**Preparação para o segundo simulado**

Exceções (3)

Asserts (2)

Wrappers(3)

Gargage (3)

StringBuilder e StringBuffer (1)

IO (5)

Serialização (2)

**Exceções**

11. class A {  
12. public void process() { System.out.print("A,"); }  
13. class B extends A {  
14. public void process() throws IOException {  
15. super.process();  
16. System.out.print("B,");  
17. throw new IOException();  
18. }  
19. public static void main(String[] args) {  
20. try { new B().process(); }  
21. catch (IOException e) { System.out.println("Exception"); }}

**What is the result?**

A. Exception  
B. A,B,Exception  
C. Compilation fails because of an error in line 20.  
D. Compilation fails because of an error in line 14.  
E. A NullPointerException is thrown at runtime.

**Resposta: D (Não pode lançar mais exceções)**

import java.io.\*;  
class Master {  
String doFileStuff() throws FileNotFoundException { return "a"; }  
}  
class Slave extends Master {  
public static void main(String[] args) {  
String s = null;  
try { s = new Slave().doFileStuff();  
} catch ( Exception x) {  
s = "b"; }  
System.out.println(s);  
}  
// insert code here  
}

**Which, inserted independently at // insert code here, will compile, and produce the output**  
b? (Choose all that apply.)

A. String doFileStuff() { return "b"; }  
B. String doFileStuff() throws IOException { return "b"; }  
C. String doFileStuff(int x) throws IOException { return "b"; }  
D. String doFileStuff() throws FileNotFoundException { return "b"; }  
E. String doFileStuff() throws NumberFormatException { return "b"; }  
F. String doFileStuff() throws NumberFormatException,  
FileNotFoundException { return "b"; }

**Answer:**  
**->  A , D, E,** and **F** are correct. It’s okay for an overriding method to throw the same  
exceptions, narrower exceptions, or no exceptions. And it’s okay for the overriding  
method to throw any runtime exceptions.  
->   **B** is incorrect, because the overriding method is trying to throw a broader exception.  
**C** is incorrect. This method doesn’t override, so the output is a.

1. class Ping extends Utils {  
2. public static void main(String [] args) {  
3. Utils u = new Ping();  
4. System.out.print(u.getInt(args[0]));  
5. }  
6. int getInt(String arg) {  
7. return Integer.parseInt(arg);  
8. }  
9. }  
10. class Utils {  
11. int getInt(String x) throws Exception { return 7; }  
12. }

**And the following three possible changes:**

**C1. Declare that main() throws an Exception.**  
**C2. Declare that Ping.getInt() throws an Exception.**  
**C3. Wrap the invocation of getInt() in a try / catch block.**

**Which change(s) allow the code to compile? (Choose all that apply.)**

A. Just C1 is sufficient.  
B. Just C2 is sufficient.  
C. Just C3 is sufficient.  
D. Both C1 and C2 are required.  
E. Both C1 and C3 are required.  
F. Both C2 and C3 are required.  
G. All three changes are required.

**Answer:**  
-> **A** and **C** are correct. Remember that line 4 is making a polymorphic call so the compiler knows that an exception might be thrown. If C1 is implemented the exception has been sufficiently declared, and if C3 is implemented the exception has been sufficiently handled. C2 is not necessary in either case.  
-> **B, D, E, F,** and **G** are incorrect based on the above.

**Asserts**

**23. int z = 5;**

**24.**

**25. public void stuff1(int x) {**

**26. assert (x > 0);**

**27. switch(x) {**

**28. case 2: x = 3;**

**29. default: assert false; } }**

**30.**

**31. private void stuff2(int y) { assert (y < 0); }**

**32.**

**33. private void stuff3() { assert (stuff4()); }**

**34.**

**35. private boolean stuff4() { z = 6; return false; }**

**Which statement is true?**

A. All of the assert statements are used appropriately.

B. Only the assert statement on line 31 is used appropriately.

C. The assert statements on lines 29 and 31 are used appropriately.

D. The assert statements on lines 26 and 29 are used appropriately.

E. The assert statements on lines 29 and 33 are used appropriately.

F. The assert statements on lines 29, 31, and 33 are used appropriately.

G. The assert statements on lines 26, 29, and 31 are used appropriately.

**Answer: C**

**1. public class Test {**

**2.**

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

**4. boolean assert = true;**

*5.* **if(assert) {**

**6. System.out.println(”assert is true”);**

**7. }**

**8. }**

**9.**

**10. }**

**Given:**

**javac -source 1.3 Test.java**

**What is the result?**

A. Compilation fails.

B. Compilation succeeds with errors.

C. Compilation succeeds with warnings.

D. Compilation succeeds without warnings or errors.

**Answer: C**

**Wrappers**

The code is given:  
  
3.  public class Bertha {  
4.    static String s = "";  
5.    public static void main(String[] args) {  
6.        int x = 4; Boolean y = true; short[] sa = {1,2,3};  
7.        doStuff(x, y);  
8.        doStuff(x);  
9.        doStuff(sa, sa);  
10.       System.out.println(s);  
11.   }  
12.   static void doStuff(Object o) { s += "1"; }  
13.   static void doStuff(Object... o) { s += "2"; }  
14.   static void doStuff(Integer... i) { s += "3"; }  
15.   static void doStuff(Long L) { s += "4"; }  
16. }  
  
What is the result?

A. 212  
B. 232  
C. 234  
D. 312  
E. 332  
F. 334  
G. Compilation fails

**Answer:  A is correct.**  
  
**Explanation:** It's legal to autobox and then widen. The first call to doStuff(), x is boxed from int to an Integer then passes these two objects (Integer,Boolean) to varargs (Object.... 0).  
  
The second call cannot widen and then box (making the Long method unusable), so it boxes the int to an Integer. And then it goes to doStuff(Object o) because Integer is-A Object. As always, a  
var-args method will be chosen only if no non-var-arg method is possible.  
  
The third call is passing two objects—they are of type 'short array.'

1) Widening beats boxing  
2) Widening beats var-args

3) Boxing beats var-args

Now see the rules for overloading methods using widening,boxing and varargs...

1) Primitive widening uses the "smallest" method argument possible.

2) Used individually, boxing and var-args are compatible with overloading.

3) You CANNOT widen from one wrapper type to another. (IS-A fails.)

4) You CANNOT widen and then box. (An int can't become a Long.)

5) You can box and then widen. (An int can become an Object, via Integer.)

6) You can combine var-args with either widening or boxing

public class Boxing1 {  
public static void main(String[] args) {  
Integer i = null;  
method(i);  
}  
static void method(int k){  
System.out.println(k);  
}  
}  
  
What is the output of the above program?

1)Null Pointer Exception  
2)Number Format Exception  
3)null  
4)0

Resposta: Null Pointer Exception  
  
Explanation :  
When wrapper type is null, we cannot do the boxing conversion. It will throw the NullpointerException when its is trying the convert to primitive type.

public class Boxing2 {  
public static void main(String[] args) {  
byte b = 10;  
method(b);  
}  
static void method(int i){  
System.out.println("Primitivae Type call");  
}  
static void method(Integer i){  
System.out.println("Wrapper Type Call");  
}  
}

What will be output for the above program?

1)Wrapper Type Call  
2)Primitive Type Call  
3)Compiler Error  
4)Compiles fine, throws runtime exception

2)Primitive Type Call  
  
Explanation :  
When comes to method overloading in Java 5.0, it works like the previous versions. First JVM will check for the matching primitive types, then it will search for the Wrapper types.

**Garbage**

class CardBoard {  
Short story = 5;  
CardBoard go(CardBoard cb) {  
cb = null;  
return cb;  
}  
public static void main(String[] args) {  
CardBoard c1 = new CardBoard();  
CardBoard c2 = new CardBoard();  
CardBoard c3 = c1.go(c2);  
c1 = null;  
// do Stuff  
} }

**When // doStuff is reached, how many objects are eligible for GC?**

A. 0  
B. 1  
C. 2  
D. Compilation fails.  
E. It is not possible to know.  
F. An exception is thrown at runtime.

**Answer:**  
-> **C** is correct. Only one CardBoard object (c1) is eligible, but it has an associated Short wrapper object that is also eligible.  
->**A, B, D, E,** and **F** are incorrect based on the above.

11. public void genNumbers() {  
12. ArrayList numbers = new ArrayList();  
13. for (int i=0; i<10; i++) {  
14. int value = i \* ((int) Math.random());  
15. Integer intObj = new Integer(value);  
16. numbers.add(intObj);  
17. }  
18. System.out.println(numbers);  
19. }

**Which line of code marks the earliest point that an object referenced by intObj becomes a candidate for**  
**garbage collection?**

A. Line 16  
B. Line 17  
C. Line 18  
D. Line 19  
E. The object is NOT a candidate for garbage collection.

**Answer: D, ou seja, só quando acaba o método**

1. public class GC {  
2. private Object o;  
3. private void doSomethingElse(Object obj) { o = obj; }  
4. public void doSomething() {  
5. Object o = new Object();  
6. doSomethingElse(o);  
7. o = new Object();  
8. doSomethingElse(null);  
9. o = null;  
10. }  
11. }

**When the doSomething method is called, after which line does the Object created in line 5 become**  
**available for garbage collection?**

A. Line 5  
B. Line 6  
C. Line 7  
D. Line 8  
E. Line 9  
F. Line 10  
----------------------------------------------------------------------------------------------------------

Answer: D

----------------------------------------------------------------------------------------------------------

**StringBuilder e StringBuffer**

What is the difference between String Buffer and String Builder APIs?

1. String Buffer is the base class class for String Builder.
2. String Builder provides a more simplified API over the String Buffer class
3. The only difference between String Buffer and String Builder class is that String Buffer is thread safe whereas String Builder is not.
4. The only difference between String Buffer and String Builder class is that String Builder is thread safe whereas String Buffer is not.

Resposta: 3

**IO**

10. class MakeFile {

11. public static void main(String[] args) {

12. try {

13. File directory = new File(”d”);

14. File file = new File(directory,”f”);

*15.* if(!file.exists()) {

16. file.createNewFile();

17. }

18. } catch (IOException e) {

19. e.printStackTrace

20. }

21. }

**22. }**

**The current directory does NOT contain a directory named “d.”**

**Which three are true? (Choose three.)**

A. Line 16 is never executed.

B. An exception is thrown at runtime.

C. Line 13 creates a File object named “d.”

D. Line 14 creates a File object named “f.’

E. Line 13 creates a directory named “d” in the file system.

F. Line 16 creates a directory named “d” and a file ‘f’ within it in the

file system.

G. Line 14 creates a file named ‘f’ inside of the directory named “d” in

the file system.

**Answer: BCD**

**Given:**

**bw is a reference to a valid BufferedWriter And the snippet:**

**15. BufferedWriter b1 = new BufferedWriter(new File("f"));**

**16. BufferedWriter b2 = new BufferedWriter(new FileWriter("f1"));**

**17. BufferedWriter b3 = new BufferedWriter(new PrintWriter("f2"));**

**18. BufferedWriter b4 = new BufferedWriter(new BufferedWriter(bw));**

**What is the result?**

A. Compilation succeeds.

B. Compilation fails due only to an error on line 15.

C. Compilation fails due only to an error on line 16.

D. Compilation fails due only to an error on line 17.

E. Compilation fails due only to an error on line 18.

F. Compilation fails due to errors on multiple lines.

**Answer:**

-> **B** is correct. BufferedWriters can be constructed only by wrapping a Writer. Lines 16, 17, and 18 are correct because BufferedWriter, FileWriter, and PrintWriter all extend Writer. (Note: BufferedWriter is a decorator class. Decorator classes are used extensively in the java.io package to allow you to extend the functionality of other classes.)

-> **A, C, D, E,** and **F** are incorrect based on the above. (Objective 3.2)

What happens when this code is compiled and executed? (1 correct answer)

void test() {

FileWriter writer = new FileWriter("/fun.log");

writer.write("Hello!");

writer.close();

}

1. A file fun.log is created with the content “Hello!”.
2. A file fun.log is created but it’s empty, because flush() was not called.
3. A runtime exception is thrown because flush() was not called.
4. Compilation fails.

Resposta: D os métodos de escrita e leitura levantam IOException

Consider a file fun.log whose first line is “Hello!”. What happens when this code is compiled and executed? (1 correct answer)

void test() throws IOException {

File file = new File("C:/fun.log");

BufferedReader reader = new BufferedReader(file);

System.out.println(reader.readLine());

}

1. A runtime exception is thrown because the file already exists.
2. It prints “Hello!”.
3. It prints “null”.
4. Compilation fails.

Resposta D: BufferedReader tem que receber um Reader no construtor

What happens when this code is compiled and executed? (1 correct answer)

void test() throws IOException {

for (int index = 1; index <= 2; index++) {

PrintWriter writer = new PrintWriter("/apa");

writer.print("apa");

writer.close();

}

}

1. A file apa is created with content “apa”.
2. A file apa is created with content “apaapa”.
3. Two files are created.
4. An exception is thrown at runtime.

Resposta: A

**Serialização**

10. public class Foo implements java.io.Serializable {

11. private int x;

12. public int getX() { return x; }

12.publicFoo(int x){this.x=x; }

13. private void writeObject( ObjectOutputStream s)

14. throws IOException {

15. // insert code here

16. }

17. }

**Which code fragment, inserted at line 15, will allow Foo objects to be**

**correctly serialized and deserialized?**

A. s.writeInt(x);

B. s.serialize(x);

C. s.writeObject(x);

D. s.defaultWriteObject();

**Resposta: D**

import java.io.\*;

class Player {

Player() { System.out.print("p"); }

}

class CardPlayer extends Player implements Serializable {

CardPlayer() { System.out.print("c"); }

public static void main(String[] args) {

CardPlayer c1 = new CardPlayer();

try {

FileOutputStream fos = new FileOutputStream("play.txt");

ObjectOutputStream os = new ObjectOutputStream(fos);

os.writeObject(c1);

os.close();

FileInputStream fis = new FileInputStream("play.txt");

ObjectInputStream is = new ObjectInputStream(fis);

CardPlayer c2 = (CardPlayer) is.readObject();

is.close();

} catch (Exception x ) { }

}

}

**What is the result?**

A. pc B. pcc

C. pcp D. pcpc

**E. Compilation fails. F. An exception is thrown at runtime.**

**Answer:**

-> **C** is correct. It's okay for a class to implement Serializable even if its superclass doesn't. However, when you deserialize such an object, the non-serializable superclass must run its constructor. Remember, constructors don't run on deserialized classes that implement Serializable.

**-> A, B, D, E, and F are incorrect based on the above.**

**\**