

OCAJP-1Z0-808 Certification Preparation



By Hiba FATNASSI

Assessment Test

1. What is the result of the following class? (Choose all that apply)

```
1: public class _C {  
2: private static int $;  
3: public static void main(String[] main) {  
4:     String a_b;  
5:     System.out.print($);  
6:     System.out.print(a_b);  
7: } }
```

- A. Compiler error on line 1.
- B. Compiler error on line 2.
- C. Compiler error on line 4.
- D. Compiler error on line 5.
- E. Compiler error on line 6.
- F. Onull
- G. nullnull

Assessment Test

2. What is the result of the following code?

```
String s1 = "Java";
String s2 = "Java";
StringBuilder sb1 = new StringBuilder();
sb1.append("Ja").append("va");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(sb1.toString() == s1);
System.out.println(sb1.toString().equals(s1));
```

- A. true is printed out exactly once.
- B. true is printed out exactly twice.
- C. true is printed out exactly three times.
- D. true is printed out exactly four times.
- E. The code does not compile.

Assessment Test

3. What is the output of the following program?

```
1: public class FeedingSchedule {  
2: public static void main(String[] args) {  
3: boolean keepGoing = true;  
4: int count = 0;  
5: int x = 3;  
6: while(count++ < 3) {  
7: int y = (1 + 2 * count) % 3;  
8: switch(y) {  
9: default:  
10: case 0: x -= 1; break;  
11: case 1: x += 5;  
12: }  
13: }
```

```
14: System.out.println(x);
```

```
15: } }
```

A. 4

B. 5

C. 6

D. 7

E. 13

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

Assessment Test

4. What is the result of the following program?

```
1: public class MathFunctions {  
2: public static void addToInt(int x, int amountToAdd) {  
3: x = x + amountToAdd;  
4: }  
5: public static void main(String[] args) {  
6: int a = 15;  
7: int b = 10;  
8: MathFunctions.addToInt(a, b);  
9: System.out.println(a); } }
```

- A. 10
- B. 15
- C. 25
- D. Compiler error on line 3.
- E. Compiler error on line 8.
- F. None of the above.

Assessment Test

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

```
public interface CanSwim {}  
public class Amphibian implements CanSwim {}  
class Tadpole extends Amphibian {}  
public class FindAllTadPole {  
    public static void main(String[] args) {  
        List<Tadpole> tadpoles = new ArrayList<Tadpole>();  
        for(Amphibian amphibian : tadpoles) {  
            _____ tadpole = amphibian;  
        } } }
```

- A. CanSwim
- B. Long
- C. Amphibian
- D. Tadpole
- E. Object

Chapter 1: Java building blocks

Chapter 1: Java building blocks

Structure of Java Class

- What is a class in Java ?
- A **java class** is a **model** or **template** that contains all the information of objects which are the set of attributes (variables) and methods.

class = stat + behavior

- What is an object ?

- **An object** is an instance of a single class.

```
1: public class Animal {  
2:     String name;  
3:     public String getName() {  
4:         return name;  
5:     }  
6:     public void setName(String newName) {  
7:         name = newName;  
8:     }  
9: }
```

Chapter 1: Java building blocks

Structure of Java Class

- What's a comment?
- It's a code that belongs to the class but is not executable.
- It is used to clarify the code.
- In Java, there are 3 types of comments:
 - Single line comment.

// comment until end of line

- Multi-line comment.

```
/* Multiple  
 * line comment  
 */
```

- Javadoc multi-line comment.

```
/**  
 * Javadoc multiple-line comment  
 * @author Jeanne and Scott  
 */
```

Chapter 1: Java building blocks

Structure of Java Class

- As a practice, can you identify the type of comment for each of these five statements? Is this a comment on a single line or multiple lines?

Multi-line comment

```
/*  
 * // anteater  
 */
```

Single line comment

```
// bear
```

Single line comment

```
// // cat
```

Single line comment

```
// /* dog */
```

Multi-line comment

```
/* elephant */
```

Multi-line comment

```
/*  
 * /* ferret */  
 */
```

Chapter 1: Java building blocks

Class vs. file

- A file includes one or more classes.
- It must include only one public class.
- He takes the name of his public class as his own name.

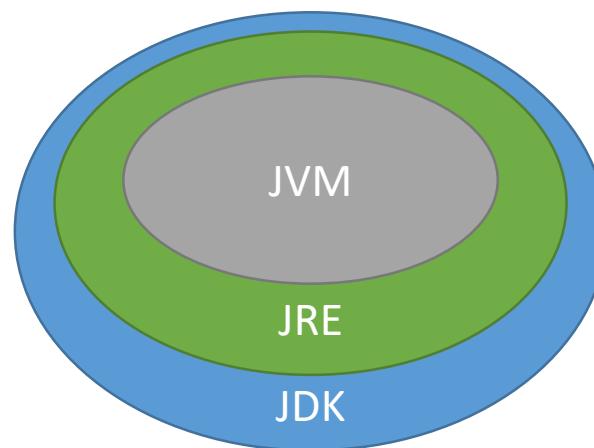
The name of the file is : **Animal.java**

```
1: public class Animal {  
2:     private String name;  
3: }  
4: class Animal2 {  
5: }
```

Chapter 1: Java building blocks

Main method

- **JVM:** Java Virtual Machine is an abstract machine that provides the runtime environment where you can run the java byte code.
- **JRE:** Java Runtime Environment is the runtime environment, it implements the JVM, as it provides APIs and other files that the JVM uses at runtime.
- **JDK:** Java Development Kit is a necessary tool for the compilation of java code.



Chapter 1: Java building blocks

Main method

- A Java program begins execution with its *main()* *method*.
- A *main()* method is the gateway between the startup of a Java process, which is managed by the *Java Virtual Machine* (JVM), and the beginning of the programmer's code.
- To compile Java code, the file must have the extension `.java`.
- The name of the file must match the name of the class.
- The result is a file of *bytecode* by the same name, but with a `.class` filename extension.
- **Note that we must omit the extension.`.class` to execute the file.**

Chapter 1: Java building blocks

Main method

1. Javac Zoo.java
2. Java Zoo

```
1: public class Zoo {  
2:     public static void main(String[] args) {  
3:         //  
4:     }  
5: }
```

- Signature of the Main method:
 - Access Modifier: Public declares the exposure level of this method to potential callers of the program.
 - The static keyword binds a method to its class so that it can be called simply by the name of the class. Java does not need to create an object to call this method.

Chapter 1: Java building blocks

Main method

- If a main () method is not present in the class we name with the .java extension, the process throws an error and terminates. Even if a main () method is present, Java throws an exception if it is not static. A non-static main () method might as well be invisible from the point of view of the JVM.
- The keyword void represents the type of return. A method that does not return data, silently returns control to the caller.
- Finally, we come to the parameter list of the main () method, represented by an array of java.lang.String objects. In practice, you can write String [] args, String args [] or String ... args. The args variable name indicates that this list contains values (arguments) at startup of the JVM.

```
1: public class Zoo {  
2:     public static void main(String[] args) {  
3:         //  
4:     }  
5: }
```

Chapter 1: Java building blocks

Main method

- To execute this method one must type:

```
$ javac Zoo.java  
$ java Zoo Bronx Zoo
```

- The program will display:

Bronx
Zoo

```
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 arguments. Spaces are used to separate arguments. If you want include spaces in an argument, you must use quotation marks.

```
$ javac Zoo.java  
$ java Zoo "San Diego" Zoo
```

Chapter 1: Java building blocks

Main method

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

```
$ javac Zoo.java  
$ java Zoo Zoo 2
```



```
Zoo  
2
```

- Finally, what will happen if you do not pass enough arguments?

```
$ javac Zoo.java  
$ java Zoo Zoo
```



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

```
public class Zoo {  
    public static void main(String[] args) {  
        System.out.println(args[0]);  
        System.out.println(args[1]);  
    } }
```

Chapter 1: Java building blocks

Package Declarations and Imports

- Java code is organized into folders called packages. To reference classes in other packages, you use an import declaration. The "*" (wildcard) character ending an import declaration means that you want to import all classes in this package. This does not include packages that are inside that one.
- `java.lang` is a special package that does not need to be imported.

```
public class ImportExample {  
    public static void main(String[] args) {  
        Random r = new Random(); // DOES NOT COMPILE  
        System.out.println(r.nextInt(10));  
    }  
}
```

Chapter 1: Java building blocks

Package Declarations and Imports

```
1: import java.lang.System;           Redundancy must be eliminated
2: import java.lang.*;
3: import java.util.Random;
4: import java.util.*;
5: public class ImportExample {
6:   public static void main(String[] args) {
7:     Random r = new Random();
8:     System.out.println(r.nextInt(10));
9:   }
10: }
```

Chapter 1: Java building blocks

Package Declarations and Imports

```
public class InputImports {  
    public void read(Files files) {  
        Paths.get("name");  
    }  
}  
  
import java.nio.*; // NO GOOD – a wildcard only matches class names, not "file.*Files"  
  
import java.nio.files.Paths.*; // NO GOOD – you cannot import methods only class names  
  
import java.nio.*.*; //NO GOOD – you can only have one wildcard and it must be at the end
```



```
import java.nio.file.*;  
import java.nio.file.Files;  
import java.nio.file.Paths;
```

Chapter 1: Java building blocks

Package Declarations and Imports

```
import java.util.*;
import java.sql.* ,  
  
public class Conflicts {
    Date date;
    // some more code
}
```

Chapter 1: Java building blocks

Code Formatting on the Exam

- Not all questions will include the imports. If the exam isn't asking about imports in the question, it will often omit the imports to save space.
- This means when you do see the line number 1 or no line numbers at all, you have to make sure imports aren't missing.

```
6: public void method(ArrayList list) {  
7:   if (list.isEmpty()) { System.out.println("e");  
8:   } else { System.out.println("n");  
9:   } }
```

```
1: public class LineNumbers {  
2:   public void method(ArrayList list) {  
3:     if (list.isEmpty()) { System.out.println("e");  
4:     } else { System.out.println("n");  
5:     } }
```

Chapter 1: Java building blocks

Creating Objects

- Constructors create Java objects. A constructor is a method that matches the class name and omits the return type. When an object is instantiated, fields and code blocks are initialized first. Then the constructor is executed.

```
Random r = new Random();
```

```
public class Chick {  
    public Chick() {  
        System.out.println("in constructor");  
    }  
}
```

```
public void Chick() { } // NOT A CONSTRUCTOR
```

- The purpose of a constructor is to initialize fields, although you can put any code in there. Another way to initialize fields is to do so directly on the line on which they're declared. This example shows both approaches:

```
public class Chicken {  
    int numEggs = 0; // initialize on line  
    String name;  
    public Chicken() {  
        name = "Duke"; // initialize in constructor  
    } }
```

Chapter 1: Java building blocks

Instance Initializer Blocks

```
3: public static void main(String[] args) {  
4:   { System.out.println("Feathers"); }  
5: }  
6: { System.out.println("Snowy"); }
```

3 code blocks and 1 instance initializer

Chapter 1: Java building blocks

Order of Initialization

- **Fields and instance initializer blocks are run in the order in which they appear in the file.**
- **The constructor runs after all fields and instance initializer blocks have run.**

```
1: public class Chick {  
2:     private String name = "Fluffy";  
3:     { System.out.println("setting field"); }  
4:     public Chick() {  
5:         name = "Tiny";  
6:         System.out.println("setting constructor");  
7:     }  
8:     public static void main(String[] args) {  
9:         Chick chick = new Chick();  
10:        System.out.println(chick.name); } }
```

Chapter 1: Java building blocks

Order of Initialization

Order matters for the fields and blocks of code.

```
{ System.out.println(name); } // DOES NOT COMPILE  
private String name = "Fluffy";
```

You can't refer to a variable before it has been initialized.

```
public class Egg {  
    public Egg() {  
        number = 5;  
    }  
    public static void main(String[] args) {  
        Egg egg = new Egg();  
        System.out.println(egg.number);  
    }  
    private int number = 3;  
    { number = 4; } }
```

Fields and blocks are run first in order, setting number to 3 and then 4. Then the constructor runs, setting number to 5.

Chapter 1: Java building blocks

Distinguishing Between Object References and Primitives (Primitive types)

Keyword	Type	Example
boolean	true or false	true
byte	8-bit integral value	123
short	16-bit integral value	123
int	32-bit integral value	123
long	64-bit integral value	123
float	32-bit floating-point value	123.45f
double	64-bit floating-point value	123.456
char	16-bit Unicode value	'a'

Chapter 1: Java building blocks

Distinguishing Between Object References and Primitives (Primitive types)

```
long max = 3123456789;
```



Compilation fail

```
long max = 3123456789L;
```

Another way to specify numbers is to change the “base.” When you learned how to count, you studied the digits 0–9. This numbering system is called base 10 since there are 10 numbers. It is also known as the decimal number system. Java allows you to specify digits in several other formats:

- 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

Chapter 1: Java building blocks

Distinguishing Between Object References and Primitives (Primitive types)

The last thing you need to know about numeric literals is a feature added in Java 7. You can have underscores in numbers to make them easier to read:

```
int million1 = 1000000;  
int million2 = 1_000_000;
```

You can add underscores anywhere except at the beginning of a literal, the end of a literal, right before a decimal point, or right after a decimal point.point décimal, ou juste après un point décimal.

```
double notAtStart = _1000.00;  
double notAtEnd = 1000.00_;  
double notByDecimal = 1000_.00;  
double annoyingButLegal = 1_00_0.0_0;
```

Chapter 1: Java building blocks

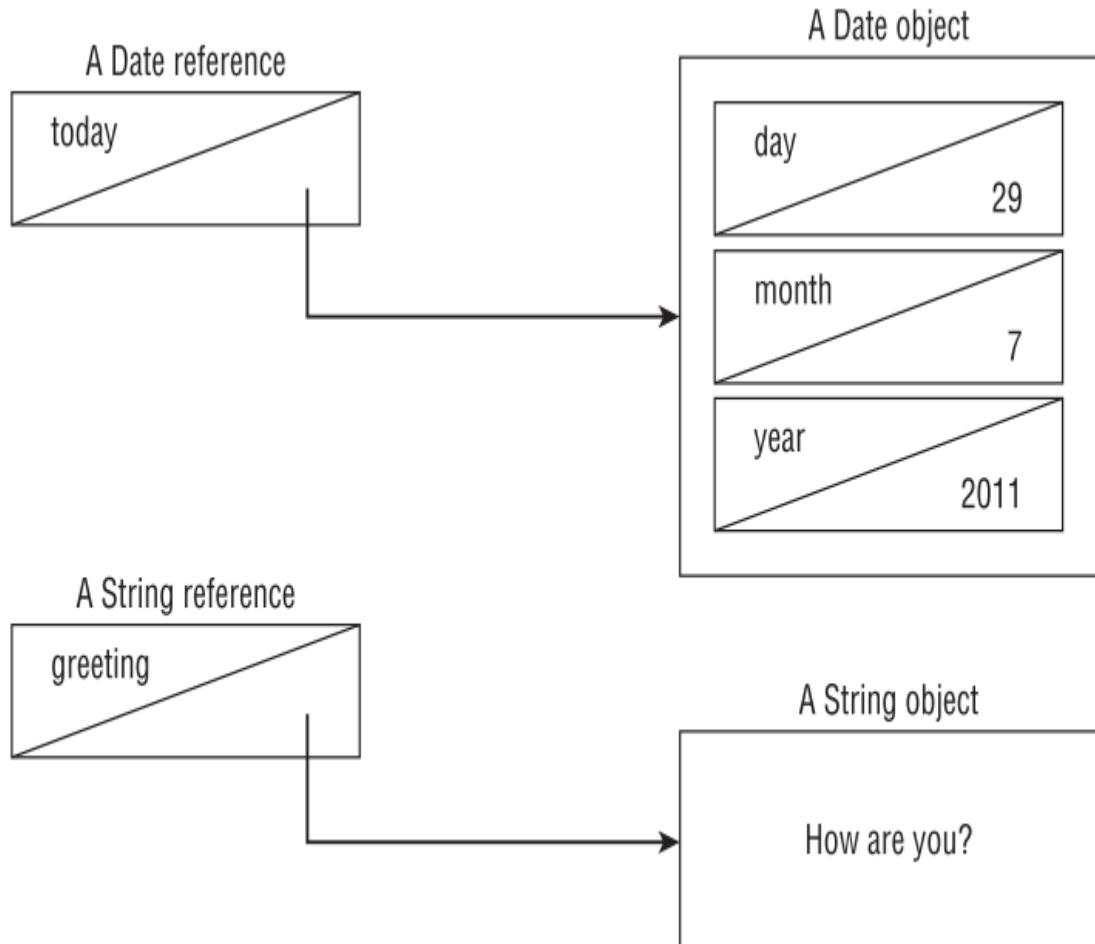
Distinguishing Between Object References and Primitives (Reference types)

- A *reference type* refers to an object (an instance of a class). Unlike primitive types that hold their values in the memory where the variable is allocated, references do not hold the value of the object they refer to.
- Instead, a reference “points” to an object by storing the memory address where the object is located, a concept referred to as a *pointer*. **Unlike other languages, Java does not allow you to learn what the physical memory address is. You can only use the reference to refer to the object.**

Chapter 1: Java building blocks

Distinguishing Between Object References and Primitives (Reference types)

```
java.util.Date today;  
String greeting;  
  
today = new java.util.Date();  
greeting = "How are you?";
```



Chapter 1: Java building blocks

Declaring Multiple Variables

- Several variables of the **same type** can be declared in a single line.

```
String s1, s2;
```

```
boolean b1, b2;
```

```
String s3 = "yes", s4 = "no";
```

```
String s1 = "1", s2;
```

```
int i1, i2, i3 = 0;
```

```
double d1, double d2;
```

```
int num, String value;
```

```
int i1; int i2;
```

```
int i3; i4;
```

Chapter 1: Java building blocks

Identifiers

- It probably comes as no surprise that Java has precise rules about *identifier* names.
- Luckily, the same rules for identifiers apply to anything you are free to name, including variables, methods, classes, and fields.
- There are only three rules to remember for legal identifiers:
 - The name must begin with a letter or the symbol \$ or _.
 - Subsequent characters may also be numbers.
 - You cannot use the same name as a Java *reserved word*. As you might imagine, a reserved word is a keyword that Java has reserved so that you are not allowed to use it.
- Remember that Java is case sensitive, so you can use versions of the keywords that only differ in case. Please don't, though.

Chapter 1: Java building blocks

Identifiers

abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto*	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

• Chapter 1: Java building blocks

Identifiers

- Prepare to be tested on these rules. The following examples are legal:
 - okidentifier
 - \$OK2Identifier
 - _alsoOK1d3ntifi3r
 - __SStillOkbutKnotsonice\$
- These examples are not legal:
 - 3DPointClass // identifiers cannot begin with a number
 - hollywood@vine // @ is not a letter, digit, \$ or _
 - *\$coffee // * is not a letter, digit, \$ or _
 - public // public is a reserved word

• Chapter 1: Java building blocks

Understanding Default Initialization of Variables

- Before you can use a variable, it needs a value. Some types of variables get this value set automatically, and others require the programmer to specify it. In the following sections, we'll look at the differences between the defaults for local, instance, and class variables.

• Chapter 1: Java building blocks

Understanding Default Initialization of Variables (variable locale)

- A *local variable* is a variable defined within a method. Local variables must be initialized before use. They do not have a default value and contain garbage data until initialized.
- The compiler will not let you read an uninitialized value:

```
4: public int notValid() {  
5:     int y = 10;  
6:     int x;  
7:     int reply = x + y; // DOES NOT COMPILE  
8:     return reply;  
9: }
```

- Chapter 1: Java building blocks

Understanding Default Initialization of Variables (variable locale)

```
public void findAnswer(boolean check) {  
    int answer;  
    if (check) {  
        onlyOneBranch = 1;  
        answer = 1;  
    } else {  
        answer = 2;  
    }  
    System.out.println(answer);  
    System.out.println(onlyOneBranch); // DOES NOT COMPILE  
}
```

• Chapter 1: Java building blocks

Understanding Default Initialization of Variables (variable de classe et variable d'instance)

- Variables that are not local variables are known as *instance variables* or *class variables*. Instance variables are also called fields. Class variables are shared across multiple objects.
- You can tell a variable is a class variable because it has the keyword static before it.
- Instance and class variables do not require you to initialize them. As soon as you declare these variables, they are given a default value.

• Chapter 1: Java building blocks

Understanding Default Initialization of Variables (variable de classe et variable d'instance)

Variable type	Default initialization value
boolean	false
byte, short, int, long	0 (in the type's bit-length)
float, double	0.0 (in the type's bit-length)
char	'\u0000' (NUL)
All object references (everything else)	null

• Chapter 1: Java building blocks

Understanding Variable Scope

- You've learned that local variables are declared within a method. How many local variable do you see in this example?

```
public void eat(int piecesOfCheese) {  
    int bitesOfCheese = 1;  
}
```

- There are two local variables in this method. `bitesOfCheese` is declared inside the method. `piecesOfCheese` is called a method parameter. It is also local to the method. Both of these variables are said to have **a scope local** to the method. This means they cannot be used outside the method.
- Local variables can never have a scope larger than the method they are defined in. However, they can have a smaller scope. Consider this example:

```
3: public void eatIfHungry(boolean hungry) {      6: } // bitesOfCheese goes out of scope here  
4: if (hungry) {                                7: System.out.println(bitesOfCheese); // DOES NOT COMPILE  
5:     int bitesOfCheese = 1;                      8: }
```

• Chapter 1: Java building blocks

Understanding Variable Scope

```
1: public class Mouse {  
2:     static int MAX_LENGTH = 5;  
3:     int length;  
4:  
5:     public void grow(int inches) {  
6:         if (length < MAX_LENGTH) {  
7:             int newSize = length + inches;  
8:             length = newSize;  
9:         }  
10:    }
```

- Local variables—**in scope from declaration to end of block**
- Instance variables—**in scope from declaration until object is garbage collected**
- Class variables—**in scope from declaration until program ends**

• Chapter 1: Java building blocks

Ordering Elements in a Class

- Now that you've seen the most common parts of a class, let's take a look at the correct order to type them into a file. Comments can go anywhere in the code. Beyond that, you need to memorize the rules in next table :

Element	Example	Required?	Where does it go?
Package declaration	package abc;	No	First line in the file
Import statements	import java.util.*;	No	Immediately after the package
Class declaration	public class C	Yes	Immediately after the import
Field declarations	int value;	No	Anywhere inside a class
Method declarations	void method()	No	Anywhere inside a class

• Chapter 1: Java building blocks

Ordering Elements in a Class

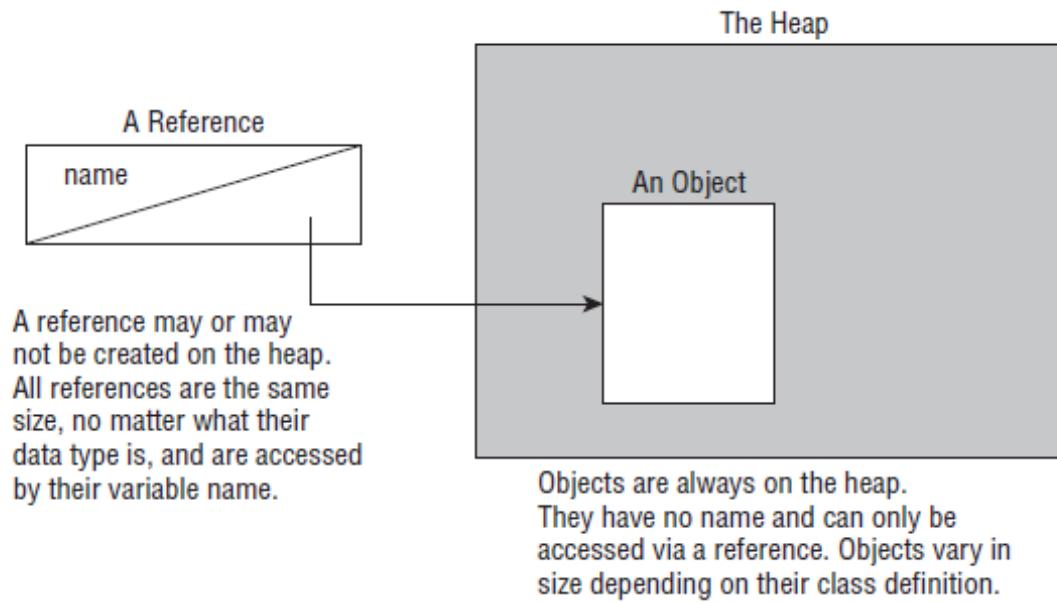
```
package structure; // package must be first non-comment
import java.util.*; // import must come after package
public class Meerkat { // then comes the class
    double weight; // fields and methods can go in either order
    public double getWeight() {
        return weight;
    }
    double height; // another field - they don't need to be together
}
```

```
import java.util.*;
package structure; // DOES NOT COMPILE
String name; // DOES NOT COMPILE
public class Meerkat { }
```

• Chapter 1: Java building blocks

Destroying Objects

- Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program.
- An object is no longer reachable when one of two situations occurs:
 - The object no longer has any references pointing to it.
 - All references to the object have gone out of scope.



• Chapter 1: Java building blocks

Destroying Objects

- Java provides a method called `System.gc()`. It meekly *suggests* that now might be a good time for Java to kick off a garbage collection run. Java is free to ignore the request.
- Java allows objects to implement a method called `finalize()` that might get called. This method gets called if the garbage collector tries to collect the object. If the garbage collector doesn't run, the method doesn't get called. If the garbage collector fails to collect the object and tries to run it again later, the method doesn't get called a second time.

• Chapter 1: Java building blocks

Destroying Objects

```
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:     } }
```

• Chapter 1: Java building blocks

Benefits of Java

- **Object Oriented** Java is an object-oriented language, which means all code is defined in classes and most of those classes can be instantiated into objects.
- **Encapsulation** Java supports access modifiers to protect data from unintended access and modification.
- **Platform Independent** Java is an interpreted language because it gets compiled to bytecode.
- **Robust** One of the major advantages of Java over C++ is that it prevents memory leaks. Java manages memory on its own and does garbage collection automatically.
- **Simple** Java was intended to be simpler than C++. In addition to eliminating pointers, it got rid of operator overloading.
- **Secure** Java code runs inside the JVM. This creates a sandbox that makes it hard for Java code to do evil things to the computer it is running on.

QCM Chapitre 1

OCAJP-IZ0-808

Q1

A,B,E

- Which of the following are valid Java identifiers? (Choose all that apply)
- **A.** A\$B
- **B.** _helloWorld
- **C.** true
- **D.** java.lang
- **E.** Public
- **F.** 1980_s

Q2

D

- What is the output of the following program?
- 1: public class WaterBottle {
- 2: private String brand;
- 3: private boolean empty;
- 4: public static void main(String[] args) {
- 5: WaterBottle wb = new WaterBottle();
- 6: System.out.print("Empty = " + wb.empty);
- 7: System.out.print(", Brand = " + wb.brand);
- 8: } }
- **A.** Line 6 generates a compiler error.
- **B.** Line 7 generates a compiler error.
- **C.** There is no output.
- **D.** Empty = false, Brand = null
- **E.** Empty = false, Brand =
- **F.** Empty = null, Brand = null

Q3

B,D,E

- Which of the following are true?
(Choose all that apply)
- 4: short numPets = 5;
- 5: int numGrains = 5.6;
- 6: String name = "Scruffy";
- 7: numPets.length();
- 8: numGrains.length();
- 9: name.length();
- **A.** Line 4 generates a compiler error.
- **B.** Line 5 generates a compiler error.
- **C.** Line 6 generates a compiler error.
- **D.** Line 7 generates a compiler error.
- **E.** Line 8 generates a compiler error.
- **F.** Line 9 generates a compiler error.
- **G.** The code compiles as is.

Q4

A,B

- Given the following class, which of the following is true? (Choose all that apply)
- 1: public class Snake {
- 2:
- 3: public void shed(boolean time) {
- 4:
- 5: if (time) {
- 6:
- 7: }
- 8: System.out.println(result);
- 9:
- 10: }
- 11: }
- **A.** If String result = "done"; is inserted on line 2, the code will compile.
- **B.** If String result = "done"; is inserted on line 4, the code will compile.
- **C.** If String result = "done"; is inserted on line 6, the code will compile.
- **D.** If String result = "done"; is inserted on line 9, the code will compile.
- **E.** None of the above changes will make the code compile.

Q5

C,D

- Given the following classes, which of the following can independently replace INSERT
- IMPORTS HERE to make the code compile? (Choose all that apply)
 - package aquarium;
 - public class Tank { }
 - package aquarium.jellies;
 - public class Jelly { }
 - package visitor;
- INSERT IMPORTS HERE
 - public class AquariumVisitor {
 - public void admire(Jelly jelly) { } }
 - **A.** import aquarium.*;
 - **B.** import aquarium.*.Jelly;
 - **C.** import aquarium.jellies.Jelly;
 - **D.** import aquarium.jellies.*;
 - **E.** import aquarium.jellies.Jelly.*;
 - **F.** None of these can make the code compile.

Q6

E

- 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.

Q7

A,B,C

- 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

Q8

B

- Given the following class, which of the following calls print out Blue Jay? (Choose all that apply)
- public class BirdDisplay {
- public static void main(String[] name) {
- System.out.println(name[1]);
- } }
- A. java BirdDisplay Sparrow Blue Jay
- B. java BirdDisplay Sparrow "Blue Jay"
- C. java BirdDisplay Blue Jay Sparrow
- D. java BirdDisplay "Blue Jay" Sparrow
- E. java BirdDisplay.class Sparrow "Blue Jay"
- F. java BirdDisplay.class "Blue Jay" Sparrow
- G. Does not compile.

Q9

A,C,D,E

- Which of the following legally fill in the blank so you can run the main() method from the command line? (Choose all that apply)
- public static void main()
- **A.** String[] _names
- **B.** String[] 123
- **C.** String abc[]
- **D.** String _Names[]
- **E.** String... \$n
- **F.** String names
- **G.** None of the above.

Q10

E

- 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.

Q11

C,D

- Which of the following are true? (Choose all that apply)
- **A.** An instance variable of type double defaults to null.
- **B.** An instance variable of type int defaults to null.
- **C.** An instance variable of type String defaults to null.
- **D.** An instance variable of type double defaults to 0.0.
- **E.** An instance variable of type int defaults to 0.0.
- **F.** An instance variable of type String defaults to 0.0.
- **G.** None of the above.

Q12

G

- 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.

Q13

A,D

- Which of the following are true? (Choose all that apply)
- **A.** An instance variable of type boolean defaults to false.
- **B.** An instance variable of type boolean defaults to true.
- **C.** An instance variable of type boolean defaults to null.
- **D.** An instance variable of type int defaults to 0.
- **E.** An instance variable of type int defaults to 0.0.
- **F.** An instance variable of type int defaults to null.
- **G.** None of the above.

Q14

D

- Given the following class in the file
/my/directory/named/A/Bird.java
:
• INSERT CODE HERE
- public class Bird { }
- Which of the following replaces
INSERT CODE HERE if we compile
from /my/directory?
• (Choose all that apply)
 - A. package my.directory.named.a;
 - B. package my.directory.named.A;
 - C. package named.a;
 - D. package named.A;
 - E. package a;
 - F. package A;
 - G. Does not compile.

Q15

A,E

- Which of the following lines of code compile? (Choose all that apply)
- **A.** int i1 = 1_234;
- **B.** double d1 = 1_234_.0;
- **C.** double d2 = 1_234._0;
- **D.** double d3 = 1_234.0_;
- **E.** double d4 = 1_234.0;
- **F.** None of the above.

Q16

B,C,D

- Given the following class, which of the following lines of code can replace INSERT CODE
 - } }
- HERE to make the code compile?
(Choose all that apply)
 - A. int amount = 9L;
 - B. int amount = 0b101;
 - C. int amount = 0xE;
 - D. double amount = 0xE;
 - E. double amount = 1_2_.0_0;
 - F. int amount = 1_2_;
 - G. None of the above.
- public class Price {
- public void admission() {
- INSERT CODE HERE
- System.out.println(amount);

Q17

A,E

- Which of the following are true?
(Choose all that apply)
- public class Bunny {
- public static void main(String[] args) {
- Bunny bun = new Bunny();
- } }
- **A.** Bunny is a class.
- **B.** bun is a class.
- **C.** main is a class.
- **D.** Bunny is a reference to an object.
- **E.** bun is a reference to an object.
- **F.** main is a reference to an object.
- **G.** None of the above.

Q18

C,D,E

- Which represent the order in which the following statements can be assembled into a program that will compile successfully?
(Choose all that apply)
- A: class Rabbit {}
- B: import java.util.*;
- C: package animals;
- D. B, C, A
- E. C, A
- F. A, C
- G. A, B

Q19

B,D

- 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

Q20

B,E

- What is true about the following code? (Choose all that apply)
- public class Bear {
- protected void finalize() {
- System.out.println("Roar!");
- }
- public static void main(String[] args) {
- Bear bear = new Bear();
- bear = null;
- System.gc();
- } }
- A. finalize() is guaranteed to be called.
- B. finalize() might or might not be called
- C. finalize() is guaranteed not to be called.
- D. Garbage collection is guaranteed to run.
- E. Garbage collection might or might not run.
- F. Garbage collection is guaranteed not to run.
- G. The code does not compile.

Q21

A

- What does the following code output?
 - 1: public class Salmon {
 - 2: int count;
 - 3: public void Salmon() {
 - 4: count = 4;
 - 5: }
 - 6: public static void main(String[] args) {
 - 7: Salmon s = new Salmon();
 - 8: System.out.println(s.count);
 - 9: } }
- **A.** 0
 - **B.** 4
 - **C.** Compilation fails on line 3.
 - **D.** Compilation fails on line 4.
 - **E.** Compilation fails on line 7.
 - **F.** Compilation fails on line 8.

Q22

B,E

- Which of the following are true statements? (Choose all that apply)
- **A.** Java allows operator overloading.
- **B.** Java code compiled on Windows can run on Linux.
- **C.** Java has pointers to specific locations in memory.
- **D.** Java is a procedural language.
- **E.** Java is an object-oriented language.
- **F.** Java is a functional programming language.

Q23

C,D

- Which of the following are true? (Choose all that apply)
- **A.** javac compiles a .class file into a .java file.
- **B.** javac compiles a .java file into a .bytecode file.
- **C.** javac compiles a .java file into a .class file.
- **D.** Java takes the name of the class as a parameter.
- **E.** Java takes the name of the .bytecode file as a parameter.
- **F.** Java takes the name of the .class file as a parameter.

Tests

1 minute and 30 seconds

Q1

- Given the following application, what is the expected output?
- public class Keyboard {
- private boolean numLock = true;
- static boolean capLock = false;
- public static void main(String... shortcuts) {
- System.out.print(numLock+""+capLock);
- }
- }
- A. true false
- B. false false
- C. It does not compile.
- D. It compiles but throws an exception at runtime.

Q2

- What is the result of compiling and executing the following class?
- public class RollerSkates {
- static int wheels = 1;
- int tracks = 5;
- public static void main(String[] arguments) {
- RollerSkates s = new RollerSkates();
- int feet=4, tracks = 15;
- System.out.print(feet + tracks + s.wheels);
- }
- }
- **A.** The code does not compile.
- **B.** 5
- **C.** 10
- **D.** 20

Q3

- What is the result of compiling and executing the following class?
- package sports;
- public class Bicycle {
- String color = "red";
- private void printColor(String color) {
- color = "purple";
- System.out.print(color);
- }
- public static void main(String[] rider) {
- new Bicycle().printColor("blue");
- }
- }
- **A.** red
- **B.** purple
- **C.** blue
- **D.** It does not compile.

Q4

- What is the result of compiling and executing the following application?
- package forecast;
- public class Weather {
- private static boolean heatWave = true;
- public static void main() {
- boolean heatWave = false;
- System.out.print(heatWave);
- }
- }
- **A.** true
- **B.** false
- **C.** It does not compile.
- **D.** It compiles but throws an error at runtime.

Q5

- 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.

Q6

- How many of the String objects are eligible for garbage collection right before the end of
 - the main method?
 - `public static void main(String[] fruits) {`
 - `String fruit1 = new String("apple");`
 - `String fruit2 = new String("orange");`
- `String fruit3 = new String("pear");`
- `fruit3 = fruit1;`
- `fruit2 = fruit3;`
- `fruit1 = fruit2;`
- `}`
- **A.** None
- **B.** One
- **C.** Two
- **D.** Three

Q7

- What is true of the finalize() method?
- **A.** It may be called zero or one times.
- **B.** It may be called zero or more times.
- **C.** It will be called exactly once.
- **D.** It may be called one or more times.

Q8

- `Integer integer = new Integer(4);`
- `System.out.print(integer.byteValue());`
- `System.out.print("-");`
- `int i = new Integer(4);`
- `System.out.print(i.byteValue());`
- **A.** 4-0
- **B.** 4-4
- **C.** The code does not compile.
- **D.** The code compiles but throws an exception at runtime.

Q9

- What does the following output?
- 1: public class InitOrder {
- 2: public String first = "instance";
- 3: public InitOrder() {
- 4: first = "constructor";
- 5: }
- 6: { first = "block"; }
- 7: public void print() {
- 8: System.out.println(first);
- 9: }
- 10: public static void main(String... args) {
- 11: new InitOrder().print();
- 12: }
- 13: }
- **A.** block
- **B.** constructor
- **C.** instance
- **D.** The code does not compile.

Q10

- Which of the following is the output of this code, assuming it runs to completion?
- package store;
- public class Toy {
- public void play() {
- System.out.print("play-");
- }
- public void finalizer() {
- System.out.print("clean-");
- }
- public static void main(String[] fun) {
- Toy car = new Toy();
- car.play();
- System.gc();
- Toy doll = new Toy();
- doll.play();
- }
- }
- **A.** play-
- **B.** play-play-
- **C.** play-clean-play-
- **D.** play-play-clean-clean

Corrections

Q1

C

- Given the following application, what is the expected output?
- public class Keyboard {
- private boolean numLock = true;
- static boolean capLock = false;
- public static void main(String... shortcuts) {
- System.out.print(numLock+""+capLock);
- }
- }
- A. true false
- B. false false
- C. It does not compile.
- D. It compiles but throws an exception at runtime.

Q2

D

- What is the result of compiling and executing the following class?
- public class RollerSkates {
- static int wheels = 1;
- int tracks = 5;
- public static void main(String[] arguments) {
- RollerSkates s = new RollerSkates();
- int feet=4, tracks = 15;
- System.out.print(feet + tracks + s.wheels);
- }
- }
- **A.** The code does not compile.
- **B.** 5
- **C.** 10
- **D.** 20

Q3

B

- What is the result of compiling and executing the following class?
- package sports;
- public class Bicycle {
- String color = "red";
- private void printColor(String color) {
- color = "purple";
- System.out.print(color);
- }
- public static void main(String[] rider) {
- new Bicycle().printColor("blue");
- }
- }
- **A.** red
- **B.** purple
- **C.** blue
- **D.** It does not compile.

Q4

D

- What is the result of compiling and executing the following application?
- package forecast;
- public class Weather {
- private static boolean heatWave = true;
- public static void main() {
- boolean heatWave = false;
- System.out.print(heatWave);
- }
- }
- **A.** true
- **B.** false
- **C.** It does not compile.
- **D.** It compiles but throws an error at runtime.

Q5

C

- 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.

Q6

C

- How many of the String objects are eligible for garbage collection right before the end of the main method?
- public static void main(String[] fruits) {
- String fruit1 = new String("apple");
- String fruit2 = new String("orange");
- String fruit3 = new String("pear");
- fruit3 = fruit1;
- fruit2 = fruit3;
- fruit1 = fruit2;
- }
- **A.** None
- **B.** One
- **C.** Two
- **D.** Three

Q7

A

- What is true of the finalize() method?
- **A.** It may be called zero or one times.
- **B.** It may be called zero or more times.
- **C.** It will be called exactly once.
- **D.** It may be called one or more times.

Q8

C

- `Integer integer = new Integer(4);`
- `System.out.print(integer.byteValue());`
- `System.out.print("-");`
- `int i = new Integer(4);`
- `System.out.print(i.byteValue());`
- **A.** 4-0
- **B.** 4-4
- **C.** The code does not compile.
- **D.** The code compiles but throws an exception at runtime.

Q9

B

- What does the following output?
- 1: public class InitOrder {
- 2: public String first = "instance";
- 3: public InitOrder() {
- 4: first = "constructor";
- 5: }
- 6: { first = "block"; }
- 7: public void print() {
- 8: System.out.println(first);
- 9: }
- 10: public static void main(String... args) {
- 11: new InitOrder().print();
- 12: }
- 13: }
- **A. block**
- **B. constructor**
- **C. instance**
- **D. The code does not compile.**

Q10

B

- Which of the following is the output of this code, assuming it runs to completion?
- package store;
- public class Toy {
- public void play() {
- System.out.print("play-");
- }
- public void finalizer() {
- System.out.print("clean-");
- }
- public static void main(String[] fun) {
- Toy car = new Toy();
- car.play();
- System.gc();
- Toy doll = new Toy();
- doll.play();
- }
- }
- **A.** play-
- **B.** play-play-
- **C.** play-clean-play-
- **D.** play-play-clean-clean

Chapter 2:Operators and Statements

• Chapter 2: Operators and Statements

Understanding Java Operators

Unless overridden with parentheses, Java operators follow order of operation, listed in the next Table, by decreasing order of operator precedence. **If two operators have the same level of precedence, then Java guarantees left-to-right evaluation.** You need to know only those operators in bold for the OCA exam.

Operator	Symbols and examples
Post-unary operators	<i>expression++</i> , <i>expression--</i>
Pre-unary operators	<i>++expression</i>, <i>--expression</i>
Other unary operators	<i>+, -, !</i>
Multiplication/Division/Modulus	<i>*, /, %</i>
Addition/Subtraction	<i>+, -</i>
Shift operators	<i><<, >>, >>></i>
Relational operators	<i><, >, <=, >=, instanceof</i>
Equal to/not equal to	<i>==, !=</i>
Logical operators	<i>&, ^, </i>
Short-circuit logical operators	<i>&&, </i>
Ternary operators	<i>boolean expression ? expression1 : expression2</i>
Assignment operators	<i>=, +=, -=, *=, /=, %=, &=, ^=, !=, <=>, >=>, >>>=</i>

• Chapter 2: Operators and Statements

Working with Binary Arithmetic Operators

- **Arithmetic Operators**

Arithmetic operators are often encountered in early mathematics and include addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). They also include the unary operators, ++ and --.

Exp : int x = 2 * 5 + 3 * 4 - 8;

- **Numeric Promotion**

Now that you understand the basics of arithmetic operators, it is vital we talk about primitive numeric promotion, as Java may do things that seem unusual to you at first. If you recall in Chapter 1, “Java Building Blocks,” where we listed the primitive numeric types, each primitive has a bit-length. You don’t need to know the exact size of these types for the exam, but you should know which are bigger than others.

• Chapter 2: Operators and Statements

Working with Binary Arithmetic Operators

- **Numeric Promotion (cont.)**

You should memorize certain rules Java will follow when applying operators to data types:

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.

• Chapter 2: Operators and Statements

Working with Binary Arithmetic Operators

- **Numeric Promotion (suite)**

What is the data type of `x * y`?

```
int x = 1;  
long y = 33;
```

What is the data type of `x + y`?

```
double x = 39.21;  
float y = 2.1;
```

What is the data type of `x / y`?

```
short x = 10;  
short y = 3;
```

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

```
short x = 14;  
float y = 13;  
double z = 30;
```

• Chapter 2: Operators and Statements

Working with Unary Operators

By definition, a *unary* operator is one that requires exactly one operand, or variable, to function. As shown in [The Table](#), they often perform simple tasks, such as increasing a numeric variable by one, or negating a boolean value.

Unary operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrements a value by 1
!	Inverts a Boolean's logical value

• Chapter 2: Operators and Statements

Working with Unary Operators

- **Logical Complement and Negation Operators :**

The *logical complement operator*, `!`, flips the value of a boolean expression.

Likewise, the *negation operator*, `-`, reverses the sign of a numeric expression.

boolean x = false;	double x = 1.21;
System.out.println(x); // false	System.out.println(x); // 1.21
x = !x;	x = -x;
System.out.println(x); // true	System.out.println(x); // -1.21
	x = -x;
	System.out.println(x); // 1.21

```
int x = !5; // DOES NOT COMPILE
boolean y = -true; // DOES NOT COMPILE
boolean z = !0; // DOES NOT COMPILE
```

• Chapter 2: Operators and Statements

Working with Unary Operators

- **Increment and Decrement Operators :**
- Increment and decrement operators, ++ and --, respectively, can be applied to numeric operands and have the higher order or precedence, as compared to binary operators. In other words, they often get applied first to an expression.
- Increment and decrement operators require special care because the order they are applied to their associated operand can make a difference in how an expression is processed.

```
int counter = 0;  
System.out.println(counter); // Outputs 0  
System.out.println(++counter); // Outputs 1  
System.out.println(counter); // Outputs 1  
System.out.println(counter--); // Outputs 1  
System.out.println(counter); // Outputs 0
```

```
int x = 3;  
int y = ++x * 5 / x-- + --x;  
System.out.println("x is " + x);  
System.out.println("y is " + y);
```

• Chapter 2: Operators and Statements

Using Additional Binary Operators

- **Assignment Operators :**

An assignment operator is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation. The simplest assignment operator is the = assignment, which you have seen already:

- int x = 1;

```
int x = 1.0; // DOES NOT COMPILE
```

```
short y = 1921222; // DOES NOT COMPILE
```

```
int z = 9f; // DOES NOT COMPILE
```

```
long t = 192301398193810323; // DOES NOT COMPILE
```

- **Casting Primitive Values**

```
short x = 10;  
short y = 3;  
short z = x * y;
```



```
short x = 10;  
short y = 3;  
short z = (short)(x * y);
```

• Chapter 2: Operators and Statements

Using Additional Binary Operators

- **Compound Assignment Operators:**

Besides the simple assignment operator, `=`, there are also numerous compound assignment operators. Only two of the compound operators are required for the exam, `+=` and `-=`.

Compound operators are useful for more than just shorthand—they can also save us from having to explicitly cast a value.

```
long x = 10;  
int y = 5;  
y = y * x; // DOES NOT COMPILE
```

```
long x = 10;  
int y = 5;  
y *= x;
```

• Chapter 2: Operators and Statements

Relational Operators

<	Strictly less than
<=	Less than or equal to
>	Strictly greater than
>=	Greater than or equal to
a instanceof b	True if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b

• Chapter 2: Operators and Statements

Logical Operators

x & y (AND)		
	y = true	y = false
x = true	true	false
x = false	false	false

x y (INCLUSIVE OR)		
	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y (EXCLUSIVE OR)		
	y = true	y = false
x = true	false	true
x = false	true	false

Here are some tips to help remember this table:

- AND is only true if both operands are true.
- Inclusive OR is only false if both operands are false.
- Exclusive OR is only true if the operands are different.

Finally, we present the conditional operators, `&&` and `||`, which are often referred to as short-circuit operators. The *short-circuit operators* are nearly identical to the logical operators, `&` and `|`, respectively, except that the right-hand side of the expression may never be evaluated if the final result can be determined by the left-hand side of the expression.

```
boolean x = true || (y < 4);
int x = 6;
boolean y = (x >= 6) || (++x <= 7);
System.out.println(x);
```

• Chapter 2: Operators and Statements

Equality Operators

Let's start with the basics, the *equals* operator == and *not equals* operator !=. Like the relational operators, they compare two operands and return a boolean value about whether the expressions or values are equal, or not equal, respectively.

The equality operators are used in one of three scenarios:

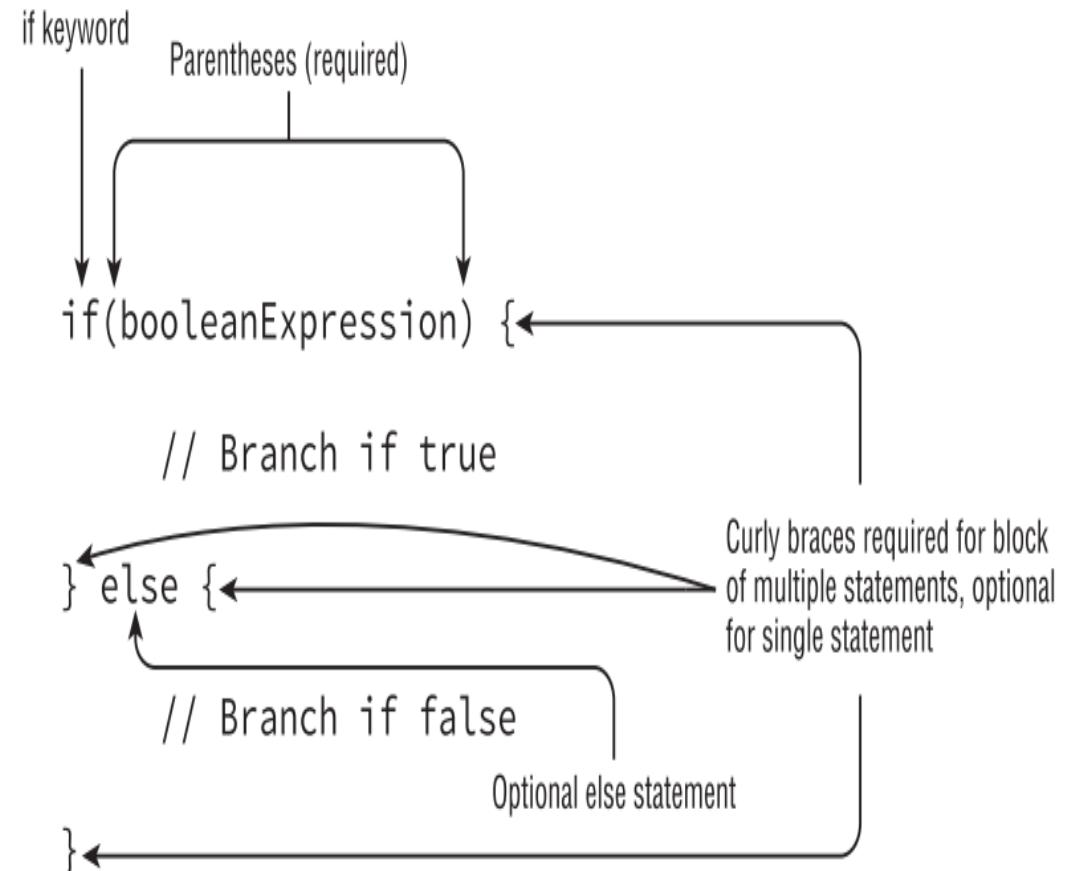
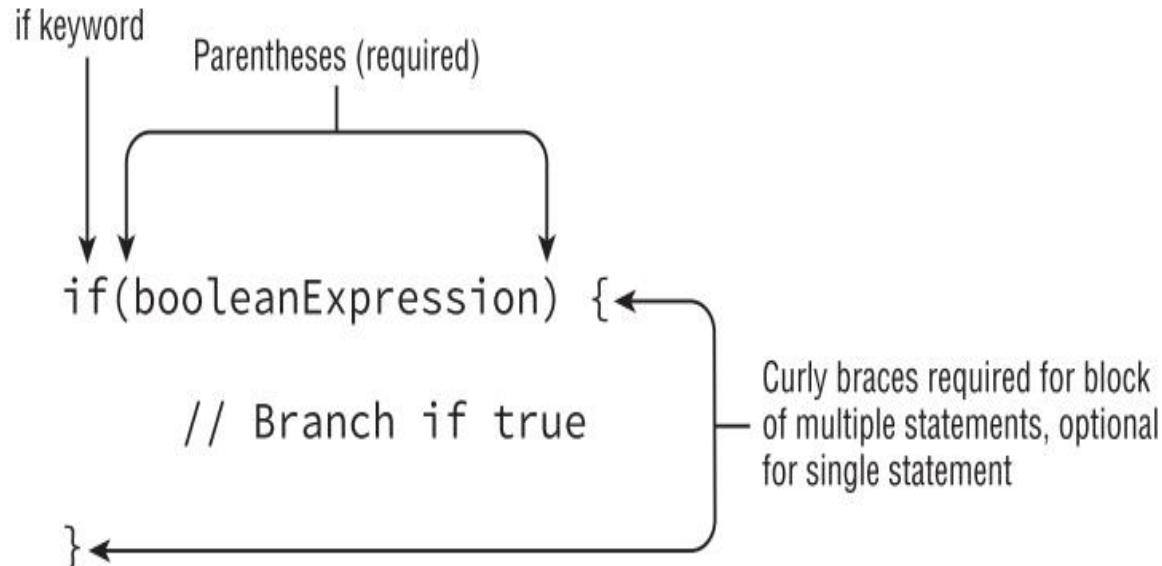
1. Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted as previously described. For example, 5 == 5.00 returns true since the left side is promoted to a double.
2. Comparing two boolean values.
3. Comparing two objects, including null and String values.

```
boolean x = true == 3; // DOES NOT COMPILE
boolean y = false != "Giraffe"; // DOES NOT COMPILE
boolean z = 3 == "Kangaroo"; // DOES NOT COMPILE
boolean y = false;
boolean x = (y = true);
System.out.println(x); // Outputs true
```

```
File x = new File("myFile.txt");
File y = new File("myFile.txt");
File z = x;
System.out.println(x == y); // Outputs false
System.out.println(x == z); // Outputs true
```

• Chapter 2: Operators and Statements

Understanding Java Statements



• Chapter 2: Operators and Statements

Understanding Java Statements

Exp : if-then Statement

```
if(hourOfDay < 11)
    System.out.println("Good Morning");
```

```
if(hourOfDay < 11) {
    System.out.println("Good Morning");
    morningGreetingCount++;
}
```

```
if(hourOfDay < 11)
    System.out.println("Good Morning");
morningGreetingCount++;
```

• Chapter 2: Operators and Statements

Understanding Java Statements

Exp : if-then-else Statement

```
if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
}  
if(hourOfDay >= 11) {  
    System.out.println("Good Afternoon");  
}
```



```
if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else {  
    System.out.println("Good Afternoon");  
}
```

```
if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else if(hourOfDay < 15) {  
    System.out.println("Good Afternoon");  
} else {  
    System.out.println("Good Evening");  
}
```

```
if(hourOfDay < 15) {  
    System.out.println("Good Afternoon");  
} else if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else {  
    System.out.println("Good Evening");  
}
```

// UNREACHABLE CODE

• Chapter 2: Operators and Statements

Understanding Java Statements

```
int x = 1;  
if(x = 5) { // DOES NOT COMPILE  
    ...  
}
```

```
int x = 1;  
if(x) { // DOES NOT COMPILE  
    ...  
}
```

Ternary Operator

Now that we have discussed if-then-else statements, we can briefly return to our discussion of operators and present the final operator that you need to learn for the exam. The conditional operator, ?:, otherwise known as the ternary operator, is the only operator that takes three operands and is of the form:

```
booleanExpression ? expression1 : expression2
```

The first operand must be a boolean expression, and the second and third can be any expression that returns a value. The ternary operation is really a condensed form of an if-then-else statement that returns a value. For example, the following two snippets of code are equivalent:

• Chapter 2: Operators and Statements

Understanding Java Statements

Ternary Operator

```
int y = 10;  
final int x;  
if(y > 5) {  
    x = 2 * y;  
} else {  
    x = 3 * y;  
}
```

```
int y = 10;  
int x = (y > 5) ? (2 * y) : (3 * y);
```

Note that it is often helpful for readability to add parentheses around the expressions in ternary operations, although it is certainly not required.

There is no requirement that second and third expressions in ternary operations have the same data types, although it may come into play when combined with the assignment operator. Compare the following two statements:

```
System.out.println((y > 5) ? 21 : "Zebra");  
int animal = (y < 91) ? 9 : "Horse"; // DOES NOT COMPILE
```

• Chapter 2: Operators and Statements

Understanding Java Statements

Ternary Operator

As of Java 7, only one of the right-hand expressions of the ternary operator will be evaluated at runtime. In a manner similar to the short-circuit operators, if one of the two right hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime.

Let's illustrate this principle with the following example:

```
int y = 1;  
int z = 1;  
final int x = y<10 ? y++ : z++;  
System.out.println(y+","+z); // Outputs 2,1
```

```
int y = 1;  
int z = 1;  
final int x = y>=10 ? y++ : z++;  
System.out.println(y+","+z); // Outputs 1,2
```

For the exam, be wary of any question that includes a ternary expression in which a variable is modified in one of the right-hand side expressions.

• Chapter 2: Operators and Statements

Understanding Java Statements

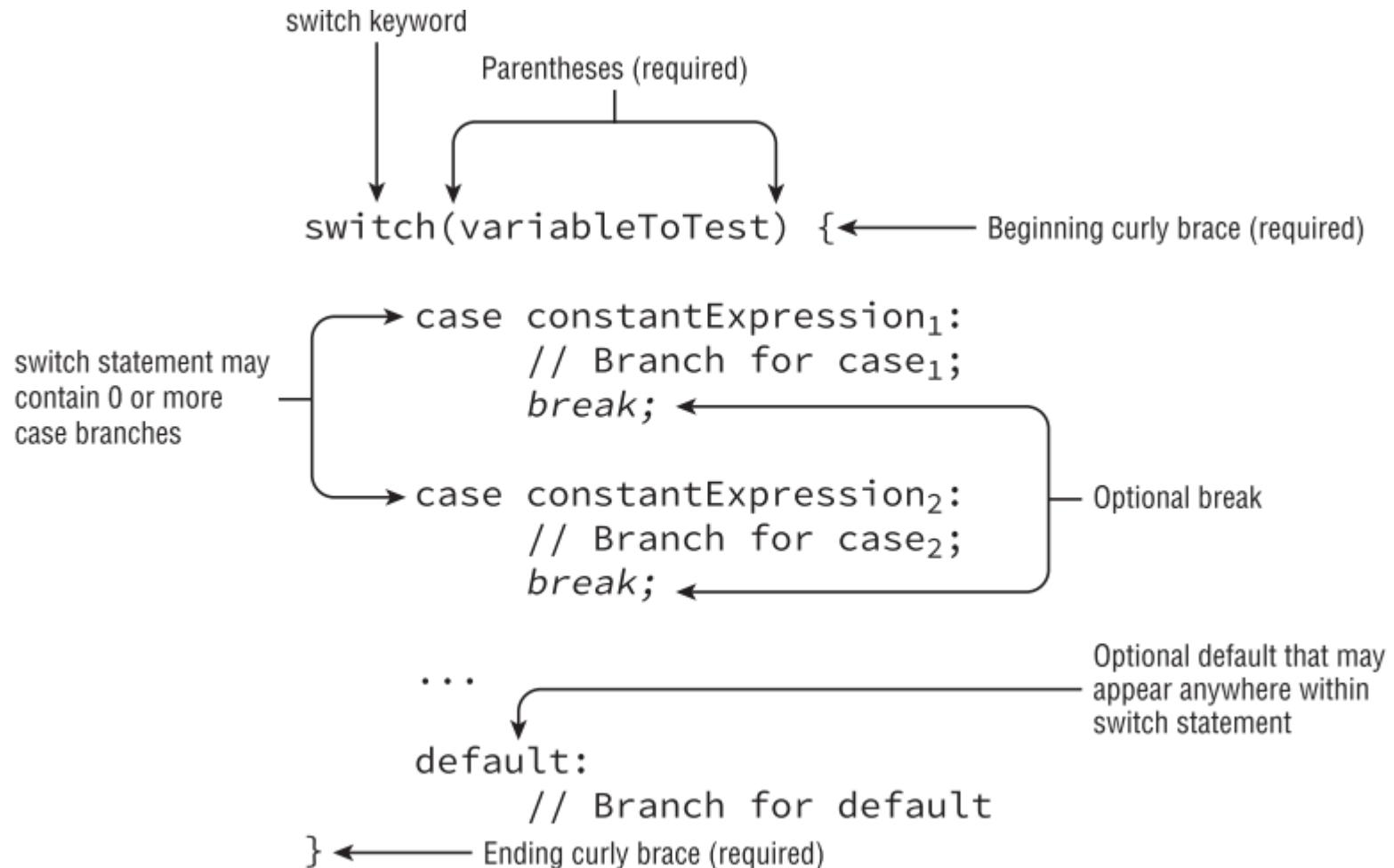
The switch Statement

A *switch* statement is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch, known as a *case* statement. If no such case statement is found that matches the value, an optional *default* statement will be called. If no such default option is available, the entire switch statement will be skipped.

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement



• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement

Data types supported by switch statements include the following:

- int and Integer
- byte and Byte
- short and Short
- char and Character
- int and Integer
- String
- enum values

For the exam, we recommend you memorize this list. Note that boolean and long, and their associated wrapper classes, are not supported by switch statements.

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement : Compile-time Constant Values

The values in each case statement must be compile-time constant values of the same data type as the switch value. This means you can use only literals, enum constants, or final constant variables of the same data type. By final constant, we mean that the variable must be marked with the final modifier and initialized with a literal value in the same expression in which it is declared.

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement : Compile-time Constant Values

```
int dayOfWeek = 5;
switch(dayOfWeek) {
    default:
        System.out.println("Weekday");
        break;
    case 0:
        System.out.println("Sunday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
}
```

With a value of dayOfWeek of 5, this code will output:

Weekday

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement

The first thing you may notice is that there is a break statement at the end of each case and default section. We'll discuss break statements in detail when we discuss loops, but for now all you need to know is that they terminate the switch statement and return flow control to the enclosing statement. As we'll soon see, if you leave out the break statement, flow will continue to the next proceeding case or default block automatically.

Another thing you might notice is that the default block is not at the end of the switch statement. There is no requirement that the case or default statements be in a particular order, unless you are going to have pathways that reach multiple sections of the switch block in a single execution.

```
int dayOfWeek = 5;  
switch(dayOfWeek) {  
    default:  
        System.out.println("Weekday");  
        break;  
    case 0:  
        System.out.println("Sunday");  
        break;  
    case 6:  
        System.out.println("Saturday");  
        break;  
}
```

With a value of dayOfWeek of 5, this code will output:

Weekday

• Chapter 2: Operators and Statements

Understanding Java Statements

The switch Statement

```
int dayOfWeek = 5;  
switch(dayOfWeek) {  
    case 0:  
        System.out.println("Sunday");  
    default:  
        System.out.println("Weekday");  
    case 6:  
        System.out.println("Saturday");  
        break;  
}
```

• Chapter 2: Operators and Statements

Understanding Java Statements

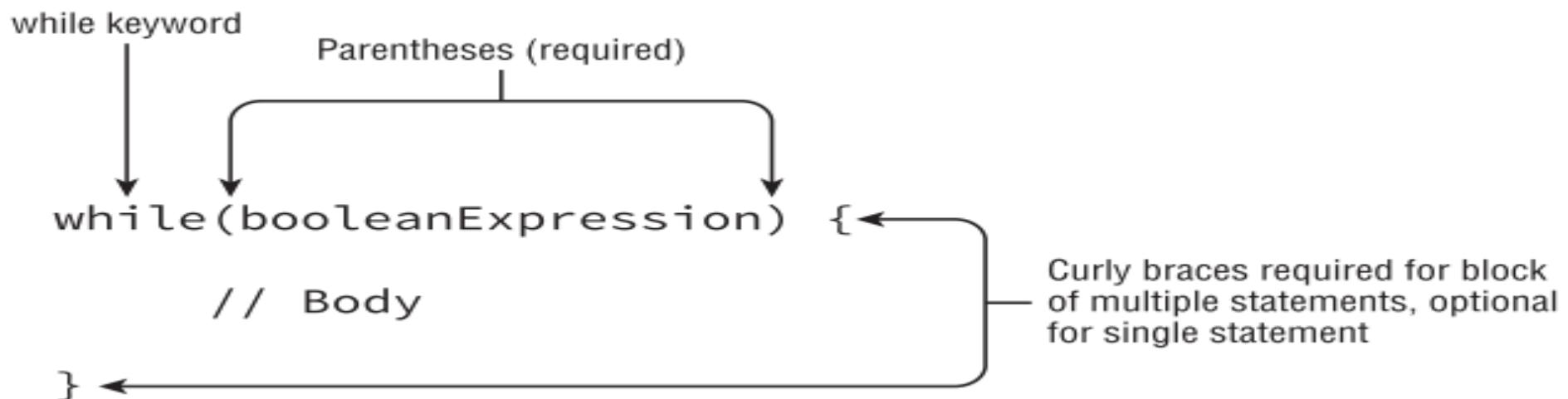
```
private int getSortOrder(String firstName, final String lastName) {  
  
    String middleName = "Patricia";  
    final String suffix = "JR";  
    int id = 0;  
    switch(firstName) {  
        case "Test":  
            return 52;  
  
        case middleName: // DOES NOT COMPILE  
            id = 5;  
            break;  
        case suffix:  
            id = 0;  
            break;  
        case lastName: // DOES NOT COMPILE  
            id = 8;  
            break;  
        case 5: // DOES NOT COMPILE  
            id = 7;  
            break;  
        case 'J': // DOES NOT COMPILE  
            id = 10;  
            break;  
        case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE  
            id=15;  
            break;  
    }  
    return id;  
}
```

• Chapter 2: Operators and Statements

Understanding Java Statements

The while Statement

A repetition control structure, which we refer to as a *loop*, executes a statement of code multiple times in succession. By using non constant variables, each repetition of the statement may be different.



• Chapter 2: Operators and Statements

Understanding Java Statements

The while Statement

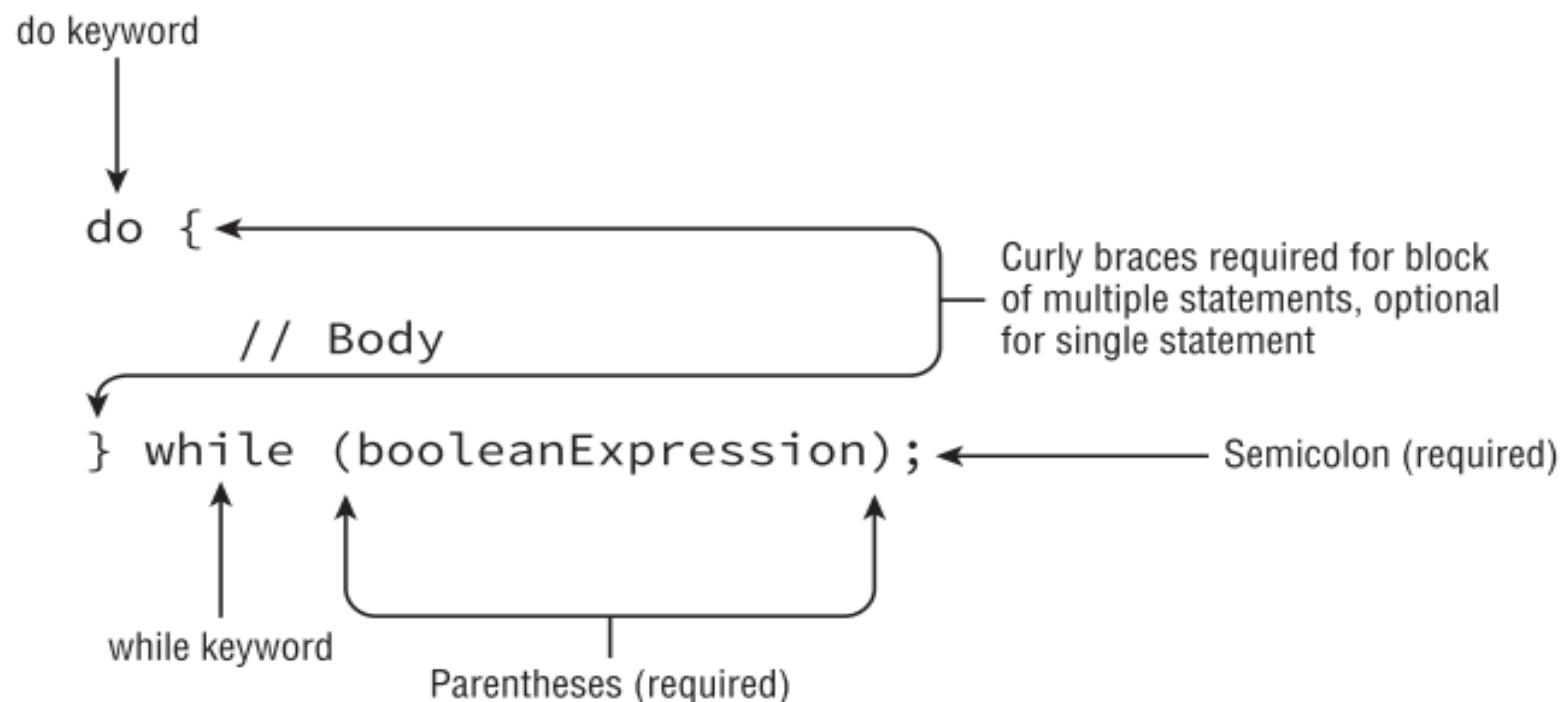
```
int roomInBelly = 5;                                int x = 2;  
public void eatCheese(int bitesOfCheese) {           int y = 5;  
    while (bitesOfCheese > 0 && roomInBelly > 0) {  
        bitesOfCheese--;  
        roomInBelly--;  
    }  
    System.out.println(bitesOfCheese+" pieces of cheese left");  
}  
  
while(x < 10)  
    y++;
```

• Chapter 2: Operators and Statements

Understanding Java Statements

The do-while Statement

Java also allows for the creation of a *do-while* loop, which like a while loop, is a repetition control structure with a termination condition and statement, or block of statements. Unlike a while loop, though, a do-while loop guarantees that the statement or block will be executed at least once.



• Chapter 2: Operators and Statements

Understanding Java Statements

The do-while Statement

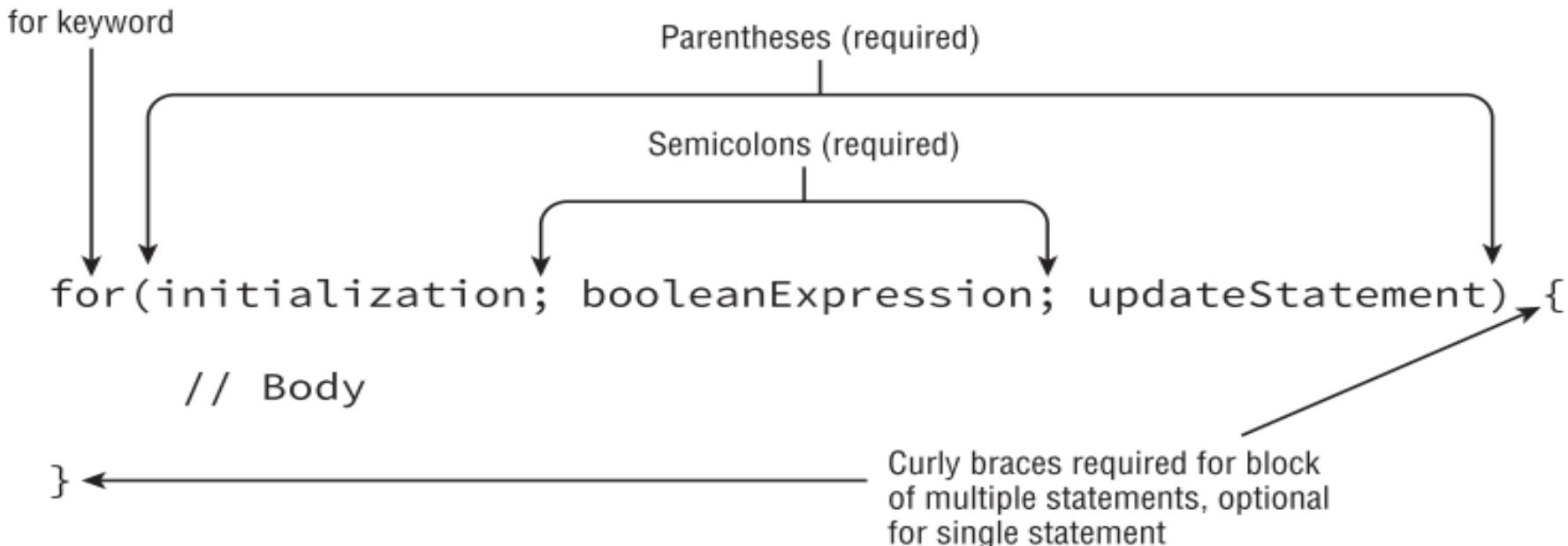
```
int x = 0;  
do {  
    x++;  
} while(false);  
System.out.println(x); // Outputs 1
```

Java will execute the statement block first, and then check the loop condition. Even though the loop exits right away, the statement block was still executed once and the program outputs a 1.

• Chapter 2: Operators and Statements

Understanding Java Statements

The Basic For Statement



- ① Initialization statement executes
- ② If booleanExpression is true continue, else exit loop
- ③ Body executes
- ④ Execute updateStatements
- ⑤ Return to Step 2

• Chapter 2: Operators and Statements

Understanding Java Statements

The Basic For Statement

Note that each section is separated by a semicolon. The initialization and update sections may contain multiple statements, separated by commas.

Variables declared in the initialization block of a for loop have limited scope and are only accessible within the for loop. Be wary of any exam questions in which a variable declared within the initialization block of a for loop is available outside the loop.

Alternatively, variables declared before the for loop and assigned a value in the initialization block may be used outside the for loop because their scope precedes the for loop creation.

```
for(int i = 0; i < 10; i++) {  
    System.out.print(i + " ");  
}
```

```
for( ; ; ) {  
    System.out.println("Hello World");  
}
```

• Chapter 2: Operators and Statements

Understanding Java Statements

The infinite loop

```
for( ; ; ) {  
    System.out.println("Hello World");  
}
```

Although this for loop may look like it will throw a compiler error, it will in fact compile and run without issue. It is actually an infinite loop that will print the same statement repeatedly. This example reinforces the fact that the components of the for loop are each optional. Note that the semicolons separating the three sections are required, as **for(;)** and **for()** will not compile.

• Chapter 2: Operators and Statements

Understanding Java Statements

Adding Multiple Terms to the for Statement

```
int x = 0;  
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {  
    System.out.print(y + " ");  
}
```

This code demonstrates three variations of the for loop you may not have seen. First, you can declare a variable, such as x in this example, before the loop begins and use it after it completes. Second, your initialization block, boolean expression, and update statements can include extra variables that may not reference each other..

For example, z is defined in the initialization block and is never used. Finally, the update statement can modify multiple variables. This code will print the following when executed: 0 1 2 3 4

• Chapter 2: Operators and Statements

Understanding Java Statements

Redeclaring a Variable in the Initialization Block

```
int x = 0;  
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // DOES NOT COMPILE  
    System.out.print(x + " ");  
}
```

This example looks similar to the previous one, but it does not compile because of the initialization block. The difference is that x is repeated in the initialization block after already being declared before the loop, resulting in the compiler stopping because of a duplicate variable declaration.

We can fix this loop by changing the declaration of x and y as follows :

```
int x = 0;  
long y = 10;  
for(y = 0, x = 4; x < 5 && y < 10; x++, y++) {  
    System.out.print(x + " ");  
}
```



Note that this variation will now compile because the initialization block simply assigns a value to x and does not declare it.

• Chapter 2: Operators and Statements

Understanding Java Statements

Using Incompatible Data Types in the Initialization Block

```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) { // DOES NOT COMPILE  
    System.out.print(x + " ");  
}
```

The variables in the initialization block must all be of the same type.

Using Loop Variables Outside the Loop

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {  
    System.out.print(y + " ");  
}  
System.out.print(x); // DOES NOT COMPILE
```

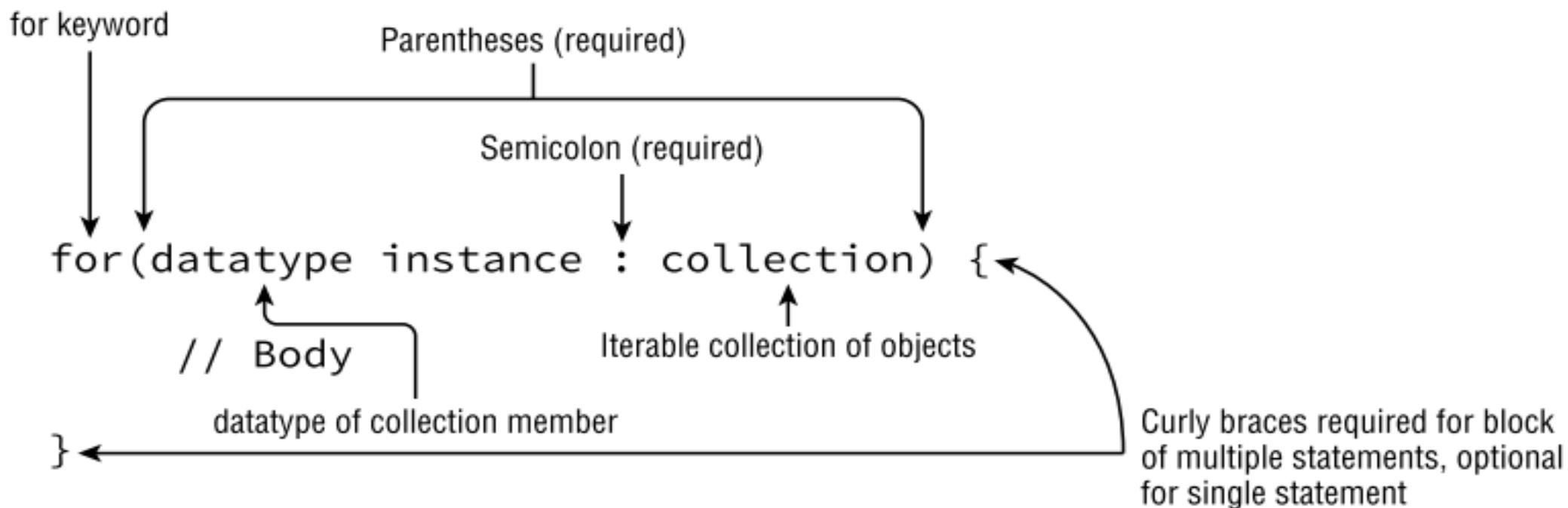
The final variation on the second example will not compile for a different reason than the previous examples. If you notice, x is defined in the initialization block of the loop, and then used after the loop terminates. Since x was only scoped for the loop, using it outside the loop will throw a compiler error.

• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

Starting with Java 5.0, Java developers have had a new type of enhanced for loop at their disposal, one specifically designed for iterating over arrays and Collection objects. This enhanced for loop, which for clarity we'll refer to as a *for-each* loop,



• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

- The for-each loop declaration is composed of an initialization section and an object to be iterated over.
- The right-hand side of the for-each loop statement must be a built-in Java array or an object whose class implements `java.lang.Iterable`, which includes most of the Java Collections framework.

For the OCA exam, the only members of the Collections framework that you need to be aware of are `List` and `ArrayList`.

- The left-hand side of the for-each loop must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement.
- On each iteration of the loop, the named variable on the left-hand side of the statement is assigned a new value from the array or collection on the right-hand side of the statement.

• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

- What will this code output?

```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
    System.out.print(name + ", ");
}
```

What will this code output?

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
    System.out.print(value + ", ");
}
```

• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

When you see a for-each loop on the exam, make sure the right-hand side is an array or Iterable object and the left-hand side has a matching type. For example, the two examples that follow will not compile.

- Why will the following fail to compile?

```
String names = "Lisa";
for(String name : names) {    // DOES NOT COMPILE
    System.out.print(name + " ");
}
```

In this example, the String names is not an array, nor does it implement java.lang.Iterable, so the compiler will throw an exception since it does not know how to iterate over the String.

- Why will the following fail to compile?

```
String[] names = new String[3];
for(int name : names) {    // DOES NOT COMPILE
    System.out.print(name + " ");
}
```

This code will fail to compile because the left-hand side of the for-each statement does not define an instance of String. Notice that in this last example, the array is initialized with three null pointer values. In and of itself, that will not cause the code to not compile, as a corrected loop would just output null three times.

• Chapter 2: Operators and Statements

Understanding Java Statements

The for-each Statement

```
java.util.List<String> names = new java.util.ArrayList<String>();  
names.add("Lisa");  
names.add("Kevin");  
names.add("Roger");  
for(int i=0; i<names.size(); i++) {  
    String name = names.get(i);  
    if(i>0) {  
        System.out.print(", ");  
    }  
    System.out.print(name);  
}
```

This sample code would output the following:

Lisa, Kevin, Roger

```
int[] values = new int[3];  
values[0] = 10;  
values[1] = new Integer(5);  
values[2] = 15;  
for(int i=1; i<values.length; i++) {  
    System.out.print(values[i]-values[i-1]);  
}
```

This sample code would output the following:

-5, 10,

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

Nested Loops

First off, loops can contain other loops. For example, consider the following code that iterates over a two-dimensional array, an array that contains other arrays as its members. We'll cover multidimensional arrays in detail in Chapter 3, but for now assume the following is how you would declare a two-dimensional array.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
    for(int i=0; i<mySimpleArray.length; i++) {
        System.out.print(mySimpleArray[i]+"\t");
    }
    System.out.println();
}
```

5	2	1	3
3	9	8	9
5	7	12	7

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

Nested Loops

Nested loops can include while and do-while, as shown in this example. See if you can determine what this code will output.

```
int x = 20;
while(x>0) {
    do {
        x -= 2
    } while (x>5);
    x--;
    System.out.print(x+"\t");
}
```

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

Adding Optional Labels

One thing we skipped when we presented if-then statements, switch statements, and loops is that they can all have optional labels. A *label* is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single word that is proceeded by a colon (:). For example, we can add optional labels to one of the previous examples:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP: for(int[] mySimpleArray : myComplexArray) {
    INNER_LOOP: for(int i=0; i<mySimpleArray.length; i++) {
        System.out.print(mySimpleArray[i]+\t");
    }
    System.out.println();
}
```

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The break Statement

As you saw when working with switch statements, a *break* statement transfers the flow of control out to the enclosing statement. The same holds true for break statements that appear inside of while, do-while, and for loops.

Optional reference to head of loop

optionalLabel: while(booleanExpression) {

// Body

// Somewhere in loop

break optionalLabel;

}

break keyword

Colon (required if optionalLabel is present)

Semicolon (required)

The diagram shows a code snippet for a while loop. An annotation 'Optional reference to head of loop' points to the label 'optionalLabel'. Another annotation 'Colon (required if optionalLabel is present)' points to the colon after 'optionalLabel'. A bracket under 'optionalLabel' indicates it is optional. An annotation 'Semicolon (required)' points to the semicolon at the end of the loop body. An annotation 'break keyword' points to the 'break' keyword inside the loop body. Brackets under 'Body' and 'Somewhere in loop' indicate they are parts of the loop's body.

Notice in the figure that the break statement can take an optional label parameter. Without a label parameter, the break statement will terminate the nearest inner loop it is currently in the process of executing. The optional label parameter allows us to break out of a higher level outer loop.

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The break Statement

```
public class SearchSample {  
    public static void main(String[] args) {  
        int[][] list = {{1,13,5},{1,2,5},{2,7,2}};  
        int searchValue = 2;  
        int positionX = -1;  
        int positionY = -1;  
        PARENT_LOOP: for(int i=0; i<list.length; i++) {  
            for(int j=0; j<list[i].length; j++) {  
                if(list[i][j]==searchValue) {  
                    positionX = i;  
                    positionY = j;  
                    break PARENT_LOOP;  
                }  
            }  
        }  
        if(positionX==-1 || positionY==-1) {  
            System.out.println("Value "+searchValue+" not found");  
        } else {  
            System.out.println("Value "+searchValue+" found at: " +  
                "("+positionX+","+positionY+")");  
        }  
    }  
}
```

Value 2 found at: (1,1)

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The break Statement

```
if(list[i][j]==searchValue) {  
    positionX = i;  
    positionY = j;  
    break;  
}
```

Value 2 found at: (2,0)

```
if(list[i][j]==searchValue) {  
    positionX = i;  
    positionY = j;  
}
```

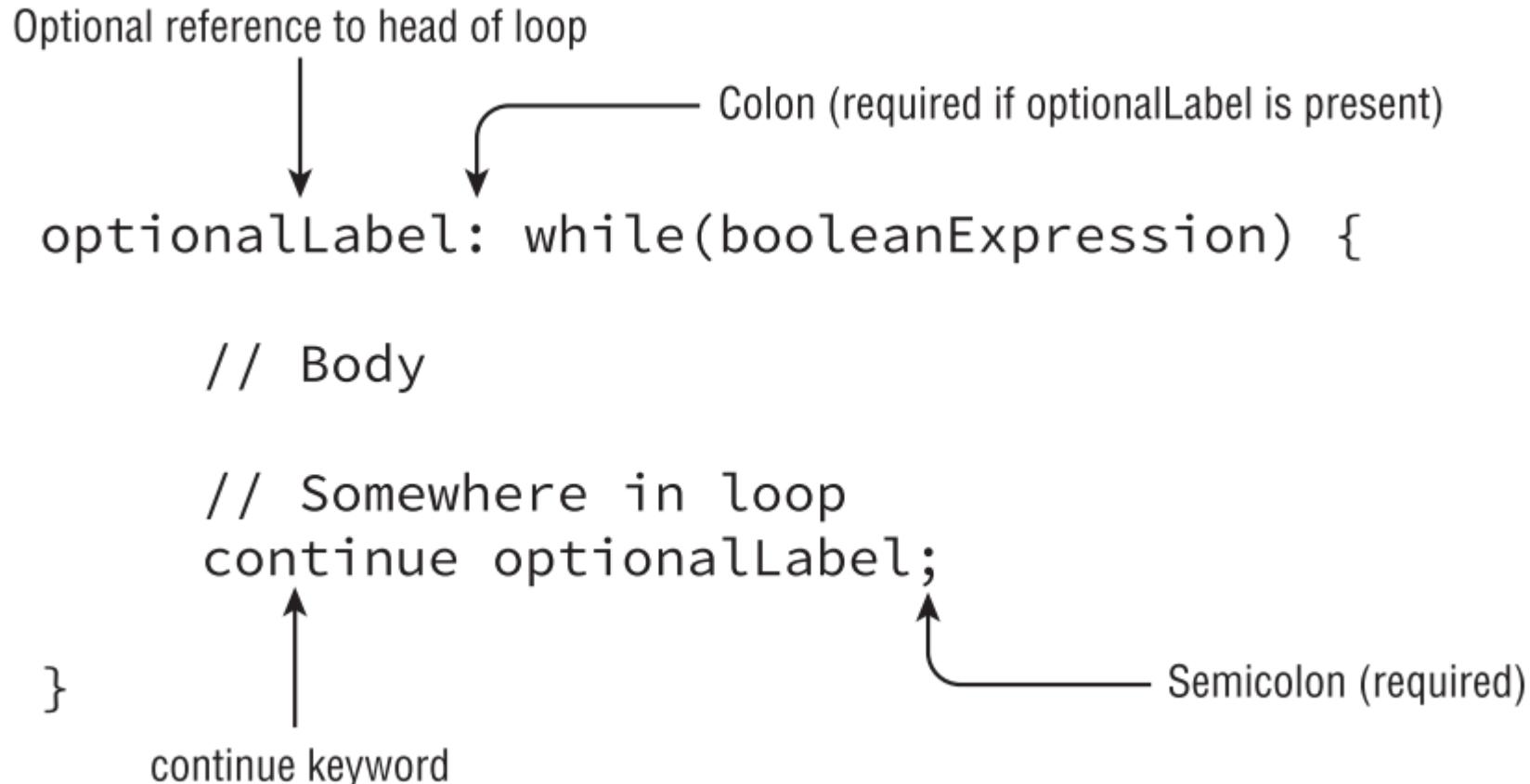
Value 2 found at: (2,2)

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The `continue` Statement

the `continue` statement, is a statement that causes flow to finish the execution of the current loop



• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

The continue Statement

```
public class SwitchSample {  
    public static void main(String[] args) {  
        FIRST_CHAR_LOOP: for (int a = 1; a <= 4; a++) {  
            for (char x = 'a'; x <= 'c'; x++) {  
                if (a == 2 || x == 'b')  
                    continue FIRST_CHAR_LOOP;  
                System.out.print(" " + a + x);  
            }  
        }  
    }  
}
```

1a 3a 4a

If we remove the FIRST_CHAR_LOOP label the result will be:

1a 1c 3a 3c 4a 4c

If we remove the continue

1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c

• Chapter 2: Operators and Statements

Understanding Advanced Flow Control

	Allows optional labels	Allows <i>break</i> statement	Allows <i>continue</i> statement
if	Yes *	No	No
while	Yes	Yes	Yes
do while	Yes	Yes	Yes
for	Yes	Yes	Yes
switch	Yes	Yes	No

* Labels are allowed for any block statement, including those that are preceded with an if-then statement.

QCM Chapitre 2

OCAJP-IZ0-808

Q1

A,D

- Which of the following Java operators can be used with boolean variables? (Choose all that apply)
- **A.** ==
- **B.** +
- **C.** --
- **D.** !
- **E.** %
- **F.** <=

Q2

A,B,D

- 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

Q3

F

- What is the output of the following application?
 - 1: public class CompareValues {
 - 2: public static void main(String[] args) {
 - 3: int x = 0;
 - 4: while(x++ < 10) {}
 - 5: String message = x > 10 ? "Greater than" : false;
 - 6:
System.out.println(message+"," +x);
 - 7: }
 - 8: }
- **A.** Greater than,10
 - **B.** false,10
 - **C.** Greater than,11
 - **D.** false,11
 - **E.** The code will not compile because of line 4.
 - **F.** The code will not compile because of line 5.

Q4

B,C,D,F

- What change would allow the following code snippet to compile? (Choose all that apply)
- 3: long x = 10;
- 4: int y = 2 * x;
- **A.** No change; it compiles as is.
- **B.** Cast x on line 4 to int.
- **C.** Change the data type of x on line 3 to short.
- **D.** Cast 2 * x on line 4 to int.
- **E.** Change the data type of y on line 4 to short.
- **F.** Change the data type of y on line 4 to long.

Q5

C

- What is the output of the following code snippet?
- 3: `java.util.List<Integer> list = new java.util.ArrayList<Integer>();`
- 4: `list.add(10);`
- 5: `list.add(14);`
- 6: `for(int x : list) {`
- 7: `System.out.print(x + ", ");`
- 8: `break;`
- 9: `}`
- A. 10, 14,
- B. 10, 14
- C. 10,
- D. The code will not compile because of line 7.
- E. The code will not compile because of line 8.
- F. The code contains an infinite loop and does not terminate.

Q6

F

- What is the output of the following code snippet?
- 3: int x = 4;
- 4: long y = x * 4 - x++;
- 5: if(y<10)
System.out.println("Too Low");
- 6: else System.out.println("Just right");
- 7: else System.out.println("Too High");
- A. Too Low
- B. Just Right
- C. Too High
- D. Compiles but throws a NullPointerException.
- E. The code will not compile because of line 6.
- F. The code will not compile because of line 7.

Q7

D

- What is the output of the following code?
 - C. 10
 - D. 8
 - E. 7
 - F. The code will not compile because line 4.
- 1: public class TernaryTester {
- 2: public static void main(String[] args) {
- 3: int x = 5;
- 4: System.out.println(x > 2 ? x < 4 ? 10 : 8 : 7);
- 5: }
- A. 5
- B. 4

Q8

B

- What is the output of the following code snippet?
- 3: boolean x = true, z = true;
- 4: int y = 20;
- 5: x = (y != 10) ^ (z=false);
- 6: System.out.println(x+", "+y+", "+z);
- **A.** true, 10, true
- **B.** true, 20, false
- **C.** false, 20, true
- **D.** false, 20, false
- **E.** false, 20, true
- **F.** The code will not compile because of line 5.

Q9

F

- 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.

Q10

D

- 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.

Q11

A

- What is the output of the following code?
 - A. 2
 - B. 3
 - C. 5
 - D. 6
 - E. The code will not compile because of line 3.
- ```
1: public class ArithmeticSample {
2: public static void main(String[] args) {
3: int x = 5 * 4 % 3;
4: System.out.println(x);
5: }
6: }
```

# Q12

D

- What is the output of the following code snippet?
- 3: int x = 0;
- 4: String s = null;
- 5: if(x == s) System.out.println("Success");
- 6: else System.out.println("Failure");
- A. Success
- B. Failure
- C. The code will not compile because of line 4.
- D. The code will not compile because of line 5.

# Q13

A

- What is the output of the following code snippet?
- 3: int x1 = 50, x2 = 75;
- 4: boolean b = x1 >= x2;
- 5: if(b = true) System.out.println("Success");
- 6: else System.out.println("Failure");
- **A.** Success
- **B.** Failure
- **C.** The code will not compile because of line 4.
- **D.** The code will not compile because of line 5.

# Q14

C

- What is the output of the following code snippet?
- 3: int c = 7;
- 4: int result = 4;
- 5: result += ++c;
- 6: System.out.println(result);
- **A.** 8
- **B.** 11
- **C.** 12
- **D.** 15
- **E.** 16
- **F.** The code will not compile because of line 5.

# Q15

E

- 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.

# Q16

D

- What is the output of the following code snippet?
- 3: do {
- 4: int y = 1;
- 5: System.out.print(y++ + " ");
- 6: } while(y <= 10);
- **A.** 1 2 3 4 5 6 7 8 9
- **B.** 1 2 3 4 5 6 7 8 9 10
- **C.** 1 2 3 4 5 6 7 8 9 10 11
- **D.** The code will not compile because of line 6.
- **E.** The code contains an infinite loop and does not terminate.

# Q17

D

- What is the output of the following code snippet?
- 3: boolean keepGoing = true;
- 4: int result = 15, i = 10;
- 5: do {
- 6: i--;
- 7: if(i==8) keepGoing = false;
- 8: result -= 2;
- 9: } while(keepGoing);
- 10: System.out.println(result);
- A. 7
- B. 9
- C. 10
- D. 11
- E. 15
- F. The code will not compile because of line 8.

# Q18

B

- What is the output of the following code snippet?
- 3: int count = 0;
- 4: ROW\_LOOP: for(int row = 1; row <=3; row++)
- 5: for(int col = 1; col <=2 ; col++) {
- 6: if(row \* col % 2 == 0) continue  
ROW\_LOOP;
- 7: count++;
- 8: }
- 9: System.out.println(count);
- A. 1
- B. 2
- C. 3
- D. 4
- E. 6
- F. The code will not compile because of line 6.

# Q19

D

- What is the result of the following code snippet?
- 3: int m = 9, n = 1, x = 0;
- 4: while(m > n) {
- 5: m--;
- 6: n += 2;
- 7: x += m + n;
- 8: }
- 9: System.out.println(x);
- A. 11
- B. 13
- C. 23
- D. 36
- E. 50
- F. The code will not compile because of line 7.

# Q20

B

- What is the result of the following code snippet?
  - 3: final char a = 'A', d = 'D';
  - 4: char grade = 'B';
  - 5: switch(grade) {
  - 6: case a:
  - 7: case 'B':  
System.out.print("great");
  - 8: case 'C': System.out.print("good");  
break;
  - 9: case d:  
  
• 10: case 'F': System.out.print("not  
good");
  - 11: }
- **A.** great
  - **B.** greatgood
  - **C.** The code will not compile because of line 3.
  - **D.** The code will not compile because of line 6.
  - **E.** The code will not compile because of lines 6 and 9.

# Tests

1 minute and 30 seconds

# Q1

- What is the output of the following application?
- package transporter;
- public class TurtleVsHare {
- public static void main(String[] arguments) {
- int turtle = 10 \* (2 + (3 + 2) / 5);
- int hare = turtle < 5 ? 10 : 25;
- System.out.print(turtle < hare ? "Hare wins!" : "Turtle wins!");
- }
- }
- **A.** Hare wins!
- **B.** Turtle wins!
- **C.** The code does not compile.
- **D.** The code compiles but throws a division by zero error at runtime.

## Q2

- What is the output of the following application?
  - public class CountEntries {
  - public static int getResult(int threshold) {
  - return threshold > 5 ? 1 : 0;
  - }
  - public static final void main(String[] days) {
  - System.out.print(getResult(5)+get
- Result(1)
- +getResult(0)+getResult(2)+"");
  - }
  - }
  - **A.** 0
  - **B.** 1
  - **C.** 0000
  - **D.** 1000

# Q3

- What is the output of the following application?
- package yoyo;
- public class TestGame {
- public String runTest(boolean spinner, boolean roller) {
- if(spinner = roller) return "up";
- else return roller ? "down" : "middle";
- }
- public static final void main(String pieces[]) {
- final TestGame tester = new TestGame();
- System.out.println(tester.runTest(false,true));
- }
- }
- **A.** up
- **B.** middle
- **C.** down
- **D.** The code does not compile.

# Q4

- Given the following code snippet, what is the value of movieRating after it is executed?
- int characters = 5;
- int story = 3;
- double movieRating = characters <= 4 ? 3 : story>1 ? 2 : 1;
- A. 2.0
- B. 3.0
- C. The code does not compile but would compile if parentheses were added.
- D. None of the above

## Q5

- Which of the following is not a possible result of executing the following application?
- public class OutsideLogic {
- public static void main(String... weather) {
- System.out.print(weather[0]!=null  
• && weather[0].equals("sunny")  
• && !false
  - ? "Go Outside" : "Stay Inside");
  - }
  - }
- A. Nothing is printed.
- B. The application throws an exception at runtime.
- C. Go Outside is printed.
- D. Stay Inside is printed.

## Q6

- What is the value of  $(5 + (!2 + 8) * 3 - 3 \% 2)/2$  in Java?
- **A.** 2
- **B.** 11
- **C.** 16
- **D.** None of the above

# Q7

- What is the output of the following application?
- public class Baby {
- public static String play(int toy, int age) {
- final String game;
- if(toy<2)
- game = age > 1 ? 1 : 10; // p1
- else
- game = age > 3 ? "Ball" : "Swim"; // p2
- return game;
- }
- public static void main(String[] variables) {
- System.out.print(play(5,2));
- }
- }
- **A.** Ball
- **B.** Swim
- **C.** The code does not compile due to p1.
- **D.** The code does not compile due to p2.

## Q8

- What is the output of the following code snippet?
- int hops = 0;
- int jumps = 0;
- jumps = hops++;
- if(jumps)
  - System.out.print("Jump!");
  - else
    - System.out.print("Hop!");
- **A.** Jump!
- **B.** Hop!
- **C.** The code does not compile.
- **D.** The code compiles but throws an exception at runtime.

## Q9

- What is the output of the following application?
  - package jungle;
  - public class TheBigRace {
  - public static void main(String[] in) {
  - int tiger = 2;
  - short lion = 3;
  - long winner = lion+2\*(tiger + lion);
  - System.out.print(winner);
  - }
  - }
- **A.** 11
  - **B.** 13
  - **C.** 25
  - **D.** None of the above

## Q10

- Given the following code snippet, what is the value of dinner after it is executed?
- int time = 11;
- int day = 4;
- String dinner = time > 10 ? day ? "Takeout" : "Salad" : "Leftovers";
- **A.** Takeout
- **B.** Salad
- **C.** The code does not compile but would compile if parentheses were added.
- **D.** None of the above

# Corrections

# Q1

B

- What is the output of the following application?
- package transporter;
- public class TurtleVsHare {
- public static void main(String[] arguments) {
- int turtle = 10 \* (2 + (3 + 2) / 5);
- int hare = turtle < 5 ? 10 : 25;
- System.out.print(turtle < hare ?

```
"Hare wins!" : "Turtle wins!");
 • }
 • }
 • A. Hare wins!
 • B. Turtle wins!
 • C. The code does not compile.
 • D. The code compiles but throws
 a division by zero error at
 runtime.
```

## Q2

A

- What is the output of the following application?
  - public class CountEntries {
  - public static int getResult(int threshold) {
  - return threshold > 5 ? 1 : 0;
  - }
  - public static final void main(String[] days) {
  - System.out.print(getResult(5)+get
- Result(1)
- +getResult(0)+getResult(2)+"");
  - }
  - }
  - **A.** 0
  - **B.** 1
  - **C.** 0000
  - **D.** 1000

# Q3

A

- What is the output of the following application?
- package yoyo;
- public class TestGame {
- public String runTest(boolean spinner, boolean roller) {
- if(spinner = roller) return "up";
- else return roller ? "down" : "middle";
- }
- public static final void main(String pieces[]) {
- final TestGame tester = new TestGame();
- System.out.println(tester.runTest(false,true));
- }
- }
- **A.** up
- **B.** middle
- **C.** down
- **D.** The code does not compile.

# Q4

A

- Given the following code snippet, what is the value of movieRating after it is executed?
- int characters = 5;
- int story = 3;
- double movieRating = characters <= 4 ? 3 : story>1 ? 2 : 1;
- A. 2.0
- B. 3.0
- C. The code does not compile but would compile if parentheses were added.
- D. None of the above

## Q5

A

- Which of the following is not a possible result of executing the following application?
- public class OutsideLogic {
- public static void main(String... weather) {
- System.out.print(weather[0]!=null  
• && weather[0].equals("sunny")  
• && !false
  - ? "Go Outside" : "Stay Inside");
  - }
  - }
- A. Nothing is printed.
- B. The application throws an exception at runtime.
- C. Go Outside is printed.
- D. Stay Inside is printed.

Q6

D

- What is the value of  $(5 + (!2 + 8) * 3 - 3 \% 2)/2$  in Java?
- **A.** 2
- **B.** 11
- **C.** 16
- **D.** None of the above

# Q7

C

- What is the output of the following application?
  - public class Baby {
  - public static String play(int toy, int age) {
  - final String game;
  - if(toy<2)
  - game = age > 1 ? 1 : 10; // p1
  - else
  - game = age > 3 ? "Ball" : "Swim"; // p2
  - return game;
  - }
- public static void main(String[] variables) {
  - System.out.print(play(5,2));
  - }
  - }
  - **A.** Ball
  - **B.** Swim
  - **C.** The code does not compile due to p1.
  - **D.** The code does not compile due to p2.

# Q8

C

- What is the output of the following code snippet?
- int hops = 0;
- int jumps = 0;
- jumps = hops++;
- if(jumps)
  - System.out.print("Jump!");
  - else
    - System.out.print("Hop!");
- **A.** Jump!
- **B.** Hop!
- **C.** The code does not compile.
- **D.** The code compiles but throws an exception at runtime.

## Q9

B

- What is the output of the following application?
  - package jungle;
  - public class TheBigRace {
  - public static void main(String[] in) {
  - int tiger = 2;
  - short lion = 3;
  - long winner = lion+2\*(tiger + lion);
  - System.out.print(winner);
  - }
  - }
- **A.** 11
  - **B.** 13
  - **C.** 25
  - **D.** None of the above

# Q10

D

- Given the following code snippet, what is the value of dinner after it is executed?
- int time = 11;
- int day = 4;
- String dinner = time > 10 ? day ? "Takeout" : "Salad" : "Leftovers";
- **A.** Takeout
- **B.** Salad
- **C.** The code does not compile but would compile if parentheses were added.
- **D.** None of the above

# **Chapter 3:Core Java APIs**

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Concatenation

The OCA exam creators like string concatenation because the + operator can be used in two ways within the same line of code. There aren't a lot of rules to know for this, but you have to know them well:

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
```

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);
```

```
4: String s = "1"; // s currently holds "1"
5: s += "2"; // s currently holds "12"
6: s += 3; // s currently holds "123"
7: System.out.println(s); // 123
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Immutability

Once a String object is created, it is not allowed to change. It cannot be made larger or smaller, and you cannot change one of the characters inside it.

Mutable is another word for changeable. Immutable is the opposite—an object that can't be changed once it's created. On the OCA exam, you need to know that String is immutable.

```
class Mutable {
 private String s;
 public void setS(String newS){ s = newS; } // Setter makes it mutable
 public String getS() { return s; }
}

final class Immutable {
 private String s = "name";
 public String getS() { return s; }
}
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Immutability

- Immutable classes in java are classes whose objects can not be changed or changed once created. Any change in an immutable object causes a new object or exception.
- Immutable classes in Java are final, and subclasses can't add mutable behavior.
- We can also set instance variables and setters as private and make the change only in the constructor.

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### The String Pool

Since strings are everywhere in Java, they use up a lot of memory. In some production applications, they can use up 25–40 percent of the memory in the entire program. Java realizes that many strings repeat in the program and solves this issue by reusing common ones. The *string pool*, also known as the *intern pool*, is a location in the Java virtual machine (JVM) that collects all these strings.

```
String name = "Fluffy";
```



Sauvegardée dans string pool

```
String name = new String("Fluffy");
```



Sauvegardée dans la pile de mémoire

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

The method **length()** returns the number of characters in the String.

```
String string = "animals";
System.out.println(string.length()); // 7
```

The method **charAt()** lets you query the string to find out what character is at a specific index.

```
char charAt(int index)
```

The following code shows how to use **charAt()**:

```
String string = "animals";
System.out.println(string.charAt(0)); // a
System.out.println(string.charAt(6)); // s
System.out.println(string.charAt(7)); // throws exception
```

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 7
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

The method **indexOf()** looks at the characters in the string and finds the first index that matches the desired value. **indexOf** can work with an individual character or a whole String as input. It can also start from a requested position.

```
int indexOf(char ch) String string = "animals";
int indexOf(char ch, int fromIndex) System.out.println(string.indexOf('a')); // 0
int indexOf(String str) System.out.println(string.indexOf("al")); // 4
int indexOf(String str, int fromIndex) System.out.println(string.indexOf('a', 4)); // 4
 System.out.println(string.indexOf("al", 5)); // -1
```

The method **substring()** also looks for characters in a string. It returns parts of the string.

The first parameter is the index to start with for the returned string. As usual, this is a zero-based index. There is an optional second parameter, which is the end index you want to stop at.

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

```
int substring(int beginIndex)
int substring(int beginIndex, int endIndex)

String string = "animals";
System.out.println(string.substring(3)); // als
System.out.println(string.substring(string.indexOf('m'))); // als
System.out.println(string.substring(3, 4)); // m
System.out.println(string.substring(3, 7)); // als

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
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

#### `toLowerCase()` and `toUpperCase()`

```
String toLowerCase(String str)
String toUpperCase(String str)
```

These methods leave alone any characters other than letters.  
Also, remember that strings are immutable, so the original string stays the same.

The following code shows how to use these methods:

```
String string = "animals";
System.out.println(string.toUpperCase()); // ANIMALS
System.out.println("Abc123".toLowerCase()); // abc123
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

#### equals() and equalsIgnoreCase()

The `equals()` method checks whether two `String` objects contain exactly the same characters in the same order. The `equalsIgnoreCase()` method checks whether two `String` objects contain the same characters with the exception that it will convert the characters' case if needed.

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
```

The following code shows how to use these methods:

```
System.out.println("abc".equals("ABC")); // false
System.out.println("ABC".equals("ABC")); // true
System.out.println("abc".equalsIgnoreCase("ABC")); // true
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

#### startsWith() and endsWith()

The `startsWith()` and `endsWith()` methods look at whether the provided value matches part of the String.

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

The following code shows how to use these methods:

```
System.out.println("abc".startsWith("a")); // true
System.out.println("abc".startsWith("A")); // false
System.out.println("abc".endsWith("c")); // true
System.out.println("abc".endsWith("a")); // false
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

#### contains()

```
boolean contains(String str)
```

The following code shows how to use these methods:

```
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
```

#### replace()

The replace() method does a simple search and replace on the string. There's a version that takes char parameters as well as a version that takes CharSequence parameters.

```
String replace(char oldChar, char newChar)
String replace(CharSequence oldChar, CharSequence newChar)
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important String Methods

#### replace()

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

#### trim()

The trim() method removes whitespace from the beginning and end of a String. In terms of the exam, whitespace consists of spaces along with the \t (tab) and \n (newline) characters. Other characters, such as \r (carriage return), are also included in what gets trimmed.

```
public String trim()
```

The following code shows how to use this method:

```
System.out.println("abc".trim()); // abc
System.out.println("\t a b c\n".trim()); // a b c
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Method Chaining

```
String start = "AniMaL ";
String trimmed = start.trim();

String lowercase = trimmed.toLowerCase(); // "animal"
String result = lowercase.replace('a', 'A'); // "Animal"
System.out.println(result);

String result = "AniMaL ".trim().toLowerCase().replace('a', 'A');
System.out.println(result); //AnimAl

5: String a = "abc";
6: String b = a.toUpperCase();
7: b = b.replace("B", "2").replace('C', '3');
8: System.out.println("a=" + a); "abc"
9: System.out.println("b=" + b); "A23"
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Using the `StringBuilder` Class

```
10: String alpha = "";
11: for(char current = 'a'; current <= 'z'; current++)
12: alpha += current;
13: System.out.println(alpha);

15: StringBuilder alpha = new StringBuilder();
16: for(char current = 'a'; current <= 'z'; current++)
17: alpha.append(current);
18: System.out.println(alpha);
```

"" object becomes eligible for garbage collection. The next time through the loop, alpha is assigned a new String object, "ab", and the "a" object becomes eligible for garbage collection. The next iteration assigns alpha to "abc" and the "ab" object becomes eligible for garbage collection, and so on.

This sequence of events continues, and after 26 iterations through the loop, a total of 27 objects are instantiated, most of which are immediately eligible for garbage collection.

This is very inefficient. Luckily, Java has a solution. The `StringBuilder` class creates a String without storing all those interim String values. Unlike the `String` class, `StringBuilder` is not immutable.

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### StringBuilder Class Mutability and Chaining

Chaining makes this even more interesting. When we chained String method calls, the result was a new String with the answer. Chaining StringBuilder objects doesn't work this way. Instead, the StringBuilder changes its own state and returns a reference to itself!

```
4: StringBuilder sb = new StringBuilder("start");
5: sb.append("+middle"); // sb = "start+middle"
6: StringBuilder same = sb.append("+end"); // "start+middle+end"
```

sb and same point to the exact same object and would print out the same value.

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
6: b = b.append("f").append("g");
7: System.out.println("a=" + a); a = abcdefg
8: System.out.println("b=" + b); b = abcdefg
```

- Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Creating a StringBuilder

There are three ways to construct a `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

The first says to create a `StringBuilder` containing an empty sequence of characters and assign `sb1` to point to it. The second says to create a `StringBuilder` containing a specific value and assign `sb2` to point to it. For the first two, it tells Java to manage the implementation details. The final example tells Java that we have some idea of how big the eventual value will be and would like the `StringBuilder` to reserve a certain number of slots for characters.

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

Important `StringBuilder` Methods

### ***charAt(), indexOf(), length(), and substring()***

These four methods work exactly the same as in the `String` class. Be sure you can identify the output of this example:

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub + " " + len + " " + ch); anim 7 s
```

Notice that `substring()` returns a `String` rather than a `StringBuilder`. That is why `sb` is not changed. `substring()` is really just a method that inquires about where the substring happens to be.

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important StringBuilder Methods

#### append()

it adds the parameter to the StringBuilder and returns a reference to the current StringBuilder.

```
StringBuilder append(String str)
```

Notice that we said one of the method signatures. There are more than 10 method signatures that look similar but that take different data types as parameters.

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb); // 1c-true
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important StringBuilder Methods

#### insert()

The insert() method adds characters to the StringBuilder at the requested index and returns a reference to the current StringBuilder. Just like append(), there are lots of method signatures for different types.

```
StringBuilder insert(int offset, String str)
```

```
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);
```

# • Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important StringBuilder Methods

#### delete() and deleteCharAt()

The delete() method is the opposite of the insert() method. It removes characters from the sequence and returns a reference to the current StringBuilder. The deleteCharAt() method is convenient when you want to delete only one character.

```
StringBuilder delete(int start, int end)
```

```
StringBuilder deleteCharAt(int index)
```

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3); // sb = adef
sb.deleteCharAt(5); // throws an exception
```

#### reverse()

it reverses the characters in the sequences and returns a reference to the current StringBuilder.

```
StringBuilder reverse()
```

```
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb); //CBA
```

- Chapter 3: Core Java APIs

## Creating and Manipulating Strings

### Important StringBuilder Methods

`toString()`

The last method converts a `StringBuilder` into a `String`.

```
String toString()
```

The following code shows how to use this method:

```
String s = sb.toString();
```

# • Chapter 3: Core Java APIs

## 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
```

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y); // true
```

```
String x = "Hello World";
String z = "Hello World".trim();
System.out.println(x == z); // false
```

```
String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y); // false
```

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;

```
String x = "Hello World";
String z = "Hello World".trim();
System.out.println(x == z); // false
```

```
String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y); // false
```

# • Chapter 3: Core Java APIs

## Understanding Equality

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z)); // true
```

The lesson is to never use == to compare String objects. The only time you should have to deal with == for Strings is on the exam.

```
1: public class Tiger {
2: String name;
3: public static void main(String[] args) {
4: Tiger t1 = new Tiger();
5: Tiger t2 = new Tiger();
6: Tiger t3 = t1;
7: System.out.println(t1 == t1); // true
8: System.out.println(t1 == t2); // false
9: System.out.println(t1.equals(t2)); // false
10: } }
```

Line 9 prints false since Tiger does not implement equals(). Don't worry—you aren't expected to know how to implement equals() for the OCA exam.

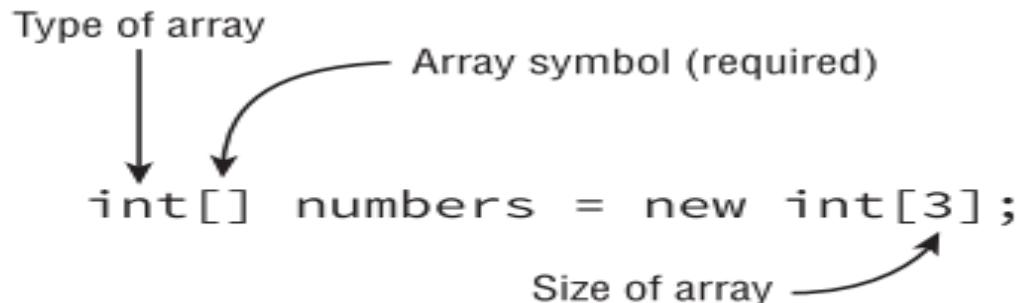
# • Chapter 3: Core Java APIs

## Understanding Java Arrays

An array is an **ordered** list. It can contain **duplicates**. You will learn about data structures that cannot contain duplicates for the OCP exam.

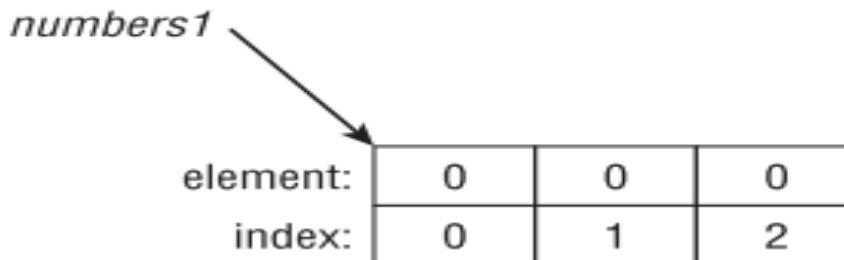
### **Creating an Array of Primitives**

```
int[] numbers1 = new int[3];
```



When using this form to instantiate an array, set all the elements to the default value for that type. As you learned in Chapter 1, the default value of an int is 0. Since `numbers1` is a reference variable, it points to the array object, as shown in Figure. As you can see, the default value for all the elements is 0. Also, the indexes start with 0 and count up.

An empty array

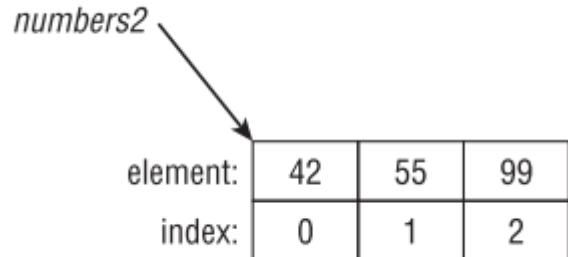


# • Chapter 3: Core Java APIs

## Understanding Java Arrays

### Creating an Array of Primitives

```
int[] numbers2 = new int[] {42, 55, 99};
```



```
int[] numbers2 = {42, 55, 99};
```

This approach is called an anonymous array. It is anonymous because you don't specify the type and size. Finally, 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 [];
```

- Chapter 3: Core Java APIs

## Understanding Java Arrays

### Creating an Array of Primitives

`int[] ids, types;` two variables of type `int[]`

`int ids[], types;` one variable of type `int[]` and one variable of type `int`

Needless to say, you shouldn't write code that looks like this. But you do still need to understand it for the exam.

# • Chapter 3: Core Java APIs

## Understanding Java Arrays

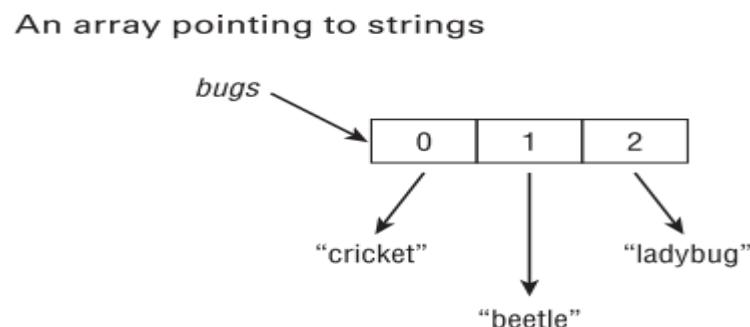
### Creating an Array with Reference Variables

You can choose any Java type to be the type of the array. This includes classes you create yourself. Let's take a look at a built-in type with String:

```
public class ArrayType {
 public static void main(String args[]) {
 String [] bugs = { "cricket", "beetle", "ladybug" };
 String [] alias = bugs;
 System.out.println(bugs.equals(alias)); // true

 } }
```

Make sure you understand Figure . The array does not allocate space for the String objects. Instead, it allocates space for a reference to where the objects are really stored.



- Chapter 3: Core Java APIs

## Understanding Java Arrays

### Creating an Array with Reference Variables

As a quick review, what do you think this array points to?

```
class Names {
 String names[];
}
```

The code never instantiated the array so it is just a reference variable to null.

```
class Names {
 String names[] = new String[2];
}
```

It is an array because it has brackets. It is an array of type String since that is the type mentioned in the declaration. It has two elements because the length is 2. Each of those two slots currently is null, but has the potential to point to a String object.

- Chapter 3: Core Java APIs

## Understanding Java Arrays

### Creating an Array with Reference Variables

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder(); // DOES NOT COMPILE
7: objects[0] = new StringBuilder(); // careful!
```

Line 6 doesn't compile because a `String[]` only allows `String` objects and `StringBuilder` is not a `String`. Line 7 is where this gets interesting. From the point of view of the compiler, this is just fine. A `StringBuilder` object can clearly go in an `Object[]`. The problem is that we don't actually have an `Object[]`. We have a `String[]` referred to from an `Object[]` variable. At runtime, the code throws an [ArrayStoreException](#). You don't need to memorize the name of this exception, but you do need to know that the code will throw an exception.

# • Chapter 3: Core Java APIs

## Understanding Java Arrays

### Using an Array

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length); // 3
6: System.out.println(mammals[0]); // monkey
7: System.out.println(mammals[1]); // chimp
8: System.out.println(mammals[2]); // donkey
```

```
String[] birds = new String[6];
System.out.println(birds.length);
```

The answer is 6. Even though all 6 elements of the array are null, there are still 6 of them. `length` does not consider what is in the array; it only considers how many slots have been allocated.

```
5: int[] numbers = new int[10];
6: for (int i = 0; i <= numbers.length; i++){ ArrayIndexOutOfBoundsException
7: numbers[i] = i + 5;}
```

# • Chapter 3: Core Java APIs

## Understanding Java Arrays

### Sorting

Java makes it easy to sort an array by providing a sort method `Arrays.sort()`.

`Arrays` is the first class provided by Java we have used that requires an import.

Remember that if you are shown a code snippet with a line number that doesn't begin with 1, you can assume the necessary imports are there.

```
import java.util.* // import whole package including Arrays
import java.util.Arrays; // import just Arrays

int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
 System.out.print (numbers[i] + " ");
```

The result is 1 6 9

Try this again with String types:

```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
 System.out.print(string + " ");
```

This code outputs 10 100 9. The problem is that String sorts in alphabetic order, and 1 sorts before 9. (Numbers sort before letters and uppercase sorts before lowercase, in case you were wondering.)

# • Chapter 3: Core Java APIs

## Understanding Java Arrays

### Searching

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

```
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));
```

---

#### Scenario

---

Target element found in sorted array

Target element not found in sorted array

---

#### Unsorted array

---

# • Chapter 3: Core Java APIs

## Understanding Java Arrays

### Multidimensional Arrays

Multiple array separators are all it takes to declare arrays with multiple dimensions. You can locate them with the type or variable name in the declaration, just as before:

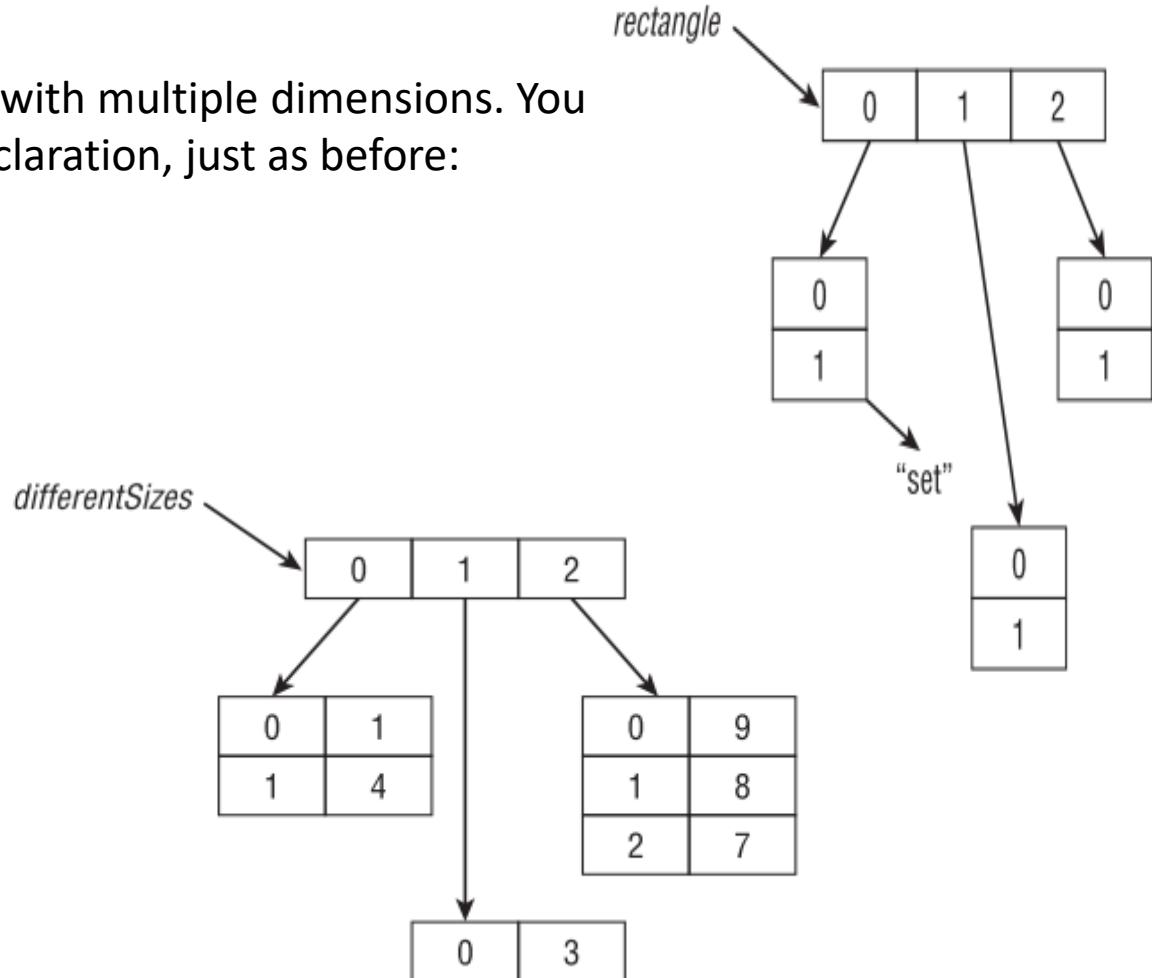
```
int[][] vars1; // 2D array
int vars2 [][]; // 2D array
int[] vars3[]; // 2D array
int[] vars4 [], space [][]; // a 2D AND a 3D array
```

```
String [][] rectangle = new String[3][2];

rectangle[0][1] = "set";

int[][] differentSize = {{1, 4}, {3}, {9,8,7}};

int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```



# • Chapter 3: Core Java APIs

## Understanding Java Arrays

### Using a Multidimensional Array

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
 for (int j = 0; j < twoD[i].length; j++)
 System.out.print(twoD[i][j] + " ");
 System.out.println(); // time for a new row
}
```

```
for (int[] inner : twoD) {
 for (int num : inner)
 System.out.print(num + " ");
 System.out.println();
}
```

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### Creating an ArrayList

```
import java.util.* // import whole package including ArrayList
import java.util.ArrayList; // import just 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<>();

List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>(); // DOES NOT COMPILE
```

ArrayList implements an interface called List. In other words, an ArrayList is a List. You will learn about interfaces in Chapter 5. In the meantime, just know that you can store an ArrayList in a List reference variable but not vice versa.

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### **add()**

The add() methods insert a new value in the ArrayList. The method signatures are as follows:

```
boolean add(E element)
void add(int index, E element)
```

Don't worry about the boolean return value. It always returns true. It is there because other classes in the collections family need a return value in the signature when adding an 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]
```

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### **remove()**

The remove() methods remove the first matching value in the ArrayList or remove the element at a specified index. The method signatures are as follows:

boolean remove(Object object)

E 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); // []
```

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### **set()**

The set() method changes one of the elements of the ArrayList without changing the size. The method signature is as follows:

E set(int index, E newElement)

The E return type is the element that got replaced. The following shows how to use this method:

```
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
```

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### **isEmpty() and size()**

The isEmpty() and size() methods look at how many of the slots are in use. The method signatures are as follows:

```
boolean isEmpty()
int size()
```

The following shows how to use these methods:

```
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
```

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### **clear()**

The clear() method provides an easy way to discard all elements of the ArrayList. The method signature is as follows:

`void clear()`

The following shows how to use this method:

```
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
```

After we call `clear()`, `birds` is back to being an empty ArrayList of size 0.

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### **contains()**

The contains() method checks whether a certain value is in the ArrayList. The method signature is as follows:

```
boolean contains(Object object)
```

The following shows how to use this method:

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
System.out.println(birds.contains("hawk")); // true
System.out.println(birds.contains("robin")); // false
```

This method calls equals() on each element of the ArrayList to see whether there are any matches. Since String implements equals(), this works out well.

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### **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)

The following shows how to use this method:

```
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
```

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### Wrapper Classes

| Primitive type | Wrapper class | Example of constructing |
|----------------|---------------|-------------------------|
| boolean        | Boolean       | new Boolean(true)       |
| byte           | Byte          | new Byte((byte) 1)      |
| short          | Short         | new Short((short) 1)    |
| int            | Integer       | new Integer(1)          |
| long           | Long          | new Long(1)             |
| float          | Float         | new Float(1.0)          |
| double         | Double        | new Double(1.0)         |
| char           | Character     | new Character('c')      |

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### Wrapper Classes

| Wrapper class | Converting String to primitive | Converting String to wrapper class |
|---------------|--------------------------------|------------------------------------|
| Boolean       | Boolean.parseBoolean("true");  | Boolean.valueOf("TRUE");           |
| Byte          | Byte.parseByte("1");           | Byte.valueOf("2");                 |
| Short         | Short.parseShort("1");         | Short.valueOf("2");                |
| Integer       | Integer.parseInt("1");         | Integer.valueOf("2");              |
| Long          | Long.parseLong("1");           | Long.valueOf("2");                 |
| Float         | Float.parseFloat("1");         | Float.valueOf("2.2");              |
| Double        | Double.parseDouble("1");       | Double.valueOf("2.2");             |
| Character     | None                           | None                               |

# • Chapter 3: Core Java APIs

## Understanding an ArrayList

### Autoboxing

Since Java 5, you can just type the primitive value and Java will convert it to the relevant wrapper class for you. This is called autoboxing.

```
4: List<Double> weights = new ArrayList<>();
5: weights.add(50.5); // [50.5]
6: weights.add(new Double(60)); // [50.5, 60.0]
7: weights.remove(50.5); // [60.0]
8: double first = weights.get(0); // 60.0
```

Line 5 autoboxes the double primitive into a Double object and adds that to the List.

Line 6 shows that you can still write code the long way and pass in a wrapper object.

Line 7 again autoboxes into the wrapper object and passes it to remove().

Line 8 retrieves the Double and unboxes it into a double primitive.

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### Autoboxing

What do you think happens if you try to unbox a null?

```
3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0); // NullPointerException
```

On line 4, we add a null to the list. This is legal because a null reference can be assigned to any reference variable. On line 5, we try to unbox that null to an int primitive. This is a problem. Java tries to get the int value of null. Since calling any method on null gives a NullPointerException, that is just what we get. Be careful when you see null in relation to autoboxing.

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### Autoboxing

Be careful when autoboxing into Integer. What do you think this code outputs?

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);
```

It actually outputs 1

After adding the two values, the List contains [1, 2]. We then request the element with index 1 be removed. That's right: index 1. Because there's already a remove() method that takes an int parameter, Java calls that method rather than autoboxing. If you want to remove the 2, you can write numbers.remove(new Integer(2)) to force wrapper class use.

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### Converting Between array and List

```
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
```

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### Converting Between array and List

```
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 UnsupportedOperationException
```

- Chapter 3: Core Java APIs

## Understanding an ArrayList

### 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]
```

# • Chapter 3: Core Java APIs

## Working with Dates and Times

In Java 8, Oracle completely revamped how we work with dates and times.

You need an import statement to work with the date and time classes.

```
import java.time.*; // import time classes
```

### Creating Dates and Times

**LocalDate** Contains just a date

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

**LocalTime** Contains just a time

**LocalDateTime** Contains both a date and time

Each of the three classes has a static method called now() that gives the current date and time. Your output is going to depend on what date/time you run it and where you live.

2015-01-20

12:45:18.401

2015-01-20T12:45:18.401

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Creating Dates and Times

```
LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);
LocalDate date2 = LocalDate.of(2015, 1, 20);

public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int dayOfMonth)
```

Month is a special type of class called an enum. You don't need to know about enums on the OCA exam and can just treat them as constants.

```
LocalTime time1 = LocalTime.of(6, 15); // hour and minute
LocalTime time2 = LocalTime.of(6, 15, 30); // + seconds
LocalTime time3 = LocalTime.of(6, 15, 30, 200); // + nanoseconds

public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanos)

LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
```

- Chapter 3: Core Java APIs

## Working with Dates and Times

### Creating Dates and Times

This time there are a lot of method signatures since there are more combinations.

```
public static LocalDateTime of(int year, int month,
 int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month,
 int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month,
 int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month,
 int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, Month month,
 int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, Month month,
 int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime)
```

- Chapter 3: Core Java APIs

## Working with Dates and Times

### Creating Dates and Times

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

Don't fall for this. 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
```

| <b>Old way</b>                                    | <b>New way (Java 8 and later)</b>                                                                                                                                                                                     |
|---------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Importing                                         | <code>import java.util.*;</code>                                                                                                                                                                                      |
| Creating an object with the current date          | <code>Date d = new Date();</code>                                                                                                                                                                                     |
| Creating an object with the current date and time | <code>Date d = new Date();</code>                                                                                                                                                                                     |
| Creating an object representing January 1, 2015   | <pre>Calendar c = Calendar.getInstance(); c.set(2015, Calendar.JANUARY, 1); Date jan = c.getTime();</pre> <p>or</p> <pre>Calendar c = new GregorianCalendar(2015, Calendar.JANUARY, 1); Date jan = c.getTime();</pre> |
| Creating January 1, 2015 without the constant     | <code>Calendar c = Calendar.getInstance(); c.set(2015, 0, 1); Date jan = c.getTime();</code>                                                                                                                          |
|                                                   | <code>LocalDate jan = LocalDate.of(2015, 1, 1)</code>                                                                                                                                                                 |

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Manipulating Dates and Times

```
12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
13: System.out.println(date); // 2014-01-20
14: date = date.plusDays(2);
15: System.out.println(date); // 2014-01-22
16: date = date.plusWeeks(1);

17: System.out.println(date); // 2014-01-29
18: date = date.plusMonths(1);
19: System.out.println(date); // 2014-02-28
20: date = date.plusYears(5);
21: System.out.println(date); // 2019-02-28
```

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Manipulating Dates and Times

```
22: LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
23: LocalTime time = LocalTime.of(5, 15);
24: LocalDateTime dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime); // 2020-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime); // 2020-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime); // 2020-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime); // 2020-01-18T19:14:30
```

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time)
.minusDays(1).minusHours(10).minusSeconds(30);
```

- Chapter 3: Core Java APIs

## Working with Dates and Times

### Manipulating Dates and Times

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);
```

It prints January 20, 2020. Adding 10 days was useless because we ignored the result. Whenever you see immutable types, pay attention to make sure the return value of a method call isn't ignored.

```
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.

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Manipulating Dates and Times

|                          | Can call on<br><b>LocalDate</b> ? | Can call on<br><b>LocalTime</b> ? | Can call on<br><b>LocalDateTime</b> ? |
|--------------------------|-----------------------------------|-----------------------------------|---------------------------------------|
| plusYears/minusYears     | Yes                               | No                                | Yes                                   |
| plusMonths/minusMonths   | Yes                               | No                                | Yes                                   |
| plusWeeks/minusWeeks     | Yes                               | No                                | Yes                                   |
| plusDays/minusDays       | Yes                               | No                                | Yes                                   |
| plusHours/minusHours     | No                                | Yes                               | Yes                                   |
| plusMinutes/minusMinutes | No                                | Yes                               | Yes                                   |
| plusSeconds/minusSeconds | No                                | Yes                               | Yes                                   |
| plusNanos/minusNanos     | No                                | Yes                               | Yes                                   |

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Working with Periods

```
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
 }
}
```

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Working with Periods

```
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

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

Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(7);
```



- Chapter 3: Core Java APIs

## Working with Dates and Times

### Working with Periods

```
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
```

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Formatting Dates and Times

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
System.out.println(date.getDayOfWeek()); // MONDAY DateTimeFormatter is in the package java.time.format.
System.out.println(date.getMonth()); // JANUARY
System.out.println(date.getYear()); // 2020
System.out.println(date.getDayOfYear()); // 20
-
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));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

ISO is a standard for dates. The output of the previous code looks like this:

2020-01-20

11:12:34

2020-01-20T11:12:34

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Formatting Dates and Times

```
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 shortDateTime =
 DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Formatting Dates and Times

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
```

```
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
```

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f)); // January 20, 2020, 11:12
```

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Formatting Dates and Times

**MMMM** M represents the month. The more Ms you have, the more verbose the Java output. For example, M outputs 1, MM outputs 01, MMM outputs Jan, and MMMM outputs January.

**dd** d represents the date in the month. As with month, the more ds you have, the more verbose the Java output. dd means to include the leading zero for a single-digit month.

• Use , if you want to output a comma (this also appears after the year).

**yyyy** y represents the year. yy outputs a two-digit year and yyyy outputs a four-digit year.

**hh** h represents the hour. Use hh to include the leading zero if you're outputting a single-digit hour.

• Use : if you want to output a colon.

**mm** m represents the minute.

|                      | Old way                                                                                  | New way (Java 8 and later)                                                                        |
|----------------------|------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Formatting the times | <pre>SimpleDateFormat sf = new<br/>SimpleDateFormat("hh:mm");<br/>sf.format(jan3);</pre> | <pre>DateTimeFormatter f<br/>= DateTimeFormatter.<br/>ofPattern("hh:mm");<br/>dt.format(f);</pre> |

# • Chapter 3: Core Java APIs

## Working with Dates and Times

### Formatting Dates and Times

```
4: DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");
5: f.format(dateTime);
6: f.format(date);
7: f.format(time);
```

If you get this question on the exam, think about what the symbols represent. We have h for hour and m for minute. Remember M (uppercase) is month and m (lowercase) is minute. We can only use this formatter with objects containing times. Therefore, line 6 will throw an exception.

- Chapter 3: Core Java APIs

## Working with Dates and Times

### Parsing Dates and Times

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date); // 2015-01-02
System.out.println(time); // 11:22
```

Parsing is consistent in that if anything goes wrong, Java throws a runtime exception.

# QCM Chapitre 3

OCAJP-IZ0-808

# Q1

G

What is output by the following code? 7: System.out.println(numFish + " " +  
(Choose all that apply) 1);

- |                                                         |                               |
|---------------------------------------------------------|-------------------------------|
| 1: public class Fish {                                  | 8: } }                        |
| 2: public static void main(String[] args)               | A. 4 1                        |
| {                                                       | B. 41                         |
| 3: int numFish = 4;                                     | C. 5                          |
| 4: String fishType = "tuna";                            | D. 5 tuna                     |
| 5: String anotherFish = numFish + 1;                    | E. 5tuna                      |
| 6: System.out.println(anotherFish + " "<br>+ fishType); | F. 51tuna                     |
|                                                         | G. The code does not compile. |

## Q2

A, C, D

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.

# Q3

B, C, E

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.

# Q4

B

What is the result of the following code?

```
7: StringBuilder sb = new
StringBuilder();
8: sb.append("aaa").insert(1,
"bb").insert(4, "ccc");
9: System.out.println(sb);
```

- A. abbaaccc
- B. abbaccca

- C. bbaaaccc
- D. bbaaccca
- E. An exception is thrown.
- F. The code does not compile.

Q5

F

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.

# Q6

B

What is the result of the following code?

```
public class Lion {
 public void roar(String roar1,
 StringBuilder roar2) {
 roar1.concat("!!!");
 roar2.append("!!!");
 }

 public static void main(String[] args) {
 String roar1 = "roar";
 StringBuilder roar2 = new
```

```
 StringBuilder("roar");
 new Lion().roar(roar1, roar2);
 System.out.println(roar1 + " " +
 roar2);
 } }
```

- A. roar roar
- B. roar roar!!!
- C. roar!!! roar
- D. roar!!! roar!!!
- E. An exception is thrown.
- F. The code does not compile.

# Q7

B, D, E

Which are the results of the following code? (Choose all that apply)

String letters = "abcdef";

System.out.println(letters.length());

System.out.println(letters.charAt(3));

System.out.println(letters.charAt(6));

A. 5

B. 6

C. c

D. d

E. An exception is thrown.

F. The code does not compile.

## Q8

A, D, E

Which are the results of the following code? (Choose all that apply)

```
String numbers = "012345678";
```

```
System.out.println(numbers.substring(1, 3));
```

```
System.out.println(numbers.substring(7, 7));
```

```
System.out.println(numbers.substring(7));
```

ng(7));

A. 12

B. 123

C. 7

D. 78

E. A blank line.

F. An exception is thrown.

G. The code does not compile.

Q9

C

What is the result of the following code?

```
3: String s = "purr";
4: s.toUpperCase();
5: s.trim();
6: s.substring(1, 3);
7: s += " two";
8: System.out.println(s.length());
```

- A. 2
- B. 4
- C. 8
- D. 10
- E. An exception is thrown.
- F. The code does not compile.

# Q10

F

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.

# Q11

E

What is the result of the following code?

```
4: int total = 0;
5: StringBuilder letters = new
StringBuilder("abcdefg");
6: total += letters.substring(1,
2).length();
7: total += letters.substring(6,
6).length();
8: total += letters.substring(6,
```

- ```
5).length();  
9: System.out.println(total);
```
- A. 1
 - B. 2
 - C. 3
 - D. 7
 - E. An exception is thrown.
 - F. The code does not compile

Q12

A

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

```
StringBuilder numbers = new  
StringBuilder("0123456789");  
numbers.delete(2, 8);  
numbers.append("-").insert(2, "+");  
System.out.println(numbers);
```

- A. 01+89-
- B. 012+9-

- C. 012+-9
- D. 0123456789
- E. An exception is thrown.
- F. The code does not compile.

Q13

F

What is the result of the following code?

```
StringBuilder b = "rumble";
b.append(4).deleteCharAt(3).delete(3, b.length() - 1);
System.out.println(b);
```

- A. rum
- B. rum4
- C. rumb4
- D. rumble4
- E. An exception is thrown.
- F. The code does not compile.

Q14

A, C

Which of the following can replace line 4 to print "avaJ"? (Choose all that apply)

3: StringBuilder puzzle = new
StringBuilder("Java");

4: // INSERT CODE HERE

5: System.out.println(puzzle);

A. puzzle.reverse();

B.

puzzle.append("vaJ\$").substring(0,
4);

C. puzzle.append("vaJ\$").delete(0,
3).deleteCharAt(puzzle.length() -
1);

D. puzzle.append("vaJ\$").delete(0,
3).deleteCharAt(puzzle.length());

E. None of the above.

Q15

C, E, F

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

- F. `int[][] java = new int[][];`
- 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[]{};`

Q16

C

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.

Q17

F

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();
- G. None of the above.

Q18

A, C, D, E

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

Q19

B, C

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.
- E. None of the above.

Q20

D

What is the result of the following statements?

```
6: List<String> list = new  
ArrayList<String>();  
7: list.add("one");  
8: list.add("two");  
9: list.add(7);  
10: for(String s : list)  
System.out.print(s);
```

- A. onetwo
- B. onetwo7
- C. onetwo followed by an exception
- D. Compiler error on line 9.
- E. Compiler error on line 10.

Q21

C

What is the result of the following statements?

```
3: ArrayList<Integer> values = new
ArrayList<>();
4: values.add(4);
5: values.add(5);
6: values.set(1, 6);
7: values.remove(0);
8: for (Integer v : values)
```

- System.out.print(v);
- A. 4
- B. 5
- C. 6
- D. 46
- E. 45
- F. An exception is thrown.
- G. The code does not compile.

Q22

D

What is the result of the following? **C.** 6

```
int[] random = { 6, -4, 12, 0, -10 };    D. The result is undefined.  
int x = 12;                                E. An exception is thrown.  
int y =                                     F. The code does not compile.  
Arrays.binarySearch(random, x);  
System.out.println(y);
```

- A.** 2
- B.** 4

Q23

A

What is the result of the following? C. Compiler error on line 4.

```
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

D. Compiler error on line 5.

E. Compiler error on line 6.

F. An exception is thrown.

Q24

C

What is the result of the following? C. Compiler error on line 7.

6: String [] names = {"Tom", "Dick", D. Compiler error on line 8.
"Harry"}; E. An exception is thrown.

7: List<String> list = names.asList();

8: list.set(0, "Sue");

9: System.out.println(names[0]);

A. Sue

B. Tom

Q25

D

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. None of the above.
- F. The code doesn't compile.

Q26

A, B, D

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

4: List<Integer> ages = new ArrayList<>();

5: ages.add(Integer.parseInt("5"));

6: ages.add(Integer.valueOf("6"));

7: ages.add(7);

8: ages.add(null);

9: for (int age : ages)
System.out.print(age);

- A. The code compiles.
- B. The code throws a runtime exception.
- C. Exactly one of the add statements uses autoboxing.
- D. Exactly two of the add statements use autoboxing.
- E. Exactly three of the add statements use autoboxing.

Q27

B

What is the result of the following?

```
else if (one.equals(two))  
List<String> one = new  
ArrayList<String>();  
one.add("abc");  
List<String> two = new  
ArrayList<>();  
two.add("abc");  
if (one == two)  
System.out.println("A");
```

```
System.out.println("B");  
else  
System.out.println("C");  
A. A  
B. B  
C. C  
D. An exception is thrown.  
E. The code does not compile.
```

Q28

D, F

Which of the following can be inserted into the blank to create a date of June 21, 2014?

(Choose all that apply)

```
import java.time.*;  
public class StartOfSummer {  
    public static void main(String[] args) {  
        LocalDate date = _____  
    }  
}
```

- }
- A. new LocalDate(2014, 5, 21);
- B. new LocalDate(2014, 6, 21);
- C. LocalDate.of(2014, 5, 21);
- D. LocalDate.of(2014, 6, 21);
- E. LocalDate.of(2014, Calendar.JUNE, 21);
- F. LocalDate.of(2014, Month.JUNE, 21);

Q29

D

What is the output of the following " " + date.getMonth() + " "
code?

```
LocalDate date =  
    LocalDate.parse("2018-04-30",  
        DateTimeFormatter.ISO_LOCAL_  
        DATE);  
  
    date.plusDays(2);  
  
    date.plusHours(3);  
  
    System.out.println(date.getYear() +
```

- + date.getDayOfMonth());
- A. 2018 APRIL 2
- B. 2018 APRIL 30
- C. 2018 MAY 2
- D. The code does not compile.
- E. A runtime exception is thrown.

Q30

F

What is the output of the following code?

```
LocalDate date =  
    LocalDate.of(2018, Month.APRIL,  
40);  
  
System.out.println(date.getYear() +  
" " + date.getMonth() + " "  
+ date.getDayOfMonth());
```

A. 2018 APRIL 4

- B. 2018 APRIL 30
- C. 2018 MAY 10
- D. Another date.
- E. The code does not compile.
- F. A runtime exception is thrown.

Q31

B

What is the output of the following + date.getDayOfMonth());
code?

```
LocalDate date =  
    LocalDate.of(2018, Month.APRIL,  
30);  
  
date.plusDays(2);  
  
date.plusYears(3);  
  
System.out.println(date.getYear() +  
" " + date.getMonth() + " "
```

- A. 2018 APRIL 2
- B. 2018 APRIL 30
- C. 2018 MAY 2
- D. 2021 APRIL 2
- E. 2021 APRIL 30
- F. 2021 MAY 2
- G. A runtime exception is thrown.

Q32

E

```
LocalDateTime d =  
    LocalDateTime.of(2015, 5, 10, 11,  
                     22, 33);  
  
Period p = Period.of(1, 2, 3);  
  
d = d.minus(p);  
  
DateTimeFormatter f =  
    DateTimeFormatter.ofLocalizedTim  
e(FormatStyle.SHORT);  
  
System.out.println(d.format(f));
```

A. 3/7/14 11:22 AM
B. 5/10/15 11:22 AM
C. 3/7/14
D. 5/10/15
E. 11:22 AM
F. The code does not compile.
G. A runtime exception is thrown.

Q33

B

What is the output of the following eTime(FormatStyle 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.ofLocalizedDat
```

```
.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.

Tests

1 minute and 30 seconds

Q1

What is the result of the following when called as java counting.Binary?

```
package counting;  
import java.util.*;  
public class Binary {  
    public static void main(String...  
        args) {  
        Arrays.sort(args);  
    }  
}
```

```
    System.out.println(Arrays.toString(  
        args));  
}  
}
```

- A. null
- B. []
- C. The code does not compile.
- D. The code compiles but throws an exception at runtime.

Q2

What does the following output?

```
String[] os = new String[] { "Mac",  
    "Linux", "Windows" };
```

```
System.out.println(Arrays.binarySe  
arch(os, "Linux"));
```

- A. 0
- B. 1
- C. 2
- D. The output is not defined.

Q3

What is the result of running the following program?

```
1: package fun;  
2: public class Sudoku {  
3: static int[][] game;  
4:  
5: public static void main(String[] args)  
{  
6: game[3][3] = 6;  
7: Object[] obj = game;  
8: game[3][3] = "X";  
9: System.out.println(game[3][3]);  
10: }  
11: }  
A. X  
B. The code does not compile.  
C. The code compiles but throws a  
NullPointerException at runtime.  
D. The code compiles but throws a  
different exception at runtime.
```

Q4

What is the output of the following?

```
String[][] listing = new String[][] { { "Book" }, { "Game", "29.99" } };  
System.out.println(listing.length + " " + listing[0].length);
```

- A. 2 1
- B. 2 2
- C. The code does not compile.
- D. The code compiles but throws an exception at runtime.

Q5

What is the output of the following when run as java FirstName?

```
public class FirstName {  
    public static void main(String[]  
        names) {  
        System.out.println(names[0]);  
    }  
}
```

A. FirstName

- B. The code does not compile.
- C. The code throws an ArrayIndexOutOfBoundsException.
- D. The code throws a NullPointerException.

Q6

What is the output of the following?

```
List<String> tools = new  
ArrayList<>();
```

```
tools.add("hammer");
```

```
tools.add("nail");
```

```
tools.add("hex key");
```

```
System.out.println(tools.get(1));
```

A. hammer

- B. hex key
- C. nail
- D. None of the above

Q7

What is the result of the following code?

```
StringBuilder sb = new  
StringBuilder("radical")  
.insert(sb.length(), "robots");  
  
System.out.println(sb);
```

- A. radicarobots
- B. radicalrobots
- C. The code does not compile.

- D. The code compiles but throws an exception at runtime.

Q8

What is the output of the following?

```
List<String> museums = new  
ArrayList<>(1);  
  
museums.add("Natural History");  
  
museums.add("Science");  
  
museums.add("Art");  
  
museums.remove(2);  
  
System.out.println(museums);
```

- A. [Natural History, Science]
- B. [Natural History, Art, Science]
- C. The code does not compile.
- D. The code compiles but throws an exception at runtime.

Q9

What is the output of the following?

```
12: StringBuilder b = new  
StringBuilder("12");
```

```
13: b = b.append("3");
```

```
14: b.reverse();
```

```
15: System.out.println(b.toString());
```

A. 12

B. 123

C. 321

D. The code does not compile.

Q10

What is the output of the following?

```
5: StringBuilder line = new  
StringBuilder("-");
```

```
6: StringBuilder anotherLine =  
line.append("-");
```

```
7: System.out.print(line ==  
anotherLine);
```

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

```
9: System.out.print(line.length());
```

A. false 1

B. false 2

C. true 1

D. true 2

Corrections

Q1

B

What is the result of the following when called as java counting.Binary?

```
package counting;  
import java.util.*;  
public class Binary {  
    public static void main(String...  
        args) {  
        Arrays.sort(args);  
    }  
}
```

```
    System.out.println(Arrays.toString(  
        args));  
}  
}
```

- A. null
- B. []
- C. The code does not compile.
- D. The code compiles but throws an exception at runtime.

Q2

D

What does the following output?

```
String[] os = new String[] { "Mac",  
"Linux", "Windows" };
```

```
System.out.println(Arrays.binarySe  
arch(os, "Linux"));
```

- A. 0
- B. 1
- C. 2
- D. The output is not defined.

Q3

B

What is the result of running the following program?

```
1: package fun;  
2: public class Sudoku {  
3: static int[][] game;  
4:  
5: public static void main(String[] args)  
{  
6: game[3][3] = 6;  
7: Object[] obj = game;
```

```
8: game[3][3] = "X";  
9: System.out.println(game[3][3]);  
10: }  
11: }  
  


- A. X
- B. The code does not compile.
- C. The code compiles but throws a NullPointerException at runtime.
- D. The code compiles but throws a different exception at runtime.

```

Q4

A

What is the output of the following?

```
String[][] listing = new String[][] { { "Book" }, { "Game", "29.99" } };  
System.out.println(listing.length + " " + listing[0].length);
```

- A. 2 1
- B. 2 2
- C. The code does not compile.
- D. The code compiles but throws an exception at runtime.

Q5

C

What is the output of the following when run as java FirstName?

```
public class FirstName {  
    public static void main(String[]  
        names) {  
        System.out.println(names[0]);  
    }  
}
```

A. FirstName

- B. The code does not compile.
- C. The code throws an ArrayIndexOutOfBoundsException.
- D. The code throws a NullPointerException.

Q6

C

What is the output of the following?

```
List<String> tools = new  
ArrayList<>();
```

```
tools.add("hammer");
```

```
tools.add("nail");
```

```
tools.add("hex key");
```

```
System.out.println(tools.get(1));
```

A. hammer

- B. hex key
- C. nail
- D. None of the above

Q7

C

What is the result of the following code?

```
StringBuilder sb = new  
StringBuilder("radical").insert(sb.le  
ngth(), "robots");
```

```
System.out.println(sb);
```

A. radicarobots

B. radicalrobots

C. The code does not compile.

D. The code compiles but throws an exception at runtime.

Q8

A

What is the output of the following?

```
List<String> museums = new  
ArrayList<>(1);  
  
museums.add("Natural History");  
  
museums.add("Science");  
  
museums.add("Art");  
  
museums.remove(2);  
  
System.out.println(museums);
```

- A. [Natural History, Science]
- B. [Natural History, Art, Science]
- C. The code does not compile.
- D. The code compiles but throws an exception at runtime.

Q9

C

What is the output of the following?

```
12: StringBuilder b = new  
StringBuilder("12");
```

```
13: b = b.append("3");
```

```
14: b.reverse();
```

```
15: System.out.println(b.toString());
```

A. 12

B. 123

C. 321

D. The code does not compile.

Q10

D

What is the output of the following?

```
5: StringBuilder line = new  
StringBuilder("-");
```

```
6: StringBuilder anotherLine =  
line.append("-");
```

```
7: System.out.print(line ==  
anotherLine);
```

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

```
9: System.out.print(line.length());
```

A. false 1

B. false 2

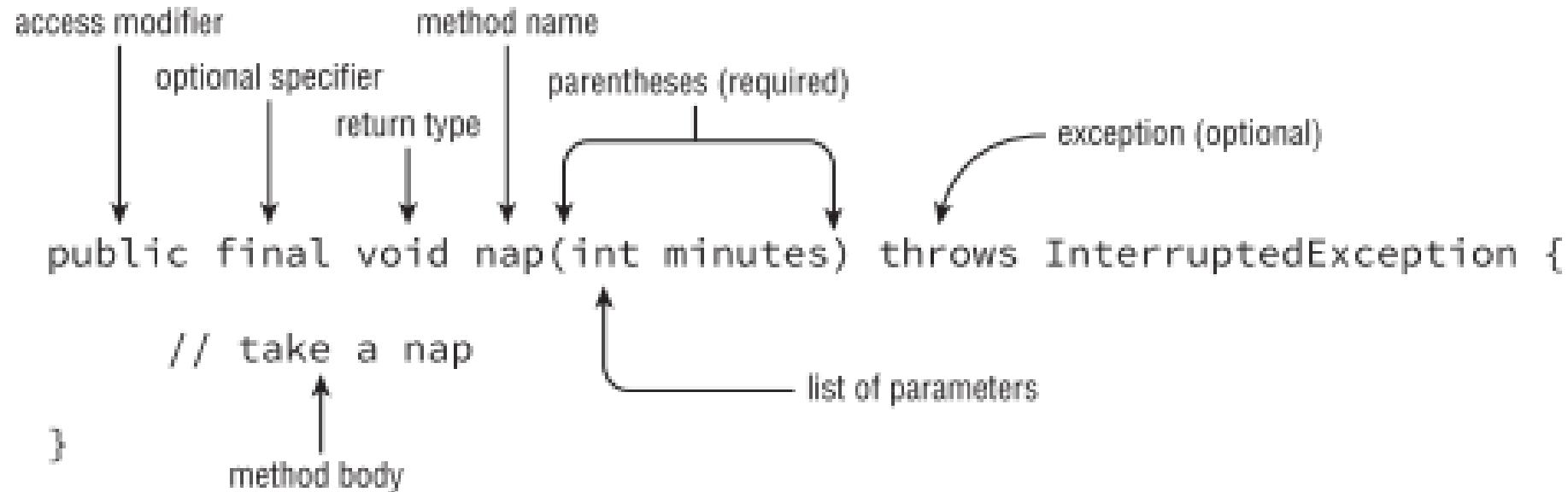
C. true 1

D. true 2

Chapter 4:Methods and Encapsulation

- Chapter 4: Methods and Encapsulation

Designing Methods



• Chapter 4: Methods and Encapsulation

Designing Methods

Element	Value in nap() example	Required?
Access modifier	public	No
Optional specifier	final	No
Return type	void	Yes
Method name	nap	Yes
Parameter list	(int minutes)	Yes, but can be empty parentheses
Optional exception list	throws InterruptedException	No
Method body	{ // take a nap }	Yes, but can be empty braces

- Chapter 4: Methods and Encapsulation

Designing Methods

Access Modifiers

Java offers four choices of access modifier:

public The method can be called from any class.

private The method can only be called from within the same class.

protected The method can only be called from classes in the same package or subclasses.

You'll learn about subclasses in Chapter 5.

Default (Package Private) Access The method can only be called from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.

```
public void walk1() {}
```

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

```
void public walk3() {} // DOES NOT COMPILE
```

```
void walk4() {}
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Optional Specifiers

static Covered later in this chapter. Used for class methods.

abstract Covered in Chapter 5. Used when not providing a method body.

final Covered in Chapter 5. Used when a method is not allowed to be overridden by a subclass.

```
public void walk1() {}  
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 COMPILE  
final public void walk7() {}
```

walk7() does compile. Java allows the optional specifiers to appear before the access modifier.

• Chapter 4: Methods and Encapsulation

Designing Methods

Return Type

Methods with a return type other than void are required to have a return statement inside the method body. This return statement must include the primitive or object to be returned. Methods that have a return type of void are permitted to have a return statement with no value returned or omit the return statement entirely.

```
public void walk1() { }
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() { } // DOES NOT COMPILE
public walk5() { } // DOES NOT COMPILE
String walk6(int a) { if (a == 4) return ""; } // DOES NOT COMPILE
```

Method Name

```
public void walk1() { }
public void 2walk() { } // DOES NOT COMPILE
public walk3 void() { } // DOES NOT COMPILE
public void Walk_$( ) { }
public void() { } // DOES NOT COMPILE
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Parameter List

Although the parameter list is required, it doesn't have to contain any parameters. This means you can just have an empty pair of parentheses after the method name, such as `void nap(){}`.

```
public void walk1() { }
public void walk2 { } // DOES NOT COMPILE
public void walk3(int a) { }
public void walk4(int a; int b) { } // DOES NOT COMPILE
public void walk5(int a, int b) { }
```

Optional Exception List

```
public void zeroExceptions() { }
public void oneException() throws IllegalArgumentException { }
public void twoExceptions() throws
    IllegalArgumentException, InterruptedException { }
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Method Body

The final part of a method declaration is the method body (except for abstract methods and interfaces, but you don't need to know about either of those until next chapter). A method body is simply a code block. It has braces that contain zero or more Java statements.

```
public void walk1() { }
public void walk2; // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

Working with Varargs

As you saw in the previous chapter, a method may use a vararg parameter (variable argument) as if it is an array. It is a little different than an array, though. A vararg parameter must be the last element in a method's parameter list. This implies you are only allowed to have one vararg parameter per method.

```
public void walk1(int... nums) { }
public void walk2(int start, int... nums) { }
public void walk3(int... nums, int start) { } // DOES NOT COMPILE
public void walk4(int... start, int... nums) { } // DOES NOT COMPILE
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Working with Varargs

```
15: public static void walk(int start, int... nums) {  
16:     System.out.println(nums.length);  
17: }  
18: public static void main(String[] args) {  
19:     walk(1);                      // 0  
20:     walk(1, 2);                  // 1  
21:     walk(1, 2, 3);              // 2  
22:     walk(1, new int[] {4, 5});    // 2  
23: }  
  
walk(1, null);      // throws a NullPointerException
```

Since null isn't an int, Java treats it as an array reference that happens to be null. It just passes on the null array object to walk. Then the walk() method throws an exception because it tries to determine the length of null.

• Chapter 4: Methods and Encapsulation

Designing Methods

Working with Varargs

```
16: public static void run(int... nums) {  
17:     System.out.println(nums[1]);  
18: }  
19: public static void main(String[] args) {  
20:     run(11, 22);      // 22  
21: }
```

Accessing a vararg parameter is also just like accessing an array.
It uses array indexing.

Applying Access Modifiers

- **private:** Only accessible within the same class
- **default (package private) access:** private and other classes in the same package
- **protected:** default access and child classes
- **public:** protected and classes in the other packages

• Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Private Access

Private access is easy. Only code in the same class can call private methods or access private fields.

```
1: package pond.duck;
2: public class FatherDuck {
3:   private String noise = "quack";
4:   private void quack() {
5:     System.out.println(noise);    // private access is ok
6:   }
7:   private void makeNoise() {
8:     quack();                  // private access is ok
9:   } }
```

```
1: package pond.duck;
2: public class BadDuckling {
3:   public void makeNoise() {
4:     FatherDuck duck = new FatherDuck();
5:     duck.quack();           // DOES NOT COMPILE
6:     System.out.println(duck.noise); // DOES NOT COMPILE
7:   } }
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Default (Package Private) Access

When there is no access modifier, Java uses the default, which is package private access. This means that the member is “private” to classes in the same package. In other words, only classes in the package may access it.

```
package pond.duck;  
  
public class MotherDuck {  
    String noise = "quack";  
    void quack() {  
        System.out.println(noise);      // default access is ok  
    }  
    private void makeNoise() {  
        quack();                      // default access is ok  
    } }
```

```
package pond.duck;  
  
public class GoodDuckling {  
    public void makeNoise() {  
        MotherDuck duck = new MotherDuck();  
        duck.quack();                  // default access  
        System.out.println(duck.noise);  // default access  
    } }
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Default (Package Private) Access

```
package pond.swan;
import pond.duck.MotherDuck;          // import another package

public class BadCygnet {
    public void makeNoise() {
        MotherDuck duck = new MotherDuck();
        duck.quack();                  // DOES NOT COMPILE
        System.out.println(duck.noise); // DOES NOT COMPILE
    }
}
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Protected Access

Protected access allows everything that default (package private) access allows and more. The protected access modifier adds the ability to access members of a parent class.

```
package pond.shore;
public class Bird {
    protected String text = "floating";          // protected access
    protected void floatInWater() {               // protected access
        System.out.println(text);
    }
}

package pond.goose;
import pond.shore.Bird;                      // in a different package
public class Gosling extends Bird {           // extends means create subclass
    public void swim() {
        floatInWater();                      // calling protected member
        System.out.println(text);            // calling protected member
    }
}
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Protected Access

```
package pond.shore;           // same package as Bird
public class BirdWatcher {
    public void watchBird() {
        Bird bird = new Bird();
        bird.floatInWater();      // calling protected member
        System.out.println(bird.text); // calling protected member
    } }
```

```
package pond.inland;
import pond.shore.Bird;       // different package than Bird
public class BirdWatcherFromAfar {
    public void watchBird() {
        Bird bird = new Bird();
        bird.floatInWater();      // DOES NOT COMPILE
        System.out.println(bird.text); // DOES NOT COMPILE
    } }
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Protected Access

```
1: package pond.swan;
2: import pond.shore.Bird;      // in different package than Bird
3: public class Swan extends Bird {    // but subclass of bird
4:     public void swim() {
5:         floatInWater();          // package access to superclass
6:         System.out.println(text); // package access to superclass
7:     }
8:     public void helpOtherSwanSwim() {
9:         Swan other = new Swan();
10:        other.floatInWater();       // package access to superclass
11:        System.out.println(other.text); // package access to superclass
12:    }
13:    public void helpOtherBirdSwim() {
14:        Bird other = new Bird();
15:        other.floatInWater();       // DOES NOT COMPILE
16:        System.out.println(other.text); // DOES NOT COMPILE
17:    }
18: }
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Public Access

Public means anyone can access the member from anywhere.

```
package pond.duck;
public class DuckTeacher {
    public String name = "helpful";      // public access
    public void swim() {                  // public access
        System.out.println("swim");
    }
}

package pond.goose;
import pond.duck.DuckTeacher;
public class LostDuckling {
    public void swim() {
        DuckTeacher teacher = new DuckTeacher();
        teacher.swim();                  // allowed
        System.out.println("Thanks" + teacher.name); // allowed
    }
}
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Applying Access Modifiers

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in the same class	Yes	Yes	Yes	Yes
Member in another class in same package	No	Yes	Yes	Yes

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in a superclass in a different package	No	No	Yes	Yes
Method/field in a non-superclass class in a different package	No	No	No	Yes

• Chapter 4: Methods and Encapsulation

Designing Methods

Designing Static Methods and Fields

Static methods don't require an instance of the class. They are shared among all users of the class.

```
public class Koala {  
    public static int count = 0;           // static variable  
    public static void main(String[] args) { // static method  
        System.out.println(count);  
    }  
}
```

Static vs. Instance

```
public class Static {  
    private String name = "Static class";  
    public static void first() { }  
    public static void second() { }  
    public void third() { System.out.println(name); }  
    public static void main(String args[]) {  
        first();  
        second();  
        third();      // DOES NOT COMPILE  
    } }
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Designing Static Methods and Fields

Type	Calling	Legal?	How?
Static method	Another static method or variable	Yes	Using the classname
Static method	An instance method or variable	No	
Instance method	A static method or variable	Yes	Using the classname or a reference variable
Instance method	Another instance method or variable	Yes	Using a reference variable

- Chapter 4: Methods and Encapsulation

Designing Methods

Static Variables

Some static variables are meant to change as the program runs. Counters are a common example of this. We want the count to increase over time. Just as with instance variables, you can initialize a static variable on the line it is declared:

```
public class Initializers {  
    private static int counter = 0; // initialization  
}
```

Uses the final modifier to ensure the variable never changes. *static final* constants use a different naming convention than other variables. They use all uppercase letters with underscores between “words.” For example:

```
public class Initializers {  
    private static final int NUM_BUCKETS = 45;  
    public static void main(String[] args) {  
        NUM_BUCKETS = 5; // DOES NOT COMPILE  
    } }
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Static Variables

```
private static final ArrayList<String> values = new ArrayList<>();  
public static void main(String[] args) {  
    values.add("changed");  
}
```

It actually does compile. `values` is a reference variable. We are allowed to call methods on reference variables. All the compiler can do is check that we don't try to reassign the `final` `values` to point to a different object.

- Chapter 4: Methods and Encapsulation

Designing Methods

Static Initialization

```
14: private static int one;
15: private static final int two;
16: private static final int three = 3;
17: private static final int four;      // DOES NOT COMPILE
18: static {
19:     one = 1;
20:     two = 2;
21:     three = 3;      // DOES NOT COMPILE
22:     two = 4;      // DOES NOT COMPILE
23: }
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Static Initialization

```
public class Person {  
    {  
        personCount = 10;  
    }  
    static {  
        personCount = 20;  
    }  
    private static int personCount = 1000;  
    {  
        personCount = 30;  
    }  
    static {  
        personCount = 40;  
    }  
    public static void main(String[] args) {  
        System.out.println(Person.personCount); // Line A  
        Person p = new Person(); // Line B  
        System.out.println(Person.personCount); // Line C  
        Person.personCount = 100; // Line D  
        p = new Person(); // Line E  
        System.out.println(Person.personCount); // Line F  
    }  
}
```

• Chapter 4: Methods and Encapsulation

Designing Methods

Static Imports

Static imports are for importing static members of classes. Just like regular imports, you can use a wildcard or import a specific member. The idea is that you shouldn't have to specify where each static method or variable comes from each time you use it.

```
import java.util.List;
import java.util.Arrays;
public class Imports {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("one", "two");
    }
}
```

```
import java.util.List;
import static java.util.Arrays.asList; // static import
public class StaticImports {
    public static void main(String[] args) {
        List<String> list = asList("one", "two"); // no Arrays.
    }
}
```

- Chapter 4: Methods and Encapsulation

Designing Methods

Static Imports

The exam will try to trick you with misusing static imports. This example shows almost everything you can do wrong. Can you figure out what is wrong with each one?

```
1: import static java.util.Arrays;      // DOES NOT COMPILE
2: import static java.util.Arrays.asList;
3: static import java.util.Arrays.*;    // DOES NOT COMPILE
4: public class BadStaticImports {
5:     public static void main(String[] args) {
6:         Arrays.asList("one"); // DOES NOT COMPILE
7:     } }
```

• Chapter 4: Methods and Encapsulation

Passing Data Among Methods

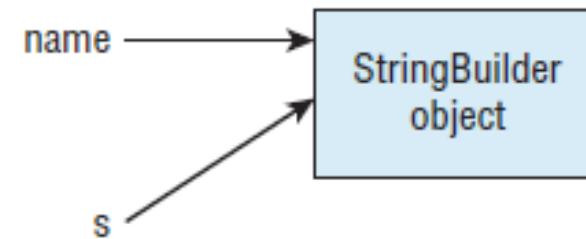
Java is a “pass-by-value” language. This means that a copy of the variable is made and the method receives that copy. Assignments made in the method do not affect the caller.

```
2: public static void main(String[] args) {  
3: int num = 4;  
4: newNumber(5);  
5: System.out.println(num); // 4  
6: }  
7: public static void newNumber(int num) {  
8: num = 8;  
9: }
```

```
public static void main(String[] args) {  
String name = "Webby";  
speak(name);  
System.out.println(name); // Webby  
}  
public static void speak(String name) {  
name = "Sparky";  
}
```

```
public static void main(String[] args) {  
StringBuilder name = new StringBuilder();  
speak(name);  
System.out.println(name); // Webby  
}  
public static void speak(StringBuilder s) {  
s.append("Webby");  
}
```

Copying a reference with pass-by-value



- Chapter 4: Methods and Encapsulation

Passing Data Among Methods

```
1: public class ReturningValues {  
2: public static void main(String[] args) {  
3: int number = 1;  
4: String letters = "abc";  
5: number(number);  
6: letters = letters(letters);  
7: System.out.println(number + letters); // 1abcd  
8: }  
9: public static int number(int number) {  
10: number++;  
11: return number;  
12: }  
13: public static String letters(String letters) {  
14: letters += "d";  
15: return letters;  
16: }  
17: }
```

- Chapter 4: Methods and Encapsulation

Overloading Methods

Method overloading occurs when there are different method signatures with the same name but different type parameters.

```
public double getMyFundsFromBank(String bankName)
```

signature is method name +
parameters only

```
public void fly(int numMiles) { }  
public void fly(short numFeet) { }  
public boolean fly() { return false; }  
void fly(int numMiles, short numFeet) { }  
public void fly(short numFeet, int numMiles) throws Exception { }
```

- Chapter 4: Methods and Encapsulation

Overloading Methods

Method overloading occurs when there are different method signatures with the same name but different type parameters.

```
public double getMyFundsFromBank(String bankName)
```

signature is method name +
parameters only

```
public void fly(int numMiles) { }  
public void fly(short numFeet) { }  
public boolean fly() { return false; }  
void fly(int numMiles, short numFeet) { }  
public void fly(short numFeet, int numMiles) throws Exception { }
```

- Chapter 4: Methods and Encapsulation

Overloading Methods

```
public void fly(int numMiles) { }
public int fly(int numMiles) { }      // DOES NOT COMPILE
```

```
public void fly(int numMiles) { }
public static void fly(int numMiles) { }      // DOES NOT COMPILE
```

```
public void fly(int[] lengths) { }
public void fly(int... lengths) { }      // DOES NOT COMPILE
```

Autoboxing

```
public void fly(int numMiles) { }
public void fly(Integer numMiles) { }
```

Java will match the int numMiles version. Java tries to use the most specific parameter list it can find. When the primitive int version isn't present, it will autobox. However, when the primitive int version is provided, there is no reason for Java to do the extra work of autoboxing.

• Chapter 4: Methods and Encapsulation

Overloading Methods

Reference Types

```
public class ReferenceTypes {  
    public void fly(String s) {  
        System.out.print("string ");  
    }  
    public void fly(Object o) {  
        System.out.print("object ");  
    }  
    public static void main(String[] args) {  
        ReferenceTypes r = new ReferenceTypes();  
        r.fly("test");  
        r.fly(56);  
    } }
```

Rule	Example of what will be chosen for <code>glide(1,2)</code>
Exact match by type	<code>public String glide(int i, int j) {}</code>
Larger primitive type	<code>public String glide(long i, long j) {}</code>
Autoboxed type	<code>public String glide(Integer i, Integer j) {}</code>
Varargs	<code>public String glide(int... nums) {}</code>

The first call is a String and finds a direct match. There's no reason to use the Object version when there is a nice String parameter list just waiting to be called. The second call looks for an int parameter list. When it doesn't find one, it autoboxes to Integer. Since it still doesn't find a match, it goes to the Object one.

• Chapter 4: Methods and Encapsulation

Creating Constructors

Overloaded constructors often call each other. One common technique is to have each constructor add one parameter until getting to the constructor that does all the work. This approach is called *constructor chaining*. In this example, all three constructors are chained.

```
public class Mouse {  
    private int numTeeth;  
    private int numWhiskers;  
    private int weight;  
  
    public Mouse(int weight) {  
        this(weight, 16); // calls constructor with 2 parameters  
    }  
  
    public Mouse(int weight, int numTeeth) {  
        this(weight, numTeeth, 6); // calls constructor with 3  
        parameters  
    }
```

```
public Mouse(int weight, int numTeeth, int numWhiskers) {  
    this.weight = weight;  
    this.numTeeth = numTeeth;  
    this.numWhiskers = numWhiskers;  
}  
  
public void print() {  
    System.out.println(weight + " " + numTeeth + " " +  
    numWhiskers);  
}  
  
public static void main(String[] args) {  
    Mouse mouse = new Mouse(15);  
    mouse.print();  
}
```

- Chapter 4: Methods and Encapsulation

Encapsulating Data

Encapsulation means we set up the class so only methods in the class with the variables can refer to the instance variables. Callers are required to use these methods.

```
1: public class Swan {  
2: private int numberEggs; // private  
3: public int getNumberEggs() { // getter  
4: return numberEggs;  
5: }  
6: public void setNumberEggs(int numberEggs) { // setter  
7: if (numberEggs >= 0) // guard condition  
8: this.numberEggs = numberEggs;  
9: } }
```

• Chapter 4: Methods and Encapsulation

Encapsulating Data

Rules for JavaBeans naming conventions

Rule	Example
Properties are private.	<code>private int numEggs;</code>
Getter methods begin with <code>is</code> if the property is a boolean.	<code>public boolean isHappy() { return happy; }</code>
Getter methods begin with <code>get</code> if the property is not a boolean.	<code>public int getNumEggs() { return numEggs; }</code>
Setter methods begin with <code>set</code> .	<code>public void setHappy(boolean happy) { this.happy = happy; }</code>
The method name must have a prefix of <code>set/get/is</code> , followed by the first letter of the property in uppercase, followed by the rest of the property name.	<code>public void setNumEggs(int num) { numEggs = num; }</code>

- Chapter 4: Methods and Encapsulation

Encapsulating Data

Creating Immutable Classes

Another common technique is making classes immutable so they cannot be changed at all. Immutable classes are helpful because you know they will always be the same.

One step in making a class immutable is to omit the setters. But wait: we still want the caller to be able to specify the initial value—we just don't want it to change after the object is created. Constructors to the rescue!

```
public class ImmutableSwan {  
    private int numberEggs;  
    public ImmutableSwan(int numberEggs) {  
        this.numberEggs = numberEggs;  
    }  
    public int getNumberEggs() {  
        return numberEggs;  
    } }
```

- Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

In Java 8, the language added the ability to write code using another style. *Functional programming* is a way of writing code more declaratively.

Functional programming uses lambda expressions to write code. A *lambda expression* is a block of code that gets passed around. You can think of a lambda expression as an anonymous method.

It has parameters and a body just like full-fledged methods do, but it doesn't have a name like a real method.

In other words, a lambda expression is like a method that you can pass as if it were a variable.

- Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

Lambda Example

```
public class Animal {  
    private String species;  
    private boolean canHop;  
    private boolean canSwim;  
    public Animal(String speciesName, boolean hopper,  
    boolean swimmer) {  
        species = speciesName;  
        canHop = hopper;  
        canSwim = swimmer;  
    }  
    public boolean canHop() { return canHop; }  
    public boolean canSwim() { return canSwim; }  
    public String toString() { return species; }  
}
```

```
public interface CheckTrait {  
    boolean test(Animal a);  
}
```

```
public class CheckIfHopper implements CheckTrait {  
    public boolean test(Animal a) {  
        return a.canHop();  
    }  
}
```

• Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

Lambda Example

```
1: public class TraditionalSearch {  
2:     public static void main(String[] args) {  
3:         List<Animal> animals = new ArrayList<Animal>(); // list of animals  
4:         animals.add(new Animal("fish", false, true));  
5:         animals.add(new Animal("kangaroo", true, false));  
6:         animals.add(new Animal("rabbit", true, false));  
7:         animals.add(new Animal("turtle", false, true));  
8:  
9:         print(animals, new CheckIfHopper()); // pass class that does check  
10:    }  
11:    private static void print(List<Animal> animals, CheckTrait checker) {  
12:        for (Animal animal : animals) {  
13:            if (checker.test(animal)) // the general check  
14:                System.out.print(animal + " ");  
15:        }  
16:        System.out.println();  
17:    }  
18: }
```

Now what happens if we want to print the Animals that swim? Sigh. We need to write another class `CheckIfSwims`.

Or simply, we could replace line 9 with the following, which uses a lambda:

9: `print(animals, a -> a.canSwim());`

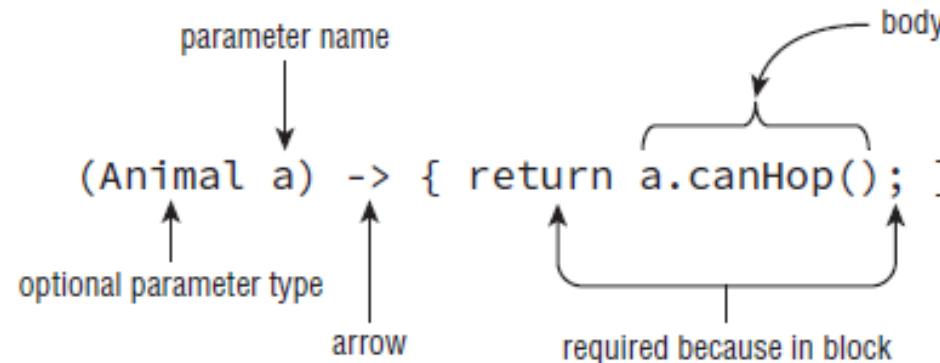
How about Animals that cannot swim?
`print(animals, a -> ! a.canSwim());`

• Chapter 4: Methods and Encapsulation

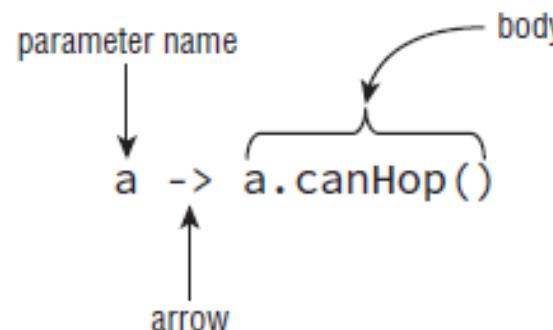
Writing Simple Lambdas

Lambda syntax

Lambda syntax, including optional parts



Lambda syntax omitting optional parts



• Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

Lambda syntax

```
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

(a, b) -> { int a = 0; return 5;}                   // DOES NOT COMPILE

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

- Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

Predicates

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

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);  
}
```

• Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

Predicates

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:     public static void main(String[] args) {
5:         List<Animal> animals = new ArrayList<Animal>();
6:         animals.add(new Animal("fish", false, true));
7:
8:         print(animals, a -> a.canHop());
9:     }
10:    private static void print(List<Animal> animals, Predicate<Animal> checker) {
11:        for (Animal animal : animals) {
12:            if (checker.test(animal))
13:                System.out.print(animal + " ");
14:        }
15:        System.out.println();
16:    }
17: }
```

- Chapter 4: Methods and Encapsulation

Writing Simple Lambdas

Predicates

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]
```

QCM Chapitre 4

OCAJP-IZ0-808

Q1

B, C

Which of the following can fill in the blank in this code to make it compile? (Choose all that apply)

```
public class Ant {  
    _____ void method() {}  
}
```

- A. default
- B. final

- C. private
- D. Public
- E. String
- F. zzz:

Q2

A, D

Which of the following compile?
(Choose all that apply)

- A. final static void method4() { }
- B. public final int void method() { }
- C. private void int method() { }
- D. static final void method3() { }
- E. void final method() {}
- F. void public method() { }

Q3

A, C, D

Which of the following methods compile? (Choose all that apply)

- A. public void methodA() { return;}
- B. public void methodB() { return null;}
- C. public void methodD() {}
- D. public int methodD() { return 9;}
- E. public int methodE() { return 9.0;}
- F. public int methodF() { return;}
- G. public int methodG() { return null;}

Q4

A, B, G

- Which of the following compile? (Choose all that apply)
 - A. public void moreA(int... nums) {}
 - B. public void moreB(String values, int... nums) {}
 - C. public void moreC(int... nums, String values) {}
 - D. public void moreD(String... values, int... nums) {}
 - E. public void moreE(String[] values, ...int nums) {}
 - F. public void moreF(String... values, int[] nums) {}
 - G. public void moreG(String[] values, int[] nums) {}

Q5

D, F, G

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]);

Q6

D

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

- A. Package private access is more lenient than protected access.
- B. A public class that has private fields and package private methods is not visible to classes outside the package.
- C. You can use access modifiers so only some of the classes in a

package see a particular package private class.

- D. You can use access modifiers to allow read access to all methods, but not any instance variables.
- E. You can use access modifiers to restrict read access to all classes that begin with the word Test.

Q7

B, C, D, F

Given the following my.school.ClassRoom and my.city.School class definitions, which line numbers in main() generate a compiler error? (Choose all that apply)

```
1: package my.school;
2: public class Classroom {
3: private int roomNumber;
4: protected String teacherName;
5: static int globalKey = 54321;
6: public int floor = 3;
7: Classroom(int r, String t) {
8: roomNumber = r;
9: teacherName = t; } }
```

```
1: package my.city;
2: import my.school.*;
```

```
3: public class School {
4: public static void main(String[] args) {
5: System.out.println(Classroom.globalKey);
6: Classroom room = new Classroom(101, ""Mrs.
Anderson");
7: System.out.println(room.roomNumber);
8: System.out.println(room.floor);
9: System.out.println(room.teacherName); } }
```

- A. None, the code compiles fine.
- B. Line 5
- C. Line 6
- D. Line 7
- E. Line 8
- F. Line 9

Q8

B, C, E

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.

Q9

C, E

Which are methods using JavaBeans naming conventions for accessors and mutators?

(Choose all that apply)

A. public boolean getCanSwim() {
return canSwim;}

B. public boolean canSwim() {
return numberWings;}

C. public int getNumWings() {

return numberWings;}

D. public int numWings() { return
numberWings;}

E. public void setCanSwim(boolean
b) { canSwim = b;}

Q10

B

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.

Q11

B, E

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.

Q12

D

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.
```

Q13

E

How many compiler errors are in the following code?

```
1: public class RopeSwing {  
2: private static final String leftRope;  
3: private static final String rightRope;  
4: private static final String bench;  
5: private static final String name = "name";  
6: static {  
7: leftRope = "left";  
8: rightRope = "right";  
9: }  
10: static {  
11: name = "name";
```

```
12: rightRope = "right";  
13: }  
14: public static void main(String[] args) {  
15: bench = "bench";  
16: }  
17: }  
A. 0  
B. 1  
C. 2  
D. 3  
E. 4  
F. 5
```

Q14

B

Which of the following can replace line 2 to make this code compile? (Choose all 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>
;)

Q15

E

What is the result of the following statements?

```
1: public class Test {  
2:   public void print(byte x) {  
3:     System.out.print("byte");  
4:   }  
5:   public void print(int x) {  
6:     System.out.print("int");  
7:   }  
8:   public void print(float x) {  
9:     System.out.print("float");  
10: }  
11:  public void print(Object x) {  
12:    System.out.print("Object");  
13: }
```

```
14: public static void main(String[] args) {  
15:   Test t = new Test();  
16:   short s = 123;  
17:   t.print(s);  
18:   t.print(true);  
19:   t.print(6.789);  
20: }  
21: }  
A. bytefloatObject  
B. intfloatObject  
C. byteObjectfloat  
D. intObjectfloat  
E. intObjectObject  
F. byteObjectObject
```

Q16

B

What is the result of the following program?

```
1: public class Squares {  
2:     public static long square(int x) {  
3:         long y = x * (long) x;  
4:         x = -1;  
5:         return y;  
6:     }  
7:     public static void main(String[] args)  
{
```

- 8: int value = 9;
 - 9: long result = square(value);
 - 10: System.out.println(value);
 - 11: } }
- A. -1
 - B. 9
 - C. 81
 - D. Compiler error on line 9.
 - E. Compiler error on a different line.

Q17

B, D, E

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

```
public class StringBuilders {  
    public static StringBuilder  
    work(StringBuilder a, StringBuilder b) {  
        a = new StringBuilder("a");  
        b.append("b");  
        return a;  
    }  
    public static void main(String[] args) {  
        StringBuilder s1 = new StringBuilder("s1");  
        StringBuilder s2 = new StringBuilder("s2");  
        StringBuilder s3 = work(s1, s2);  
    }  
}
```

```
System.out.println("s1 = " + s1);  
System.out.println("s2 = " + s2);  
System.out.println("s3 = " + s3);  
}  
}  
A. s1 = a  
B. s1 = s1  
C. s2 = s2  
D. s2 = s2b  
E. s3 = a  
F. s3 = null  
G. The code does not compile.
```

Q18

C, G

Which of the following are true?
(Choose 2)

- A. `this()` can be called from anywhere in a constructor.
- B. `this()` can be called from any instance method in the class.
- C. `this.variableName` can be called from any instance method in the class.
- D. `this.variableName` can be called

from any static method in the class.
E. You must include a default constructor in the code if the compiler does not include one.
F. You can call the default constructor written by the compiler using `this()`.
G. You can access a private constructor with the `main()` method.

Q19

A, G

Which of these classes compile and use a default constructor? (Choose all that apply)

- A. public class Bird { }
- B. public class Bird { public bird() {} }
- C. public class Bird { public bird(String name) {} }
- D. public class Bird { public Bird() {} }
- E. public class Bird { Bird(String name)

- { } }
- F. public class Bird { private Bird(int age) {} }
- G. public class Bird { void Bird() {} }

Q20

E

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);

Q21

C

Which of the following complete the constructor so that this code prints out 50? (Choose all that apply)

```
public class Cheetah {  
    int numSpots;  
    public Cheetah(int numSpots) {  
        // INSERT CODE HERE  
    }  
    public static void main(String[] args) {
```

```
        System.out.println(new  
            Cheetah(50).numSpots);  
    }  
}
```

- A. numSpots = numSpots;
- B. numSpots = this.numSpots;
- C. this.numSpots = numSpots;
- D. numSpots = super.numSpots;
- E. super.numSpots = numSpots;
- F. None of the above.

Q22

E

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.

Q23

A

What is the result of the following?

```
1: public class Order {  
2:     String value = "t";  
3:     { value += "a"; }  
4:     { value += "c"; }  
5:     public Order() {  
6:         value += "b";  
7:     }  
8:     public Order(String s) {  
9:         value += s;  
10:    }  
11:    public static void main(String[] args) {
```

```
12:         Order order = new Order("f");  
13:         order = new Order();  
14:         System.out.println(order.value);  
15:     } }
```

- A. tacb
- B. tacf
- C. tacbf
- D. tacfb
- E. tacftacb
- F. The code does not compile.
- G. An exception is thrown.

Q24

B, C, E

Which of the following will compile when inserted in the following code? (Choose all that apply)

```
public class Order3 {  
    final String value1 = "1";  
    static String value2 = "2";  
    String value3 = "3";  
}  
// CODE SNIPPET 1  
  
static {  
// CODE SNIPPET 2
```

```
    }  
}
```

- A. value1 = "d"; instead of // CODE SNIPPET 1
- B. value2 = "e"; instead of // CODE SNIPPET 1
- C. value3 = "f"; instead of // CODE SNIPPET 1
- D. value1 = "g"; instead of // CODE SNIPPET 2
- E. value2 = "h"; instead of // CODE SNIPPET 2
- F. value3 = "i"; instead of // CODE SNIPPET 2

Q25

A, E

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).

Q26

A

What is the result of the following class?

```
1: import java.util.function.*;
2:
3: public class Panda {
4: int age;
5: public static void main(String[] args) {
6: Panda p1 = new Panda();
7: p1.age = 1;
8: check(p1, p -> p.age < 5);
9: }
10: private static void check(Panda panda, Predicate<Panda> pred) {
11: String result = pred.test(panda) ?
"match" : "not match";
12: System.out.print(result);
13: } }
```

- A. match
- B. not match
- C. Compiler error on line 8.
- D. Compiler error on line 10.
- E. Compiler error on line 11.
- F. A runtime exception is thrown.

Q27

C

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.

## Q28

A, D, F

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

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

```
list.removeIf(_____);
```

- 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()

## Q29

A, F

Which lambda can replace the MySecret A. caller((e) -> "Poof");  
class to return the same value? (Choose all that apply)

interface Secret { String magic(double d);  
}

class MySecret implements Secret {  
 public String magic(double d) {  
 return "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"; });

# Tests

1 minute and 30 seconds

# Q1

Which of the following method signatures does not contain a compiler error?

- A. public void sing(String key, String... harmonies)
- B. public void sing(int note, String... sound, int music)
- C. public void sing(String... keys, String... pitches)

D. public void sing(String... notes, String melodies)

## Q2

Which of the following is not a reason to use encapsulation when designing a class?

- A. Promote usability by other developers.
- B. Maintain class data integrity of data elements.
- C. Prevent users from modifying the internal attributes of a class.

D. Increase concurrency and improve performance.

# Q3

How many final modifiers would need to be removed for this application to compile?

```
package end;
public final class Games {
 public final static int finish(final int score)
 {
 final int win = 3;
 final int result = score++ < 5 ? 2 : win;
 return result+=win;
 }
}
```

```
 public static void main(final String[] v) {
 System.out.print(finish(Integer.parseInt(v[0])));
 }
}
```

- 01:30,00
- A. None
  - B. One
  - C. Two
  - D. The code will not compile regardless of the number of final modifiers that are removed.

# Q4

What is a possible output of the following application?

```
package wrap;
public class Gift {
 private final Object contents;
 protected Object getContents() {
 return contents;
 }
 protected void setContents(Object
 contents) {
 this.contents = contents;
 }
 public void showPresent() {
```

```
 System.out.print("Your gift: "+contents);
 }
 public static void main(String[] treats) {
 Gift gift = new Gift();
 gift.setContents(gift);
 gift.showPresent();
 }
}
```

- A. Your gift: wrap.Gift@29ca2745
- B. Your gift: Your gift:
- C. It does not compile.
- D. It compiles but throws an exception at runtime.

# Q5

Given the following two classes, each in a different package, which line inserted below allows the second class to compile?

```
package clothes;
public class Store {
 public static String getClothes() { return
 "dress"; }
}
package wardrobe;
// INSERT CODE HERE
```

```
public class Closet {
 public void borrow() {
 System.out.print("Borrowing clothes:
 "+getClothes());
 }
}
```

- A. static import clothes.Store.getClothes;
- B. import clothes.Store.\*;
- C. import static clothes.Store.getClothes;
- D. import static clothes.Store;

# Q6

What is the output of the following application?

```
package ship;
public class Phone {
 private int size;
 public Phone(int size) {this.size=size;}
 public static void sendHome(Phone p,
 int newSize) {
 p = new Phone(newSize);
 p.size = 4;
 }
}
```

```
public static final void main(String...
 params) {
```

```
 final Phone phone = new Phone(3);
 sendHome(phone,7);
 System.out.print(phone.size);
```

- A. 3
- B. 4
- C. 7
- D. The code does not compile.

# Q7

What is the output of the following application?

```
public class ChooseWisely {
 public ChooseWisely() { super(); }
 public int choose(int choice) { return
 5; }
 public int choose(short choice) {
 return 2; }
 public int choose(long choice) { return
 11; }
```

```
 public static void main(String[] path) {
 System.out.print(new
 ChooseWisely().choose((byte)2+1));
 }
 }
```

- A. 5
- B. 2
- C. 11
- D. The code does not compile.

## Q8

Which portion of code can be removed so that this line of code continues to compile?

```
Predicate<StringBuilder> p = (StringBuilder b) -> {return true;};
```

- A. Remove StringBuilder b
- B. Remove ->
- C. Remove { and ;}
- D. Remove { return and ;}

## Q9

How many of these lines fail to compile? D. Four

Predicate<String> pred1 = s -> false;

Predicate<String> pred2 = (s) -> false;

Predicate<String> pred3 = String s -> false;

Predicate<String> pred4 = (String s) -> false;

A. One

B. Two

C. Three

# Q10

How many lines does this code output?

```
import java.util.*;
import java.util.function.*;
public class PrintNegative {
 public static void main(String[] args) {
 List<String> list = new ArrayList<>();
 list.add("-5");
 list.add("0");
 list.add("5");
 print(list, e -> e < 0);
 }
}
```

```
 public static void print(List<String> list,
 Predicate<Integer> p) {
 for (String num : list)
 if (p.test(Integer.parseInt(num)))
 System.out.println(num);
 }
}
```

- A. One
- B. Two
- C. None. The code does not compile.
- D. None. The code throws an exception at runtime.

# Corrections

# Q1

A

Which of the following method signatures does not contain a compiler error?

- A. public void sing(String key, String... harmonies)
- B. public void sing(int note, String... sound, int music)
- C. public void sing(String... keys, String... pitches)
- D. public void sing(String... notes, String melodies)

Q2

D

Which of the following is not a reason to use encapsulation when designing a class?

- A. Promote usability by other developers.
- B. Maintain class data integrity of data elements.
- C. Prevent users from modifying the internal attributes of a class.

D. Increase concurrency and improve performance.

# Q3

C

How many final modifiers would need to be removed for this application to compile?

```
public final class Games {
 public final static int finish(final int score) {
 final int win = 3;
 final int result = score++ < 5 ? 2 : win;
 return result+=win;
 }
}

public static void main(final String[] v) {
 System.out.print(finish(Integer.parseInt(v[0])));
}
```

- A. None
- B. One
- C. Two
- D. The code will not compile regardless of the number of final modifiers that are removed.

# Q4

C

What is a possible output of the following application?

```
package wrap;
public class Gift {
 private final Object contents;
 protected Object getContents() {
 return contents;
 }
 protected void setContents(Object contents) {
 this.contents = contents;
 }
 public void showPresent() {
```

```
 System.out.print("Your gift: "+contents);
 }
 public static void main(String[] treats) {
 Gift gift = new Gift();
 gift.setContents(gift);
 gift.showPresent();
 }
}
```

- A. Your gift: wrap.Gift@29ca2745
- B. Your gift: Your gift:
- C. It does not compile.
- D. It compiles but throws an exception at runtime.

# Q5

C

Given the following two classes, each in a different package, which line inserted below allows the second class to compile?

```
package clothes;
public class Store {
 public static String getClothes() { return
 "dress"; }
}
package wardrobe;
// INSERT CODE HERE
```

```
public class Closet {
 public void borrow() {
 System.out.print("Borrowing clothes:
 "+getClothes());
 }
}
```

- A. static import clothes.Store.getClothes;
- B. import clothes.Store.\*;
- C. import static clothes.Store.getClothes;
- D. import static clothes.Store;

# Q6

A

What is the output of the following application?

```
package ship;
public class Phone {
 private int size;
 public Phone(int size) {this.size=size;}
 public static void sendHome(Phone p,
 int newSize) {
 p = new Phone(newSize);
 p.size = 4;
 }
}
```

```
public static final void main(String...
 params) {
 final Phone phone = new Phone(3);
 sendHome(phone,7);
 System.out.print(phone.size);
}
```

- A. 3
- B. 4
- C. 7
- D. The code does not compile.

# Q7

A

What is the output of the following application?

```
public class ChooseWisely {
 public ChooseWisely() { super(); }
 public int choose(int choice) { return
 5; }
 public int choose(short choice) {
 return 2; }
 public int choose(long choice) { return
 11; }
```

```
 public static void main(String[] path) {
 System.out.print(new
 ChooseWisely().choose((byte)2+1));
 }
 }
```

- A. 5
- B. 2
- C. 11
- D. The code does not compile.

Q8

D

Which portion of code can be removed so that this line of code continues to compile?

Predicate<StringBuilder> p = (StringBuilder b) -> {return true;};

- A. Remove StringBuilder b
- B. Remove ->
- C. Remove { and ;}
- D. Remove { return and ;}

Q9

A

How many of these lines fail to compile? D. Four

Predicate<String> pred1 = s -> false;

Predicate<String> pred2 = (s) -> false;

Predicate<String> pred3 = String s -> false;

Predicate<String> pred4 = (String s) -> false;

A. One

B. Two

C. Three

# Q10

C

How many lines does this code output?

```
import java.util.*;
import java.util.function.*;
public class PrintNegative {
 public static void main(String[] args) {
 List<String> list = new ArrayList<>();
 list.add("-5");
 list.add("0");
 list.add("5");
 print(list, e -> e < 0);
 }
}
```

```
public static void print(List<String> list,
Predicate<Integer> p) {
 for (String num : list)
 if (p.test(Integer.parseInt(num)))
 System.out.println(num);
 }
}
```

- A. One
- B. Two
- C. None. The code does not compile.
- D. None. The code throws an exception at runtime.

# **Chapter 5: Class Design**

# • Chapter 5: Class Design

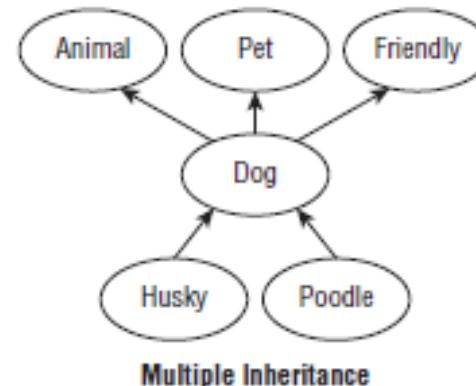
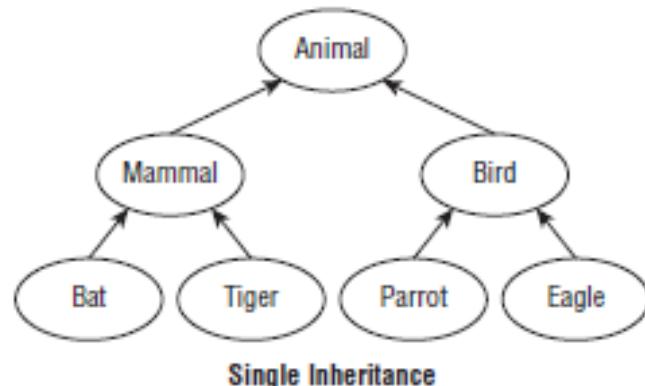
## Introducing Class Inheritance

When creating a new class in Java, you can define the class to **inherit** from an existing class. **Inheritance** is the process by which the new child subclass automatically includes any public or protected primitives, objects, or methods defined in the parent class.

we refer to any class that inherits from another class as a *child class*, or a descendent of that class.

Alternatively, we refer to the class that the child inherits from as the *parent class*, or an ancestor of the class.

Java supports **only single inheritance**, by which a class may inherit from only one direct parent class.



# • Chapter 5: Class Design

## Introducing Class Inheritance

### Extending a Class

The diagram illustrates the structure of a Java class definition with the following annotations:

- public or default access modifier**: Points to the word "public".
- abstract or final keyword (optional)**: Points to the word "abstract".
- class keyword (required)**: Points to the word "class".
- class name**: Points to the identifier "ElephantSeal".
- extends parent class (optional)**: Points to the word "extends". A bracket groups "extends" and "Seal", with an arrow pointing to the "Seal" identifier.

```
public abstract class ElephantSeal extends Seal {
 // Methods and Variables defined here
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Extending a Class

```
public class Animal {
 private int age;
 public int getAge() {
 return age;
 }
 public void setAge(int age) {
 this.age = age;
 }
}
```

```
public class Lion extends Animal {
 private void roar() {
 System.out.println("The "+getAge()+" year old lion says: Roar!");
 }
}

public class Lion extends Animal {
 private void roar() {
 System.out.println("The "+age+" year old lion says: Roar!");
 // DOES NOT COMPILE
 }
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Applying Class Access Modifiers

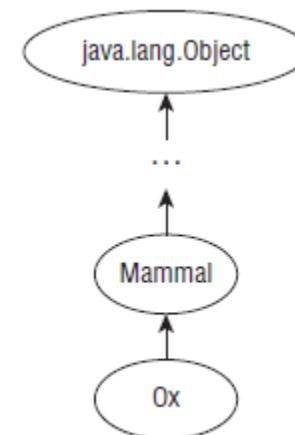
You can apply access modifiers (public, private, protected, default) to both class methods and variables. It probably comes as no surprise that you can also apply access modifiers to class definitions, since we have been adding the public access modifier to nearly every class up to now.

### Creating Java Objects

Throughout our discussion of Java in this book, we have thrown around the word *object* numerous times—and with good reason. In Java, all classes inherit from a single class, `java.lang.Object`. Furthermore, `java.lang.Object` is the only class that doesn't have any parent classes.

```
public class Zoo {
}

public class Zoo extends java.lang.Object {
}
```



All objects inherit `java.lang.Object`

414

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Defining Constructors

In Java, the first statement of every constructor is either a call to another constructor within the class, using `this()`, or a call to a constructor in the direct parent class, using `super()`.

```
public class Animal {
 private int age;
 public Animal(int age) {
 super();
 this.age = age;
 }
}
```

```
public class Zebra extends Animal {
 public Zebra(int age) {
 super(age);
 }
 public Zebra() {
 this(4);
 }
}
```

```
public class Zoo {
 public Zoo() {
 System.out.println("Zoo created");
 super(); // DOES NOT COMPILE
 }
}
```

```
public class Zoo {
 public Zoo() {
 super();
 System.out.println("Zoo created");
 super(); // DOES NOT COMPILE
 }
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Understanding Compiler Enhancements

```
public class Mammal {
 public Mammal(int age) {
 }
}
public class Elephant extends Mammal { // DOES NOT COMPILE
}
```

```
public class Elephant extends Mammal {
 public Elephant() { // DOES NOT COMPILE
 }
}
```

```
public class Elephant extends Mammal {
 public Elephant() {
 super(10);
 }
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Constructor Definition Rules

1. The first statement of every constructor is a call to another constructor within the class using this(), or a call to a constructor in the direct parent class using super().
2. The super() call may not be used after the first statement of the constructor.
3. If no super() call is declared in a constructor, Java will insert a no-argument super() as the first statement of the constructor.
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.

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Calling Constructors

```
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();
 }
}
```

Primate  
Ape

- Chapter 5: Class Design

## Introducing Class Inheritance

### *super() vs. super*

The exam may try to trick you by using both super() and super in a constructor.

```
public Rabbit(int age) {
 super();
 super.setAge(10);
}
```

```
public Rabbit(int age) {
 super; // DOES NOT COMPILE
 super().setAge(10); // DOES NOT COMPILE
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Overriding a Method

```
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());
 }
}
```

```
public double getAverageWeight() {
 return getAverageWeight()+20; // INFINITE LOOP
}
```

50.00

70.00

- Chapter 5: Class Design

## Introducing Class Inheritance

### Overriding a Method

Overriding a method is not without limitations, though. The compiler performs the following checks when you override a non private method:

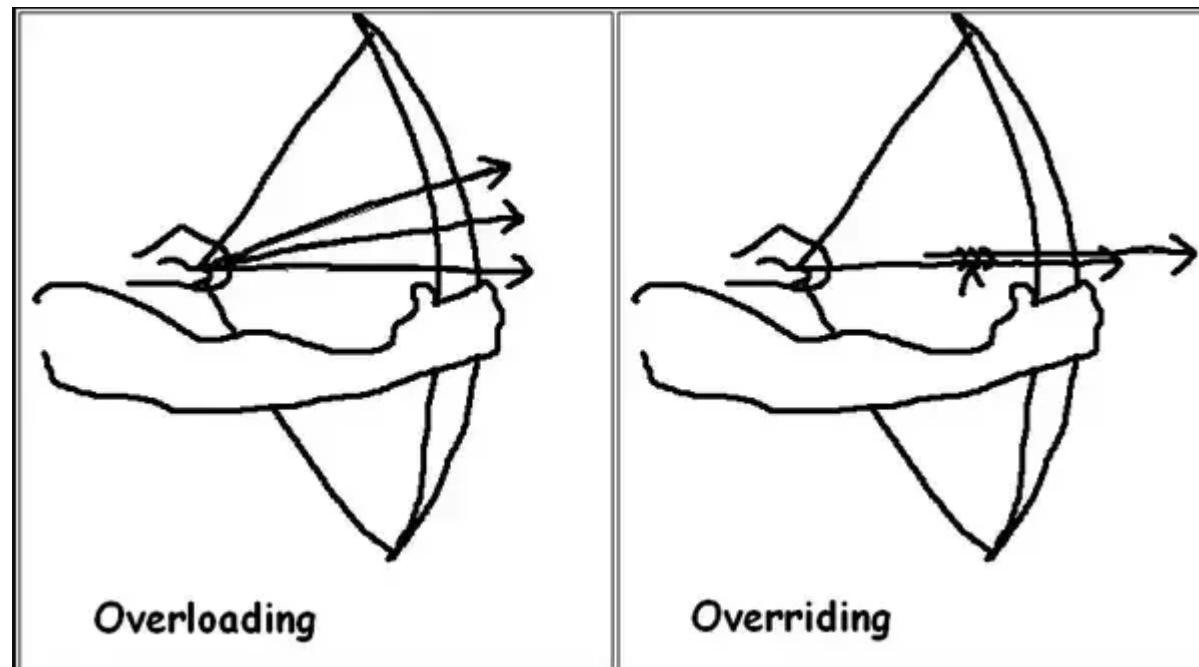
1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.

- Chapter 5: Class Design

## Introducing Class Inheritance

### Overloading vs. Overriding

Overloading a method and overriding a method are similar in that they both involve redefining a method using the same name. They differ in that an overloaded method will use a different signature than an overridden method.



# • Chapter 5: Class Design

## Introducing Class Inheritance

### 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, so none of the rules for overriding methods are invoked.

```
public class Camel {
 private String getNumberOfHumps() {
 return "Undefined";
 }
}
```

```
public class BactrianCamel extends Camel {
 private int getNumberOfHumps() {
 return 2;
 }
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Hiding Static Methods

A *hidden method* occurs when a child class defines a static method with the same name and signature as a static method defined in a parent class.

Method hiding is similar but not exactly the same as method overriding. The following list summarizes the five rules for hiding a 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 checked exception that is new or broader than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the 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).

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Creating *final* methods

final methods cannot be overridden and final class cannot be extended.

```
public class Bird {
 public final boolean hasFeathers() {
 return true;
 }
}

public class Penguin extends Bird {
 public final boolean hasFeathers() { // DOES NOT COMPILE
 return false;
 }
}
```

# • Chapter 5: Class Design

## Introducing Class Inheritance

### Hiding Variables

When you hide a variable, you define a variable with the same name as a variable in a parent class. This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference.

```
public class Rodent {
 protected int tailLength = 4;
 public void getRodentDetails() {
 System.out.println("[parentTail="+tailLength+"]");
 }
}

public class Mouse extends Rodent {
 protected int tailLength = 8;
 public void getMouseDetails() {
 System.out.println("[tail="+tailLength +",parentTail="+super.tailLength+"]");
 }
 public static void main(String[] args) {
 Mouse mouse = new Mouse();
 mouse.getRodentDetails();
 mouse.getMouseDetails();
 }
}
```

[parentTail=4]  
[tail=8,parentTail=4]

# • Chapter 5: Class Design

## Creating Abstract Classes

An *abstract class* is a class that is marked with the abstract keyword and cannot be instantiated.

An *abstract method* is a method marked with the abstract keyword defined in an abstract class, for which no implementation is provided in the class in which it is declared.

```
public abstract class Cow {
}

public class Chicken {
 public abstract void peck(); // DOES NOT COMPILE
}
```

```
public abstract class Turtle {
 public abstract void swim() {} // DOES NOT COMPILE
 public abstract int getAge() {} // DOES NOT COMPILE
 return 10;
}
```

- Chapter 5: Class Design

## Creating Abstract Classes

An *abstract class* is a class that is marked with the `abstract` keyword and cannot be instantiated.

An *abstract method* is a method marked with the `abstract` keyword defined in an abstract class, for which no implementation is provided in the class in which it is declared.

```
public final abstract class Tortoise { // DOES NOT COMPILE
}
```

```
public abstract class Goat {
 public abstract final void chew(); // DOES NOT COMPILE
}
```

```
public abstract class Whale {
 private abstract void sing(); // DOES NOT COMPILE
}
```

# • Chapter 5: Class Design

## Creating a Concrete Class

An abstract class becomes useful when it is extended by a concrete subclass. A *concrete class* is the first non abstract subclass that extends an abstract class and is required to implement all inherited abstract methods.

When you see a concrete class extending an abstract class on the exam, check that it implements all of the required abstract methods.

```
public abstract class Animal {
 public abstract String getName();
}

public class Walrus extends Animal { // DOES NOT COMPILE
}

public abstract class Eagle extends Animal {
}
```

- Chapter 5: Class Design

## Creating a Concrete Class

Abstract Class Definition Rules:

1. Abstract classes cannot be instantiated directly.
2. Abstract classes may be defined with any number, including zero, of abstract and non abstract methods.
3. Abstract classes may not be marked as private or final.
4. An abstract class that extends another abstract class inherits all of its abstract methods as its own abstract methods.
5. The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods.

# • Chapter 5: Class Design

## Creating a Concrete Class

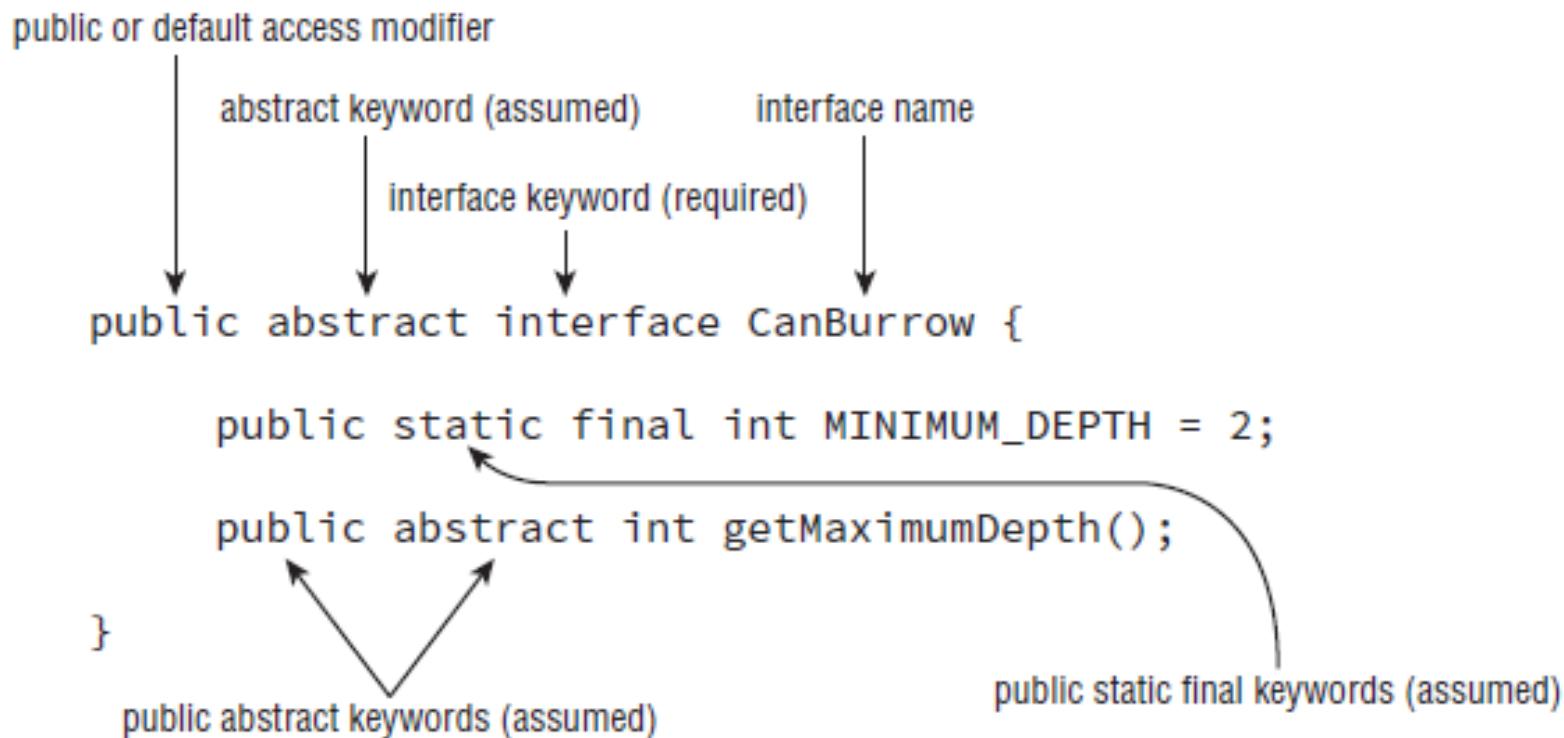
Abstract Method Definition Rules:

1. Abstract methods may only be defined in abstract classes.
2. Abstract methods may not be declared private or final.
3. Abstract methods must not provide a method body/implementation in the abstract class for which it is declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method. For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class.

# • Chapter 5: Class Design

## Implementing Interfaces

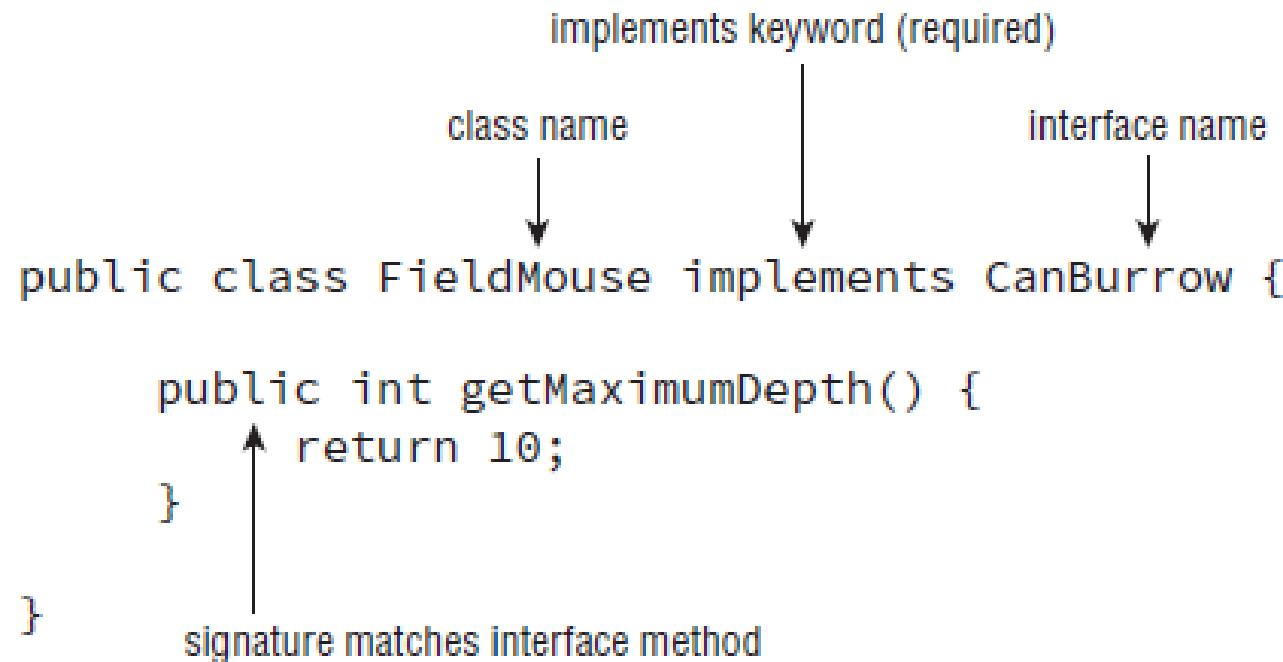
An *interface* is an abstract data type that defines a list of abstract public methods that any class implementing the interface must provide.



# • Chapter 5: Class Design

## Implementing Interfaces

A class invokes the interface by using the implements keyword in its class definition.



- Chapter 5: Class Design

## Implementing Interfaces

### 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.

- Chapter 5: Class Design

## Implementing Interfaces

### 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.

- Chapter 5: Class Design

## Implementing Interfaces

### Inheriting an Interface

There are two inheritance rules you should keep in mind when extending an interface:

1. An interface that extends another interface, as well as an abstract class that implements an interface, inherits all of the abstract methods as its own abstract methods.
2. The first concrete class that implements an interface, or extends an abstract class that implements an interface, must provide an implementation for all of the inherited abstract methods.

### Classes, Interfaces, and Keywords

Although a class can implement an interface, a class cannot extend an interface. Likewise, whereas an interface can extend another interface, an interface cannot implement another interface.

# • Chapter 5: Class Design

## Implementing Interfaces

### Abstract Methods and Multiple Inheritance

```
public interface Herbivore {
 public void eatPlants();
}
```

```
public interface Omnivore {
 public void eatPlants();
 public void eatMeat();
}
```

```
public interface Herbivore {
 public int eatPlants();
}
```

```
public interface Omnivore {
 public void eatPlants();
}
```

```
public class Bear implements Herbivore, Omnivore {
 public void eatMeat() {
 System.out.println("Eating meat");
 }
```

```
 public void eatPlants() {
 System.out.println("Eating plants");
 }
}
```

```
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");
 }
}
```

The code doesn't compile, as the class defines two methods with the same name and input parameters but different return types. If we were to remove either definition of eatPlants(), the compiler would stop because the definition of Bear would be missing one of the required methods.

# • Chapter 5: Class Design

## Implementing Interfaces

### Abstract Methods and Multiple Inheritance

```
public interface Herbivore {
 public void eatPlants();
}
```

```
public interface Omnivore {
 public void eatPlants();
 public void eatMeat();
}
```

```
public interface Herbivore {
 public int eatPlants();
}
```

```
public interface Omnivore {
 public void eatPlants();
}
```

```
public class Bear implements Herbivore, Omnivore {
 public void eatMeat() {
 System.out.println("Eating meat");
 }
```

```
 public void eatPlants() {
 System.out.println("Eating plants");
 }
}
```

```
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");
 }
}
```

The code doesn't compile, as the class defines two methods with the same name and input parameters but different return types. If we were to remove either definition of eatPlants(), the compiler would stop because the definition of Bear would be missing one of the required methods.

# • Chapter 5: Class Design

## Implementing Interfaces

### 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.

```
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
}
```

- Chapter 5: Class Design

## Implementing Interfaces

### Default Interface Methods

A ***default method*** is a method defined within an interface with the `default` keyword in which a method body is provided. Contrast default methods with “regular” methods in an interface, which are assumed to be abstract and may not have a method body.

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.

The following are the default interface method rules you need to be familiar with:

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 assumed to 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`.

# • Chapter 5: Class Design

## Implementing Interfaces

### 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());
 }
}
```

```
public class Cat implements Walk, Run {
 public int getSpeed() {
 return 1;
 }

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

- Chapter 5: Class Design

## Implementing Interfaces

### Static Interface Methods

Java 8 also now includes support for static methods within interfaces.

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.

- Chapter 5: Class Design

## Implementing Interfaces

### Static Interface Methods

```
public interface Hop {
 static int getJumpHeight() {
 return 8;
 }
}

public class Bunny implements Hop {
 public void printDetails() {
 System.out.println(getJumpHeight()); // DOES NOT COMPILE
 }
}
```

```
public class Bunny implements Hop {
 public void printDetails() {
 System.out.println(Hop.getJumpHeight());
 }
}
```

# • Chapter 5: Class Design

## Understanding Polymorphism

Java supports *polymorphism*, the property of an object to take on many different forms.

A Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements, either directly or through a superclass.

```
public class Primate {
 public boolean hasHair() {
 return true;
 }
}

public interface HasTail {
 public boolean isTailStriped();
}
```

10  
false  
true

```
public class Lemur extends Primate implements HasTail {
 public boolean isTailStriped() {
 return false;
 }
 public int age = 10;
 public static void main(String[] args) {
 Lemur lemur = new Lemur();
 System.out.println(lemur.age);

 HasTail hasTail = lemur;
 System.out.println(hasTail.isTailStriped());

 Primate primate = lemur;
 System.out.println(primate.hasHair());
 }
}
```

# • Chapter 5: Class Design

## Understanding Polymorphism

```
HasTail hasTail = lemur;
System.out.println(hasTail.age); // DOES NOT COMPILE
```

```
Primate primate = lemur;
System.out.println(primate.isTailStriped()); // DOES NOT COMPILE
```

### Object vs. Reference

In Java, all objects are accessed by reference, so as a developer you never have direct access to the object itself.

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

# • Chapter 5: Class Design

## Understanding Polymorphism

### Casting Objects

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
 }
}
```

```
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
 }
}
```

# • Chapter 5: Class Design

## Understanding Polymorphism

### Virtual Methods

A *virtual method* is a method in which the specific implementation is not determined until runtime.

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

What makes a virtual method special in Java is that if you call a method on an object that overrides a method, you get the overridden method, even if the call to the method is on a parent reference or within the parent class.

```
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();
 }
}
```

The bird name is: Peacock

# • Chapter 5: Class Design

## Understanding Polymorphism

### Polymorphic Parameters

One of the most useful applications of polymorphism is the ability to pass instances of a subclass or interface to a method. Since you're casting from a subtype to a supertype, an explicit cast is not required. This property is referred to as polymorphic parameters of a method.

```
public class Reptile {
 public String getName() {
 return "Reptile";
 }
}

public class Alligator extends Reptile {
 public String getName() {
 return "Alligator";
 }
}

public class Crocodile extends Reptile {
 public String getName() {
 return "Crocodile";
 }
}
```

```
public class ZooWorker {
 public static void feed(Reptile reptile) {
 System.out.println("Feeding reptile "+reptile.getName());
 }
 public static void main(String[] args) {
 feed(new Alligator());
 feed(new Crocodile());
 feed(new Reptile());
 }
}
```

Feeding: Alligator  
Feeding: Crocodile  
Feeding: Reptile

# QCM Chapitre 5

OCAJP-IZ0-808

Q1

B

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

# Q2

E

What is the output of the following code?

```
1: class Mammal {
2: public Mammal(int age) {
3: System.out.print("Mammal");
4: }
5: }
6: public class Platypus extends Mammal {
7: public Platypus() {
8: System.out.print("Platypus");
9: }
10: public static void main(String[] args) {
11: new Mammal(5);
```

- 12: }
- 13: }
- A. Platypus
- B. Mammal
- C. PlatypusMammal
- D. MammalPlatypus
- E. The code will not compile because of line 8.
- F. The code will not compile because of line 11

# Q3

A, B, D, E

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

```
public interface CanHop {}
```

```
public class Frog implements CanHop {
 public static void main(String[] args) {
 frog = new TurtleFrog();
 }
```

```
 public class BrazilianHornedFrog extends
 Frog {}
```

```
 public class TurtleFrog extends Frog {}
```

- A. Frog
- B. TurtleFrog
- C. BrazilianHornedFrog
- D. CanHop
- E. Object
- F. Long

# Q4

C, E

Which statement(s) are correct about the following code? (Choose all that apply)

```
public class Rodent {
 protected static Integer chew() throws Exception {
 System.out.println("Rodent is chewing");
 return 1;
 }
}

public class Beaver extends Rodent {
 public Number chew() throws RuntimeException {
 System.out.println("Beaver is chewing on wood");
 return 2;
 }
}
```

- A. It will compile without issue.
- B. It fails to compile because the type of the exception the method throws is a subclass of the type of exception the parent method throws.
- C. It fails to compile because the return types are not covariant.
- D. It fails to compile because the method is protected in the parent class and public in the subclass.
- E. It fails to compile because of a static modifier mismatch between the two methods.

Q5

A, D, E, F

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

# Q6

D

Choose the correct statement about the following code:

```
1: interface HasExoskeleton {
2: abstract int
getNumberOfSections();
3: }
4: abstract class Insect implements
HasExoskeleton {
5: abstract int getNumberOfLegs();
6: }
7: public class Beetle extends Insect {
```

- 8: int getNumberOfLegs() { return 6; }  
9: }
- A. It compiles and runs without issue.
- B. The code will not compile because of line 2.
- C. The code will not compile because of line 4.
- D. The code will not compile because of line 7.
- E. It compiles but throws an exception at runtime.

# Q7

B, C

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, you guarantee that the specific method will be called in the parent class at runtime.

# Q8

F

Choose the correct statement about the following code:

```
1: public interface Herbivore {
2: int amount = 10;
3: public static void eatGrass();
4: public int chew() {
5: return 13;
6: }
7: }
```

A. It compiles and runs without issue.

- B. The code will not compile because of line 2.
- C. The code will not compile because of line 3.
- D. The code will not compile because of line 4.
- E. The code will not compile because of lines 2 and 3.
- F. The code will not compile because of lines 3 and 4.

# Q9

A

Choose the correct statement about the following code:

```
1: public interface CanFly {
2: void fly();
3: }
4: interface HasWings {
5: public abstract Object getWindSpan();
6: }
7: abstract class Falcon implements
CanFly, HasWings {
8: }
```

A. It compiles without issue.

- B. The code will not compile because of line 2.
- 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 lines 2 and 5.
- F. The code will not compile because the class Falcon doesn't implement the methods.

# Q10

B, C, E, F

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.

# Q11

A, D, E

What modifiers are assumed for all interface variables? (Choose all that apply)

- A. public
- B. protected
- C. private
- D. static
- E. final
- F. abstract

# Q12

B

What is the output of the following code? 10: }

```
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: }
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.
```

# Q13

A

What is the output of the following code?

```
9: spider.printName(9.0);
1: class Arthropod
2: public void printName(double input) {
System.out
 .print("Arthropod");
}
3: }
4: public class Spider extends Arthropod {
5: public void printName(int input) {
System.out.print("Spider");
}
6: public static void main(String[] args) {
7: Spider spider = new Spider();
8: spider.printName(4);
```

10: }

11: }

- A. SpiderArthropod
- B. ArthropodSpider
- C. SpiderSpider
- D. ArthropodArthropod
- E. The code will not compile because of line 5.
- F. The code will not compile because of line 9.

# Q14

C

Which statements are true about the following code? (Choose all that apply)

```
1: interface HasVocalCords {
2: public abstract void makeSound();
3: }
4: public interface CanBark extends
HasVocalCords {
5: public void bark();
6: }
```

A. The CanBark interface doesn't compile.

- B. A class that implements HasVocalCords must override the makeSound() method.
- C. A class that implements CanBark inherits both the makeSound() and bark() methods.
- D. A class that implements CanBark only inherits the bark() method.
- E. An interface cannot extend another interface.

# Q15

B

- Which of the following is true about a concrete subclass? (Choose all that apply)
- A. A concrete subclass can be declared as abstract.
  - B. A concrete subclass must implement all inherited abstract methods.
  - C. A concrete subclass must implement all methods defined in an inherited interface.
  - D. A concrete subclass cannot be marked as final.
  - E. Abstract methods cannot be overridden by a concrete subclass.

# Q16

E

What is the output of the following code?

```
1: abstract class Reptile {
2: public final void layEggs() {
3: System.out.println("Reptile laying eggs");
4: }
5: public static void main(String[] args) {
6: Reptile reptile = new Lizard();
7: reptile.layEggs();
8: }
9: public class Lizard extends Reptile {
10: }
```

- 9: public void layEggs() {  
10: 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.

# Q17

B

What is the output of the following code?

```
1: public abstract class Whale {
2: public abstract void dive() {};
3: public static void main(String[] args) {
4: Whale whale = new Orca();
5: whale.dive();
6: }
7: }
8: class Orca extends Whale {
```

- ```
9:   public void dive(int depth) {  
10:     System.out.println("Orca diving"); }  
11: }
```
- A. Orca diving
 - B. The code will not compile because of line 2.
 - C. The code will not compile because of line 8.
 - D. The code will not compile because of line 9.
 - E. The output cannot be determined from the code provided.

Q18

E

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.

# Q19

A,C,F

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

# Q20

A

What is the result of the following code?

```
1: public abstract class Bird {
2: private void fly() {
3: System.out.println("Bird is flying"); }
4: public static void main(String[] args) {
5: Bird bird = new Pelican();
6: bird.fly();
7: }
8: class Pelican extends Bird {
9: protected void fly() {
10: System.out.println("Pelican is flying"); }
11: }
12: }
```

- A. Bird is flying
- B. Pelican is flying
- 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.

# Tests

1 minute and 30 seconds

# Q1

How many compiler errors does the following code contain?

```
package animal;
interface CanFly {
 public void fly() {}
}

final class Bird {
 public int fly(int speed) {}
}
```

```
public class Eagle extends Bird
implements CanFly {
```

```
 public void fly() {}
}
```

- A. None
- B. One
- C. Two
- D. Three

## Q2

What is the output of the following application?

```
package musical;
interface SpeakDialogue { default int
talk() { return 7; } }

interface SingMonologue { default int
talk() { return 5; } }

public class Performance implements
SpeakDialogue, SingMonologue {
 public int talk(String... x) {
 return x.length;
 }
}
```

```
 public static void main(String[]
notes) {
 System.out.print(new
Performance().talk(notes));
}
}

A. 7
B. 5
C. The code does not compile.
D. The code compiles without issue,
but the output cannot be determined
until runtime.
```

# Q3

What is the output of the following application?

```
class Math {
 public final double secret = 2;
}
```

```
class ComplexMath extends Math {
 public final double secret = 4;
}
```

```
public class InfiniteMath extends
ComplexMath {
 public final double secret = 8;
```

```
 public static void main(String[]
numbers) {
 Math math = new InfiniteMath();
 System.out.print(math.secret);
 }
 }
```

- A. 2
- B. 4
- C. 8
- D. The code does not compile.

## Q4

Given the following method and the fact that `FileNotFoundException` is a subclass of `IOException`, which of the following method signatures is a valid override by a subclass?

`protected void dance() throws FileNotFoundException {}`

- A. `void dance() throws IOException`
- B. `public void dance() throws IOException`
- C. `private void dance() throws FileNotFoundException`
- D. `public final void dance()`

# Q5

Given the class definitions below, which value, when inserted into the blank line, does not allow the class to compile?

```
public class Canine {}
public class Dog extends Canine {}
public class Wolf extends Canine {}
public final class Husky extends Dog {}
public class Zoologist {
 Canine animal;
 public final void setAnimal(Dog
 animal) { this.animal = animal; }
```

```
 public static void main(String[]
 furryFriends) {
 new Zoologist().setAnimal(____);
 }
}
A. new Husky()
B. new Dog()
C. new Wolf()
D. null
```

# Q6

Which statement about the following application is true?

```
package party;
abstract class House {
 protected abstract Object getSpace();
}
abstract class Room extends House {
 abstract Object getSpace(Object list);
}
abstract public class Ballroom extends
House {
 protected abstract Object getSpace();
```

```
public static void main(String[]
squareFootage) {
 System.out.print("Let's start the
party!");
}
```

- A. It compiles and at runtime prints Let's start the party!
- B. It does not compile for one reason.
- C. It does not compile for two reasons.
- D. It does not compile for three reasons.

# Q7

Given that Integer and Long are subclasses of Number, what type can be used to fill in the blank in the class below to allow it to compile?

```
package orchestra;
interface MusicCreator { public
Number play(); }
abstract class StringInstrument {
public Long play() {return 3L;} }
public class Violin extends
```

StringInstrument implements  
MusicCreator {

```
public _____ play() {
 return 12;
}
```

}

- A. Long
- B. Integer
- C. Long or Integer
- D. Long or Number

## Q8

Which of the following is the best reason for creating a default interface method?

- A. Allow interface methods to be inherited.
- B. Add backward compatibility to existing interfaces.
- C. Give an interface the ability to create concrete methods.
- D. Allow an interface to define a method at the class level.

# Q9

Given that EOFException is a subclass of IOException, what is the output of the following application?

```
package ai;
import java.io.*;
class Machine {
 public boolean turnOn() throws
EOFException {return true;}
}
public class Robot extends Machine {
 public boolean turnOn() throws
IOException {return false;}
```

```
public static void main(String[]
doesNotCompute) throws Exception {
 Machine m = new Robot();
 System.out.print(m.turnOn());
}
```

- A. true
- B. false
- C. The code does not compile.
- D. The code compiles but produces an exception at runtime.

# Q10

What is the output of the following application?

```
package sports;
abstract class Ball {
 protected final int size;
 public Ball(int size) {
 this.size = size;
 }
}
interface Equipment {}
public class SoccerBall extends Ball implements Equipment {
 public SoccerBall() {
 super(5);
 }
}
```

```
public Ball get() { return this; }
public static void main(String[] passes) {
 Equipment equipment = (Equipment)(Ball)new
 SoccerBall().get();
 System.out.print(((SoccerBall)equipment).size);
}
A. 5
B. The code does not compile due an invalid
cast.
C. The code does not compile for a different
reason.
D. The code compiles but throws a
ClassCastException at runtime.
```

# Corrections

# Q1

D

How many compiler errors does the following code contain?

```
package animal;
interface CanFly {
 public void fly() {}
}

final class Bird {
 public int fly(int speed) {}
}
```

```
public class Eagle extends Bird
implements CanFly {
```

```
 public void fly() {}
}
```

- A. None
- B. One
- C. Two
- D. Three

## Q2

C

What is the output of the following application?

```
package musical;
interface SpeakDialogue { default int
talk() { return 7; } }
interface SingMonologue { default int
talk() { return 5; } }
public class Performance implements
SpeakDialogue, SingMonologue {
public int talk(String... x) {
 return x.length;
}
```

```
public static void main(String[]
notes) {
 System.out.print(new
Performance().talk(notes));
}
```

A. 7  
B. 5  
C. The code does not compile.  
D. The code compiles without issue,  
but the output cannot be determined  
until runtime.

# Q3

A

What is the output of the following application?

```
class Math {
 public final double secret = 2;
}

class ComplexMath extends Math {
 public final double secret = 4;
}

public class InfiniteMath extends
ComplexMath {
 public final double secret = 8;
```

```
 public static void main(String[]
numbers) {
 Math math = new InfiniteMath();
 System.out.print(math.secret);
}
}

A. 2
B. 4
C. 8
D. The code does not compile.
```

# Q4

D

Given the following method and the fact that  
FileNotFoundException is a subclass of IOException, which of the  
following method signatures is a valid override by a subclass?

protected void dance() throws FileNotFoundException {}

- A. void dance() throws IOException
- B. public void dance() throws IOException
- C. private void dance() throws FileNotFoundException
- D. public final void dance()

# Q5

C

Given the class definitions below, which value, when inserted into the blank line, does not allow the class to compile?

```
public class Canine {}
public class Dog extends Canine {}
public class Wolf extends Canine {}
public final class Husky extends Dog {}
public class Zoologist {
 Canine animal;
 public final void setAnimal(Dog
animal) { this.animal = animal; }
```

```
 public static void main(String[]
furryFriends) {
 new Zoologist().setAnimal(____);
 }
}
A. new Husky()
B. new Dog()
C. new Wolf()
D. null
```

# Q6

Which statement about the following application is true?

```
package party;
abstract class House {
 protected abstract Object getSpace();
}
abstract class Room extends House {
 abstract Object getSpace(Object list);
}
abstract public class Ballroom extends
House {
 protected abstract Object getSpace();
```

```
public static void main(String[]
squareFootage) {
 System.out.print("Let's start the
party!");
}
```

- A. It compiles and at runtime prints Let's start the party!
- B. It does not compile for one reason.
- C. It does not compile for two reasons.
- D. It does not compile for three reasons.

A

# Q7

A

Given that Integer and Long are subclasses of Number, what type can be used to fill in the blank in the class below to allow it to compile?

```
package orchestra;
interface MusicCreator { public
Number play(); }
abstract class StringInstrument {
public Long play() {return 3L;} }
public class Violin extends
```

StringInstrument implements  
MusicCreator {

```
public _____ play() {
 return 12;
}
```

}

- A. Long
- B. Integer
- C. Long or Integer
- D. Long or Number

## Q8

B

Which of the following is the best reason for creating a default interface method?

- A. Allow interface methods to be inherited.
- B. Add backward compatibility to existing interfaces.
- C. Give an interface the ability to create concrete methods.

D. Allow an interface to define a method at the class level.

# Q9

C

Given that EOFException is a subclass of IOException, what is the output of the following application?

```
package ai;
import java.io.*;
class Machine {
 public boolean turnOn() throws
 EOFException {return true;}
}
public class Robot extends Machine {
 public boolean turnOn() throws
 IOException {return false;}}
```

```
public static void main(String[]
 doesNotCompute) throws Exception {
 Machine m = new Robot();
 System.out.print(m.turnOn());
}
```

- A. true
- B. false
- C. The code does not compile.
- D. The code compiles but produces an exception at runtime.

# Q10

A

What is the output of the following application?

```
package sports;
abstract class Ball {
 protected final int size;
 public Ball(int size) {
 this.size = size;
 }
}
interface Equipment {}
public class SoccerBall extends Ball implements Equipment {
 public SoccerBall() {
 super(5);
 }
}
```

```
public Ball get() { return this; }
public static void main(String[] passes) {
 Equipment equipment = (Equipment)(Ball)new
 SoccerBall().get();
```

```
System.out.print(((SoccerBall)equipment).size);
 }
```

```
}
```

A. 5

B. The code does not compile due an invalid cast.

C. The code does not compile for a different reason.

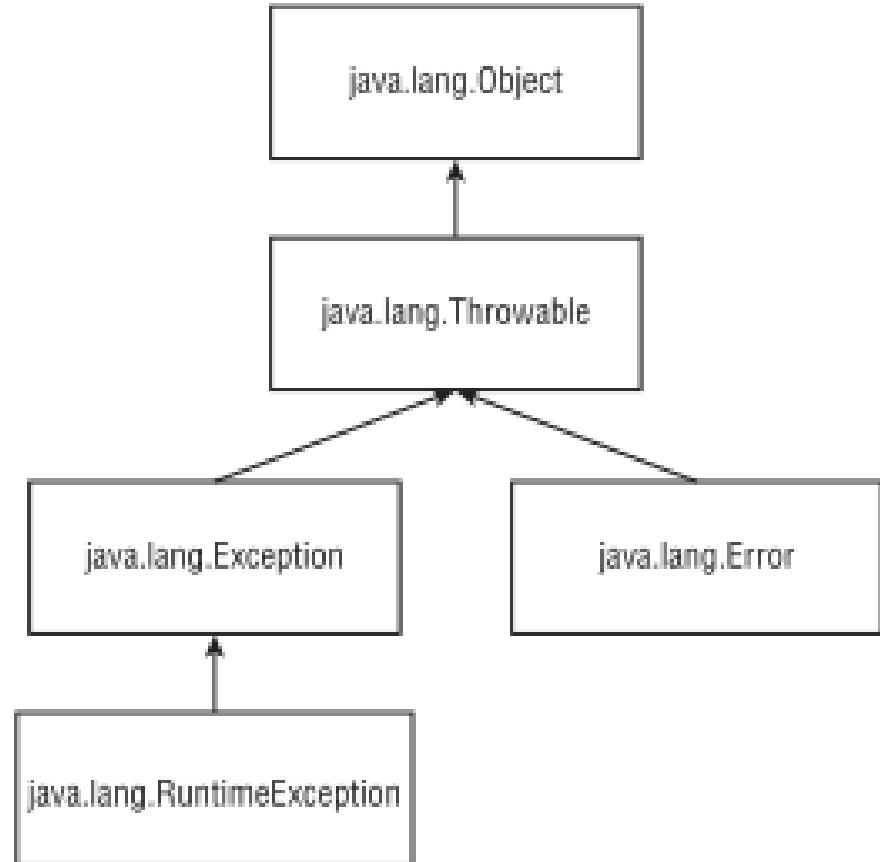
D. The code compiles but throws a ClassCastException at runtime.

# **Chapter 6: Exceptions**

- Chapter 6: Exceptions

## Understanding Exception Types

An *exception* is Java's way of saying, "I give up. I don't know what to do right now. You deal with it." When you write a method, you can either deal with the exception or make it the calling code's problem.



# • Chapter 6: Exceptions

## Understanding Exception Types

An exception indicates something unexpected happened. A method can handle an exception by catching it or declaring it for the caller to deal with. Many exceptions are thrown by Java libraries. You can throw your own exception with code such as `throw new Exception()`.

Subclasses of `java.lang.Error` are exceptions that a programmer should not attempt to handle. Subclasses of `java.lang.RuntimeException` are runtime (unchecked) exceptions.

Subclasses of `java.lang.Exception`, but not `java.lang.RuntimeException` are checked exceptions. Java requires checked exceptions to be handled or declared.

Common runtime exceptions include:

- `ArithmaticException`
- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `IllegalArgumentException`
- `NullPointerException`
- `NumberFormatException`

# • Chapter 6: Exceptions

## Understanding Exception Types

IllegalArgumentException and NumberFormatException are typically thrown by the programmer, whereas the others are typically thrown by the JVM.

Common checked exceptions include:

- IOException
- FileNotFoundException

Common errors include:

- ExceptionInInitializerError
- StackOverflowError
- NoClassDefFoundError

When a method overrides a method in a superclass or interface, it is not allowed to add checked exceptions. It is allowed to declare fewer exceptions or declare a subclass of a declared exception. Methods declare exceptions with the keyword throws.

# • Chapter 6: Exceptions

## Understanding Exception Types

### Throwing an Exception

On the exam, you will see two types of code that result in an exception. The first is code that's wrong. For example:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

This code throws an `ArrayIndexOutOfBoundsException`. That means questions about exceptions can be hidden in questions that appear to be about something else.

The second way for code to result in an exception is to explicitly request Java to throw one. Java lets you write statements like these:

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");
```

The `throw` keyword tells Java you want some other part of the code to deal with the exception.

# • Chapter 6: Exceptions

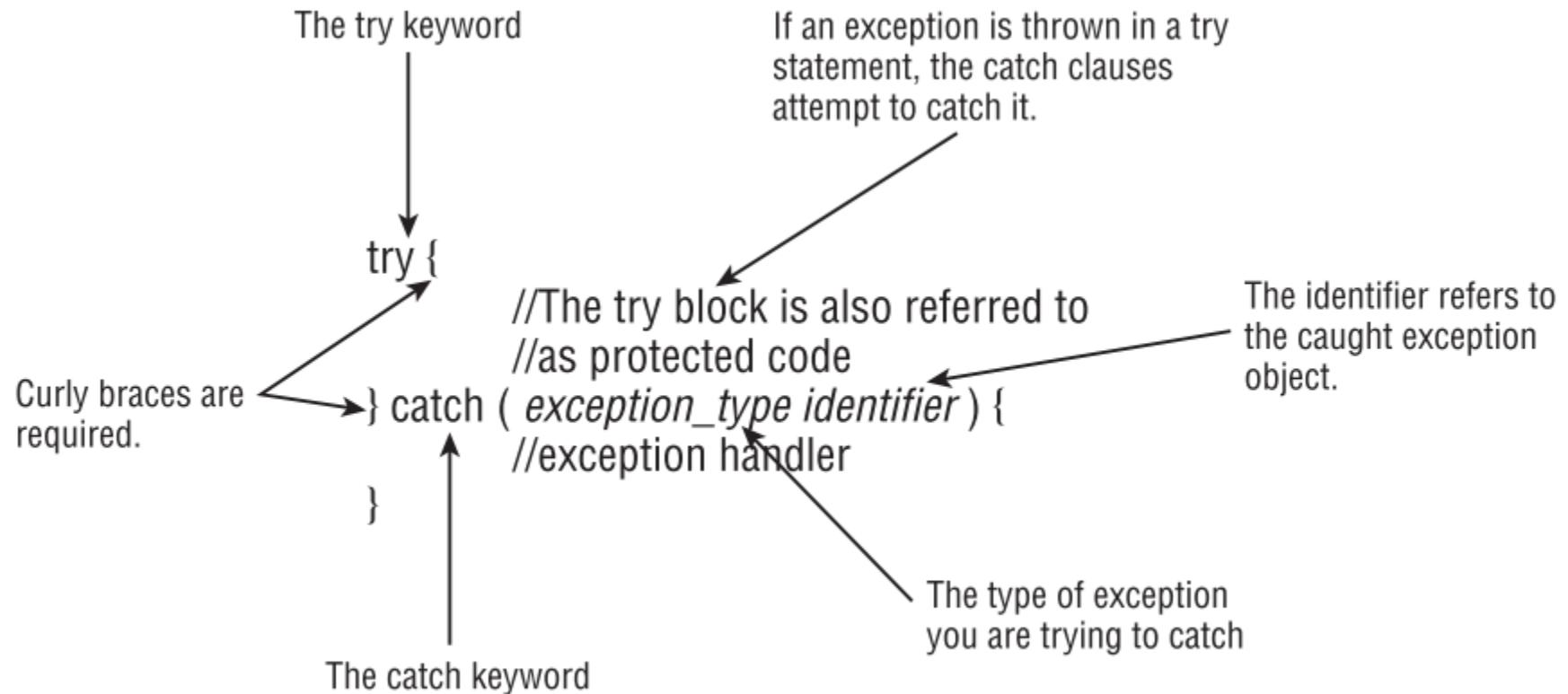
## Understanding Exception Types

### Throwing an Exception

| Type              | How to recognize                                           | Okay for program to catch? | Is program required to handle or declare? |
|-------------------|------------------------------------------------------------|----------------------------|-------------------------------------------|
| Runtime exception | Subclass of RuntimeException                               | Yes                        | No                                        |
| Checked exception | Subclass of Exception but not subclass of RuntimeException | Yes                        | Yes                                       |
| Error             | Subclass of Error                                          | No                         | No                                        |

# • Chapter 6: Exceptions

## Using a try Statement



# • Chapter 6: Exceptions

## Using a try Statement

The code in the try block is run normally. If any of the statements throw an exception that can be caught by the exception type listed in the catch block, the try block stops running and execution goes to the catch statement. If none of the statements in the try block throw an exception that can be caught, the catch clause is not run.

```
3: void explore() {
4: try {
5: fall();
6: System.out.println("never get here");
7: } catch (RuntimeException e) {
8: getUp();
9: }
10: seeAnimals();
11: }
12: void fall() { throw new RuntimeException(); }
```

First, line 5 calls the fall() method. Line 12 throws an exception. This means Java jumps straight to the catch block, skipping line 6. The girl gets up on line 8. Now the try statement is over and execution proceeds normally with line 10.

# • Chapter 6: Exceptions

## Using a try Statement

### Using a try Statement

```
try // DOES NOT COMPILE
 fall();
catch (Exception e)
 System.out.println("get up");
```

The problem is that the braces are missing. It needs to look like this:

```
try {
 fall();
} catch (Exception e) {
 System.out.println("get up");
}
```

```
try { // DOES NOT COMPILE
 fall();
}
```

This code doesn't compile because the try block doesn't have anything after it.

Remember, the point of a try statement is for something to happen if an exception is thrown. Without another clause, the try statement is lonely.

# • Chapter 6: Exceptions

## Using a try Statement

### Adding a finally Block

The try statement also lets you run code at the end with a finally clause regardless of whether an exception is thrown.

A finally block can  
only appear as part  
of a try statement.

```
try {
 //protected code
} catch (exceptiontype identifier) {
 //exception handler
} finally {
 //finally block
}
```

The finally keyword

The finally block  
always executes,  
whether or not an  
exception occurs  
in the try block.

- Chapter 6: Exceptions

## Using a try Statement

### Adding a finally Block

```
12: void explore() {
13: try {
14: seeAnimals();
15: fall();
16: } catch (Exception e) {
17: getHugFromDaddy();
18: } finally {
19: seeMoreAnimals();
20: }
21: goHome();
22: }
```

# • Chapter 6: Exceptions

## Using a try Statement

### Adding a finally Block

```
25: try { // DOES NOT COMPILE
26: fall();
27: } finally {
28: System.out.println("all better");
29: } catch (Exception e) {
30: System.out.println("get up");
31: }

32:
33: try { // DOES NOT COMPILE
34: fall();
35: }
36:
37: try {
```

```
38: fall();
39: } finally {
40: System.out.println("all better");
41: }
```

The first example (lines 25–31) does not compile because the catch and finally blocks are in the wrong order. The second example (lines 33–35) does not compile because there must be a catch or finally block. The third example (lines 37–41) is just fine. catch is not required if finally is present.

```
String s = "";
try {
 s += "t";
} catch(Exception e) {
 s += "c";
} finally {
 s += "f";
}
s += "a";
System.out.print(s);
```

# • Chapter 6: Exceptions

## Catching Various Types of Exceptions

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }

public void visitPorcupine() {
 try {
 seeAnimal();
 } catch (AnimalsOutForAWalk e) { // first catch block
 System.out.print("try back later");
 } catch (ExhibitClosed e) { // second catch block
 System.out.print("not today");
 }
}
```

```
public void visitMonkeys() {
 try {
 seeAnimal();
 } catch (ExhibitClosedForLunch e) { // subclass exception
 System.out.print("try back later");
 } catch (ExhibitClosed e) { // superclass exception
 System.out.print("not today");
 }
}
```

```
public void visitMonkeys() {
 try {
 seeAnimal();
 } catch (ExhibitClosed e) {
 System.out.print("not today");
 } catch (ExhibitClosedForLunch e) { // DOES NOT COMPILE
 System.out.print("try back later");
 }
}
```

# • Chapitre 6: Exceptions

## Catching Various Types of Exceptions

Creating your own exceptions is not on the OCA exam, but it is on the OCP exam. However, the OCA exam can define basic exceptions to show you the hierarchy. You only need to do two things with this information.

First, you must be able to recognize if the exception is a checked or an unchecked exception.

Second, you need to determine if any of the exceptions are subclasses of the others.

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }
```

In this example, there are three custom exceptions. All are unchecked exceptions because they directly or indirectly extend RuntimeException.

# • Chapitre 6: Exceptions

## Catching Various Types of Exceptions

Now we catch both types of exceptions and handle them by printing out the appropriate message:

```
public void visitPorcupine() {
 try {
 seeAnimal();
 } catch (AnimalsOutForAWalk e) { // first catch block
 System.out.print("try back later");
 } catch (ExhibitClosed e) { // second catch block
 System.out.print("not today");
 }
}
```

A rule exists for the order of the catch blocks. Java looks at them in the order they appear. If it is impossible for one of the catch blocks to be executed, a compiler error about unreachable code occurs. This happens when a superclass is caught before a subclass. Remember, we warned you to pay attention to any subclass exceptions.

In the porcupine example, the order of the catch blocks could be reversed because the exceptions don't inherit from each other. And yes, we have seen a porcupine be taken for a walk on a leash.

# • Chapitre 6: Exceptions

## Catching Various Types of Exceptions

The following example shows exception types that do inherit from each other:

```
public void visitMonkeys() {
 try {
 seeAnimal();
 } catch (ExhibitClosedForLunch e) { // subclass exception
 System.out.print("try back later");
 } catch (ExhibitClosed e) { // superclass exception
 System.out.print("not today");
 }
}
```

```
public void visitMonkeys() {
 try {
 seeAnimal();
 } catch (ExhibitClosed e) {
 System.out.print("not today");
 } catch (ExhibitClosedForLunch e) { // DOES NOT COMPILE
 System.out.print("try back later");
 }
}
```

If the more specific ExhibitClosedForLunch exception is thrown, the first catchblock runs. If not, Java checks if the superclass ExhibitClosed exception is thrown and catches it. This time, the order of the catch blocks does matter. The reverse does not work.

This time, if the more specific ExhibitClosedForLunch exception is thrown, the catch block for ExhibitClosed runs—which means there is no way for the second catch block to ever run. Java correctly tells us there is an unreachable catch block.

# • Chapitre 6: Exceptions

## Catching Various Types of Exceptions

Let's try this one more time. Do you see why this code doesn't compile?

```
public void visitSnakes() {
 try {
 seeAnimal();
 } catch (RuntimeException e) {
 System.out.print("runtime exception");
 } catch (ExhibitClosed e) {// DOES NOT COMPILE
 System.out.print("not today");
 } catch (Exception e) {
 System.out.print("exception");
 }
}
```

It's the same problem. **ExhibitClosed** is a **RuntimeException**. If it is thrown, the first catch block takes care of it, making sure there no way to get to the second catch block.

To review catching multiple exceptions, remember that at most one catch block will run and it will be the first catch block that can handle it.

# • Chapter 6: Exceptions

## Throwing a Second Exception

```
26: try {
27: throw new RuntimeException();
28: } catch (RuntimeException e) {
29: throw new RuntimeException();
30: } finally {
31: throw new Exception();
32: }
```

```
30: public String exceptions() {
31: String result = "";
32: String v = null;
33: try {
34: try {
35: result += "before";
36: v.length();
37: result += "after";
38: } catch (NullPointerException e) {
39: result += "catch";
40: throw new RuntimeException();
41: } finally {
42: result += "finally";
43: throw new Exception();
44: }
45: } catch (Exception e) {
46: result += "done";
47: }
48: return result;
49: }
```

before catch finally done.

- Chapter 6: Exceptions

## Recognizing Common Exception Types

# Runtime Exceptions

Runtime exceptions extend RuntimeException. They don't have to be handled or declared.

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

**ArithmaticException** 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** Thrown by the JVM when an attempt is made to cast an exception to a subclass of which it is not an instance

**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 when an attempt is made

- Chapter 6: Exceptions

## Recognizing Common Exception Types

# Checked Exceptions

Checked exceptions have `Exception` in their hierarchy but not `RuntimeException`. They must be handled or declared. 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

- Chapter 6: Exceptions

## Recognizing Common Exception Types

# Errors

Errors extend the Error class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see these:

**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 (this is called *infinite recursion* because the method typically calls itself without end)

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

# • Chapter 6: Exceptions

## Calling Methods That Throw Exceptions

### Subclasses

```
class CanNotHopException extends Exception { }
class Hopper {
 public void hop() { }
}
class Bunny extends Hopper {
 public void hop() throws CanNotHopException { } // DOES NOT COMPILE
}
```

Java knows hop() isn't allowed to throw any checked exceptions because the superclass Hopper doesn't declare any. Imagine what would happen if subclasses could add checked exceptions

```
class Hopper {
 public void hop() throws CanNotHopException { }
}
class Bunny extends Hopper {
 public void hop() { }
}
```

A subclass not declaring an exception is similar to a method declaring it throws an exception that it never actually throws. This is perfectly legal.

- Chapter 6: Exceptions

## Calling Methods That Throw Exceptions

### Subclasses

```
class Hopper {
 public void hop() throws Exception { }
}

class Bunny extends Hopper {
 public void hop() throws CanNotHopException { }
}
```

Bunny could declare that it throws Exception directly, or it could declare that it throws a more specific type of Exception. It could even declare that it throws nothing at all. This rule applies only to checked exceptions.

```
class Hopper {
 public void hop() { }
}

class Bunny extends Hopper {
 public void hop() throws IllegalStateException { }
}
```

The reason that it's okay to declare new runtime exceptions in a subclass method is that the declaration is redundant. Methods are free to throw any runtime exceptions they want without mentioning them in the method declaration.

- Chapter 6: Exceptions

## Calling Methods That Throw Exceptions

### Printing an Exception

```
5: public static void main(String[] args) {
6: try {
7: hop();
8: } catch (Exception e) {
9: System.out.println(e);
10: System.out.println(e.getMessage());
11: e.printStackTrace();
12: }
13: }
14: private static void hop() {
15: throw new RuntimeException("cannot hop");
16: }
```

This code results in the following output:

```
java.lang.RuntimeException: cannot hop
cannot hop
java.lang.RuntimeException: cannot hop
at trycatch.Handling.hop(Handling.java:15)
at trycatch.Handling.main(Handling.java:7)
```

The first line shows what Java prints out by default: the exception type and message. The second line shows just the message. The rest shows a stack trace. The stack trace is usually the most helpful one because it shows where the exception occurred in each method that it passed through. On the OCA exam, you will mostly see the first approach. This is because the exam often shows code snippets.

# QCM Chapitre 6

OCAJP-IZ0-808

Q1

B

- Which of the following statements are true? (Choose all that apply)
- D. You can declare only unchecked exceptions.
  - A. Runtime exceptions are the same thing as checked exceptions.
  - E. You can handle only Exception subclasses.
  - B. Runtime exceptions are the same thing as unchecked exceptions.
  - C. You can declare only checked exceptions.

# Q2

B, D

Which of the following pairs fill in the blanks to make this code compile? (Choose all that apply)

```
7: public void ohNo() _____
Exception {
8: _____ Exception();
9: }
```

A. On line 7, fill in throw

- B. On line 7, fill in throws
- C. On line 8, fill in throw
- D. On line 8, fill in throw new
- E. On line 8, fill in throws
- F. On line 8, fill in throws new

# Q3

C

When are you required to use a finally block in a regular try statement (not a try-with-resources)?

- A. Never.
- B. When the program code doesn't terminate on its own.
- C. When there are no catch blocks in a try statement.

- D. When there is exactly one catch block in a try statement.
- E. When there are two or more catch blocks in a try statement.

## Q4

B

Which exception will the following throw?

```
Object obj = new Integer(3);
```

```
String str = (String) obj;
```

```
System.out.println(str);
```

- A. ArrayIndexOutOfBoundsException
- B. ClassCastException
- C. IllegalArgumentException

- D. NumberFormatException
- E. None of the above.

Q5

A, B, D

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

# Q6

C

- What will happen if you add the statement `System.out.println(5 / 0);` to a working `main()` method?
- A. It will not compile.
  - B. It will not run.
  - C. It will run and throw an `ArithmaticException`.
  - D. It will run and throw an `IllegalArgumentException`.
  - E. None of the above.

# Q7

C

What is printed besides the stack trace caused by the NullPointerException from line 16?

```
1: public class DoSomething {
2: public void go() {
3: System.out.print("A");
4: try {
5: stop();
6: } catch (ArithmetricException e) {
7: System.out.print("B");
8: } finally {
9: System.out.print("C");
10: }
11: System.out.print("D");
12: }
13: public void stop() {
```

```
14: System.out.print("E");
15: Object x = null;
16: x.toString();
17: System.out.print("F");
18: }
19: public static void main(String[] args) {
20: new DoSomething().go();
21: }
22: }
A. AE
B. AEBCD
C. AEC
D. AECD
E. No output appears other than the stack trace.
```

# Q8

E

What is the output of the following snippet, assuming a and b are both 0?

```
3: try {
4: return a / b;
5: } catch (RuntimeException e) {
6: return -1;
7: } catch (ArithmetricException e) {
8: return 0;
9: } finally {
10: System.out.print("done");
11: }
```

- A. -1
- B. 0
- C. done-1
- D. done0
- E. The code does not compile.
- F. An uncaught exception is thrown.

# Q9

B

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.

Q10

E

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.
  - G. An uncaught exception is thrown.

# Q11

A

What is the output of the following program?

```
1: public class Cat {
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 (NullPointerException e) {
10: System.out.print("4");
11: }
12: System.out.print("5");
13: }
14: public static void main(String[] args) {
```

- ```
15:    Cat leo = new Cat();  
16:    leo.name = "Leo";  
17:    leo.parseName();  
18:    System.out.print("6");  
19:  }  
20: }
```
- A. 12, followed by a stack trace for a NumberFormatException
 - B. 124, followed by a stack trace for a NumberFormatException
 - C. 12456
 - D. 12456
 - E. 1256, followed by a stack trace for a NumberFormatException
 - F. The code does not compile.
 - G. An uncaught exception is thrown.

Q12

A, B, D, G

What is printed by the following? (Choose all that apply) 14: }

```
1: public class Mouse {  
2:     public String name;  
3:     public void run() {  
4:         System.out.print("1");  
5:         try {  
6:             System.out.print("2");  
7:             name.toString();  
8:             System.out.print("3");  
9:         } catch (NullPointerException e) {  
10:             System.out.print("4");  
11:             throw e;  
12:         }  
13:         System.out.print("5");  
14:     }  
15:     public static void main(String[] args) {  
16:         Mouse jerry = new Mouse();  
17:         jerry.run();  
18:         System.out.print("6");  
19:     } }  
A. 1  
B. 2  
C. 3  
D. 4  
E. 5  
F. 6  
G. The stack trace for a NullPointerException
```

Q13

A, B, C, E

- Which of the following statements declaration.
are true? (Choose all that apply)
- A. You can declare a method with Exception as the return type.
 - B. You can declare any subclass of Error in the throws part of a method declaration.
 - C. You can declare any subclass of Exception in the throws part of a method
 - D. You can declare any subclass of Object in the throws part of a method declaration.
 - E. You can declare any subclass of RuntimeException in the throws part of a method declaration.

Q14

A, C, D, E

- Which of the following can be inserted on line 8 to make this code compile? (Choose all that 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();
```

# Q15

A, B, D, E

Which of the following are unchecked exceptions? (Choose all that apply)

- A. `ArrayIndexOutOfBoundsException`
- B. `IllegalArgumentException`
- C. `IOException`
- D. `NumberFormatException`
- E. Any exception that extends

`RuntimeException`

F. Any exception that extends `Exception`

# Q16

B

Which scenario is the best use of an exception?

- A. An element is not found when searching a list.
- B. An unexpected parameter is passed into a method.
- C. The computer caught fire.
- D. You want to loop through a list.
- E. You don't know how to code a method.

# Q17

A, C, D, E

Which of the following can be inserted into Lion to make this code compile?  
(Choose all that 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{}

# Q18

B,C, E

- Which of the following are true?      handled or declared.  
(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.

# Q19

C,E

Which of the following can be inserted }  
in the blank to make the code compile? }  
(Choose all

that apply)

```
public static void main(String[] args) {
 try {
 System.out.println("work real hard");
 } catch (e) {
 } catch (RuntimeException e) {
```

- A. Exception
- B. IOException
- C. IllegalArgumentException
- D. RuntimeException
- E. StackOverflowError
- F. None of the above.

# Q20

A, E

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: } finally {
23: System.out.print("e");
24: throw new RuntimeException("3");
25: }
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.

# Tests

1 minute and 30 seconds

# Q1

What is the output of the following application?

```
package zoo;
class BigCat {
 void roar(int level) throw RuntimeException { // m1
 if(level<3) throw new
IllegalArgumentException("Incomplete");
 System.out.print("Roar!");
 }
}
public class Lion extends BigCat {
 public void roar() { // m2
 System.out.print("Roar!!!");
 }
}
public static void main(String[] cubs) {
 final BigCat kitty = new Lion(); // m3
 kitty.roar(2);
}
```

- A. The code does not compile because of line m1.
- B. The code does not compile because of line m2.
- C. The code does not compile because of line m3.
- D. The code compiles but a stack trace is printed at runtime.

## Q2

Given the following code snippet, which specific exception will be thrown?

```
final Object exception = new Exception();
final Exception data =
(RuntimeException)exception;
System.out.print(data);
```

- A. ClassCastException
- B. RuntimeException
- C. NullPointerException
- D. None of the above

# Q3

In the following application, the values of street and city have been omitted. Which one of the following is a possible output of executing this class?

- I. 350 5th Ave - New York
- II. Posted:350 5th Ave - New York

```
package registration;
public class Address {
```

```
 public String getAddress(String street, String city) {
 try {
 return street.toString() + " : " + city.toString();
 } finally {
 System.out.print("Posted:");
 }
 }
```

```
 public static void main(String[] form) {
 String street = // value omitted
 String city = // value omitted
 System.out.print(new
Address().getAddress(street,city));
 }
}
```

- A. I only
- B. II only
- C. I and II
- D. None of the above

# Q4

What is the output of the following application?

```
package body;
class Organ {
 public void operate() throws
RuntimeException {
 throw new RuntimeException("Not
supported");
 }
}
public class Heart extends Organ {
 public void operate() throws Exception {
 System.out.print("beat");
 }
}
```

```
public static void main(String... cholesterol)
throws Exception {
```

```
 try {
 new Heart().operate();
 } finally {
 }
}
```

- A. beat
- B. Not supported
- C. The code does not compile.
- D. The code compiles but a stack trace is
printed at runtime.

# Q5

What is the output of the following application?

```
package clothing;
public class Coat {
 public Long zipper() throws Exception {
 try {
 String checkZipper = (String) new Object();
 } catch (Exception e) {
 throw RuntimeException("Broken!");
 }
 return null;
 }
}
```

```
public static void main(String... warmth) {
 try {
 new Coat().zipper();
 System.out.print("Finished!");
 } catch (Throwable t) {}
}
```

- A. Finished!
- B. Finished!, followed by a stack trace
- C. The application does not produce any output at runtime.
- D. The code does not compile.

# Q6

Given the following application, which type of exception will be printed in the stack trace at runtime?

```
package carnival;
public class WhackAnException {
 public static void main(String... hammer) {
 try {
 throw new ClassCastException();
 } catch (IllegalArgumentException e) {
 throw new IllegalArgumentException();
 } catch (RuntimeException e) {
 throw new NullPointerException();
 } finally {
 throw new RuntimeException();
 }
 }
}
```

- A. `IllegalArgumentException`
- B. `NullPointerException`
- C. `RuntimeException`
- D. The code does not compile.

# Q7

Which of these method signatures is allowed in a class implementing the Outfielder interface?

```
class OutOfBoundsException
extends BadCatchException {}

class BadCatchException extends
Exception {}

public interface Outfielder {
 public void catchBall() throws
```

```
 OutOfBoundsException;
}
```

- A. public int catchBall() throws OutOfBoundsException
- B. public int catchBall() throws BadCatchException
- C. public int catchBall() throws Exception
- D. None of the above

# Q8

What is the result of compiling and running the following application?

```
package castles;
class DragonException extends Exception {}
public class Lair {
 public void openDrawbridge() throws Exception // r1
 try {
 throw new Exception("This Exception");
 } catch (RuntimeException e) {
 throw new DragonException(); // r2
 } finally {
 throw new RuntimeException("Or maybe
this one");
```

```
 }
}
public static void main(String[] moat) throws
Exception {
 new Lair().openDrawbridge(); // r3
}
```

- A. The code does not compile because of line r1.
- B. The code does not compile because of line r2.
- C. The code does not compile because of line r3.
- D. The code compiles, but a stack trace is printed at runtime.

# Q9

What is the output of the following application?

```
package broken;
class Problem implements RuntimeException {}
public class BiggerProblem extends Problem {
 public static void main(String uhOh[]) {
 try {
 throw new BiggerProblem();
 } catch (BiggerProblem re) {
 System.out.print("Problem?");
 } catch (Problem e) {
```

```
 System.out.print("Handled");
 } finally {
 System.out.print("Fixed!");
 }
}
```

- A. Problem?Fixed!
- B. Handled.Fixed!
- C. Problem?Handled.Fixed!
- D. The code does not compile.

# Q10

Given that `FileNotFoundException` is a subclass of `IOException`, what is the output of the following application?

```
package storage;
import java.io.*;
public class Backup {
 public void performBackup() {
 try {
 throw new IOException("Disk not found");
 } catch (Exception e) {
 try {
 throw new FileNotFoundException("File not
found");
 } catch (FileNotFoundException e) { // z1

```

```
 System.out.print("Failed");
 }
 }
 }
 public static void main(String... files) {
 new Backup().performBackup(); // z2
 }
}
```

- A. Failed
- B. The application compiles but a stack trace is printed at runtime.
- C. The code does not compile because of line z1.
- D. The code does not compile because of line z2.

# Corrections

# Q1

A

What is the output of the following application?

```
package zoo;
class BigCat {
 void roar(int level) throw RuntimeException { // m1
 if(level<3) throw new
IllegalArgumentException("Incomplete");
 System.out.print("Roar!");
 }
}
public class Lion extends BigCat {
 public void roar() { // m2
 System.out.print("Roar!!!");
 }
}
```

```
 }
 public static void main(String[] cubs) {
 final BigCat kitty = new Lion(); // m3
 kitty.roar(2);
 }
}
```

- A. The code does not compile because of line m1.
- B. The code does not compile because of line m2.
- C. The code does not compile because of line m3.
- D. The code compiles but a stack trace is printed at runtime.

## Q2

A

Given the following code snippet, which specific exception will be thrown?

```
final Object exception = new Exception();
final Exception data =
(RuntimeException)exception;
System.out.print(data);
```

- A. ClassCastException

- B. RuntimeException
- C. NullPointerException
- D. None of the above

# Q3

B

In the following application, the values of street and city have been omitted. Which one of the following is a possible output of executing this class?

- I. 350 5th Ave : New York
- II. Posted:350 5th Ave : New York

package registration;

public class Address {

    public String getAddress(String street, String city) {  
        try {

            return street.toString() + " :" + city.toString();

    } finally {

        System.out.print("Posted:");

    }

}

```
public static void main(String[] form) {
 String street = // value omitted
 String city = // value omitted
 System.out.print(new
 Address().getAddress(street,city));
}
```

- A. I only
- B. II only
- C. I and II
- D. None of the above

# Q4

C

What is the output of the following application?

```
package body;
class Organ {
 public void operate() throws
RuntimeException {
 throw new RuntimeException("Not
supported");
 }
}
public class Heart extends Organ {
 public void operate() throws Exception {
 System.out.print("beat");
 }
}
```

```
public static void main(String... cholesterol)
throws Exception {
```

```
 try {
 new Heart().operate();
 } finally {
 }
}
```

- A. beat
- B. Not supported
- C. The code does not compile.
- D. The code compiles but a stack trace is
printed at runtime.

# Q5

D

What is the output of the following application?

```
package clothing;
public class Coat {
 public Long zipper() throws Exception {
 try {
 String checkZipper = (String)new Object();
 } catch (Exception e) {
 throw RuntimeException("Broken!");
 }
 return null;
 }
}
```

```
public static void main(String... warmth) {
 try {
 new Coat().zipper();
 System.out.print("Finished!");
 } catch (Throwable t) {}
}
```

- A. Finished!
- B. Finished!, followed by a stack trace
- C. The application does not produce any output at runtime.
- D. The code does not compile.

# Q6

C

Given the following application, which type of exception will be printed in the stack trace at runtime?

```
package carnival;
public class WhackAnException {
 public static void main(String... hammer) {
 try {
 throw new ClassCastException();
 } catch (IllegalArgumentException e) {
 throw new IllegalArgumentException();
 } catch (RuntimeException e) {
 throw new NullPointerException();
 }
 }
}
```

```
} finally {
 throw new RuntimeException();
}
}
```

- A. `IllegalArgumentException`
- B. `NullPointerException`
- C. `RuntimeException`
- D. The code does not compile.

Q7

D

Which of these method signatures  
is allowed in a class implementing  
the Outfielder  
interface?

```
class OutOfBoundsException
extends BadCatchException {}

class BadCatchException extends
Exception {}

public interface Outfielder {
 public void catchBall() throws
```

- ```
    OutOfBoundsException;  
}
```
- A. public int catchBall() throws OutOfBoundsException
 - B. public int catchBall() throws BadCatchException
 - C. public int catchBall() throws Exception
 - D. None of the above

Q8

D

What is the result of compiling and running the following application?

```
package castles;
class DragonException extends Exception {}
public class Lair {
    public void openDrawbridge() throws Exception // r1
    try {
        throw new Exception("This Exception");
    } catch (RuntimeException e) {
        throw new DragonException(); // r2
    } finally {
        throw new RuntimeException("Or maybe
this one");
```

```
    }
}
public static void main(String[] moat) throws
Exception {
    new Lair().openDrawbridge(); // r3
}
```

- A. The code does not compile because of line r1.
- B. The code does not compile because of line r2.
- C. The code does not compile because of line r3.
- D. The code compiles, but a stack trace is printed at runtime.

Q9

D

What is the output of the following application?

```
package broken;
class Problem implements RuntimeException {}
public class BiggerProblem extends Problem {
    public static void main(String uhOh[]) {
        try {
            throw new BiggerProblem();
        } catch (BiggerProblem re) {
            System.out.print("Problem?");
        } catch (Problem e) {
```

```
        System.out.print("Handled");
    } finally {
        System.out.print("Fixed!");
    }
}
```

- A. Problem?Fixed!
- B. Handled.Fixed!
- C. Problem?Handled.Fixed!
- D. The code does not compile.

Q10

C

Given that FileNotFoundException is a subclass of IOException, what is the output of the following application?

```
package storage;
import java.io.*;
public class Backup {
    public void performBackup() {
        try {
            throw new IOException("Disk not found");
        } catch (Exception e) {
            try {
                throw new FileNotFoundException("File not
found");
            } catch (FileNotFoundException e) { // z1

```

```
        System.out.print("Failed");
    }
}
public static void main(String... files) {
    new Backup().performBackup(); // z2
}
```

- A. Failed
- B. The application compiles but a stack trace is printed at runtime.
- C. The code does not compile because of line z1.
- D. The code does not compile because of line z2.

More Questions

Q1

D

What is the output of the following application?

```
package ballroom;
public class Dance {
    public static void swing(int... beats) throws
ClassCastException {
        try {
            System.out.print("1"+beats[2]); // p1
        } catch (RuntimeException e) {
            System.out.print("2");
        } catch (Exception e) {
            System.out.print("3");
        } finally {
            System.out.print("4");
        }
    }
}
```

```
    }
    public static void main(String... music) {
        new Dance().swing(0,0); // p2
        System.out.print("5");
    }
}
A. 145
B. 1045
C. 24, followed by a stack trace
D. 245
E. The code does not compile because of line p1.
F. The code does not compile because of line p2.
```

Q2

E

What is the output of the following?

```
List<String> drinks =  
Arrays.asList("can", "cup");  
for (int container = drinks.size();  
container > 0; container++) {  
  
System.out.print(drinks.get(container - 1) + ",");
```

- }
- A. can,cup,
- B. cup,can,
- C. The code does not compile.
- D. This is an infinite loop.
- E. The code compiles but throws an exception at runtime.

Q3

A, E, F

Which of the following method signatures are valid declarations of an entry point in a Java application? (Choose three.)

- A. public static void main(String... widgets)
- B. public static void main(String sprockets)
- C. protected static void

- main(String[] args)
- D. public static int void main(String[] arg)
- E. public static final void main(String []a)
- F. public static void main(String[] data)

Q4

C, D, E

Given the application below and the choices available, which lines must all be removed to allow the code to compile? (Choose three.)

```
1: package year;
2: public class Seasons {
3:   public static void main(String[] time) {
4:     final long winter = 10;
5:     final byte season = 2;
6:     int fall = 4;
7:     final short summer = 3;
8:     switch(season) {
9:       case 1:
10:      case winter: System.out.print("winter");
11:      default:
```

```
12:       case fall: System.out.print("fall");
13:       case summer: System.out.print("summer");
14:       default:
15:     }
16:   }
17: }
```

- A. Line 8
- B. Line 9
- C. Line 10
- D. Line 11
- E. Line 12
- F. Line 13

Q5

A, B, C

Given the application below, which lines do not compile? (Choose three.)

```
package furryfriends;  
interface Friend {  
    protected String getName(); // h1  
}  
class Cat implements Friend {  
    String getName() { // h2  
        return "Kitty";  
    }  
}  
public class Dog implements Friend {  
    String getName() throws RuntimeException { // h3  
        return "Doggy";  
    }  
}
```

```
}  
public static void main(String[] adoption) {  
    Friend friend = new Dog(); // h4  
    System.out.print(((Cat)friend).getName()); // h5  
    System.out.print(((Dog)null).getName()); // h6  
}  
}
```

- A. Line h1
- B. Line h2
- C. Line h3
- D. Line h4
- E. Line h5
- F. Line h6

Q6

B, E, F

Which of the following are unchecked exceptions? (Choose three.)

- A. FileNotFoundException
- B. ArithmeticException
- C. IOException
- D. Exception
- E. IllegalArgumentException
- F. RuntimeException

Q7

What is the result of compiling and executing the following application?

```
package ranch;  
public class Cowboy {  
    private int space = 5;  
    private double ship = space < 2 ? 1 : 10; // }  
g1  
    public void printMessage() {  
        if(ship>1) {  
            System.out.println("Goodbye");  
        } if(ship<10 && space>=2)  
        System.out.println("Hello"); // g2  
        else System.out.println("See you again");  
    }  
}
```

```
    }  
    public static final void main(String... stars)  
    {  
        new Cowboy().printMessage();  
    }
```

- A. It only prints Hello.
- B. It only prints Goodbye.
- C. It only prints See you again.
- D. It does not compile because of line g1.
- E. It does not compile because of line g2.
- F. None of the above

Q8

B, C, D

Given the following three class declarations, which sets of access modifiers can be inserted, in order, into the blank lines below that would allow all of the classes to compile? (Choose three.)

```
package wake;
public class Alarm {
    static int clock;
    long getTime() {return clock;}
}
package wake;
public class Coffee {
    private boolean bringCoffee() { return new
        Alarm().clock<10;}
}
```

```
}
```

package sleep;

```
public class Snooze extends wake.Alarm {
    private boolean checkTime() { return
        getTime()>10;}
}
```

- A. protected and package-private (blank)
- B. public and public
- C. package-private (blank) and protected
- D. protected and protected
- E. private and public
- F. package-private (blank) and package-private (blank)

Q9

Given that FileNotFoundException is a subclass of IOException and Long is a subclass of Number, what is the output of the following application?

```
package materials;  
import java.io.*;  
class CarbonStructure {  
    protected long count;  
    public abstract Number getCount() throws IOException;  
// q1  
    public CarbonStructure(int count) { this.count = count; } }  
}  
public class Diamond extends CarbonStructure {  
    public Diamond() { super(15); }  
    public Long getCount() throws FileNotFoundException {  
// q2  
        return count;  
    }
```

```
public static void main(String[] cost) {  
    try {  
        final CarbonStructure ring = new Diamond(); // q3  
        System.out.print(ring.getCount()); // q4  
    } catch (IOException e) {  
        e.printStackTrace();  
    } }
```

- A. 15
- B. It does not compile because of line q1.
- C. It does not compile because of line q2.
- D. It does not compile because of line q3.
- E. It does not compile because of line q4.
- F. It compiles but throws an exception at runtime.

B

Q10

C

How many lines contain a compile error? 9:

```
1: import java.time.*;
2: import java.time.format.*;
3:
4: public class HowLong {
5:     public void main(String h) {
6:         LocalDate newYears = new
LocalDate(2017, 1, 1);
7:         Period period =
Period.ofYears(1).ofDays(1);
8:         DateTimeFormat format =
DateTimeFormat.ofPattern("MM-dd-
yyyy");
```

```
System.out.print(format.format(newYears
.minus(period)));
10: }
11: }
A. None
B. One
C. Two
D. Three
E. Four
F. Five
```

Q11

A, E

Which of the following statements about try-catch blocks are correct? (Choose two.)

- E. A try block can have zero or more catch blocks.
- F. A try block can have zero or more finally blocks.
- A. A catch block can never appear after a finally block.
- B. A try block must be followed by a catch block.
- C. A finally block can never appear after a catch block.
- D. A try block must be followed by a finally block.

Q12

B

What is printed by the following code snippet?

```
int fish = 1 + 2 * 5>=2 ? 4 : 2;
```

```
int mammals = 3 < 3 ? 1 : 5>=5 ? 9 :  
7;
```

```
System.out.print(fish+mammals+"")  
);
```

- A. 49
- B. 13

C. 18

D. 99

E. It does not compile.

Q13

A, C, E

Which of the following statements about objects, reference types, and casting are correct?
(Choose three.)

- A. An object can be assigned to an inherited interface reference variable without an explicit cast.
- B. The compiler can prevent all explicit casts that lead to an exception at runtime.
- C. Casting an object to a reference variable does not modify the object in

memory.

- D. An object can be assigned to a subclass reference variable without an explicit cast.
- E. An object can be assigned to a superclass reference variable without an explicit cast.
- F. An implicit cast of an object to one of its inherited types can sometimes lead to a `ClassCastException` at runtime.

Q14

F

What is the output of the following when run as java EchoFirst seed flower plant?

```
package unix;
import java.util.*;
public class EchoFirst {
    public static void main(String[] args) {
        int result =
        Arrays.binarySearch(args, args[0]);
        System.out.println(result);
    }
}
```

- A. 0
- B. 1
- C. 2
- D. The code does not compile.
- E. The code compiles but throws an exception at runtime.
- F. The output is not guaranteed.

Q15

B

How many objects are eligible for garbage collection at the end of the main() method?

```
package store;
public class Shoes {
    static String shoe1 = new
String("sandal");
    static String shoe2 = new String("flip
flop");
    public static void shopping() {
        String shoe3 = new String("croc");
        shoe2 = shoe1;
        shoe1 = shoe3;
```

```
}  
public static void main(String... args) {  
    shopping();  
}  
}
```

- A. None
- B. One
- C. Two
- D. Three
- E. The code does not compile.

Q16

E

Fill in the blanks: The keyword is used in method declarations, the keyword is used to guarantee a statement will execute even if an exception is thrown, and

the keyword is used to throw an exception to the surrounding process.

- A. throw, finally, throws

- B. throws, catch, throw
- C. catch, finally, throw
- D. finally, catch, throw
- E. throws, finally, throw

Q17

B, E

Which statements best describe the result of this code? (Choose two.)

```
package nyc;
public class TouristBus {
    public static void main(String... args) {
        String[] nycTourLoops = new String[] {
            "Downtown", "Uptown", "Brooklyn" };
        String[] times = new String[] { "Day",
            "Night" };
        for (int i = 0, j = 0; i <
nycTourLoops.length; i++, j++)
            }
```

```
System.out.println(nycTourLoops[i] + " "
+ times[j]);}
```

- A. The println causes one line of output.
- B. The println causes two lines of output.
- C. The println causes three lines of output.
- D. The code terminates successfully.
- E. The code throws an exception at runtime.

Q18

E

Fill in the blanks: Because of _____ D. inheritance, abstract,
, it is possible to _____ a method, polymorphism
which allows Java to support _____ E. virtual methods, override,
. polymorphism.

- A. abstract methods, override,
inheritance
- B. concrete methods, overload,
inheritance
- C. virtual methods, overload,
interfaces

Q19

E

What is the result of the following?

```
package calendar;
```

```
public class Seasons {
```

```
    public static void seasons(String...  
names) {
```

```
        int l = names[1].length(); // s1
```

```
        System.out.println(names[l]); // s2
```

```
}
```

```
public static void main(String[] args) {  
    seasons("Summer", "Fall", "Winter",
```

```
        "Spring");
```

```
}
```

```
}
```

A. Fall

B. Spring

C. The code does not compile.

D. The code throws an exception on
line s1.

E. The code throws an exception on
line s2.

Q20

D

How many lines of the following application contain compilation errors?

```
1: package percussion;  
2:  
3: interface MakesNoise {}  
4: abstract class Instrument implements  
   MakesNoise {  
5:   public Instrument(int beats) {}  
6:   public void play() {}  
7: }  
8: public class Drum extends Instrument {  
9:   public void play(int count) {}  
10:  public void concert() {
```

```
11:    super.play(5);  
12: }  
13: public static void main(String[] beats) {  
14:   MakesNoise mn = new Drum();  
15:   mn.concert();  
16: }  
17: }  
A. None. The code compiles and runs without issue.  
B. One  
C. Two  
D. Three  
E. Four
```

Q21

B

What is the output of the following application?

```
package fly;
public class Helicopter {
    public int adjustPropellers(int length, String[]
        type) {
        length++;
        type[0] = "LONG";
        return length;
    }
    public static void main(String[] climb) {
        final Helicopter h = new Helicopter();
        int length = 5;
        String[] type = new String[1];
```

```
        length = h.adjustPropellers(length, type);
        System.out.print(length+","+type[0]);
    }
}
A. 5, LONG
B. 6, LONG
C. 5, null
D. 6, null
E. The code does not compile.
F. The code compiles but throws an exception at
runtime.
```

Q22

D

How many lines of the following application do not compile?

}

```
package castles;

class OpenDoorException extends Exception {}

class CableSnapException extends OpenDoorException {}

public class Palace {

    public void openDrawbridge() throws Exception {
        try {
            throw new Exception("Problem");
        } catch (OpenDoorException e) {
            throw new OpenDoorException();
        } catch (CableSnapException ex) {
            try {
                throw new OpenDoorException();
            } catch (Exception ex) {
            } finally {
                System.out.println("Almost done");
            }
        }
    }

    public static void main(String[] moat) throws IllegalArgumentException {
        new Palace().openDrawbridge();
    }
}
```

A. None. The code compiles and produces a stack trace at runtime.
B. One
C. Two
D. Three
E. Four
F. Five

- A. None. The code compiles and produces a stack trace at runtime.
 - B. One
 - C. Two
 - D. Three
 - E. Four
 - F. Five

Q23

D

Choose the best answer: _____ and _____ are two properties that go hand in hand to improve class design by structuring a class with related attributes and actions while protecting the underlying data from access by other classes.

- A. Optimization and platform independence
- B. Platform independence and

encapsulation

- C. Platform independence and inheritance
- D. Object orientation and encapsulation
- E. Inheritance and polymorphism

Q24

E

What is the output of the following?

B. false true
string bike1 = "speedy"; C. true false
string bike2 = new String("speedy"); D. true true
boolean test1 = bike1 == bike2; E. The code does not compile.
boolean test2 = bike1.equals(bike2); F. The code compiles but throws an
System.out.println(test1 + " " + exception at runtime.
test2);
A. false false

Q25

A

What is the output of the following when run as java EchoFirst seed flower plant?

```
package unix;  
import java.util.*;  
public class EchoFirst {  
    public static void main(String[] args) {  
        Arrays.sort(args);  
        int result = Arrays.binarySearch(args,  
                                         args[0]);  
        System.out.println(result);  
    }  
}
```

- A. 0
- B. 1
- C. 2
- D. The code does not compile.
- E. The code compiles but throws an exception at runtime.
- F. The output is not guaranteed.

Q26

B,C,D

Which are true statements?
(Choose three.)

- A. Every do-while loop can be rewritten as a for-each loop.
- B. Every for-each loop can be rewritten as a do-while loop.
- C. Every for-each loop can be rewritten as a traditional for loop.
- D. Every for-each loop can be

rewritten as a while loop.

- E. Every traditional for loop can be rewritten as a for-each loop.
- F. Every while loop can be rewritten as a for-each loop.

Q27

E

How many lines does this program print?

```
import java.time.*;  
public class OnePlusOne {  
    public static void main(String...  
    nums) {  
        LocalDate time = LocalDate.of(1,  
        11);  
        while (time.getHour() < 1) {  
            time.plusHours(1);  
        }  
    }  
}
```

System.out.println("in loop");

}

}

}

- A. None
- B. One
- C. Two
- D. This is an infinite loop.
- E. The code does not compile.

Q28

C

How many objects are eligible for garbage collection immediately before the end of the main() method?

```
public class Tennis {  
    public static void main(String[] game) {  
        String[] balls = new String[1];  
        int[] scores = new int[1];  
        balls = null;
```

```
        scores = null;  
    }  
}
```

- A. None
- B. One
- C. Two
- D. Three
- E. Four

Q29

B

What is the output of the following?

- 14: int count = 0;
 - 15: LocalDate date = LocalDate.of(2017, Month.JANUARY, 1);
 - 16: while (date.getMonth() != Month.APRIL)
 - 17: date = date.minusMonths(1);
 - 18: count++;
 - 19: System.out.println(count);
- A. 0
 - B. 1
 - C. 3
 - D. 9
 - E. This is an infinite loop.
 - F. The code does not compile.

Q30

D

How many lines of the following class do not compile?

```
1: package arctic;
2: abstract class Bear {
3:     protected int sing;
4:     protected abstract int grunt();
5:     int sing() {
6:         return sing;
7:     }
8: }
9: public class PolarBear extends Bear {
10:    int grunt() {
```

- 11: sing() += 10;
- 12: return super.grunt()+1;
- 13: return 10;
- 14: }
- 15: }
 - A. None, the class compiles without issue.
 - B. One
 - C. Two
 - D. Three
 - E. Four
 - F. Five