPattern("MM dd yyyy");

LocalDate d1 = LocalDate.parse("01 02 2015", df);

LocalTime t1 = LocalTime.parse("11:22");

System.out.println(d1); // 2015-01-02

System.out.println(t1); // 11:22

\*\*\*\* All substring(6,5), charAt(8) are index exceptions throws at Runtime & not at compile time

QUESTIONS

1. What is output by the following code? (Choose all that apply)

1: public class Fish {

2: public static void main(String[] args) {

3: int numFish = 4;

4: String fishType = "tuna";

5: String anotherFish = numFish + 1;

6: System.out.println(anotherFish + " " + fishType);

7: System.out.println(numFish + " " + 1);

8: } }

A. 4 1

B. 41

C. 5

D. 5 tuna

E. 5tuna

F. 51tuna

G. The code does not compile.

G. Line 5 does not compile. This question is checking to see if you are paying attention to the types. numFish is an int and 1 is an int. Therefore, we use numeric addition and get 5. The problem is that we can’t store an int in a String variable. Supposing line 5

said String anotherFish = numFish + 1 + "";. In that case, the answer would be options A and D. The variable defined on line 5 would be the string "5", and both output statements would use concatenation.

2. Which of the following are output by this code? (Choose all that apply)

3: String s = "Hello";

4: String t = new String(s);

5: if ("Hello".equals(s)) System.out.println("one");

6: if (t == s) System.out.println("two");

7: if (t.equals(s)) System.out.println("three");

8: if ("Hello" == s) System.out.println("four");

9: if ("Hello" == t) System.out.println("five");

A. one

B. two

C. three

D. four

E. five

F. The code does not compile.

A, C, D. The code compiles fine. Line 3 points to the String in the string pool. Line 4 calls the String constructor explicitly and is therefore a different object than s. Lines 5 and 7 check for object equality, which is true, and so print one and three. Line 6 uses

object reference equality, which is not true since we have different objects. Line 7 also compares references but is true since both references point to the object from the string pool. Finally, line 8 compares one object from the string pool with one that was explicitly constructed and returns false.

3. Which are true statements? (Choose all that apply)

A. An immutable object can be modified.

B. An immutable object cannot be modified.

C. An immutable object can be garbage collected.

D. An immutable object cannot be garbage collected.

E. String is immutable.

F. StringBuffer is immutable.

G. StringBuilder is immutable.

B, C, E. Immutable means the state of an object cannot change once it is created.Immutable objects can be garbage collected just like mutable objects. String is immutable.StringBuilder can be mutated with methods like append()

5. What is the result of the following code?

2: String s1 = "java";

3: StringBuilder s2 = new StringBuilder("java");

4: if (s1 == s2)

5: System.out.print("1");

6: if (s1.equals(s2))

7: System.out.print("2");

A. 1

B. 2

C. 12

D. No output is printed.

E. An exception is thrown.

F. The code does not compile

F. The question is trying to distract you into paying attention to logical equality versus object reference equality. It is hoping you will miss the fact that line 4 does not compile.

**Java does not allow you to compare String and StringBuilder using ==.**

10. What is the result of the following code? (Choose all that apply)

13: String a = "";

14: a += 2;

15: a += 'c';

16: a += false;

17: if ( a == "2cfalse") System.out.println("==");

18: if ( a.equals("2cfalse")) System.out.println("equals");

A. Compile error on line 14.

B. Compile error on line 15.

C. Compile error on line 16.

D. Compile error on another line.

E. ==

F. equals

G. An exception is thrown.

F. a += 2 expands to a = a + 2. A String concatenated with any other type gives a String. Lines 14, 15, and 16 all append to a, giving a result of "2cfalse". The if statement on line 18 returns false because the values of the two String objects are the same using object equality. The if statement on line 17 returns false because the two String objects are not the same in memory. One comes directly from the string pool and the other comes from building using String operations

15. Which of these array declarations is not legal? (Choose all that apply)

A. int[][] scores = new int[5][];

B. Object[][][] cubbies = new Object[3][0][5];

C. String beans[] = new beans[6];

D. java.util.Date[] dates[] = new java.util.Date[2][];

E. int[][] types = new int[];

F. int[][] java = new int[][];

C, E, F. Option C uses the variable name as if it were a type, which is clearly illegal.Options E , another diminesion on right side missing.F don’t specify any size. Dimensions of a multidimensional array, the first one is required.

16. Which of these compile when replacing line 8? (Choose all that apply)

7: char[]c = new char[2];

8: // INSERT CODE HERE

A. int length = c.capacity;

B. int length = c.capacity();

C. int length = c.length;

D. int length = c.length();

E. int length = c.size;

F. int length = c.size();

G. None of the above.

C. Arrays define a property called length. It is not a method, so parentheses are not allowed.

17. Which of these compile when replacing line 8? (Choose all that apply)

7: ArrayList l = new ArrayList();

8: // INSERT CODE HERE

A. int length = l.capacity;

B. int length = l.capacity();

C. int length = l.length;

D. int length = l.length();

E. int length = l.size;

F. int length = l.size();

F. The ArrayList class defines a method called size().

18. Which of the following are true? (Choose all that apply)

A. An array has a fixed size.

B. An ArrayList has a fixed size.

C. An array allows multiple dimensions.

D. An array is ordered.

E. An ArrayList is ordered.

F. An array is immutable.

G. An ArrayList is immutable

A, C, D, E. An array is not able to change size and can have multiple dimensions. Both an array and ArrayList are ordered and have indexes. Neither is immutable. The elements can change in value

19. Which of the following are true? (Choose all that apply)

A. Two arrays with the same content are equal.

B. Two ArrayLists with the same content are equal.

C. If you call remove(0) using an empty ArrayList object, it will compile successfully.

D. If you call remove(0) using an empty ArrayList object, it will run successfully.

B, C. An array does not override equals() and so uses object equality. ArrayList does override equals() and defines it as the same elements in the same order. The compiler does not know when an index is out of bounds and thus can’t give you a compiler

error. The code will throw an exception at runtime, though

23. What is the result of the following?

4: List<Integer> list = Arrays.asList(10, 4, -1, 5);

5: Collections.sort(list);

6: Integer array[] = list.toArray(new Integer[4]);

7: System.out.println(array[0]);

A. –1

B. 10

C. Compiler error on line 4.

D. Compiler error on line 5.

A. Line 4 creates a fixed size array of size 4. Line 5 sorts it. Line 6 converts it back to an array. The brackets aren’t in the traditional place, but they are still legal. Line 7 prints the first element, which is now –1.

25. What is the result of the following?

List<String> hex = Arrays.asList("30", "8", "3A", "FF");

Collections.sort(hex);

int x = Collections.binarySearch(hex, "8");

int y = Collections.binarySearch(hex, "3A");

int z = Collections.binarySearch(hex, "4F");

System.out.println(x + " " + y + " " + z);

A 0 1 –2

B. 0 1 –3

C. 2 1 –2

D. 2 1 –3

E. The code doesn’t compile.

D. After sorting, hex contains [30, 3A, 8, FF]. Remember that numbers sort before letters and strings sort alphabetically. This makes 30 come before 8. A binary search correctly finds 8 at index 2 and 3A at index 1. It cannot find 4F but notices it should be at index 2. The rule when an item isn’t found is to negate that index and subtract 1.Therefore, we get –2–1, which is –3.

27. What is the result of the following?

List<String> one = new ArrayList<String>();

one.add("abc");

List<String> two = new ArrayList<>();

two.add("abc");

if (one == two)

System.out.println("A");

else if (one.equals(two))

System.out.println("B");

else

System.out.println("C");

A. A

B. B

C. C

D. An exception is thrown.

B. The first if statement is false because the variables do not point to the same object.The second if statement is true because ArrayList implements equality to mean the same elements in the same order.

What is the output of the following code?

LocalDate date = LocalDate.of(2018, Month.APRIL, 30);

date.plusDays(2);

date.plusYears(3);

System.out.println(date.getYear() + " " + date.getMonth() + " "

+ date.getDayOfMonth());

A. 2018 APRIL 2

B. 2018 APRIL 30

C. 2018 MAY 2

D. 2021 APRIL 2

E. 2021 APRIL 30

B. The date starts out as April 30, 2018. Since Dates are immutable and the plus methods have their return values ignored, the result is unchanged. Therefore, option B is correct.

33. What is the output of the following code?

LocalDateTime d = LocalDateTime.of(2015, 5, 10, 11, 22, 33);

Period p = Period.ofDays(1).ofYears(2);

d = d.minus(p);

DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle

.SHORT);

System.out.println(f.format(d));

A. 5/9/13 11:22 AM

B. 5/10/13 11:22 AM

C. 5/9/14

D. 5/10/14

E. The code does not compile.

F. A runtime exception is thrown.

B. Period does not allow chaining. Only the last Period method called counts, so only the two years are subtracted.

## 4.Methods-Constructors-Lambdas

public final void nap(int minutes) throws InterruptedException {

// take a nap

}

default void walk2() {} // DOES NOT COMPILE

walk2() doesn’t compile because default is not a valid access modifier.

public final void walk2() {}

public static final void walk3() {}

public final static void walk4() {}

public modifier void walk5() {} // DOES NOT COMPILE

public void final walk6() {} // DOES NOT COMPIL

final public void walk7() {}

walk3() and walk4() are valid method declarations with both final and static as optional specifiers. The order of these two keywords doesn’t matter. walk7() does compile. Java allows the optional specifi ers to appear before the accessmodifier

Finally Order will be

**{Access Specifier/ Optional Specifier /or both} returntype methodname{ }**

**Static Block**

static blocks will execute at the time of class loading only.

1.Identification of static members from parent to child

2.Execution of static variable assignments and static blocks from parent to child.

3.Execution of child class main method.

**Instance blocks**

1.Identification of Instance members from parent to child

**Parent Class:**

2.Execution of Instance variable assignments and instance blocks only in parent class

3.Execution of Parent class constructor.

**Child Class:**

4.Execution of Instance variable assignments and instance blocks in the child class.

5.Execution of child class constructor

Static Blocks will execute at Class loading time, in both parent child classes,

Finally Order is :Static Block > Instance Block > Constructor

class A {

A() {

System.out.println("Class-A : Costrcutor");

}

static {

System.out.println("Class-A : Static Block");

}

{

System.out.println("Class-A : Instance Block");

}

}

public class Order extends A {

Order() {

System.out.println("Order : Costrcutor");

}

static {

System.out.println("Order : Static Block");

}

{

System.out.println("Order : Instance Block");

}

public static void main(String[] args) {

A ob = new A(); // only A Class

System.out.println("A==Completed");

Order o1 = new Order(); // Order class Contains A Class DataMembers

System.out.println("Order1 == Completed");

Order o2 = new Order(); // Order class Contains A Class DataMembers

System.out.println("Order2 == Completed");

}

}

Class-A : Static Block

Order : Static Block

Class-A : Instance Block

Class-A : Costrcutor

A==Completed

Class-A : Instance Block

Class-A : Costrcutor

Order : Instance Block

Order : Costrcutor

Order1 == Completed

Class-A : Instance Block

Class-A : Costrcutor

Order : Instance Block

Order : Costrcutor

Order2 == Completed

**Lambdas**

Lambdas work with interfaces that have only one method. These are called functional interfaces—interfaces that can be used with functional programming.

print(animals, a -> a.canSwim());

How about Animals that cannot swim?

print(animals, a -> ! a.canSwim());

The syntax of lambdas is tricky because many parts are optional. These two lines do the exact same thing:

a -> a.canHop()

(Animal a) -> { return a.canHop(); }

valid interfaces that can consume a lambda with zero, one, or two String parameters.

3: print(() -> true); // 0 parameters

4: print(a -> a.startsWith("test")); // 1 parameter

5: print((String a) -> a.startsWith("test")); // 1 parameter

6: print((a, b) -> a.startsWith("test")); // 2 parameters

7: print((String a, String b) -> a.startsWith("test")); // 2 parameters

print(a, b -> a.startsWith("test")); // DOES NOT COMPILE

print(a -> { a.startsWith("test"); }); // DOES NOT COMPILE

print(a -> { return a.startsWith("test") }); // DOES NOT COMPILE

The first line needs parentheses around the parameter list. Remember that the parentheses are only optional when there is one parameter and it doesn’t have a type declared. The second line is missing the return keyword. The last line is missing the semicolon.

You might have noticed all of our lambdas return a boolean. That is because the scope for the OCA exam limits what you need to learn.

(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE (right side arg a,b so a redeclared)

We tried to redeclare a, which is not allowed. By contrast, the following line is okay because it uses a different variable name:

(a, b) -> { int c = 0; return 5;}

**Predicates**

In our earlier example, we created an interface with one method:

boolean test(Animal a);

You can imagine that we’d have to create lots of interfaces like this to use lambdas. We want to test Animals and Strings and Plants and anything else that we come across. Luckily, Java recognizes that this is a common problem and provides such an interface

for us. It’s in the package java.util.function and the gist of it is as follows:

public interface Predicate<T> {

boolean test(T t);

}

The only difference is that it uses this type T instead of Animal. That’s the syntax for generics. It’s like when we created an ArrayList and got to specify any type that goes in it.

Java 8 even integrated the Predicate interface into some existing classes. There is only one you need to know for the exam. ArrayList declares a removeIf() method that takes a Predicate.

3: List<String> bunnies = new ArrayList<>();

4: bunnies.add("long ear");

5: bunnies.add("floppy");

6: bunnies.add("hoppy");

7: System.out.println(bunnies); // [long ear, floppy, hoppy]

8: bunnies.removeIf(s -> s.charAt(0) != 'h');

9: System.out.println(bunnies); // [hoppy]

Line 8 takes care of everything for us. It defi nes a predicate that takes a String and returns a boolean. The removeIf() method does the rest.For the OCA exam, you only need to know how to implement lambda expressions thatuse the Predicate interface. Remember the one method in the interface called test()?

Questions

**\*\*\*\* moreB(int values[], int... nums) -> No Compile time error or Runtime Error**

**case 1 : moreB(new int[]{1,2,3,4,5} ); //takes as int values[], nums will empty**

**case 2 : moreB(new int[]{1,2,3,4,5}, new int[]{6,7,8,9,10} );//2 seperate arrays , one for values[], one for nums..**

25.Which of the following are true about the following code? (Choose all that apply)

public class Create {

Create() {

System.out.print("1 ");

}

Create(int num) {

System.out.print("2 ");

}

Create(Integer num) {

System.out.print("3 ");

}

Create(Object num) {

System.out.print("4 ");

}

Create(int... nums) {

System.out.print("5 ");

}

public static void main(String[] args) {

new Create(100);

new Create(1000L);

}

}

A. The code prints out 2 4.

B. The code prints out 3 4.

C. The code prints out 4 2.

D. The code prints out 4 4.

E. The code prints 3 4 if you remove the constructor Create(int num).

F. The code prints 4 4 if you remove the constructor Create(int num).

G. The code prints 5 4 if you remove the constructor Create(int num).

1. Which of the following can fill in the blank in this code to make it compile?

public class Ant {

\_\_\_\_\_ void method() { }

}

A. default

B. final

C. private

D. Public

E. String

F. zzz:

B, C. void is a return type. Only the access modifier or optional specifiers are allowed before the return type. Option C is correct, creating a method with private access. Option B is correct, creating a method with default access and the optional specifier

final.F is incorrect because labels are not allowed for methods

5. Given the following method, which of the method calls return 2? (Choose all that apply)

public int howMany(boolean b, boolean... b2) {

return b2.length;

}

A. howMany();

B. howMany(true);

C. howMany(true, true);

D. howMany(true, true, true);

E. howMany(true, {true});

F. howMany(true, {true, true});

G. howMany(true, new boolean[2]);

D, G. Option D passes the initial parameter plus two more to turn into a vararg array of size 2. Option G passes the initial parameter plus an array of size 2. Option A does not compile because it does not pass the initial parameter. Options E and F do not compile because they do not declare an array properly. It should be new boolean[] {true}. Option B creates a vararg array of size 0 and option C creates a vararg array of size 1.

**8. Which of the following are true? (Choose all that apply**)

A. Encapsulation uses package private instance variables.

B. Encapsulation uses private instance variables.

C. Encapsulation allows setters.

D. Immutability uses package private instance variables.

E. Immutability uses private instance variables.

F. Immutability allows setters.

B, C, E. Encapsulation requires using methods to get and set instance variables so other classes are not directly using them. Instance variables must be private for this to work. Immutability takes this a step further, allowing only getters, so the instance variables do not change state.

10. What is the output of the following code?

1: package rope;

2: public class Rope {

3: public static int LENGTH = 5;

4: static {

5: LENGTH = 10;

6: }

7: public static void swing() {

8: System.out.print("swing ");

9: }

10: }

1: import rope.\*;

2: import static rope.Rope.\*;

3: public class Chimp {

4: public static void main(String[] args) {

5: Rope.swing();

6: new Rope().swing();

7: System.out.println(LENGTH);

8: }

9: }

A. swing swing 5

B. swing swing 10

C. Compiler error on line 2 of Chimp.

D. Compiler error on line 5 of Chimp.

E. Compiler error on line 6 of Chimp.

F. Compiler error on line 7 of Chimp.

B. Rope runs line 3, setting LENGTH to 5, then immediately after runs the static initializer,which sets it to 10. Line 5 calls the static method normally and prints swing.Line 6 also calls the static method. Java allows calling a static method through an instance variable. Line 7 uses the static import on line 2 to reference LENGTH.

11. Which are true of the following code? (Choose all that apply)

1: public class Rope {

2: public static void swing() {

3: System.out.print("swing ");

4: }

5: public void climb() {

6: System.out.println("climb ");

7: }

8: public static void play() {

9: swing();

10: climb();

11: }

12: public static void main(String[] args) {

13: Rope rope = new Rope();

14: rope.play();

15: Rope rope2 = null;

16: rope2.play();

17: }

18: }

A. The code compiles as is.

B. There is exactly one compiler error in the code.

C. There are exactly two compiler errors in the code.

D. If the lines with compiler errors are removed, the output is climb climb.

E. If the lines with compiler errors are removed, the output is swing swing.

F. If the lines with compile errors are removed, the code throws a NullPointerException.

11. B, E. Line 10 does not compile because static methods are not allowed to call instance

methods. Even though we are calling play() as if it were an instance method and an instance exists, Java knows play() is really a static method and treats it as such. If line 10 is removed, the code works. It does not throw a NullPointerException on line 16

because play() is a static method. Java looks at the type of the reference for rope2 and translates the call to Rope.play().

12. What is the output of the following code?

import rope.\*;

import static rope.Rope.\*;

public class RopeSwing {

private static Rope rope1 = new Rope();

private static Rope rope2 = new Rope();

{

System.out.println(rope1.length);

}

public static void main(String[] args) {

rope1.length = 2;

rope2.length = 8;

System.out.println(rope1.length);

}

}

package rope;

public class Rope {

public static int length = 0;

}

A. 02

B. 08

C. 2

D. 8

E. The code does not compile.

F. An exception is thrown.

D. There are two details to notice in this code. First, note that RopeSwing has an instance initializer and not a static initializer. Since RopeSwing is never constructed, the instance initializer does not run. The other detail is that length is static. Changes

from one object update this common static variable.

14. Which of the following can replace line 2 to make this code compile? (Chooseall that apply)

1: import java.util.\*;

2: // INSERT CODE HERE

3: public class Imports {

4: public void method(ArrayList<String> list) {

5: sort(list);

6: }

7: }

A. import static java.util.Collections;

B. import static java.util.Collections.\*;

C. import static java.util.Collections.sort(ArrayList<String>);

D. static import java.util.Collections;

E. static import java.util.Collections.\*;

F. static import java.util.Collections.sort(ArrayList<String>);

B. The two valid ways to do this are import static java.util.Collections.\*; and import static java.util.Collections.sort;. Option A is incorrect because you can only do a static import on static members. Classes such as Collections require a regular import. Option C is nonsense as method parameters have no business in an import. Options D, E, and F try to trick you into reversing the syntax of import static.

20. Which code can be inserted to have the code print 2?

public class BirdSeed {

private int numberBags;

boolean call;

public BirdSeed() {

// LINE 1

call = false;

// LINE 2

}

public BirdSeed(int numberBags) {

this.numberBags = numberBags;

}

public static void main(String[] args) {

BirdSeed seed = new BirdSeed();

System.out.println(seed.numberBags);

} }

A. Replace line 1 with BirdSeed(2);

B. Replace line 2 with BirdSeed(2);

C. Replace line 1 with new BirdSeed(2);

D. Replace line 2 with new BirdSeed(2);

E. Replace line 1 with this(2);

F. Replace line 2 with this(2);

E. Options A and B will not compile because constructors cannot be called without new. Options C and D will compile but will create a new object rather than setting the fields in this one. Option F will not compile because this() must be the first line of a constructor. Option E is correct.

22. What is the result of the following?

1: public class Order {

2: static String result = "";

3: { result += "c"; }

4: static

5: { result += "u"; }

6: { result += "r"; }

7: }

1: public class OrderDriver {

2: public static void main(String[] args) {

3: System.out.print(Order.result + " ");

4: System.out.print(Order.result + " ");

5: new Order();

6: new Order();

7: System.out.print(Order.result + " ");

8: }

9: }

A. curur

B. ucrcr

C. u ucrcr

D. u u curcur

E. u u ucrcr

F. ur ur urc

G. The code does not compile.

E. On line 3 of OrderDriver, we refer to Order for the first time. At this point the statics in Order get initialized. In this case, the statics are the static declaration of result and the static initializer. result is u at this point. On line 4, result is the same because the static initialization is only run once. On line 5, we create a new Order, which triggers the instance initializers in the order they appear in the file. Now result is ucr. Line 6 creates another Order, triggering another set of initializers. Now result is ucrcr. Notice how the static is on a different line than the initialization code in lines 4–5 of Order.

27. What is the result of the following code?

1: interface Climb {

2: boolean isTooHigh(int height, int limit);

3: }

4:

5: public class Climber {

6: public static void main(String[] args) {

7: check((h, l) -> h.append(l).isEmpty(), 5);

8: }

9: private static void check(Climb climb, int height) {

10: if (climb.isTooHigh(height, 10))

11: System.out.println("too high");

12: else

13: System.out.println("ok");

14: }

15: }

A. ok

B. too high

C. Compiler error on line 7.

D. Compiler error on line 10.

E. Compiler error on a different line.

F. A runtime exception is thrown.

C. The interface takes two int parameters. The code on line 7 attempts to use them as if one is a StringBuilder. It is tricky to use types in a lambda when they are implicitly specified. Remember to check the interface for the real type.

28. Which of the following lambda expressions can fill in the blank? (Choose all that apply)

List<String> list = new ArrayList<>();

list.removeIf(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_);

Review Questions 231

A. s -> s.isEmpty()

B. s -> {s.isEmpty()}

C. s -> {s.isEmpty();}

D. s -> {return s.isEmpty();}

E. String s -> s.isEmpty()

F. (String s) -> s.isEmpty()

A, D, F. removeIf() expects a Predicate, which takes a parameter list of one parameter using the specified type. Options B and C are incorrect because they do not use the return keyword. It is required inside braces for lambda bodies. Option E is incorrect

because it is missing the parentheses around the parameter list. This is only optional for a single parameter with an inferred type.

29. Which lambda can replace the MySecret class to return the same value? (Chooseall that apply)

interface Secret {

String magic(double d);

}

class MySecret implements Secret {

public String magic(double d) {

return "Poof";

}

}

A. caller((e) -> "Poof");

B. caller((e) -> {"Poof"});

C. caller((e) -> { String e = ""; "Poof" });

D. caller((e) -> { String e = ""; return "Poof"; });

E. caller((e) -> { String e = ""; return "Poof" });

F. caller((e) -> { String f = ""; return "Poof"; });

A, F. Option B is incorrect because it does not use the return keyword. Options C, D,and E are incorrect because the variable e is already in use from the lambda and cannotbe redefined. Additionally, option C is missing the return keyword and option E is missing the semicolon.

## 5.Class Design-Inheritance-Super-this-abstract

public class Donkey {

}

public class Donkey {

public Donkey() {

}

}

public class Donkey {

public Donkey() {

super();

}

}

Make sure you understand the differences between these three Donkey class definitions and why Java will automatically convert them all to the last defi nition.

public class Mammal {

public Mammal(int age) {

}

}

public class Elephant extends Mammal { // DOES NOT COMPILE

}

In this example no constructor is defi ned within the Elephant class, so the compiler tries to insert a default no-argument constructor with a super() call, as it did in the Donkey example. The compiler stops, though, when it realizes there is no parent constructor that

takes no arguments.

public class Mammal {

public Mammal(int age) {

}

}

public class Elephant extends Mammal {

public Elephant() { // DOES NOT COMPILE

}

}

This code still doesn’t compile, though, because the compiler tries to insert the noargument super() as the fi rst statement of the constructor in the Elephant class, and there is no such constructor in the parent class.

public class Mammal {

public Mammal(int age) {

}

}

public class Elephant extends Mammal {

public Elephant() {

super(10);

}

}

This code will compile because we have added a constructor with an explicit call to a parent constructor. Note that the class Elephant now has a no-argument constructor even though its parent class Mammal doesn’t

4. If the parent doesn’t have a no-argument constructor and the child doesn’t define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child class.

5. If the parent doesn’t have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor

class Primate {

public Primate() {

System.out.println("Primate");

}

}

class Ape extends Primate {

public Ape() {

System.out.println("Ape");

}

}

public class Chimpanzee extends Ape {

public static void main(String[] args) {

new Chimpanzee();

}

}

The compiler fi rst inserts the super() command as the fi rst statement of both the Primate and Ape constructors. Next, the compiler inserts a default no-argument constructor in the Chimpanzee class with super() as the fi rst statement of the constructor. The code

will execute with the parent constructors called fi rst and yields the following output:

Primate

Ape

public class Canine {

public double getAverageWeight() {

return 50;

}

}

public class Wolf extends Canine {

public double getAverageWeight() {

return super.getAverageWeight()+20;

}

public static void main(String[] args) {

System.out.println(new Canine().getAverageWeight());

System.out.println(new Wolf().getAverageWeight());

}

}

In this example, in which the child class Wolf overrides the parent class Canine, the method getAverageWeight() runs without issue and outputs the following:

50.00

70.00

You might be wondering, was the use of super in the child’s method required? For example, what would the following code output if we removed the super keyword in the getAverageWeight() method of the Wolf class?

public double getAverageWeight() {

return getAverageWeight()+20; // INFINITE LOOP

The compiler performs the following checks when you override a nonprivate method:

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 or more accessible than the method in the parent class.

3. The method in the child class may not throw a New Exception or Larger Exception their parent class method.

4. If the method returns a value, it must be the same or a subclass of the method in parent class, known as covariant return types.

5. The method defined in the child class must be marked as static if it is marked as static in the parent class (method hiding). Likewise, the method must not be marked as static in the child class if it is not marked as static in the parent class (method

overriding).

**Case1 :**

public class InsufficientDataException extends Exception {}

public class Reptile {

protected boolean hasLegs() throws InsufficientDataException {

throw new InsufficientDataException();

}

protected double getWeight() throws Exception {

return 2;

}

}

public class Snake extends Reptile {

protected boolean hasLegs() { //No Issue

return false;

}

protected double getWeight() throws InsufficientDataException{//No Issue

return 2;

}

}

**Case:2**

public class InsufficientDataException extends Exception {}

public class Reptile {

protected double getHeight() throws InsufficientDataException {

return 2;

}

protected int getLength() {

return 10;

}

}

public class Snake extends Reptile {

protected double getHeight() throws Exception { // DOES NOT COMPILE

return 2;

}

protected int getLength() throws InsufficientDataException { // DOES NOT COMPILE

return 10;

}

}

The getHeight() method in the parent class throws an InsufficientDataException,whereas the method in the child class throws an Exception. Since Exception is not a subclass of InsufficientDataException, the third rule of overriding methods is

violated and the code will not compile. Coincidentally, Exception is a superclass of InsufficientDataException.

Next, the getLength() method doesn’t throw an exception in the parent class, but it does throw an exception, insufficientDataException, in the child class. In this manner,the child class defi nes a new exception

**Redeclaring private Methods**

In Java, it is not possible to override a private method in a parent class since the parent method is not accessible from

the child class.Java permits you to redeclare a new method in the child class with the same or modified signature as the method in the parent class. This method in the child class is a separate and independent method, unrelated to the parent version’s method

public class Camel {

private String getNumberOfHumps() {

return "Undefined";

}

}

public class BactrianCamel extends Camel {

private int getNumberOfHumps() {

return 2;

}

}

This code compiles without issue.

**Overriding vs. Hiding Methods**

At runtime the child version of an overridden method is always executed for an instance regardless of whether the method call is defined in a parent or child class method. In this manner, the parent method is never used unless an explicit call to the parent method is referenced, using the syntax ParentClassName.method().

Alternatively, at runtime the parent version of a hidden method is always executed if the call to the method is defi ned in the

parent class.

public class Marsupial {

public static boolean isBiped() {

return false;

}

public void getMarsupialDescription() {

System.out.println("Marsupial walks on two legs: "+isBiped());

}

}

public class Kangaroo extends Marsupial {

public static boolean isBiped() {

return true;

}

public void getKangarooDescription() {

System.out.println("Kangaroo hops on two legs: "+isBiped());

}

public static void main(String[] args) {

Kangaroo joey = new Kangaroo();

joey.getMarsupialDescription();

joey.getKangarooDescription();

}

}

In this example, the code compiles and runs without issue, outputting the following:

Marsupial walks on two legs: false

Kangaroo hops on two legs: true

Notice that isBiped() returns false in the parent class and true in the child class.In the fi rst method call, the parent method getMarsupialDescription() is used. TheMarsupial class only knows about isBiped() from its own class defi nition, so it outputs

false. In the second method call, the child executes a method of isBiped(), which hides the parent method’s version and returns true.

Contrast this fi rst example with the following example, which uses an overridden version of isBiped() instead of a hidden version:

class Marsupial {

public boolean isBiped() {

return false;

}

public void getMarsupialDescription() {

System.out.println("Marsupial walks on two legs: "+isBiped());

}

}

public class Kangaroo extends Marsupial {

public boolean isBiped() {

return true;

}

public void getKangarooDescription() {

System.out.println("Kangaroo hops on two legs: "+isBiped());

}

public static void main(String[] args) {

Kangaroo joey = new Kangaroo();

joey.getMarsupialDescription();

joey.getKangarooDescription();

}

}

This code also compiles and runs without issue, but it outputs slightly different text:

Marsupial walks on two legs: true

Kangaroo hops on two legs: true

In this example, the isBiped() method is overridden, not hidden, in the child class.Therefore, it is replaced at runtime in the parent class with the call to the child class’s method.Make sure you understand these examples as they show how hidden and overridden

methods are fundamentally different.

**Abstract Classes**

public [abstract/final] class ElephantSeal extends Seal {

// Methods and Variables defined here

}

public abstract class Turtle {

public abstract void swim() {} // DOES NOT COMPILE

public abstract int getAge() { // DOES NOT COMPILE

return 10;

}

}

The first method, swim(), doesn’t compile because two brackets are provided instead of a semicolon, and Java interprets this as providing a body to an abstract method. The second method, getAge(), doesn’t compile because it also provides a body to an abstract method.

abstract class and abstract methods cannot be marked as final A method may not be marked as both abstract and private.

Even with abstract methods, the 5 rules for overriding methods must be followed.

public abstract class Whale {

protected abstract void sing();

}

public class HumpbackWhale extends Whale {

private void sing() { // DOES NOT COMPILE

System.out.println("Humpback whale is singing");

}

}

overriding a method, the subclass cannot reduce the visibility of the parent method, sing().

public abstract class Animal {

public abstract String getName();

}

public class Walrus extends Animal { // DOES NOT COMPILE

}

public abstract class Eagle extends Animal {

}

In this example, we again have an abstract class Animal with a concrete subclass Walrus that doesn’t compile since it doesn’t implement a getName() method. We also have an abstract class Eagle, which like Walrus extends Animal and doesn’t provide an implementation for getName(). In this situation, Eagle does compile because it is marked as abstract

**Interfaces**

public interface CanBurrow {

public static final int MINIMUM\_DEPTH = 2;

public abstract int getMaximumDepth();

}

A class may implement multiple interfaces, each separated by a comma, such as in the following example:

public class Elephant implements WalksOnFourLegs, HasTrunk, Herbivore {

}

**Defining an Interface**

-

1. Interfaces cannot be instantiated directly.

2. An interface is not required to have any methods.

3. An interface may not be marked as final.

4. All top-level interfaces are assumed to have public or default access, and they must include the abstract modifier in their definition. Therefore, marking an interface as private, protected, or final will trigger a compiler error, since this is incompatible

with these assumptions.

5. All nondefault methods in an interface are assumed to have the modifiers abstract and public in their definition. Therefore, marking a method as private, protected, or final will trigger compiler errors as these are incompatible with the abstract and

public keywords.

The fourth rule doesn’t apply to inner interfaces, although inner classes and interfacesare not in scope for the OCA exam.

For example, the following two interface definitions are equivalent, as the compiler will convert them both to the second example:

public interface CanFly {

void fly(int speed);

abstract void takeoff();

public abstract double dive();

}

public abstract interface CanFly {

public abstract void fly(int speed);

public abstract void takeoff();

public abstract double dive();

}

Let’s take a look at an example that violates the assumed keywords:

private final interface CanCrawl { // DOES NOT COMPILE

private void dig(int depth); // DOES NOT COMPILE

protected abstract double depth(); // DOES NOT COMPILE

public final void surface(); // DOES NOT COMPILE

}

Every single line of this example doesn’t compile. The first line doesn’t compile for two reasons. First, it is marked as final, which cannot be applied to an interface since it conflicts with the assumed abstract keyword. Next, it is marked as private, which confl icts

with the public or default required access for interfaces.

what will happen if you define a class that inherits from two interfaces that contain the same

abstract method?

public interface Herbivore {

public void eatPlants();

}

public interface Omnivore {

public void eatPlants();

public void eatMeat();

}

In this scenario, the signatures for the two interface methods eatPlants() are compatible,so you can define a class that fulfi lls both interfaces simultaneously:

public class Bear implements Herbivore, Omnivore {

public void eatMeat() {

System.out.println("Eating meat");

}

public void eatPlants() {

System.out.println("Eating plants");

}

}

Why does this work? Remember that interface methods in this example are abstract and define the “behavior” that the class implementing the interface must have. If two abstract interface methods have identical behaviors—or in this case the same method

signature— creating a class that implements one of the two methods automatically implements the second method. In this manner, the interface methods are considered duplicates since they have the same signature

What happens if the two methods have different signatures? If the method name is the same but the input parameters are different, there is no conflict because this is considered a method overload. We demonstrate this principle in the following example:

public interface Herbivore {

public int eatPlants(int quantity);

}

public interface Omnivore {

public void eatPlants();

}

public class Bear implements Herbivore, Omnivore {

public int eatPlants(int quantity) {

System.out.println("Eating plants: "+quantity);

return quantity;

}

public void eatPlants() {

System.out.println("Eating plants");

}

}

In this example, we see that the class that implements both interfaces must provide implements of both versions of eatPlants(), since they are considered separate methods.Notice that it doesn’t matter if the return type of the two methods is the same or different,

because the compiler treats these methods as independent.

Unfortunately, if the method name and input parameters are the same but the return types are different between the two methods, the class or interface attempting to inherit both interfaces will not compile. The reason the code doesn’t compile has less to do with

interfaces and more to do with class design, as discussed in Chapter 4. It is not possible in Java to define two methods in a class with the same name and input parameters but different return types.

Given the following two interface definitions for Herbivore and Omnivore,the following code will not compile:

public interface Herbivore {

public int eatPlants();

}

public interface Omnivore {

public void eatPlants();

}

public class Bear implements Herbivore, Omnivore {

public int eatPlants() { // DOES NOT COMPILE

System.out.println("Eating plants: 10");

return 10;

}

public void eatPlants() { // DOES NOT COMPILE

System.out.println("Eating plants");

}

}

public interface Herbivore {

public int eatPlants();

}

public interface Omnivore {

public void eatPlants();

}

public interface Supervore extends Herbivore, Omnivore {} // DOES NOT COMPILE

public abstract class AbstractBear implements Herbivore, Omnivore {}

// DOES NOT COMPILE

Even without implementation details, the compiler detects the problem with the abstract definition and prevents compilation

**Interface Variables**

1. Interface variables are assumed to be public, static, and final. Therefore, marking a variable as private or protected will trigger a compiler error, as will marking any variable as abstract.

2. The value of an interface variable must be set when it is declared since it is marked as final.

Based on these rules, it should come as no surprise that the following entries will not compile:

public interface CanDig {

private int MAXIMUM\_DEPTH = 100; // DOES NOT COMPILE

protected abstract boolean UNDERWATER = false; // DOES NOT COMPILE

public static String TYPE; // DOES NOT COMPILE

}

**Default Interface Methods**

Java 8, the authors of Java have introduced a new type of method to an interface, referred to as a default method. A default method is a method defined within an interface with the default keyword in which a method body is provided.In this manner, classes have the option to override the default method if they need to, but they are not required to do so. If the class doesn’t override the method, the default implementation will be used. In this manner, the method definition is concrete, not abstract.

1. A default method may only be declared within an interface and not within a class or abstract class.

2. A default method must be marked with the default keyword. If a method is marked as default, it must provide a method body.

3. A default method is not be static, final, or abstract, as it may be used or overridden by a class that implements the interface.

4. Like all methods in an interface, a default method is assumed to be public and will not compile if marked as private or protected.

The following is an example of a default method defined in an interface:

public interface IsWarmBlooded {

boolean hasScales();

public default double getTemperature() {

return 10.0;

}

}

This example defines two interface methods, one is a normal abstract method and the other a default method. Note that both methods are assumed to be public, as all methods of an interface are all public. The first method is terminated with a semicolon and doesn’t provide a body, whereas the second default method provides a body. Any class that implements IsWarmBlooded may rely on the default implementation of getTemperature() or override the method and create its own version.

public interface Carnivore {

public default void eatMeat(); // DOES NOT COMPILE

public int getRequiredFoodAmount() { // DOES NOT COMPILE

return 13;

}

}

eatMeat(), doesn’t compile because it is marked as default but doesn’t provide a method body. The second method, getRequiredFood Amount(), also doesn’t compile because it provides a method body but is not marked with

the default keyword.

When an interface extends another interface that contains a default method, it may choose to ignore the default method, in which case the default implementation for the method will be used. Alternatively, the interface may override the definition of the default

method using the standard rules for method overriding, such as not limiting the accessibility of the method and using covariant returns.

public interface HasFins {

public default int getNumberOfFins() {

return 4;

}

public default double getLongestFinLength() {

return 20.0;

}

public default boolean doFinsHaveScales() {

return true;

}

}

public interface SharkFamily extends HasFins {

public default int getNumberOfFins() {

return 8;

}

public double getLongestFinLength();

public boolean doFinsHaveScales() { // DOES NOT COMPILE

return false;

}

}

SharkFamily, extends HasFins and overrides the default method getNumberOfFins() with a new method that returns a different value. Next, the SharkFamily interface replaces the default method getLongestFinLength() with a new abstract method, forcing any class that implements the SharkFamily interface to provide an implementation of the method. Finally, the SharkFamily interface overrides the doFinsHaveScales() method but doesn’t mark the method as default.

**Default Methods and Multiple Inheritance**

public interface Walk {

public default int getSpeed() {

return 5;

}

}

public interface Run {

public default int getSpeed() {

return 10;

}

}

public class Cat implements Walk, Run { // DOES NOT COMPILE

public static void main(String[] args) {

System.out.println(new Cat().getSpeed());

}

}

If a class implements two interfaces that have default methods with the same name and signature, the compiler will throw an error. There is an exception to this rule, though: if the subclass overrides the duplicate default methods, the code will compile without

issue—the ambiguity about which version of the method to call has been removed. For example, the following modified implementation of Cat will compile and output 1:

public class Cat implements Walk, Run {

public int getSpeed() {

return 1;

}

**Static Interface Methods**

Java 8 also now includes support for static methods within interface.A static method defined in an interface is not inherited in any classes that implement the interface.

Here are the static interface method rules you need to be familiar with:

1. Like all methods in an interface, a static method is assumed to be public and will not compile if marked as private or protected.

2. To reference the static method, a reference to the name of the interface must be used.

public interface Hop {

static int getJumpHeight() {

return 8;

}

}

public class Bunny implements Hop {

public void printDetails() {

System.out.println(Hop.getJumpHeight());

System.out.println(getJumpHeight()); // DOES NOT COMPILE

}

}

It follows, then, that a class that implements two interfaces containing static methods with the same signature will still compile at runtime, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name.

Contrast this with the behavior you saw for default interface methods in the previous section: the code would compile if the subclass overrode the default methods and would fail to compile.

**Polymorphisum**

Casting variables:

1. Casting an object from a subclass to a superclass doesn’t require an explicit cast.

2. Casting an object from a superclass to a subclass requires an explicit cast.

3. The compiler will not allow casts to unrelated types.

4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class

public class Bird {}

public class Fish {

public static void main(String[] args) {

Fish fish = new Fish();

Bird bird = (Bird)fish; // DOES NOT COMPILE

}

}

In this example, the classes Fish and Bird are not related through any class hierarchy that the compiler is aware of; therefore, the code will not compile.

public class Rodent {

}

public class Capybara extends Rodent {

public static void main(String[] args) {

Rodent rodent = new Rodent();

Capybara capybara = (Capybara)rodent; // Throws ClassCastException at runtime

}

}

This code creates an instance of Rodent and then tries to cast it to a subclass of Rodent, Capybara. Although this code will compile without issue, it will throw a ClassCastException at runtime since the object being referenced is not an instance of the Capybara class. The thing to keep in mind in this example is the object that was created is not related to the Capybara class in any way.

A virtual method is a method in which the specifi c implementation is not determined until runtime. In fact, all non-fi nal, nonstatic,

and non-private Java methods are considered virtual methods, since any of them can be overridden at runtime.

public class Bird {

public String getName() {

return "Unknown";

}

public void displayInformation() {

System.out.println("The bird name is: "+getName());

}

}

public class Peacock extends Bird {

public String getName() {

return "Peacock";

}

public static void main(String[] args) {

Bird bird = new Peacock();

bird.displayInformation();

}

}

This code compiles and executes without issue and outputs the following:

The bird name is: Peacock

Questions

1. What modifiers are implicitly applied to all interface methods? (Choose all that apply)

A. protected

B. public

C. static

D. void

E. abstract

F. default

B. All interface methods are implicitly public,Option E is a tricky one, because prior to Java 8 all interface methods would be assumed to be abstract. Since Java 8 now includes default and static methods and they are never abstract, you cannot assume the abstract modifier will be implicitly applied to all methods by the compiler

5. Which of the following may only be hidden and not overridden? (Choose all that apply)

A. private instance methods

B. protected instance methods

C. public instance methods

D. static methods

E. public variables

F. private variables

A, D, E, F. First off, options B and C are incorrect because protected and public methods may be overridden, not hidden. Option A is correct because private methods are always hidden in a subclass. Option D is also correct because static methods cannot be overridden, only hidden. Options E and F are correct because variables may only be hidden, regardless of the access modifier

7. Which of the following statements about polymorphism are true? (Choose all that apply)

A. A reference to an object may be cast to a subclass of the object without an explicit cast.

B. If a method takes a superclass of three objects, then any of those classes may be passed as a parameter to the method.

C. A method that takes a parameter with type java.lang.Object will take any reference.

D. All cast exceptions can be detected at compile-time.

E. By defining a public instance method in the superclass,guarantee the specific method will be called in the parent class at runtime

B, C. A reference to an object requires an explicit cast if referenced with a subclass, so option A is incorrect. If the cast is to a superclass reference, then an explicit cast is not required. Because of polymorphic parameters, if a method takes the superclass of

an object as a parameter, then any subclass references may be used without a cast, so option B is correct. All objects extend java.lang.Object, so if a method takes that type, any valid object, including null, may be passed; therefore, option C is correct.

Some cast exceptions can be detected as errors at compile-time, but others can only be detected at runtime, so D is incorrect. Due to the nature of polymorphism, a public instance method can be overridden in a subclass and calls to it will be replaced even in

the superclass it was defined, so E is incorrect

10. Which statements are true for both abstract classes and interfaces? (Choose all that apply)

A. All methods within them are assumed to be abstract.

B. Both can contain public static final variables.

C. Both can be extended using the extend keyword.

D. Both can contain default methods.

E. Both can contain static methods.

F. Neither can be instantiated directly.

G. Both inherit java.lang.Object.

B, C, E, F. Option A is wrong, because an abstract class may contain concrete methods.Since Java 8, interfaces may also contain concrete methods in form of static ordefault methods. Although all variables in interfaces are assumed to be public staticfinal, abstract classes may contain them as well, so option B is correct. Both abstractclasses and interfaces can be extended with the extends keyword, so option C is correct.Only interfaces can contain default methods, so option D is incorrect. Bothabstract classes and interfaces can contain static methods, so option E is correct. Bothstructures require a concrete subclass to be instantiated, so option F is correct. Finally,though an instance of an object that implements an interface inherits java.lang.Object, the interface itself doesn't; otherwise, Java would support multiple inheritancefor objects, which it doesn't. Therefore, option G is incorrect

12. What is the output of the following code?

1: interface Nocturnal {

2: default boolean isBlind() { return true; }

3: }

4: public class Owl implements Nocturnal {

5: public boolean isBlind() { return false; }

6: public static void main(String[] args) {

7: Nocturnal nocturnal = (Nocturnal)new Owl();

8: System.out.println(nocturnal.isBlind());

9: }

10: }

A. true

B. false

C. The code will not compile because of line 2.

D. The code will not compile because of line 5.

E. The code will not compile because of line 7.

F. The code will not compile because of line 8.

B. This code compiles and runs without issue, outputting false, so option B is thecorrect answer. The first declaration of isBlind() is as a default interface method,assumed public. The second declaration of isBlind() correctly overrides the defaultinterface method. Finally, the newly created Owl instance may be automatically cast toa Nocturnal reference without an explicit cast, although adding it doesn’t break thecode.

16. What is the output of the following code?

1: abstract class Reptile {

2: public final void layEggs() { System.out.println("Reptile laying eggs");

}

3: public static void main(String[] args) {

4: Reptile reptile = new Lizard();

5: reptile.layEggs();

6: }

7: }

8: public class Lizard extends Reptile {

9: public void layEggs() { System.out.println("Lizard laying eggs"); }

10: }

A. Reptile laying eggs

B. Lizard laying eggs

C. The code will not compile because of line 4.

D. The code will not compile because of line 5.

E. The code will not compile because of line 9.

E. The code doesn’t compile, so options A and B are incorrect. The issue with line 9 is that layEggs() is marked as final in the superclass Reptile, which means it cannot be overridden. There are no errors on any other lines, so options C and D are incorrect

18. What is the output of the following code? (Choose all that apply)

1: interface Aquatic {

2: public default int getNumberOfGills(int input) { return 2; }

3: }

4: public class ClownFish implements Aquatic {

5: public String getNumberOfGills() { return "4"; }

6: public String getNumberOfGills(int input) { return "6"; }

7: public static void main(String[] args) {

8: System.out.println(new ClownFish().getNumberOfGills(-1));

9: }

10: }

A. 2

B. 4

C. 6

D. The code will not compile because of line 5.

E. The code will not compile because of line 6.

F. The code will not compile because of line 8.

E. The code doesn’t compile because line 6 contains an incompatible override of thegetNumberOfGills(int input) method defined in the Aquatic interface. In particular,int and String are not covariant returns types, since int is not a subclass of String.Note that line 5 compiles without issue; getNumberOfGills() is an overloaded methodthat is not related to the parent interface method that takes an int value.

19. Which of the following statements can be inserted in the blank so that the code will compile successfully? (Choose all that apply)

public class Snake {}

public class Cobra extends Snake {}

public class GardenSnake {}

public class SnakeHandler {

private Snake snake;

public void setSnake(Snake snake) { this.snake = snake; }

public static void main(String[] args) {

new SnakeHandler().setSnake( );

}

}

A. new Cobra()

B. new GardenSnake()

C. new Snake()

D. new Object()

E. new String("Snake")

F. null

A, C, F. First off, Cobra is a subclass of Snake, so option A can be used. GardenSnake isnot defined as a subclass of Snake, so it cannot be used and option B is incorrect. Theclass Snake is not marked as abstract, so it can be instantiated and passed, so optionC is correct. Next, Object is a superclass of Snake, not a subclass, so it also cannot beused and option D is incorrect. The class String is unrelated in this example, so optionE is incorrect. Finally, a null value can always be passed as an object value, regardlessof type, so option F is correct.

## 6.Exception Handling

**java.lang.Object**

**|**

**java.lang.Throwable**

/ \

**java.lang.Exception** **java.lang.Error**

|

**java.lang.RuntimeException**

A **checked exception** includes Exception and all subclasses that do not extend RuntimeException. Checked exceptions tend to be more anticipated—for example, trying to read a file that doesn’t exist.

-Java has a rule called the handle or declare rule.

-For checked exceptions, Java requires the code to either handle them or declare them in the method signature.

**Runtime exceptions** (unchecked exceptions) tend to be unexpected but not necessarily fatal.For example, accessing an invalid array index is unexpected.

Error means something went so horribly wrong that your program should not attempt to recover from it. ex, the disk drive “disappeared.”

**Runtime Exceptions**

Runtime exceptions extend RuntimeException. They don’t have to be handled or declared.They thrown by the programmer/by JVM.

-ArithmeticException: Thrown by the JVM when code attempts to divide by zero

-ArrayIndexOutOfBoundsException: Thrown by the JVM when code uses an illegal index to access an array

-ClassCastException-

-IllegalArgumentException Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument

-NullPointerException Thrown by the JVM when there is a null reference where an object is required

-NumberFormatException Thrown by the programmer string doesn’t have an appropriate format

**Checked Exceptions**

They can be thrown by the programmer or by the JVM.Common runtime exceptions include the following:

-FileNotFoundException Thrown programmatically when code tries to reference a file that does not exist

-IOException Thrown programmatically when there’s a problem reading or writing a file

**Errors**

**Errors extend the Error class**. They are thrown by the JVM and should not be handled ordeclared. Errors are rare

-ExceptionInInitializerError Thrown by the JVM when a static initializer throws an exception and doesn’t handle it

-StackOverflowError Thrown by the JVM when a method calls itself too many times

-NoClassDefFoundError Thrown by the JVM when a class that the code uses is available at compile time but not runtime

**Calling Methods That Throw Exceptions**

class NoMoreCarrotsException extends Exception {}

public class Bunny {

public static void main(String[] args) {

eatCarrot();// DOES NOT COMPILE

}

private static void eatCarrot() throws NoMoreCarrotsException {

}

}

The problem is that NoMoreCarrotsException is a checked exception. Checked exceptions must be handled or declared. The code would compile if we changed the main() method to either of these:

public static void main(String[] args)

throws NoMoreCarrotsException {// declare exception

eatCarrot();

}

public static void main(String[] args) {

try {

eatCarrot();

} catch (NoMoreCarrotsException e ) {// handle exception

System.out.print("sad rabbit");

}

}

**Subclasses**

1.When a class overrides a method from a superclass or implements a method from an interface, it’s not allowed to add new checked exceptions to the method signature.

class Hopper {

public void hop() { }

}

class Bunny extends Hopper {

public void hop() throws CanNotHopException { } // DOES NOT COMPILE

}

2.A subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them.

class Hopper {

public void hop() throws CanNotHopException { }

}

class Bunny extends Hopper {

public void hop() { } //Valid, already Handled in Super class

}

3.class is allowed to declare a subclass of an exception type. The idea is the same. The superclass or interface has already taken care of a broader type. Here’s an example:

class Hopper {

public void hop() throws Exception { }

}

class Bunny extends Hopper {

public void hop() throws CanNotHopException { }

}

-Inside catch we can throw the Exception, it will print the Stacktrace with No ISSUIE

public class Mouse {

public String name;

public void run() {

System.out.print("1");

try {

System.out.print("2");

name.toString();

System.out.print("3");

} catch (NullPointerException e) {

System.out.print("4");

throw e;

}

System.out.print("5");

}

public static void main(String[] args) {

Mouse jerry = new Mouse();

jerry.run();

System.out.print("6");

} }

Questions

5. Which of the following exceptions are thrown by the JVM? (Choose all that apply)

A. ArrayIndexOutOfBoundsException

B. ExceptionInInitializerError

C. java.io.IOException

D. NullPointerException

E. NumberFormatException

A, B, D. java.io.IOException is thrown by many methods in the java.io package,but it is always thrown programmatically. The same is true for NumberFormatException;it is thrown programmatically by the wrapper classes of java.lang. The otherthree exceptions are all thrown by the JVM when the corresponding problem arises

9. What is the output of the following program?

1: public class Laptop {

2: public void start() {

3: try {

4: System.out.print("Starting up ");

5: throw new Exception();

6: } catch (Exception e) {

7: System.out.print("Problem ");

8: System.exit(0);

9: } finally {

10: System.out.print("Shutting down ");

11: }

12: }

13: public static void main(String[] args) {

14: new Laptop().start();

15: } }

A. Starting up

B. Starting up Problem

C. Starting up Problem Shutting down

D. Starting up Shutting down

E. The code does not compile.

F. An uncaught exception is thrown

B. The main() method invokes start on a new Laptop object. Line 4 prints Starting up; then line 5 throws an Exception. Line 6 catches the exception, line 7 prints Problem, and then line 8 calls System.exit, which terminates the JVM. The finally block does not execute because the JVM is no longer running.

10. What is the output of the following program?

1: public class Dog {

2: public String name;

3: public void parseName() {

4: System.out.print("1");

5: try {

6: System.out.print("2");

7: int x = Integer.parseInt(name);

8: System.out.print("3");

9: } catch (NumberFormatException e) {

10: System.out.print("4");

11: }

12: }

13: public static void main(String[] args) {

14: Dog leroy = new Dog();

15: leroy.name = "Leroy";

16: leroy.parseName();

17: System.out.print("5");

18: } }

A. 12

B. 1234

C. 1235

D. 124

E. 1245

F. The code does not compile.

E. The parseName method is invoked within main() on a new Dog object. Line 4 prints 1. The try block executes and 2 is printed. Line 7 throws a NumberFormatException, so line 8 doesn’t execute. The exception is caught on line 9, and line 10 prints 4. Because the exception is handled, execution resumes normally. parseName runs to completion,

14. Which of the following can be inserted on line 8 to make this code compile? (Choose allthat apply)

7: public void ohNo() throws IOException {

8: // INSERT CODE HERE

9: }

A. System.out.println("it's ok");

B. throw new Exception();

C. throw new IllegalArgumentException();

D. throw new java.io.IOException();

E. throw new RuntimeException();

A, C, D, E. A method that declares an exception isn’t required to throw one, making option A correct. Runtime exceptions can be thrown in any method, making options C and E correct. Option D matches the exception type declared and so is also correct. Option B is incorrect because a broader exception is not allowed.

17. Which of the following can be inserted into Lion to make this code compile? (Choose allthat apply)

class HasSoreThroatException extends Exception {}

class TiredException extends RuntimeException {}

interface Roar {

void roar() throws HasSoreThroatException;

}

class Lion implements Roar {// INSERT CODE HERE

}

A. public void roar(){}

B. public void roar() throws Exception{}

C. public void roar() throws HasSoreThroatException{}

D. public void roar() throws IllegalArgumentException{}

E. public void roar() throws TiredException{}

A, C, D, E. The method is allowed to throw no exceptions at all, making option A correct.It is also allowed to throw runtime exceptions, making options D and E correct.Option C is also correct since it matches the signature in the interface.

18. Which of the following are true? (Choose all that apply)

A. Checked exceptions are allowed to be handled or declared.

B. Checked exceptions are required to be handled or declared.

C. Errors are allowed to be handled or declared.

D. Errors are required to be handled or declared.

E. Runtime exceptions are allowed to be handled or declared.

F. Runtime exceptions are required to be handled or declared.

A, B, C, E. Checked exceptions are required to be handled or declared. Runtime exceptions are allowed to be handled or declared. Errors are allowed to be handled or declared, but this is bad practice

20. What does the output of the following contain? (Choose all that apply)

12: public static void main(String[] args) {

13: System.out.print("a");

14: try {

15: System.out.print("b");

16: throw new IllegalArgumentException();

17: } catch (IllegalArgumentException e) {

18: System.out.print("c");

19: throw new RuntimeException("1");

20: } catch (RuntimeException e) {

21: System.out.print("d");

22: throw new RuntimeException("2");

23: } finally {

24: System.out.print("e");

25: throw new RuntimeException("3");

26: }

27: }

A. abce

B. abde

C. An exception with the message set to "1"

D. An exception with the message set to "2"

E. An exception with the message set to "3"

F. Nothing; the code does not compile.

A, E. The code begins normally and prints a on line 13, followed by b on line 15. On line 16, it throws an exception that’s caught on line 17. Remember, only the most specific matching catch is run. Line 18 prints c, and then line 19 throws another exception.Only one catch will run at a time Regardless, the finally block runs, printing e. Since the finally block also throws an exception, that’s the one printed.

# Oca warp up questions

## 1.Java Basics

22. Given the following class definition, which is the only line that does not contain a compilation error?

1: public ThisClassDoesNotCompile {

2: double int count;

3: void errors() {}

4: static void private limit; }

A. Line 1

B. Line 2

C. Line 3

D. Line 4

22. C. Line 1 is missing the class keyword. Line 2 contains two types for the same variable.Line 3 is a valid definition for a method, making C the correct answer. Finally, line 4 contains an access modifier, private, after the return type, which is not allowed. In addition,void is an invalid type for variables.

24. Which of the following is not a property of a JVM?

A. It prevents Java bytecode from being easily decoded/decompiled.

B. It supports platform independence.

C. It manages memory for the application.

D. It translates Java instructions to machine instructions.

A. Options B, C, and D are each correct statements about JVMs. Option A is incorrect.Not only is it not a statement about JVMs, it is actually false as Java bytecode can often be easily decoded/decompiled

42. is the technique of structuring programming data as a unit consisting of

attributes, with actions defined on the unit.

A. Encapsulation

B. Object orientation

C. Platform independence

D. Polymorphism

42. B. Object-oriented programming is the technique of structuring data into objects, which may contain data and a set of actions that operate on the data, making Option B the correct answer.

47. Which statements about calling the compilation command javac and the execution command

java are true?

I. java may use a period . to separate packages.

II. javac takes a .java file and returns a .class file.

III. java may use a slash (/) to separate packages.

A. I only

B. II only

C. I and II

D. I, II, and III

C. The javac command takes a text-based .java file and returns a binary bytecode.class file, making II a true statement. The java command uses a period (.) to separate packages, not a slash (/), making I a true statement and III a false statement. For these reasons, Option C is the correct answer.

50. Which statement about the JVM is true?

A. The JVM schedules garbage collection on a predictable schedule.

B. The JVM ensures that the application will always terminate.

C. The JVM requires a properly defined entry point method to execute the application.

D. A Java compiled code can be run on any computer

C. Garbage collection can happen at any time while an application is running, especially ifthe available memory suddenly becomes low, making Option A incorrect. Option B is alsoincorrect, since it is trivial to create a Java application with an infinite loop that never terminates.Option D is incorrect because the computer must be able to run the JVM in orderto execute a Java class. Option C is the only correct answer, as the JVM does require anentry point method to begin executing the application.