Explore More

Subcription : Premium CDAC NOTES & MATERIAL @99

Contact to Join
Premium Group

Click to Join
Telegram Group

For More E-Notes

Join Our Community to stay Updated

TAP ON THE ICONS TO JOIN!

	codewitharrays.in freelance project available to buy contact on 8007592194	
SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

	freelance_Project available to buy contact on 8007592194	
21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+NextJs+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql

41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48	Train Ticket Booking Project	React+Springboot+MySql
49	Quizz Application Project	JSP+Springboot+MySql
50	Hotel Room Booking Project	React+Springboot+MySql
51	Online Crime Reporting Portal Project	React+Springboot+MySql
52	Online Child Adoption Portal Project	React+Springboot+MySql
53	online Pizza Delivery System Project	React+Springboot+MySql
54	Online Social Complaint Portal Project	React+Springboot+MySql
55	Electric Vehical management system Project	React+Springboot+MySql
56	Online mess / Tiffin management System Project	React+Springboot+MySql
57		React+Springboot+MySql
58		React+Springboot+MySql
59		React+Springboot+MySql
60		React+Springboot+MySql

Spring Boot + React JS + MySQL Project List

Sr.No	Project Name	YouTube Link
1	Online E-Learning Hub Platform Project	https://youtu.be/KMjyBaWmgzg?si=YckHuNzs7eC84-IW
2	PG Mate / Room sharing/Flat sharing	https://youtu.be/4P9clHg3wvk?si=4uEsi0962CG6Xodp
3	Tour and Travel System Project Version 1.0	https://youtu.be/-UHOBywHaP8?si=KHHfE_A0uv725f12
4	Marriage Hall Booking	https://youtu.be/VXz0kZQi5to?si=llOS-QG3TpAFP5k7
5	Ecommerce Shopping project	https://youtu.be/vJ_C6LkhrZ0?si=YhcBylSErvdn7paq
6	Bike Rental System Project	https://youtu.be/FIzsAmIBCbk?si=7ujQTJqEgkQ8ju2H
7	Multi-Restaurant management system	https://youtu.be/pvV-pM2Jf3s?si=PgvnT-yFc8ktrDxB
8	Hospital management system Project	https://youtu.be/IynIouBZvY4?si=CXzQs3BsRkjKhZCw
9	Municipal Corporation system Project	https://youtu.be/cVMx9NVyI4I?si=qX0oQt-GT-LR_5jF
10	Tour and Travel System Project version 2.0	https://youtu.be/_4u0mB9mHXE?si=gDiAhKBowi2gNUKZ

Sr.No	Project Name	YouTube Link
11	Tour and Travel System Project version 3.0	https://youtu.be/Dm7nOdpasWg?si=P_Lh2gcOFhlyudug
12	Gym Management system Project	https://youtu.be/J8_7Zrkg7ag?si=LcxV51ynfUB7OptX
13	Online Driving License system Project	https://youtu.be/3yRzsMs8TLE?si=JRI_z4FDx4Gmt7fn
14	Online Flight Booking system Project	https://youtu.be/m755rOwdk8U?si=HURvAY2VnizIyJlh
15	Employee management system project	https://youtu.be/lD1iE3W_GRw?si=Y_jv1xV_BljhrD0H
16	Online student school or college portal	https://youtu.be/4A25aEKfei0?si=RoVgZtxMk9TPdQvD
17	Online movie booking system project	https://youtu.be/Lfjv_U74SC4?si=fiDvrhhrjb4KSlSm
18	Online Pizza Delivery system project	https://youtu.be/Tp3izreZ458?si=8eWAOzA8SVdNwlyM
19	Online Crime Reporting system Project	https://youtu.be/0UlzReSk9tQ?si=6vN0e70TVY1GOwPO
20	Online Children Adoption Project	https://youtu.be/3T5HC2HKyT4?si=bntP78niYH802I7N

TWO-MINUTE DRILL

Remember that in this chapter, when we talk about classes, we're referring to non-inner classes, or *top-level* classes. We'll devote all of Chapter 8 to inner classes.

Identifiers (Objective 1.3)

- ❑ Identifiers can begin with a letter, an underscore, or a currency character.
- ❑ After the first character, identifiers can also include digits.
- ❑ Identifiers can be of any length.
- ❑ JavaBeans methods must be named using camelCase, and depending on the method's purpose, must start with `set`, `get`, `is`, `add`, or `remove`.

Declaration Rules (Objective 1.1)

- ❑ A source code file can have only one `public` class.
- ❑ If the source file contains a `public` class, the filename must match the `public` class name.
- ❑ A file can have only one `package` statement, but multiple `imports`.
- ❑ The `package` statement (if any) must be the first (non-comment) line in a source file.
- ❑ The `import` statements (if any) must come after the `package` and before the class declaration.
- ❑ If there is no `package` statement, `import` statements must be the first (non-comment) statements in the source file.
- ❑ `package` and `import` statements apply to all classes in the file.
- ❑ A file can have more than one nonpublic class.
- ❑ Files with no `public` classes have no naming restrictions.

Class Access Modifiers (Objective 1.1)

- ❑ There are three access modifiers: `public`, `protected`, and `private`.
- ❑ There are four access levels: public, protected, default, and private.
- ❑ Classes can have only `public` or default access.
- ❑ A class with default access can be seen only by classes within the same package.
- ❑ A class with `public` access can be seen by all classes from all packages.

- Class visibility revolves around whether code in one class can
  - Create an instance of another class
  - Extend (or subclass), another class
  - Access methods and variables of another class

Class Modifiers (Nonaccess) (Objective 1.2)

- Classes can also be modified with `final`, `abstract`, or `strictfp`.
- A class cannot be both `final` and `abstract`.
- A `final` class cannot be subclassed.
- An `abstract` class cannot be instantiated.
- A single `abstract` method in a class means the whole class must be abstract.
- An `abstract` class can have both `abstract` and nonabstract methods.
- The first concrete class to extend an `abstract` class must implement all of its `abstract` methods.

Interface Implementation (Objective 1.2)

- Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
- Interfaces can be implemented by any class, from any inheritance tree.
- An interface is like a 100-percent `abstract` class, and is implicitly abstract whether you type the `abstract` modifier in the declaration or not.
- An interface can have only abstract methods, no concrete methods allowed.
- Interface methods are by default `public` and `abstract`—explicit declaration of these modifiers is optional.
- Interfaces can have constants, which are always implicitly `public`, `static`, and `final`.
- Interface constant declarations of `public`, `static`, and `final` are optional in any combination.
- A legal nonabstract implementing class has the following properties:
  - It provides concrete implementations for the interface's methods.
  - It must follow all legal override rules for the methods it implements.
  - It must not declare any new checked exceptions for an implementation method.

- It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.
- It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.
- It must maintain the exact signature (allowing for covariant returns) and return type of the methods it implements (but does not have to declare the exceptions of the interface).

- A class implementing an interface can itself be `abstract`.
- An `abstract` implementing class does not have to implement the interface methods (but the first concrete subclass must).
- A class can extend only one class (no multiple inheritance), but it can implement many interfaces.
- Interfaces can extend one or more other interfaces.
- Interfaces cannot extend a class, or implement a class or interface.
- When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

Member Access Modifiers (Objectives 1.3 and 1.4)

- Methods and instance (nonlocal) variables are known as "members."
- Members can use all four access levels: public, protected, default, private.
- Member access comes in two forms:
  - Code in one class can access a member of another class.
  - A subclass can inherit a member of its superclass.
- If a class cannot be accessed, its members cannot be accessed.
- Determine class visibility before determining member visibility.
- `public` members can be accessed by all other classes, even in other packages.
- If a superclass member is public, the subclass inherits it—regardless of package.
- Members accessed without the dot operator (.) must belong to the same class.
- `this.` always refers to the currently executing object.
- `this.aMethod()` is the same as just invoking `aMethod()`.
- `private` members can be accessed only by code in the same class.
- `private` members are not visible to subclasses, so private members cannot be inherited.

- Default and `protected` members differ only when subclasses are involved:
  - Default members can be accessed only by classes in the same package.
  - `protected` members can be accessed by other classes in the same package, plus subclasses regardless of package.
  - `protected` = package plus kids (kids meaning subclasses).
  - For subclasses outside the package, the `protected` member can be accessed only through inheritance; a subclass outside the package cannot access a `protected` member by using a reference to a superclass instance (in other words, inheritance is the only mechanism for a subclass outside the package to access a `protected` member of its superclass).
  - A `protected` member inherited by a subclass from another package is not accessible to any other class in the subclass package, except for the subclass' own subclasses.

Local Variables (Objective 1.3)

- Local (method, automatic, or stack) variable declarations cannot have access modifiers.
- `final` is the only modifier available to local variables.
- Local variables don't get default values, so they must be initialized before use.

Other Modifiers—Members (Objective 1.3)

- `final` methods cannot be overridden in a subclass.
- `abstract` methods are declared, with a signature, a return type, and an optional throws clause, but are not implemented.
- `abstract` methods end in a semicolon—no curly braces.
- Three ways to spot a non-abstract method:
  - The method is not marked `abstract`.
  - The method has curly braces.
  - The method has code between the curly braces.
- The first nonabstract (concrete) class to extend an `abstract` class must implement all of the `abstract` class' `abstract` methods.
- The `synchronized` modifier applies only to methods and code blocks.
- `synchronized` methods can have any access control and can also be marked `final`.

- `abstract` methods must be implemented by a subclass, so they must be inheritable. For that reason:
  - `abstract` methods cannot be `private`.
  - `abstract` methods cannot be `final`.
- The `native` modifier applies only to methods.
- The `strictfp` modifier applies only to classes and methods.

Methods with var-args (Objective 1.4)

- As of Java 5, methods can declare a parameter that accepts from zero to many arguments, a so-called var-arg method.
- A var-arg parameter is declared with the syntax `type... name`; for instance: `doStuff(int... x) { }`
- A var-arg method can have only one var-arg parameter.
- In methods with normal parameters and a var-arg, the var-arg must come last.

Variable Declarations (Objective 1.3)

- Instance variables can
  - Have any access control
  - Be marked `final` or `transient`
- Instance variables can't be `abstract`, `synchronized`, `native`, or `strictfp`.
- It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing."
- `final` variables have the following properties:
  - `final` variables cannot be reinitialized once assigned a value.
  - `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
  - `final` reference variables must be initialized before the constructor completes.
- There is no such thing as a `final` object. An object reference marked `final` does not mean the object itself is immutable.
- The `transient` modifier applies only to instance variables.
- The `volatile` modifier applies only to instance variables.

Array Declarations (Objective 1.3)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be to the left or right of the variable name.
- It is never legal to include the size of an array in the declaration.
- An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.

Static Variables and Methods (Objective 1.4)

- They are not tied to any particular instance of a class.
- No classes instances are needed in order to use `static` members of the class.
- There is only one copy of a `static` variable / class and all instances share it.
- `static` methods do not have direct access to non-static members.

Enums (Objective 1.3)

- An `enum` specifies a list of constant values assigned to a type.
- An `enum` is NOT a String or an int; an enum constant's type is the enum type. For example, SUMMER and FALL are of the enum type Season.
- An `enum` can be declared outside or inside a class, but NOT in a method.
- An `enum` declared outside a class must NOT be marked `static`, `final`, `abstract`, `protected`, or `private`.
- Enums can contain constructors, methods, variables, and constant class bodies.
- `enum` constants can send arguments to the `enum` constructor, using the syntax BIG(8), where the int literal 8 is passed to the `enum` constructor.
- `enum` constructors can have arguments, and can be overloaded.
- `enum` constructors can NEVER be invoked directly in code. They are always called automatically when an `enum` is initialized.
- The semicolon at the end of an `enum` declaration is optional. These are legal:

```
enum Foo { ONE, TWO, THREE}
enum Foo { ONE, TWO, THREE};
```

- `MyEnum.values()` returns an array of `MyEnum`'s values.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

If you have a rough time with these at first, don't beat yourself up. Be positive. Repeat nice affirmations to yourself like, "I am smart enough to understand enums" and "OK, so that other guy knows enums better than I do, but I bet he can't <insert something you *are* good at> like me."

1. Which is true? (Choose all that apply.)

A. "X extends Y" is correct if and only if X is a class and Y is an interface

B. "X extends Y" is correct if and only if X is an interface and Y is a class

C. "X extends Y" is correct if X and Y are either both classes or both interfaces

D. "X extends Y" is correct for all combinations of X and Y being classes and/or interfaces

2. Which method names follow the JavaBeans standard? (Choose all that apply.)

A. `addSize`

B. `getCust`

C. `deleteRep`

D. `isColorado`

E. `putDimensions`

3. Given:

```
1. class Voop {
2.   public static void main(String [] args) {
3.     doStuff(1);
4.     doStuff(1,2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`

B. `static void doStuff(int[] doArgs) { }`

C. `static void doStuff(int doArgs...) { }`

D. `static void doStuff(int... doArgs, int y) { }`

E. `static void doStuff(int x, int... doArgs) { }`

4. Given:

```
1. enum Animals {
2.   DOG("woof"), CAT("meow"), FISH("burble");
3.   String sound;
4.   Animals(String s) { sound = s; }
5. }
6. class TestEnum {
7.   static Animals a;
8.   public static void main(String[] args) {
9.     System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.  }
11. }
```

What is the result?

A. `woof burble`

B. Multiple compilation errors

C. Compilation fails due to an error on line 2

D. Compilation fails due to an error on line 3

E. Compilation fails due to an error on line 4

F. Compilation fails due to an error on line 9

5. Given two files:

```
1. package pkgA;
2. public class Foo {
3.   int a = 5;
4.   protected int b = 6;
5.   public int c = 7;
6. }
```

```
3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.   public static void main(String[] args) {
7.     Foo f = new Foo();
8.     System.out.print(" " + f.a);
9.     System.out.print(" " + f.b);
10.    System.out.println(" " + f.c);
11.  }
12. }
```

What is the result? (Choose all that apply.)

A. 5 6 7

B. 5 followed by an exception

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

F. Compilation fails with an error on line 10

6. Given:

```
1. public class Electronic implements Device
       { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
     { public void doIt(int x) { } }
6.
7. class Phone3 extends Electronic implements Device
     { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 1

C. Compilation fails with an error on line 3

D. Compilation fails with an error on line 5

E. Compilation fails with an error on line 7

F. Compilation fails with an error on line 9

7. Given:

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
```

```
 8.     long [] x [5];
 9.     Boolean []ba[];
10.     enum Traffic { RED, YELLOW, GREEN };
11.   }
12. }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 6

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

F. Compilation fails with an error on line 10

8. Given:

```
3. public class TestDays {
4.   public enum Days { MON, TUE, WED };
5.   public static void main(String[] args) {
6.     for(Days d : Days.values() )
7.       ;
8.     Days [] d2 = Days.values();
9.     System.out.println(d2[2]);
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. `TUE`

B. `WED`

C. The output is unpredictable

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 6

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

9. Given:

```
4. public class Frodo extends Hobbit {
5.   public static void main(String[] args) {
6.     Short myGold = 7;
7.     System.out.println(countGold(myGold, 6));
8.   }
9. }
10. class Hobbit {
11.   int countGold(int x, int y) { return x + y; }
12. }
```

What is the result?

A. 13

B. Compilation fails due to multiple errors

C. Compilation fails due to an error on line 6

D. Compilation fails due to an error on line 7

E. Compilation fails due to an error on line 11

SELF TEST ANSWERS

1. Which is true? (Choose all that apply.)

A. "X extends Y" is correct if and only if X is a class and Y is an interface

B. "X extends Y" is correct if and only if X is an interface and Y is a class

C. "X extends Y" is correct if X and Y are either both classes or both interfaces

D. "X extends Y" is correct for all combinations of X and Y being classes and/or interfaces

Answer:

☑ **C** is correct.

☒ **A** is incorrect because classes implement interfaces, they don't extend them. **B** is incorrect because interfaces only "inherit from" other interfaces. **D** is incorrect based on the preceding rules. (Objective 1.2)

2. Which method names follow the JavaBeans standard? (Choose all that apply.)

A. `addSize`

B. `getCust`

C. `deleteRep`

D. `isColorado`

E. `putDimensions`

Answer:

☑ **B** and **D** use the valid prefixes '`get`' and '`is`'.

☒ **A** is incorrect because '`add`' can be used only with Listener methods. **C** and **E** are incorrect because '`delete`' and '`put`' are not standard JavaBeans name prefixes. (Objective 1.4)

3. Given:

```
1. class Voop {
2.   public static void main(String[] args) {
3.     doStuff(1);
4.     doStuff(1,2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`
B. `static void doStuff(int[] doArgs) { }`
C. `static void doStuff(int doArgs...) { }`
D. `static void doStuff(int... doArgs, int y) { }`
E. `static void doStuff(int x, int... doArgs) { }`

Answer:

☑ **A** and **E** use valid var-args syntax.

☒ **B** and **C** are invalid var-arg syntax, and **D** is invalid because the var-arg must be the last of a method's arguments. (Objective 1.4)

4. Given:

```
1. enum Animals {
2.   DOG("woof"), CAT("meow"), FISH("burble");
3.   String sound;
4.   Animals(String s) { sound = s; }
5. }
6. class TestEnum {
7.   static Animals a;
8.   public static void main(String [] args) {
9.     System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.  }
11. }
```

What is the result?

A. `woof burble`
B. Multiple compilation errors
C. Compilation fails due to an error on line 2
D. Compilation fails due to an error on line 3
E. Compilation fails due to an error on line 4
F. Compilation fails due to an error on line 9

Answer:

☑ **A** is correct; enums can have constructors and variables.

☒ **B, C, D, E,** and **F** are incorrect; these lines all use correct syntax. (Objective 1.3)

5. Given two files:

```
1. package pkgA;
2. public class Foo {
3.   int a = 5;
4.   protected int b = 6;
5.   public int c = 7;
6. }

3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.   public static void main(String[] args) {
7.     Foo f = new Foo();
8.     System.out.print(" " + f.a);
9.     System.out.print(" " + f.b);
10.    System.out.print(" " + f.c);
11.  }
12. }
```

What is the result? (Choose all that apply.)

A. 5 6 7
B. 5 followed by an exception
C. Compilation fails with an error on line 7
D. Compilation fails with an error on line 8
E. Compilation fails with an error on line 9
F. Compilation fails with an error on line 10

Answer:

☑ **D** and **E** are correct. Variable `a` has default access, so it cannot be accessed from outside the package. Variable `b` has protected access in pkgA.

☒ **A, B, C,** and **F** are incorrect based on the above information. (Objective 1.1)

6. Given:

```
1. public class Electronic implements Device
       { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
     { public void doIt(int x) { } }
6.
```

```
7. class Phone3 extends Electronic implements Device
     { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds
B. Compilation fails with an error on line 1
C. Compilation fails with an error on line 3
D. Compilation fails with an error on line 5
E. Compilation fails with an error on line 7
F. Compilation fails with an error on line 9

Answer:

☑ **A** is correct; all of these are legal declarations.

☒ **B, C, D, E,** and **F** are incorrect based on the above information. (Objective 1.2)

7. Given:

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
8.     long [] x [5];
9.     Boolean []ba[];
10.    enum Traffic { RED, YELLOW, GREEN };
11.  }
12. }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds
B. Compilation fails with an error on line 6
C. Compilation fails with an error on line 7
D. Compilation fails with an error on line 8
E. Compilation fails with an error on line 9
F. Compilation fails with an error on line 10

Answer:

☑ **C, D,** and **F** are correct. Variable names cannot begin with a #, an array declaration can't include a size without an instantiation, and enums can't be declared within a method.

☒ **A, B,** and **E** are incorrect based on the above information. (Objective 1.3)

8. Given:

```
3. public class TestDays {
4.   public enum Days { MON, TUE, WED };
5.   public static void main(String[] args) {
6.     for(Days d : Days.values() )
7.       ;
8.     Days [] d2 = Days.values();
9.     System.out.println(d2[2]);
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. `TUE`

B. `WED`

C. The output is unpredictable

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 6

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

Answer:

☑ **B** is correct. Every `enum` comes with a `static values()` method that returns an array of the `enum`'s values, in the order in which they are declared in the `enum`.

☒ **A, C, D, E, F,** and **G** are incorrect based on the above information. (Objective 1.3)

9. Given:

```
4. public class Frodo extends Hobbit {
5.   public static void main(String[] args) {
6.     Short myGold = 7;
7.     System.out.println(countGold(myGold, 6));
8.   }
9. }
10. class Hobbit {
11.   int countGold(int x, int y) { return x + y; }
12. }
```

What is the result?

A. 13
B. Compilation fails due to multiple errors
C. Compilation fails due to an error on line 6
D. Compilation fails due to an error on line 7
E. Compilation fails due to an error on line 11

Answer:

☑ **D** is correct. The `Short myGold` is autoboxed correctly, but the `countGold()` method cannot be invoked from a static context.

☒ **A, B, C,** and **E** are incorrect based on the above information. (Objective 1.4)

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter.

Encapsulation, IS-A, HAS-A (Objective 5.1)

- Encapsulation helps hide implementation behind an interface (or API).
- Encapsulated code has two features:
  - Instance variables are kept protected (usually with the private modifier).
  - Getter and setter methods provide access to instance variables.
- IS-A refers to inheritance or implementation.
- IS-A is expressed with the keyword `extends`.
- IS-A, "inherits from," and "is a subtype of" are all equivalent expressions.
- HAS-A means an instance of one class "has a" reference to an instance of another class or another instance of the same class.

Inheritance (Objective 5.5)

- Inheritance allows a class to be a subclass of a superclass, and thereby inherit `public` and `protected` variables and methods of the superclass.
- Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.
- All classes (except class `Object`), are subclasses of type `Object`, and therefore they inherit `Object`'s methods.

Polymorphism (Objective 5.2)

- Polymorphism means "many forms."
- A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
- A single object can be referred to by reference variables of many different types—as long as they are the same type or a supertype of the object.
- The reference variable's type (not the object's type), determines which methods can be called!
- Polymorphic method invocations apply only to overridden *instance* methods.

Overriding and Overloading (Objectives 1.5 and 5.4)

- Methods can be overridden or overloaded; constructors can be overloaded but not overridden.
- Abstract methods must be overridden by the first concrete (non-abstract) subclass.
- With respect to the method it overrides, the overriding method
  - Must have the same argument list.
  - Must have the same return type, except that as of Java 5, the return type can be a subclass—this is known as a covariant return.
  - Must not have a more restrictive access modifier.
  - May have a less restrictive access modifier.
  - Must not throw new or broader checked exceptions.
  - May throw fewer or narrower checked exceptions, or any unchecked exception.
- `final` methods cannot be overridden.
- Only inherited methods may be overridden, and remember that private methods are not inherited.
- A subclass uses `super.overriddenMethodName()` to call the superclass version of an overridden method.
- Overloading means reusing a method name, but with different arguments.
- Overloaded methods
  - Must have different argument lists
  - May have different return types, if argument lists are also different
  - May have different access modifiers
  - May throw different exceptions
- Methods from a superclass can be overloaded in a subclass.
- Polymorphism applies to overriding, not to overloading.
- Object type (not the reference variable's type), determines which overridden method is used at runtime.
- Reference type determines which overloaded method will be used at compile time.

Reference Variable Casting (Objective 5.2)

- There are two types of reference variable casting: downcasting and upcasting.
- Downcasting: If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.
- Upcasting: You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

Implementing an Interface (Objective 1.2)

- When you implement an interface, you are fulfilling its contract.
- You implement an interface by properly and concretely overriding all of the methods defined by the interface.
- A single class can implement many interfaces.

Return Types (Objective 1.5)

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
- Object reference return types can accept `null` as a return value.
- An array is a legal return type, both to declare and return as a value.
- For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
- Nothing can be returned from a `void`, but you can return nothing. You're allowed to simply say `return`, in any method with a `void` return type, to bust out of a method early. But you can't return nothing from a method with a non-void return type.
- Methods with an object reference return type, can return a subtype.
- Methods with an interface return type, can return any implementer.

Constructors and Instantiation (Objectives 1.6 and 5.4)

- A constructor is always invoked when a new object is created.

- Each superclass in an object's inheritance tree will have a constructor called.
- Every class, even an abstract class, has at least one constructor.
- Constructors must have the same name as the class.
- Constructors don't have a return type. If you see code with a return type, it's a method with the same name as the class, it's not a constructor.
- Typical constructor execution occurs as follows:
  - The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the Object constructor.
  - The Object constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.
- Constructors can use any access modifier (even `private`!).
- The compiler will create a default constructor if you don't create any constructors in your class.
- The default constructor is a no-arg constructor with a no-arg call to `super()`.
- The first statement of every constructor must be a call to either `this()` (an overloaded constructor) or `super()`.
- The compiler will add a call to `super()` unless you have already put in a call to `this()` or `super()`.
- Instance members are accessible only after the super constructor runs.
- Abstract classes have constructors that are called when a concrete subclass is instantiated.
- Interfaces do not have constructors.
- If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to `super()` with arguments matching those of the superclass constructor.
- Constructors are never inherited, thus they cannot be overridden.
- A constructor can be directly invoked only by another constructor (using a call to `super()` or `this()`).
- Issues with calls to `this()`
  - May appear only as the first statement in a constructor.
  - The argument list determines which overloaded constructor is called.

- ❑ Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call `super()` or the stack will explode.
- ❑ Calls to `this()` and `super()` cannot be in the same constructor. You can have one or the other, but never both.

Statics (Objective 1.3)

- ❑ Use `static` methods to implement behaviors that are not affected by the state of any instances.
- ❑ Use `static` variables to hold data that is class specific as opposed to instance specific—there will be only one copy of a `static` variable.
- ❑ All `static` members belong to the class, not to any instance.
- ❑ A `static` method can't access an instance variable directly.
- ❑ Use the dot operator to access `static` members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable, for instance:

```
d.doStuff();
```

becomes:

```
Dog.doStuff();
```

- ❑ `static` methods can't be overridden, but they can be redefined.

Coupling and Cohesion (Objective 5.1)

- ❑ Coupling refers to the degree to which one class knows about or uses members of another class.
- ❑ Loose coupling is the desirable state of having classes that are well encapsulated, minimize references to each other, and limit the breadth of API usage.
- ❑ Tight coupling is the undesirable state of having classes that break the rules of loose coupling.
- ❑ Cohesion refers to the degree in which a class has a single, well-defined role or responsibility.
- ❑ High cohesion is the desirable state of a class whose members support a single, well-focused role or responsibility.
- ❑ Low cohesion is the undesirable state of a class whose members support multiple, unfocused roles or responsibilities.

SELF TEST

1. Given:

```
public abstract interface Frobnicate { public void twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

A.
```
public abstract class Frob implements Frobnicate {
  public abstract void twiddle(String s) { }
}
```

B.
```
public abstract class Frob implements Frobnicate { }
```

C.
```
public class Frob extends Frobnicate {
  public void twiddle(Integer i) { }
}
```

D.
```
public class Frob implements Frobnicate {
  public void twiddle(Integer i) { }
}
```

E.
```
public class Frob implements Frobnicate {
  public void twiddle(String i) { }
  public void twiddle(Integer s) { }
}
```

2. Given:

```
class Top {
  public Top(String s) { System.out.print("B"); }
}
public class Bottom2 extends Top {
  public Bottom2(String s) { System.out.print("D"); }
  public static void main(String [] args) {
    new Bottom2("C");
    System.out.println(" ");
  }
}
```

What is the result?

A. `BD`

B. `DB`

C. `BDC`

D. `DBC`

E. Compilation fails

3. Given:

```
class Clidder {
  private final void flipper() { System.out.println("Clidder"); }
}

public class Clidlet extends Clidder {
  public final void flipper() { System.out.println("Clidlet");  }
  public static void main(String [] args) {
    new Clidlet().flipper();
  }
}
```

What is the result?

A. `Clidlet`

B. `Clidder`

C. `Clidder`
`Clidlet`

D. `Clidlet`
`Clidder`

E. Compilation fails

4. Using the **fragments** below, complete the following **code** so it compiles. Note, you may not have to fill all of the slots.

Code:

```
class AgedP {
  _____ _____  _____  _____  _____
  public AgedP(int x) {
    _____  _____  _____  _____  _____  _____
  }
}
public class Kinder extends AgedP {
  _____  _____  _____  _____  _____  _____
  public Kinder(int x) {
    _____  _____  _____  _____  _____ ();
  }
}
```

Fragments: Use the following fragments zero or more times:

AgedP	super	this	
(	)	{	}
;			

5. Which statement(s) are true? (Choose all that apply.)
   A. Cohesion is the OO principle most closely associated with hiding implementation details
   B. Cohesion is the OO principle most closely associated with making sure that classes know about other classes only through their APIs
   C. Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose
   D. Cohesion is the OO principle most closely associated with allowing a single object to be seen as having many types

6. Given the following,

```
1. class X { void do1() { } }
2. class Y extends X { void do2() { } }
3.
4. class Chrome {
5.   public static void main(String [] args) {
6.     X x1 = new X();
7.     X x2 = new Y();
8.     Y y1 = new Y();
9.     // insert code here
10. } }
```

Which, inserted at line 9, will compile? (Choose all that apply.)

A. `x2.do2();`
B. `(Y)x2.do2();`
C. `((Y)x2).do2();`
D. None of the above statements will compile

7. Given:

   1. ClassA has a ClassD
   2. Methods in ClassA use public methods in ClassB
   3. Methods in ClassC use public methods in ClassA
   4. Methods in ClassA use public variables in ClassB

   Which is most likely true? (Choose the most likely.)

   A. ClassD has low cohesion
   B. ClassA has weak encapsulation
   C. ClassB has weak encapsulation
   D. ClassB has strong encapsulation
   E. ClassC is tightly coupled to ClassA

8. Given:

```
 3. class Dog {
 4.   public void bark() { System.out.print("woof "); }
 5. }
 6. class Hound extends Dog {
 7.   public void sniff() { System.out.print("sniff "); }
 8.   public void bark() { System.out.print("howl "); }
 9. }
10. public class DogShow {
11.   public static void main(String[] args) { new DogShow().go(); }
12.   void go() {
13.     new Hound().bark();
14.     ((Dog) new Hound()).bark();
15.     ((Dog) new Hound()).sniff();
16.   }
17. }
```

   What is the result? (Choose all that apply.)

   A. `howl howl sniff`
   B. `howl woof sniff`
   C. `howl howl` followed by an exception
   D. `howl woof` followed by an exception
   E. Compilation fails with an error at line 14
   F. Compilation fails with an error at line 15

9. Given:

```
3. public class Redwood extends Tree {
4.   public static void main(String[] args) {
5.     new Redwood().go();
6.   }
7.   void go() {
8.     go2(new Tree(), new Redwood());
9.     go2((Redwood) new Tree(), new Redwood());
10.  }
11.  void go2(Tree t1, Redwood r1) {
12.     Redwood r2 = (Redwood)t1;
13.     Tree t2 = (Tree)r1;
14.  }
15. }
16. class Tree { }
```

What is the result? (Choose all that apply.)

A. An exception is thrown at runtime
B. The code compiles and runs with no output
C. Compilation fails with an error at line 8
D. Compilation fails with an error at line 9
E. Compilation fails with an error at line 12
F. Compilation fails with an error at line 13

10. Given:

```
3. public class Tenor extends Singer {
4.   public static String sing() { return "fa"; }
5.   public static void main(String[] args) {
6.     Tenor t = new Tenor();
7.     Singer s = new Tenor();
8.     System.out.println(t.sing() + " " + s.sing());
9.   }
10. }
11. class Singer { public static String sing() { return "la"; } }
```

What is the result?

A. `fa fa`
B. `fa la`
C. `la la`
D. Compilation fails
E. An exception is thrown at runtime

11. Given:

```
3. class Alpha {
4.   static String s = " ";
5.   protected Alpha() { s += "alpha "; }
6. }
7. class SubAlpha extends Alpha {
8.   private SubAlpha() { s += "sub "; }
9. }
10. public class SubSubAlpha extends Alpha {
11.   private SubSubAlpha() { s += "subsub "; }
12.   public static void main(String[] args) {
13.     new SubSubAlpha();
14.     System.out.println(s);
15.   }
16. }
```

What is the result?

A. `subsub`
B. `sub subsub`
C. `alpha subsub`
D. `alpha sub subsub`
E. Compilation fails
F. An exception is thrown at runtime

12. Given:

```
3. class Building {
4.   Building() {  System.out.print("b ");  }
5.   Building(String name) {
6.     this();   System.out.print("bn " + name);
7.   }
8. }
9. public class House extends Building {
10.   House() {  System.out.print("h ");  }
11.   House(String name) {
12.     this();   System.out.print("hn " + name);
13.   }
14.   public static void main(String[] args) { new House("x "); }
15. }
```

What is the result?

A. `h hn x`
B. `hn x h`
C. `b h hn x`
D. `b hn x h`
E. `bn x h hn x`
F. `b bn x h hn x`
G. `bn x b h hn x`
H. Compilation fails

13. Given:

```
3. class Mammal {
4.   String name = "furry ";
5.   String makeNoise() { return "generic noise"; }
6. }
7. class Zebra extends Mammal {
8.   String name = "stripes ";
9.   String makeNoise() { return "bray"; }
10. }
11. public class ZooKeeper {
12.   public static void main(String[] args) { new ZooKeeper().go(); }
13.   void go() {
14.     Mammal m = new Zebra();
15.     System.out.println(m.name + m.makeNoise());
16.   }
17. }
```

What is the result?

A. `furry bray`
B. `stripes bray`
C. `furry generic noise`
D. `stripes generic noise`
E. Compilation fails
F. An exception is thrown at runtime

14. You're designing a new online board game in which Floozels are a type of Jammers, Jammers can have Quizels, Quizels are a type of Klakker, and Floozels can have several Floozets. Which of the following fragments represent this design? (Choose all that apply.)

A.
```
import java.util.*;
interface Klakker { }
class Jammer { Set<Quizel> q; }
class Quizel implements Klakker { }
public class Floozel extends Jammer { List<Floozet> f; }
interface Floozet { }
```

B.
```
import java.util.*;
class Klakker { Set<Quizel> q; }
class Quizel extends Klakker { }
class Jammer { List<Floozel> f; }
class Floozet extends Floozel { }
public class Floozel { Set<Klakker> k; }
```

C.
```
import java.util.*;
class Floozet { }
class Quizel implements Klakker { }
class Jammer { List<Quizel> q; }
interface Klakker { }
class Floozel extends Jammer { List<Floozet> f; }
```

D.
```
import java.util.*;
interface Jammer extends Quizel { }
interface Klakker { }
interface Quizel extends Klakker { }
interface Floozel extends Jammer, Floozet { }
interface Floozet { }
```

15. Given:

```
3. class A { }
4. class B extends A { }
5. public class ComingThru {
6.   static String s = "-";
7.   public static void main(String[] args) {
8.     A[] aa = new A[2];
9.     B[] ba = new B[2];
10.    sifter(aa);
11.    sifter(ba);
12.    sifter(7);
13.    System.out.println(s);
14.  }
```

```
15.   static void sifter(A[]... a2)    { s += "1"; }
16.   static void sifter(B[]... b1)    { s += "2"; }
17.   static void sifter(B[] b1)       { s += "3"; }
18.   static void sifter(Object o)     { s += "4"; }
19. }
```

What is the result?

A. `-124`

B. `-134`

C. `-424`

D. `-434`

E. `-444`

F. Compilation fails

SELF TEST ANSWERS

1. Given:

```
public abstract interface Frobnicate { public void twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

A.
```
public abstract class Frob implements Frobnicate {
  public abstract void twiddle(String s) { }
}
```

B.
```
public abstract class Frob implements Frobnicate { }
```

C.
```
public class Frob extends Frobnicate {
  public void twiddle(Integer i) { }
}
```

D.
```
public class Frob implements Frobnicate {
  public void twiddle(Integer i) { }
}
```

E.
```
public class Frob implements Frobnicate {
  public void twiddle(String i) { }
  public void twiddle(Integer s) { }
}
```

Answer:

☑ **B** is correct, an `abstract` class need not implement any or all of an interface's methods. **E** is correct, the class implements the interface method and additionally overloads the `twiddle()` method.

☒ **A** is incorrect because `abstract` methods have no body. **C** is incorrect because classes implement interfaces they don't extend them. **D** is incorrect because overloading a method is not implementing it.

(Objective 5.4)

2. Given:

```
class Top {
  public Top(String s) { System.out.print("B"); }
}
public class Bottom2 extends Top {
  public Bottom2(String s) { System.out.print("D"); }
  public static void main(String [] args) {
    new Bottom2("C");
    System.out.println(" ");
} }
```

What is the result?

A. `BD`
B. `DB`
C. `BDC`
D. `DBC`
E. Compilation fails

Answer:

☑ **E** is correct. The implied `super()` call in Bottom2's constructor cannot be satisfied because there isn't a no-arg constructor in Top. A default, no-arg constructor is generated by the compiler only if the class has no constructor defined explicitly.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 1.6)

3. Given:

```
class Clidder {
  private final void flipper() { System.out.println("Clidder"); }
}

public class Clidlet extends Clidder {
  public final void flipper() { System.out.println("Clidlet");  }
  public static void main(String [] args) {
    new Clidlet().flipper();
} }
```

What is the result?

A. `Clidlet`
B. `Clidder`
C. `Clidder`
   `Clidlet`
D. `Clidlet`
   `Clidder`
E. Compilation fails

Answer:

☑ **A** is correct. Although a final method cannot be overridden, in this case, the method is private, and therefore hidden. The effect is that a new, accessible, method flipper is created. Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.

☒ **B, C, D,** and **E** are incorrect based on the preceding. (Objective 5.3)

4. Using the **fragments** below, complete the following **code** so it compiles. Note, you may not have to fill all of the slots.

Code:

```
class AgedP {

  __________  __________  __________  __________  _________
  public AgedP(int x) {

    __________  __________  __________  _________ __________
  }
}
public class Kinder extends AgedP {

  __________  __________  __________  _________  _________  __________
  public Kinder(int x) {

    __________  __________  __________ __________  __________ ();
  }
}
```

Fragments: Use the following fragments zero or more times:

AgedP	super	this	
(	)	{	}
;			

Answer:

```
class AgedP {
  AgedP() {}
  public AgedP(int x) {
  }
}
public class Kinder extends AgedP {
  public Kinder(int x) {
    super();
  }
}
```

As there is no droppable tile for the variable x and the parentheses (in the Kinder constructor), are already in place and empty, there is no way to construct a call to the superclass constructor

that takes an argument. Therefore, the only remaining possibility is to create a call to the no-argument superclass constructor. This is done as: `super();`. The line cannot be left blank, as the parentheses are already in place. Further, since the superclass constructor called is the no-argument version, this constructor must be created. It will not be created by the compiler because there is another constructor already present.
(Objective 5.4)

5 Which statement(s) are true? (Choose all that apply.)

A. Cohesion is the OO principle most closely associated with hiding implementation details

B. Cohesion is the OO principle most closely associated with making sure that classes know about other classes only through their APIs

C. Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose

D. Cohesion is the OO principle most closely associated with allowing a single object to be seen as having many types

Answer:

☑ Answer **C** is correct.

☒ **A** refers to encapsulation, **B** refers to coupling, and **D** refers to polymorphism.
(Objective 5.1)

6. Given the following,

```
1. class X { void do1() { } }
2. class Y extends X { void do2() { } }
3.
4. class Chrome {
5.   public static void main(String [] args) {
6.     X x1 = new X();
7.     X x2 = new Y();
8.     Y y1 = new Y();
9.     // insert code here
10.  }
11. }
```

Which, inserted at line 9, will compile? (Choose all that apply.)

A. `x2.do2();`

B. `(Y)x2.do2();`

C. `((Y)x2).do2();`

D. None of the above statements will compile

Answer:

☑ **C** is correct. Before you can invoke `Y`'s `do2` method you have to cast `x2` to be of type `Y`. Statement **B** looks like a proper cast but without the second set of parentheses, the compiler thinks it's an incomplete statement.

☒ **A, B** and **D** are incorrect based on the preceding.
(Objective 5.2)

7. Given:

1. ClassA has a ClassD
2. Methods in ClassA use public methods in ClassB
3. Methods in ClassC use public methods in ClassA
4. Methods in ClassA use public variables in ClassB

Which is most likely true? (Choose the most likely.)

A. ClassD has low cohesion

B. ClassA has weak encapsulation

C. ClassB has weak encapsulation

D. ClassB has strong encapsulation

E. ClassC is tightly coupled to ClassA

Answer:

☑ **C** is correct. Generally speaking, public variables are a sign of weak encapsulation.

☒ **A, B, D,** and **E** are incorrect, because based on the information given, none of these statements can be supported.
(Objective 5.1)

8. Given:

```
3. class Dog {
4.   public void bark() { System.out.print("woof "); }
5. }
6. class Hound extends Dog {
7.   public void sniff() { System.out.print("sniff "); }
```

```
 8.   public void bark() { System.out.print("howl "); }
 9. }
10. public class DogShow {
11.   public static void main(String[] args) { new DogShow().go(); }
12.   void go() {
13.     new Hound().bark();
14.     ((Dog) new Hound()).bark();
15.     ((Dog) new Hound()).sniff();
16.   }
17. }
```

What is the result? (Choose all that apply.)

A. `howl howl sniff`

B. `howl woof sniff`

C. `howl howl` followed by an exception

D. `howl woof` followed by an exception

E. Compilation fails with an error at line 14

F. Compilation fails with an error at line 15

Answer:

☑ **F** is correct. Class `Dog` doesn't have a `sniff` method.

☒ **A, B, C, D,** and **E** are incorrect based on the above information. (Objective 5.2)

9. Given:

```
 3. public class Redwood extends Tree {
 4.   public static void main(String[] args) {
 5.     new Redwood().go();
 6.   }
 7.   void go() {
 8.     go2(new Tree(), new Redwood());
 9.     go2((Redwood) new Tree(), new Redwood());
10.   }
11.   void go2(Tree t1, Redwood r1) {
12.      Redwood r2 = (Redwood)t1;
13.      Tree t2 = (Tree)r1;
14.   }
15. }
16. class Tree { }
```

What is the result? (Choose all that apply.)

A. An exception is thrown at runtime

B. The code compiles and runs with no output

C. Compilation fails with an error at line 8

D. Compilation fails with an error at line 9

E. Compilation fails with an error at line 12

F. Compilation fails with an error at line 13

Answer:

☑ **A** is correct, a ClassCastException will be thrown when the code attempts to downcast a Tree to a Redwood.

☒ **B, C, D, E,** and **F** are incorrect based on the above information. (Objective 5.2)

10. Given:

```
3. public class Tenor extends Singer {
4.   public static String sing() { return "fa"; }
5.   public static void main(String[] args) {
6.     Tenor t = new Tenor();
7.     Singer s = new Tenor();
8.     System.out.println(t.sing() + " " + s.sing());
9.   }
10. }
11. class Singer { public static String sing() { return "la"; } }
```

What is the result?

A. `fa fa`

B. `fa la`

C. `la la`

D. Compilation fails

E. An exception is thrown at runtime

Answer:

☑ **B** is correct. The code is correct, but polymorphism doesn't apply to static methods.

☒ **A, C, D,** and **E** are incorrect based on the above information. (Objective 5.2)

11. Given:

```
3. class Alpha {
4.   static String s = " ";
5.   protected Alpha() { s += "alpha "; }
6. }
7. class SubAlpha extends Alpha {
8.   private SubAlpha() { s += "sub "; }
9. }
10. public class SubSubAlpha extends Alpha {
11.   private SubSubAlpha() { s += "subsub "; }
12.   public static void main(String[] args) {
13.     new SubSubAlpha();
14.     System.out.println(s);
15.   }
16. }
```

What is the result?

A. `subsub`

B. `sub subsub`

C. `alpha subsub`

D. `alpha sub subsub`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **C** is correct. Watch out, `SubSubAlpha` extends `Alpha`! Since the code doesn't attempt to make a `SubAlpha`, the private constructor in `SubAlpha` is okay.

☒ **A, B, D, E,** and **F** are incorrect based on the above information. (Objective 5.3)

12. Given:

```
3. class Building {
4.   Building() {  System.out.print("b ");  }
5.   Building(String name) {
6.     this();   System.out.print("bn " + name);
7.   }
8. }
9. public class House extends Building {
```

```
10.   House() {  System.out.print("h ");  }
11.   House(String name) {
12.     this();   System.out.print("hn " + name);
13.   }
14.   public static void main(String[] args) { new House("x "); }
15. }
```

What is the result?

A. `h hn x`

B. `hn x h`

C. `b h hn x`

D. `b hn x h`

E. `bn x h hn x`

F. `b bn x h hn x`

G. `bn x b h hn x`

H. Compilation fails

Answer:

☑ **C** is correct. Remember that constructors call their superclass constructors, which execute first, and that constructors can be overloaded.

☒ **A, B, D, E, F, G,** and **H** are incorrect based on the above information. (Objectives 1.6, 5.4)

13. Given:

```
3. class Mammal {
4.   String name = "furry ";
5.   String makeNoise() { return "generic noise"; }
6. }
7. class Zebra extends Mammal {
8.   String name = "stripes ";
9.   String makeNoise() { return "bray"; }
10. }
11. public class ZooKeeper {
12.   public static void main(String[] args) { new ZooKeeper().go(); }
13.   void go() {
14.     Mammal m = new Zebra();
15.     System.out.println(m.name + m.makeNoise());
16.   }
17. }
```

What is the result?

A. `furry bray`

B. `stripes bray`

C. `furry generic noise`

D. `stripes generic noise`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **A** is correct. Polymorphism is only for instance methods.

☒ **B, C, D, E,** and **F** are incorrect based on the above information. (Objectives 1.5, 5.4)

14. You're designing a new online board game in which Floozels are a type of Jammers, Jammers can have Quizels, Quizels are a type of Klakker, and Floozels can have several Floozets. Which of the following fragments represent this design? (Choose all that apply.)

A.
```
import java.util.*;
interface Klakker { }
class Jammer { Set<Quizel> q; }
class Quizel implements Klakker { }
public class Floozel extends Jammer { List<Floozet> f; }
interface Floozet { }
```

B.
```
import java.util.*;
class Klakker { Set<Quizel> q; }
class Quizel extends Klakker { }
class Jammer { List<Floozel> f; }
class Floozet extends Floozel { }
public class Floozel { Set<Klakker> k; }
```

C.
```
import java.util.*;
class Floozet { }
class Quizel implements Klakker { }
class Jammer { List<Quizel> q; }
interface Klakker { }
class Floozel extends Jammer { List<Floozet> f; }
```

D.
```
import java.util.*;
interface Jammer extends Quizel { }
interface Klakker { }
interface Quizel extends Klakker { }
interface Floozel extends Jammer, Floozet { }
interface Floozet { }
```

Answer:

☑ **A** and **C** are correct. The phrase "type of" indicates an "is-a" relationship (extends or implements), and the phrase "have" is of course a "has-a" relationship (usually instance variables).

☒ **B** and **D** are incorrect based on the above information.
(Objective 5.5)

15. Given:

```
3. class A { }
4. class B extends A { }
5. public class ComingThru {
6.   static String s = "-";
7.   public static void main(String[] args) {
8.     A[] aa = new A[2];
9.     B[] ba = new B[2];
10.    sifter(aa);
11.    sifter(ba);
12.    sifter(7);
13.    System.out.println(s);
14.  }
15.  static void sifter(A[]... a2)    { s += "1"; }
16.  static void sifter(B[]... b1)    { s += "2"; }
17.  static void sifter(B[] b1)       { s += "3"; }
18.  static void sifter(Object o)     { s += "4"; }
19. }
```

What is the result?

A. `-124`

B. `-134`

C. `-424`

D. `-434`

E. `-444`

F. Compilation fails

Answer:

☑ **D** is correct. In general, overloaded var-args methods are chosen last. Remember that arrays are objects. Finally, an int can be boxed to an Integer and then "widened" to an Object.

☒ **A, B, C, E,** and **F** are incorrect based on the above information.
(Objective 1.5)

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Stack and Heap

- ❑ Local variables (method variables) live on the stack.
- ❑ Objects and their instance variables live on the heap.

Literals and Primitive Casting (Objective 1.3)

- ❑ Integer literals can be decimal, octal (e.g. `013`), or hexadecimal (e.g. `0x3d`).
- ❑ Literals for `long`s end in `L` or `l`.
- ❑ Float literals end in `F` or `f`, `double` literals end in a digit or `D` or `d`.
- ❑ The `boolean` literals are `true` and `false`.
- ❑ Literals for `char`s are a single character inside single quotes: `'d'`.

Scope (Objectives 1.3 and 7.6)

- ❑ Scope refers to the lifetime of a variable.
- ❑ There are four basic scopes:
  - ❑ Static variables live basically as long as their class lives.
  - ❑ Instance variables live as long as their object lives.
  - ❑ Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
  - ❑ Block variables (e.g., in a `for` or an `if`) live until the block completes.

Basic Assignments (Objectives 1.3 and 7.6)

- ❑ Literal integers are implicitly ints.
- ❑ Integer expressions always result in an `int`-sized result, never smaller.
- ❑ Floating-point numbers are implicitly doubles (64 bits).
- ❑ Narrowing a primitive truncates the *high order* bits.
- ❑ Compound assignments (e.g. `+=`), perform an automatic cast.
- ❑ A reference variable holds the bits that are used to refer to an object.
- ❑ Reference variables can refer to subclasses of the declared type but not to superclasses.

- When creating a new object, e.g., `Button b = new Button();`, three things happen:
  - Make a reference variable named `b`, of type Button
  - Create a new Button object
  - Assign the Button object to the reference variable `b`

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objectives 1.3 and 7.6)

- When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of null.
- When an array of primitives is instantiated, elements get default values.
- Instance variables are always initialized with a default value.
- Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Passing Variables into Methods (Objective 7.3)

- Methods can take primitives and/or object references as arguments.
- Method arguments are always copies.
- Method arguments are never actual objects (they can be references to objects).
- A primitive argument is an unattached copy of the original primitive.
- A reference argument is another copy of a reference to the original object.
- Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs, and hard-to-answer exam questions.

Array Declaration, Construction, and Initialization (Obj. 1.3)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be left or right of the name.
- It is never legal to include the size of an array in the declaration.
- You must include the size of an array when you construct it (using `new`) unless you are creating an anonymous array.
- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.

- ❑ Arrays are indexed beginning with zero.
- ❑ An ArrayIndexOutOfBoundsException occurs if you use a bad index value.
- ❑ Arrays have a `length` variable whose value is the number of array elements.
- ❑ The last index you can access is always one less than the length of the array.
- ❑ Multidimensional arrays are just arrays of arrays.
- ❑ The dimensions in a multidimensional array can have different lengths.
- ❑ An array of primitives can accept any value that can be promoted implicitly to the array's declared type;. e.g., a `byte` variable can go in an `int` array.
- ❑ An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
- ❑ If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
- ❑ You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

Initialization Blocks (Objectives 1.3 and 7.6)

- ❑ Static initialization blocks run once, when the class is first loaded.
- ❑ Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.
- ❑ If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

Using Wrappers (Objective 3.1)

- ❑ The wrapper classes correlate to the primitive types.
- ❑ Wrappers have two main functions:
  - ❑ To wrap primitives so that they can be handled like objects
  - ❑ To provide utility methods for primitives (usually conversions)
- ❑ The three most important method families are
  - ❑ `xxxValue()` Takes no arguments, returns a primitive
  - ❑ `parseXxx()` Takes a String, returns a primitive, throws NFE
  - ❑ `valueOf()` Takes a String, returns a wrapped object, throws NFE

- ❑ Wrapper constructors can take a String or a primitive, except for Character, which can only take a `char`.
- ❑ Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.

Boxing (Objective 3.1)

- ❑ As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.
- ❑ Using == with wrappers created through boxing is tricky; those with the same small values (typically lower than 127), will be ==, larger values will not be ==.

Advanced Overloading (Objectives 1.5 and 5.4)

- ❑ Primitive widening uses the "smallest" method argument possible.
- ❑ Used individually, boxing and var-args are compatible with overloading.
- ❑ You CANNOT widen from one wrapper type to another. (IS-A fails.)
- ❑ You CANNOT widen and then box. (An `int` can't become a Long.)
- ❑ You can box and then widen. (An `int` can become an Object, via an Integer.)
- ❑ You can combine var-args with either widening or boxing.

Garbage Collection (Objective 7.4)

- ❑ In Java, garbage collection (GC) provides automated memory management.
- ❑ The purpose of GC is to delete objects that can't be reached.
- ❑ Only the JVM decides when to run the GC, you can only suggest it.
- ❑ You can't know the GC algorithm for sure.
- ❑ Objects must be considered eligible before they can be garbage collected.
- ❑ An object is eligible when no live thread can reach it.
- ❑ To reach an object, you must have a live, reachable reference to that object.
- ❑ Java applications can run out of memory.
- ❑ Islands of objects can be GCed, even though they refer to each other.
- ❑ Request garbage collection with `System.gc();` (only before the SCJP 6).
- ❑ Class Object has a `finalize()` method.
- ❑ The `finalize()` method is guaranteed to run once and only once before the garbage collector deletes an object.
- ❑ The garbage collector makes no guarantees, `finalize()` may never run.
- ❑ You can uneligibilize an object for GC from within `finalize()`.

SELF TEST

1. Given:

```
class CardBoard {
  Short story = 200;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

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

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

2. Given:

```
class Alien {
  String invade(short ships) { return "a few"; }
  String invade(short... ships) { return "many"; }
}
class Defender {
  public static void main(String [] args) {
    System.out.println(new Alien().invade(7));
  }
}
```

What is the result?

A. `many`
B. `a few`
C. Compilation fails
D. The output is not predictable
E. An exception is thrown at runtime

3. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2,}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result?

A. 2

B. 4

C. An exception is thrown at runtime

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 5

F. Compilation fails due to an error on line 6

G. Compilation fails due to an error on line 7

4. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`

B. `hi hi`

C. `hi hi hi`

D. Compilation fails
E. `hi`, followed by an exception
F. `hi hi`, followed by an exception

5. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`
B. `false true`
C. `true false`
D. `false false`
E. Compilation fails
F. An exception is thrown at runtime

6. Given:

```
class Bird {
  { System.out.print("b1 "); }
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`
B. `pre b2 b1 r2 r3 hawk`
C. `pre b2 b1 r2 r3 hawk r1 r4`
D. `r1 r4 pre b1 b2 r3 r2 hawk`
E. `r1 r4 pre b2 b1 r2 r3 hawk`
F. `pre r1 r4 b1 b2 r3 r2 hawk`
G. `pre r1 r4 b2 b1 r2 r3 hawk`
H. The order of output cannot be predicted
I. Compilation fails

7. Given:

```
3. public class Bridge {
4.   public enum Suits {
5.     CLUBS(20), DIAMONDS(20), HEARTS(30), SPADES(30),
6.     NOTRUMP(40) { public int getValue(int bid) {
                       return ((bid-1)*30)+40; } };
7.     Suits(int points) { this.points = points;  }
8.     private int points;
9.     public int getValue(int bid) { return points * bid; }
10.  }
11.  public static void main(String[] args) {
12.    System.out.println(Suits.NOTRUMP.getValue(3));
13.    System.out.println(Suits.SPADES + " " + Suits.SPADES.points);
14.    System.out.println(Suits.values());
15.  }
16. }
```

Which are true? (Choose all that apply.)

A. The output could contain `30`
B. The output could contain `@bf73fa`
C. The output could contain `DIAMONDS`
D. Compilation fails due to an error on line 6
E. Compilation fails due to an error on line 7
F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

H. Compilation fails due to an error within lines 12 to 14

8. Given:

```
3. public class Ouch {
4.   static int ouch = 7;
5.   public static void main(String[] args) {
6.     new Ouch().go(ouch);
7.     System.out.print(" " + ouch);
8.   }
9.   void go(int ouch) {
10.    ouch++;
11.    for(int ouch = 3; ouch < 6; ouch++)
12.      ;
13.    System.out.print(" " + ouch);
14.  }
15. }
```

What is the result?

A. `5 7`

B. `5 8`

C. `8 7`

D. `8 8`

E. Compilation fails

F. An exception is thrown at runtime

9. Given:

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

What is the result?

A. `212`

B. `232`

C. `234`

D. `312`

E. `332`

F. `334`

G. Compilation fails

10. Given:

```
3. class Dozens {
4.   int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.   public static void main(String[] args) {
8.     Dozens [] da = new Dozens[3];
9.     da[0] = new Dozens();
10.    Dozens d = new Dozens();
11.    da[1] = d;
12.    d = null;
13.    da[1] = null;
14.    // do stuff
15.  }
16. }
```

Which two are true about the objects created within `main()`, and eligible for garbage collection when line 14 is reached?

A. Three objects were created

B. Four objects were created

C. Five objects were created

D. Zero objects are eligible for GC

E. One object is eligible for GC

F. Two objects are eligible for GC

G. Three objects are eligible for GC

11. Given:

```
3. class Beta { }
4. class Alpha {
5.   static Beta b1;
6.   Beta b2;
7. }
8. public class Tester {
9.   public static void main(String[] args) {
10.    Beta b1 = new Beta();     Beta b2 = new Beta();
11.    Alpha a1 = new Alpha();   Alpha a2 = new Alpha();
12.    a1.b1 = b1;
13.    a1.b2 = b1;
14.    a2.b2 = b2;
15.    a1 = null;  b1 = null;  b2 = null;
16.    // do stuff
17.  }
18. }
```

When line 16 is reached, how many objects will be eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

12. Given:

```
3. class Box {
4.   int size;
5.   Box(int s) { size = s; }
6. }
7. public class Laser {
8.   public static void main(String[] args) {
9.     Box b1 = new Box(5);
10.    Box[] ba = go(b1, new Box(6));
11.    ba[0] = b1;
12.    for(Box b : ba) System.out.print(b.size + " ");
13.  }
14.  static Box[] go(Box b1, Box b2) {
15.     b1.size = 4;
16.     Box[] ma = {b2, b1};
17.     return ma;
18.  }
19. }
```

What is the result?

A. `4 4`

B. `5 4`

C. `6 4`

D. `4 5`

E. `5 5`

F. Compilation fails

13. Given:

```
3. public class Dark {
4.   int x = 3;
5.   public static void main(String[] args) {
6.     new Dark().go1();
7.   }
8.   void go1() {
9.     int x;
10.    go2(++x);
11.  }
12.  void go2(int y) {
13.    int x = ++y;
14.    System.out.println(x);
15.  }
16. }
```

What is the result?

A. 2

B. 3

C. 4

D. 5

E. Compilation fails

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. Given:

```
class CardBoard {
  Short story = 200;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

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

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

Answer:

☑ **C** is correct. Only one `CardBoard object (c1)` is eligible, but it has an associated Short wrapper object that is also eligible.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

2. Given:

```
class Alien {
  String invade(short ships) { return "a few"; }
  String invade(short... ships) { return "many"; }
}
class Defender {
  public static void main(String [] args) {
    System.out.println(new Alien().invade(7));
} }
```

What is the result?

A. `many`
B. `a few`
C. Compilation fails
D. The output is not predictable
E. An exception is thrown at runtime

Answer:

☑ **C** is correct, compilation fails. The var-args declaration is fine, but `invade` takes a `short`, so the argument 7 needs to be cast to a `short`. With the cast, the answer is **B**, '`a few`'.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 1.3)

3. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2,}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result?

A. `2`
B. `4`
C. An exception is thrown at runtime
D. Compilation fails due to an error on line 4
E. Compilation fails due to an error on line 5
F. Compilation fails due to an error on line 6
G. Compilation fails due to an error on line 7

Answer:

☑ **C** is correct. A `ClassCastException` is thrown at line 7 because `o1` refers to an `int[][]` not an `int[]`. If line 7 was removed, the output would be 4.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 1.3)

4. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`
B. `hi hi`
C. `hi hi hi`
D. Compilation fails
E. `hi`, followed by an exception
F. `hi hi`, followed by an exception

Answer:

☑ **F** is correct. The `m2` object's `m1` instance variable is never initialized, so when `m5` tries to use it a `NullPointerException` is thrown.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 7.3)

5. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`
B. `false true`
C. `true false`
D. `false false`
E. Compilation fails
F. An exception is thrown at runtime

Answer:

☑ **A** is correct. The references `f1`, `z`, and `f3` all refer to the same instance of `Fizz`. The `final` modifier assures that a reference variable cannot be referred to a different object, but `final` doesn't keep the object's state from changing.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 7.3)

6. Given:

```
class Bird {
  { System.out.print("b1 "); }
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`
B. `pre b2 b1 r2 r3 hawk`
C. `pre b2 b1 r2 r3 hawk r1 r4`
D. `r1 r4 pre b1 b2 r3 r2 hawk`
E. `r1 r4 pre b2 b1 r2 r3 hawk`

F. `pre r1 r4 b1 b2 r3 r2 hawk`

G. `pre r1 r4 b2 b1 r2 r3 hawk`

H. The order of output cannot be predicted

I. Compilation fails

Answer:

☑ **D** is correct. Static init blocks are executed at class loading time, instance init blocks run right after the call to `super()` in a constructor. When multiple init blocks of a single type occur in a class, they run in order, from the top down.

☒ **A, B, C, E, F, G, H,** and **I** are incorrect based on the above. Note: you'll probably never see this many choices on the real exam! (Objective 1.3)

7. Given:

```
3. public class Bridge {
4.   public enum Suits {
5.     CLUBS(20), DIAMONDS(20), HEARTS(30), SPADES(30),
6.     NOTRUMP(40) { public int getValue(int bid) {
                       return ((bid-1)*30)+40; } };
7.     Suits(int points) { this.points = points;  }
8.     private int points;
9.     public int getValue(int bid) { return points * bid; }
10.  }
11.  public static void main(String[] args) {
12.    System.out.println(Suits.NOTRUMP.getBidValue(3));
13.    System.out.println(Suits.SPADES + " " + Suits.SPADES.points);
14.    System.out.println(Suits.values());
15.  }
16. }
```

Which are true? (Choose all that apply.)

A. The output could contain `30`

B. The output could contain `@bf73fa`

C. The output could contain `DIAMONDS`

D. Compilation fails due to an error on line 6

E. Compilation fails due to an error on line 7

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

H. Compilation fails due to an error within lines 12 to 14

Answer:

☑ **A** and **B** are correct. The code compiles and runs without exception. The `values()` method returns an array reference, not the contents of the enum, so DIAMONDS is never printed.

☒ **C, D, E, F, G,** and **H** are incorrect based on the above. (Objective 1.3)

8. Given:

```
3. public class Ouch {
4.   static int ouch = 7;
5.   public static void main(String[] args) {
6.     new Ouch().go(ouch);
7.     System.out.print(" " + ouch);
8.   }
9.   void go(int ouch) {
10.    ouch++;
11.    for(int ouch = 3; ouch < 6; ouch++)
12.      ;
13.    System.out.print(" " + ouch);
14.  }
15. }
```

What is the result?

A. `5 7`

B. `5 8`

C. `8 7`

D. `8 8`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **E** is correct. The parameter declared on line 9 is valid (although ugly), but the variable name `ouch` cannot be declared again on line 11 in the same scope as the declaration on line 9.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 1.3)

9. Given:

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

What is the result?

A. `212`

B. `232`

C. `234`

D. `312`

E. `332`

F. `334`

G. Compilation fails

Answer:

☑ **A** is correct. It's legal to autobox and then widen. The first call to `doStuff()` boxes the int to an Integer then passes two objects. The second call cannot widen and then box (making the `Long` method unusable), so it boxes the int to an Integer. As always, a var-args method will be chosen only if no non-var-arg method is possible. The third call is passing two objects—they are of type 'short array.'

☒ **B, C, D, E, F,** and **G** are incorrect based on the above. (Objective 3.1)

10. Given:

```
3. class Dozens {
4.   int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.   public static void main(String[] args) {
```

```
 8.      Dozens [] da = new Dozens[3];
 9.      da[0] = new Dozens();
10.      Dozens d = new Dozens();
11.      da[1] = d;
12.      d = null;
13.      da[1] = null;
14.      // do stuff
15.    }
16. }
```

Which two are true about the objects created within `main()`, and eligible for garbage collection when line 14 is reached?

A. Three objects were created

B. Four objects were created

C. Five objects were created

D. Zero objects are eligible for GC

E. One object is eligible for GC

F. Two objects are eligible for GC

G. Three objects are eligible for GC

Answer:

☑ **C** and **F** are correct. `da` refers to an object of type "Dozens array," and each Dozens object that is created comes with its own "int array" object. When line 14 is reached, only the second Dozens object (and its "int array" object) are not reachable.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (Objective 7.4)

11. Given:

```
 3. class Beta { }
 4. class Alpha {
 5.   static Beta b1;
 6.   Beta b2;
 7. }
 8. public class Tester {
 9.   public static void main(String[] args) {
10.     Beta b1 = new Beta();     Beta b2 = new Beta();
11.     Alpha a1 = new Alpha();   Alpha a2 = new Alpha();
12.     a1.b1 = b1;
13.     a1.b2 = b1;
14.     a2.b2 = b2;
15.     a1 = null;  b1 = null;  b2 = null;
```

```
16.      // do stuff
17.    }
18. }
```

When line 16 is reached, how many objects will be eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

Answer:

☑ **B** is correct. It should be clear that there is still a reference to the object referred to by `a2`, and that there is still a reference to the object referred to by `a2.b2`. What might be less clear is that you can still access the other Beta object through the static variable `a2.b1`—because it's static.

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

12. Given:

```
3. class Box {
4.   int size;
5.   Box(int s) { size = s; }
6. }
7. public class Laser {
8.   public static void main(String[] args) {
9.     Box b1 = new Box(5);
10.    Box[] ba = go(b1, new Box(6));
11.    ba[0] = b1;
12.    for(Box b : ba) System.out.print(b.size + " ");
13.  }
14.  static Box[] go(Box b1, Box b2) {
15.    b1.size = 4;
16.    Box[] ma = {b2, b1};
17.    return ma;
18.  }
19. }
```

What is the result?

A. `4 4`

B. `5 4`

C. `6 4`

D. `4 5`

E. `5 5`

F. Compilation fails

Answer:

☑ **A** is correct. Although `main()`'s `b1` is a different reference variable than `go()`'s `b1`, they refer to the same Box object.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 7.3)

13. Given:

```
3. public class Dark {
4.   int x = 3;
5.   public static void main(String[] args) {
6.     new Dark().go1();
7.   }
8.   void go1() {
9.     int x;
10.    go2(++x);
11.  }
12.  void go2(int y) {
13.    int x = ++y;
14.    System.out.println(x);
15.  }
16. }
```

What is the result?

A. 2

B. 3

C. 4

D. 5

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **E** is correct. In `go1()` the local variable `x` is not initialized.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 1.3)

TWO-MINUTE DRILL

Here are some of the key points from each section in this chapter.

Relational Operators (Objective 7.6)

- Relational operators always result in a `boolean` value (`true` or `false`).
- There are six relational operators: >, >=, <, <=, ==, and !=. The last two (== and !=) are sometimes referred to as *equality operators*.
- When comparing characters, Java uses the Unicode value of the character as the numerical value.
- Equality operators
  - There are two equality operators: == and !=.
  - Four types of things can be tested: numbers, characters, booleans, and reference variables.
- When comparing reference variables, == returns `true` only if both references refer to the same object.

instanceof Operator (Objective 7.6)

- `instanceof` is for reference variables only, and checks for whether the object is of a particular type.
- The `instanceof` operator can be used only to test objects (or `null`) against class types that are in the same class hierarchy.
- For interfaces, an object passes the `instanceof` test if any of its superclasses implement the interface on the right side of the `instanceof` operator.

Arithmetic Operators (Objective 7.6)

- There are four primary math operators: add, subtract, multiply, and divide.
- The remainder operator (%), returns the remainder of a division.
- Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
- The *, /, and % operators have higher precedence than + and -.

String Concatenation Operator (Objective 7.6)

- If either operand is a `String`, the + operator concatenates the operands.
- If both operands are numeric, the + operator adds the operands.

Increment/Decrement Operators (Objective 7.6)

- Prefix operators (`++` and `--`) run before the value is used in the expression.
- Postfix operators (`++` and `--`) run after the value is used in the expression.
- In any expression, both operands are fully evaluated *before* the operator is applied.
- Variables marked `final` cannot be incremented or decremented.

Ternary (Conditional Operator) (Objective 7.6)

- Returns one of two values based on whether a `boolean` expression is `true` or `false`.
  - Returns the value after the `?` if the expression is `true`.
  - Returns the value after the `:` if the expression is `false`.

Logical Operators (Objective 7.6)

- The exam covers six "logical" operators: &, |, ^, !, &&, and ||.
- Logical operators work with two expressions (except for !) that must resolve to `boolean` values.
- The && and & operators return `true` only if both operands are `true`.
- The || and | operators return `true` if either or both operands are `true`.
- The && and || operators are known as short-circuit operators.
- The && operator does not evaluate the right operand if the left operand is `false`.
- The || does not evaluate the right operand if the left operand is `true`.
- The & and | operators always evaluate both operands.
- The ^ operator (called the "logical XOR"), returns `true` if exactly one operand is `true`.
- The ! operator (called the "inversion" operator), returns the opposite value of the `boolean` operand it precedes.

SELF TEST

1. Given:

```
class Hexy {
  public static void main(String[] args) {
    Integer i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`
B. `life`
C. `universe`
D. `everything`
E. Compilation fails
F. An exception is thrown at runtime

2. Given:

```
1. class Comp2 {
2.   public static void main(String[] args) {
3.     float f1 = 2.3f;
4.     float[][] f2 = {{42.0f}, {1.7f, 2.3f}, {2.6f, 2.7f}};
5.     float[] f3 = {2.7f};
6.     Long x = 42L;
7.     // insert code here
8.       System.out.println("true");
9.   }
10. }
```

And the following five code fragments:

```
F1.  if(f1 == f2)
F2.  if(f1 == f2[2][1])
F3.  if(x == f2[0][0])
F4.  if(f1 == f2[1,1])
F5.  if(f3 == f2[2])
```

What is true?

A. One of them will compile, only one will be `true`
B. Two of them will compile, only one will be `true`
C. Two of them will compile, two will be `true`
D. Three of them will compile, only one will be `true`
E. Three of them will compile, exactly two will be `true`
F. Three of them will compile, exactly three will be `true`

3. Given:

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`
B. `production live2`
C. `test case live2`
D. Compilation fails
E. An exception is thrown at runtime

4. Given:

```
class Feline {
  public static void main(String[] args) {
    Long x = 42L;
    Long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`

B. `9 foo47 4244foo`

C. `9 foo425 86foo`

D. `9 foo425 4244foo`

E. `72 foo47 86foo`

F. `72 foo47 4244foo`

G. `72 foo425 86foo`

H. `72 foo425 4244foo`

I. Compilation fails

5. Place the fragments into the code to produce the output 33. Note, you must use each fragment exactly once.

CODE:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x   ___  ___;
    ___ ___  ___;
    ___ ___  ___;
    ___ ___  ___;

    System.out.println(x);
  }
}
```

FRAGMENTS:

y	y	y	y
y	x	x	
-=	*=	*=	*=

6. Given:

```
3. public class Twisty {
4.   { index = 1; }
5.   int index;
6.   public static void main(String[] args) {
7.     new Twisty().go();
8.   }
9.   void go() {
10.    int [][] dd = {{9,8,7}, {6,5,4}, {3,2,1,0}};
11.    System.out.println(dd[index++][index++]);
12.  }
13. }
```

What is the result? (Choose all that apply.)

A. 1

B. 2

C. 4

D. 6

E. 8

F. Compilation fails

G. An exception is thrown at runtime

7. Given:

```
3. public class McGee {
4.   public static void main(String[] args) {
5.     Days d1 = Days.TH;
6.     Days d2 = Days.M;
7.     for(Days d: Days.values()) {
8.       if(d.equals(Days.F)) break;
9.       d2 = d;
10.    }
11.    System.out.println((d1 == d2)?"same old" : "newly new");
12.  }
13.  enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors
D. Compilation fails due only to an error on line 7
E. Compilation fails due only to an error on line 8
F. Compilation fails due only to an error on line 11
G. Compilation fails due only to an error on line 13

8. Given:

```
4. public class SpecialOps {
5.   public static void main(String[] args) {
6.     String s = "";
7.     Boolean b1 = true;
8.     boolean b2 = false;
9.     if((b2 = false) | (21%5) > 2) s += "x";
10.    if(b1 || (b2 == true))        s += "y";
11.    if(b2 == true)                s += "z";
12.    System.out.println(s);
13.  }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. x will be included in the output
C. y will be included in the output
D. z will be included in the output
E. An exception is thrown at runtime

9. Given:

```
3. public class Spock {
4.   public static void main(String[] args) {
5.     int mask = 0;
6.     int count = 0;
7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )    mask = mask + 1;
8.     if(  (6 > 8) ^ false)                            mask = mask + 10;
9.     if( !(mask > 1) && ++count > 1)                  mask = mask + 100;
10.    System.out.println(mask + " " + count);
11.  }
12. }
```

Which two are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0
B. `mask` is 1
C. `mask` is 2
D. `mask` is 10
E. `mask` is greater than 10
F. `count` is 0
G. `count` is greater than 0

10. Given:

```
3. interface Vessel { }
4. interface Toy { }
5. class Boat implements Vessel { }
6. class Speedboat extends Boat implements Toy { }
7. public class Tree {
8.   public static void main(String[] args) {
9.     String s = "0";
10.    Boat b = new Boat();
11.    Boat b2 = new Speedboat();
12.    Speedboat s2 = new Speedboat();
13.    if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.    if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.    System.out.println(s);
16.  }
17. }
```

What is the result?

A. `0`
B. `01`
C. `02`
D. `012`
E. Compilation fails
F. An exception is thrown at runtime

SELF TEST ANSWERS

1. Given:

```
class Hexy {
  public static void main(String[] args) {
    Integer i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`
B. `life`
C. `universe`
D. `everything`
E. Compilation fails
F. An exception is thrown at runtime

Answer:

☑ **D** is correct. This is a ternary nested in a ternary with a little unboxing thrown in. Both of the ternary expressions are `false`.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 7.6)

2. Given:

```
1. class Comp2 {
2.   public static void main(String[] args) {
3.     float f1 = 2.3f;
4.     float[][] f2 = {{42.0f}, {1.7f, 2.3f}, {2.6f, 2.7f}};
5.     float[] f3 = {2.7f};
6.     Long x = 42L;
7.     // insert code here
8.       System.out.println("true");
9.   }
10. }
```

And the following five code fragments:

```
F1.  if(f1 == f2)
F2.  if(f1 == f2[2][1])
F3.  if(x == f2[0][0])
F4.  if(f1 == f2[1,1])
F5.  if(f3 == f2[2])
```

What is true?

A. One of them will compile, only one will be `true`
B. Two of them will compile, only one will be `true`
C. Two of them will compile, two will be `true`
D. Three of them will compile, only one will be `true`
E. Three of them will compile, exactly two will be `true`
F. Three of them will compile, exactly three will be `true`

Answer:

☑ **D** is correct. Fragments `F2`, `F3`, and `F5` will compile, and only `F3` is `true`.
☒ **A, B, C, E,** and **F** are incorrect. `F1` is incorrect because you can't compare a primitive to an array. `F4` is incorrect syntax to access an element of a two-dimensional array. (Objective 7.6)

3. Given:

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`
B. `production live2`

C. `test case live2`
D. Compilation fails
E. An exception is thrown at runtime

Answer:

☑ **E** is correct. Because the short circuit `(||)` is not used, both operands are evaluated. Since `args[1]` is past the args array bounds, an `ArrayIndexOutOfBoundsException` is thrown.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 7.6)

4. Given:

```
class Feline {
  public static void main(String[] args) {
    Long x = 42L;
    Long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`
B. `9 foo47 4244foo`
C. `9 foo425 86foo`
D. `9 foo425 4244foo`
E. `72 foo47 86foo`
F. `72 foo47 4244foo`
G. `72 foo425 86foo`
H. `72 foo425 4244foo`
I. Compilation fails

Answer:

☑ **G** is correct. Concatenation runs from left to right, and if either operand is a `String`, the operands are concatenated. If both operands are numbers they are added together. Unboxing works in conjunction with concatenation.

☒ **A, B, C, D, E, F, H,** and **I** are incorrect based on the above. (Objective 7.6)

5. Place the fragments into the code to produce the output 33. Note, you must use each fragment exactly once.

CODE:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x    ___  ___;
    ___  ___  ___;
    ___  ___  ___;
    ___  ___  ___;

    System.out.println(x);
  }
}
```

FRAGMENTS:

y	y	y	y
y	x	x	
-=	*=	*=	*=

Answer:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x *= x;
    y *= y;
    y *= y;
    x -= y;

    System.out.println(x);
  }
}
```

Yeah, we know it's kind of puzzle-y, but you might encounter something like it on the real exam. (Objective 7.6)

6. Given:

```
3. public class Twisty {
4.   { index = 1; }
5.   int index;
6.   public static void main(String[] args) {
7.     new Twisty().go();
8.   }
9.   void go() {
10.    int [][] dd = {{9,8,7}, {6,5,4}, {3,2,1,0}};
11.    System.out.println(dd[index++][index++]);
12.  }
13. }
```

What is the result? (Choose all that apply.)

A. 1

B. 2

C. 4

D. 6

E. 8

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **C** is correct. Multidimensional arrays' dimensions can be inconsistent, the code uses an initialization block, and the increment operators are both post-increment operators.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 1.3)

7. Given:

```
3. public class McGee {
4.   public static void main(String[] args) {
5.     Days d1 = Days.TH;
6.     Days d2 = Days.M;
7.     for(Days d: Days.values()) {
8.       if(d.equals(Days.F)) break;
9.       d2 = d;
10.    }
11.    System.out.println((d1 == d2)?"same old" : "newly new");
```

```
12.    }
13.    enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors

D. Compilation fails due only to an error on line 7

E. Compilation fails due only to an error on line 8

F. Compilation fails due only to an error on line 11

G. Compilation fails due only to an error on line 13

Answer:

☑ **A** is correct. All of this syntax is correct. The for-each iterates through the enum using the `values()` method to return an array. Enums can be compared using either `equals()` or ==. Enums can be used in a ternary operator's Boolean test.

☒ **B, C, D, E, F,** and **G** are incorrect based on the above. (Objective 7.6)

8. Given:

```
4. public class SpecialOps {
5.   public static void main(String[] args) {
6.     String s = "";
7.     Boolean b1 = true;
8.     Boolean b2 = false;
9.     if((b2 = false) | (21%5) > 2) s += "x";
10.    if(b1 || (b2 = true))         s += "y";
11.    if(b2 == true)                s += "z";
12.    System.out.println(s);
13.  }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. x will be included in the output

C. y will be included in the output

D. `z` will be included in the output

E. An exception is thrown at runtime

Answer:

☑ **C** is correct. First of all, boxing takes care of the Boolean. Line 9 uses the modulus operator, which returns the remainder of the division, which in this case is 1. Also, line 9 sets `b2` to false, and it doesn't test `b2`'s value. Line 10 sets `b2` to true, and it doesn't test its value; however, the short circuit operator keeps the expression `b2 = true` from being executed.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 7.6)

9. Given:

```
3. public class Spock {
4.   public static void main(String[] args) {
5.     int mask = 0;
6.     int count = 0;
7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )    mask = mask + 1;
8.     if(  (6 > 8) ^ false)                            mask = mask + 10;
9.     if( !(mask > 1) && ++count > 1)                  mask = mask + 100;
10.    System.out.println(mask + " " + count);
11.  }
12. }
```

Which two answers are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0

B. `mask` is 1

C. `mask` is 2

D. `mask` is 10

E. `mask` is greater than 10

F. `count` is 0

G. `count` is greater than 0

Answer:

☑ **C** and **F** are correct. At line 7 the `||` keeps `count` from being incremented, but the `|` allows `mask` to be incremented. At line 8 the ^ returns true only if exactly one operand is true. At line 9 `mask` is 2 and the `&&` keeps `count` from being incremented.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (Objective 7.6)

10. Given:

```
 3. interface Vessel { }
 4. interface Toy { }
 5. class Boat implements Vessel { }
 6. class Speedboat extends Boat implements Toy { }
 7. public class Tree {
 8.   public static void main(String[] args) {
 9.     String s = "0";
10.     Boat b = new Boat();
11.     Boat b2 = new Speedboat();
12.     Speedboat s2 = new Speedboat();
13.     if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.     if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.     System.out.println(s);
16.   }
17. }
```

What is the result?

A. `0`

B. `01`

C. `02`

D. `012`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **D** is correct. First, remember that `instanceof` can look up through multiple levels of an inheritance tree. Also remember that `instanceof` is commonly used before attempting a downcast, so in this case, after line 15, it would be possible to say `Speedboat s3 = (Speedboat)b2;`.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 7.6)

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. You might want to loop through them several times.

Writing Code Using if and switch Statements (Obj. 2.1)

- The only legal expression in an `if` statement is a `boolean` expression, in other words an expression that resolves to a `boolean` or a `Boolean` variable.
- Watch out for `boolean` assignments (`=`) that can be mistaken for `boolean` equality (`==`) tests:

```
boolean x = false;
if (x = true) { } // an assignment, so x will always be true!
```

- Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.
- `switch` statements can evaluate only to `enums` or the `byte`, `short`, `int`, and `char` data types. You can't say,

```
long s = 30;
switch(s) { }
```

- The `case` constant must be a literal or `final` variable, or a constant expression, including an `enum`. You cannot have a case that includes a non-final variable, or a range of values.
- If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs.
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.
- The `default` block can be located anywhere in the `switch` block, so if no `case` matches, the `default` block will be entered, and if the `default` does not contain a `break`, then code will continue to execute (fall-through) to the end of the `switch` or until the `break` statement is encountered.

Writing Code Using Loops (Objective 2.2)

- A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
- If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop, or within the `for` loop declaration.
- A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop (in other words, code below the `for` loop won't be able to use the variable).
- You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be separated by a comma.
- An enhanced `for` statement (new as of Java 6), has two parts, the *declaration* and the *expression*. It is used only to loop through arrays or collections.
- With an enhanced `for`, the *expression* is the array or collection through which you want to loop.
- With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
- You cannot use a number (old C-style language construct) or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- The `do` loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue (Objective 2.2)

- An unlabeled `break` statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
- An unlabeled `continue` statement will cause: the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
- If the `break` statement or the `continue` statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6)

- ❑ Exceptions come in two flavors: checked and unchecked.
- ❑ Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- ❑ Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws`, or handle the exception with an appropriate try/catch.
- ❑ Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, or to declare them, but the compiler doesn't care one way or the other.
- ❑ If you use an optional `finally` block, it will always be invoked, regardless of whether an exception in the corresponding `try` is thrown or not, and regardless of whether a thrown exception is caught or not.
- ❑ The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.
- ❑ Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.
- ❑ Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()`, and `main()` is "ducking" the exception by declaring it).
- ❑ You can create your own exceptions, normally by extending `Exception` or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.
- ❑ All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining `catch` clauses that can never be reached.
- ❑ Some exceptions are created by programmers, some by the JVM.

Working with the Assertion Mechanism (Objective 2.3)

- Assertions give you a way to test your assumptions during development and debugging.
- Assertions are typically enabled during testing but disabled during deployment.
- You can use `assert` as a keyword (as of version 1.4) or an identifier, but not both together. To compile older code that uses `assert` as an identifier (for example, a method name), use the `-source 1.3` command-line flag to `javac`.
- Assertions are disabled at runtime by default. To enable them, use a command-line flag `-ea` or `-enableassertions`.
- Selectively disable assertions by using the `-da` or `-disableassertions` flag.
- If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.
- You can enable and disable assertions on a class-by-class basis, using the following syntax:

  ```
  java -ea  -da:MyClass  TestClass
  ```

- You can enable and disable assertions on a package-by-package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).
- Do not use assertions to validate arguments to `public` methods.
- Do not use `assert` expressions that cause side effects. Assertions aren't guaranteed to always run, and you don't want behavior that changes depending on whether assertions are enabled.
- Do use assertions—even in `public` methods—to validate that a particular code block will never be reached. You can use `assert false;` for code that should never be reached, so that an assertion error is thrown immediately if the `assert` statement is executed.

SELF TEST

1. Given two files:

```
1. class One {
2.   public static void main(String[] args) {
3.     int assert = 0;
4.   }
5. }
```

```
1. class Two {
2.   public static void main(String[] args) {
3.     assert(false);
4.   }
5. }
```

And the four command-line invocations:

```
javac -source 1.3 One.java
javac -source 1.4 One.java
javac -source 1.3 Two.java
javac -source 1.4 Two.java
```

What is the result? (Choose all that apply.)

A. Only one compilation will succeed
B. Exactly two compilations will succeed
C. Exactly three compilations will succeed
D. All four compilations will succeed
E. No compiler warnings will be produced
F. At least one compiler warning will be produced

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
```

```
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

What is the result?

A. `-`
B. `-c`
C. `-c2`
D. `-2c`
E. `-c22b`
F. `-2c2b`
G. `-2c2bc`
H. Compilation fails

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`
B. `IllegalStateException`
C. `NumberFormatException`
D. `IllegalArgumentException`
E. `ExceptionInInitializerError`
F. `ArrayIndexOutOfBoundsException`

4. Which are true? (Choose all that apply.)

A. It is appropriate to use assertions to validate arguments to methods marked `public`
B. It is appropriate to catch and handle assertion errors
C. It is NOT appropriate to use assertions to validate command-line arguments
D. It is appropriate to use assertions to generate alerts when you reach code that should not be reachable
E. It is NOT appropriate for assertions to change a program's state

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7.   }
8. }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

7. Given:

```
3. class SubException extends Exception { }
4. class SubSubException extends SubException { }
5.
6. public class CC { void doStuff() throws SubException { } }
7.
8. class CC2 extends CC { void doStuff() throws SubSubException { } }
9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

8. Given:

```
3. public class Ebb {
4.   static int x = 7;
5.   public static void main(String[] args) {
6.     String s = "";
7.     for(int y = 0; y < 3; y++) {
8.       x++;
9.       switch(x) {
10.         case 8: s += "8 ";
11.         case 9: s += "9 ";
12.         case 10: { s+= "10 "; break; }
13.         default: s += "d ";
14.         case 13: s+= "13 ";
```

```
15.        }
16.      }
17.      System.out.println(s);
18.    }
19.    static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

9. Given:

```
3. class Infinity { }
4. public class Beyond extends Infinity {
5.   static Integer i;
6.   public static void main(String[] args) {
7.     int sw = (int)(Math.random() * 3);
8.     switch(sw) {
9.       case 0: {  for(int x = 10; x > 5; x++)
10.                   if(x > 10000000) x = 10;
11.                break; }
12.      case 1: {  int y = 7 * i;  break;  }
13.      case 2: {  Infinity inf = new Beyond();
14.                 Beyond b = (Beyond)inf;  }
15.    }
16.  }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A ClassCastException might be thrown

C. A StackOverflowError might be thrown

D. A NullPointerException might be thrown

E. An IllegalStateException might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

10. Given:

```
3. public class Circles {
4.   public static void main(String[] args) {
5.     int[] ia = {1,3,5,7,9};
6.     for(int x : ia) {
7.       for(int j = 0; j < 3; j++) {
8.         if(x > 4 && x < 8) continue;
9.         System.out.print(" " + x);
10.        if(j == 1) break;
11.        continue;
12.      }
13.      continue;
14.    }
15.  }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

11. Given:

```
3. public class OverAndOver {
4.   static String s = "";
5.   public static void main(String[] args) {
6.     try {
7.       s += "1";
8.       throw new Exception();
9.     } catch (Exception e) { s += "2";
10.    } finally { s += "3"; doStuff(); s += "4";
11.    }
```

```
12.      System.out.println(s);
13.    }
14.    static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

12. Given:

```
 3. public class Wind {
 4.   public static void main(String[] args) {
 5.     foreach:
 6.     for(int j=0; j<5; j++) {
 7.       for(int k=0; k< 3; k++) {
 8.         System.out.print(" " + j);
 9.         if(j==3 && k==1) break foreach;
10.         if(j==0 || j==2) break;
11.       }
12.     }
13.   }
14. }
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

13. Given:

```
3. public class Gotcha {
4.   public static void main(String[] args) {
5.     // insert code here
6.
7.   }
8.   void go() {
9.     go();
10.  }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();
II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I - III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile
B. They will all compile
C. All will complete normally
D. None will complete normally
E. Only one will complete normally
F. Two of them will complete normally

14. Given:

```
3. public class Clumsy {
4.   public static void main(String[] args) {
5.     int j = 7;
6.     assert(++j > 7);
7.     assert(++j > 8): "hi";
8.     assert(j > 10): j=12;
9.     assert(j==12): doStuff();
```

```
10.      assert(j==12): new Clumsy();
11.    }
12.    static void doStuff() { }
13. }
```

Which are true? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 6

C. Compilation fails due to an error on line 7

D. Compilation fails due to an error on line 8

E. Compilation fails due to an error on line 9

F. Compilation fails due to an error on line 10

15. Given:

```
1. public class Frisbee {
2.    // insert code here
3.      int x = 0;
4.      System.out.println(7/x);
5.    }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 4, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a try/catch block around line 6 will cause compilation to fail

16. Given:

```
 2. class MyException extends Exception { }
 3. class Tire {
 4.   void doStuff() {  }
 5. }
 6. public class Retread extends Tire {
 7.   public static void main(String[] args) {
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I - IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile
B. They will all compile
C. Some, but not all, will compile
D. All of those that compile will throw an exception at runtime
E. None of those that compile will throw an exception at runtime
F. Only some of those that compile will throw an exception at runtime

SELF TEST ANSWERS

1. Given two files:

```
1. class One {
2.   public static void main(String[] args) {
3.     int assert = 0;
4.   }
5. }
1. class Two {
2.   public static void main(String[] args) {
3.     assert(false);
4.   }
5. }
```

And the four command-line invocations:

```
javac -source 1.3 One.java
javac -source 1.4 One.java
javac -source 1.3 Two.java
javac -source 1.4 Two.java
```

What is the result? (Choose all that apply.)

A. Only one compilation will succeed
B. Exactly two compilations will succeed
C. Exactly three compilations will succeed
D. All four compilations will succeed
E. No compiler warnings will be produced
F. At least one compiler warning will be produced

Answer:

☑ **B** and **F** are correct. Class One will compile (and issue a warning) using the `1.3` flag, and class Two will compile using the `1.4` flag.

☒ **A, C, D,** and **E** are incorrect based on the above. (Objective 2.3)

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
```

```
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
} }
```

What is the result?

A. `-`
B. `-c`
C. `-c2`
D. `-2c`
E. `-c22b`
F. `-2c2b`
G. `-2c2bc`
H. Compilation fails

Answer:

☑ **B** is correct. Once `s3()` throws the exception to `s2()`, `s2()` throws it to `s1()`, and no more of `s2()`'s code will be executed.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (Objective 2.5)

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`
B. `IllegalStateException`
C. `NumberFormatException`
D. `IllegalArgumentException`

E. `ExceptionInInitializerError`

F. `ArrayIndexOutOfBoundsException`

Answer:

☑ **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (i.e., a broader exception).

☒ **A, B, E,** and **F** are not in `NumberFormatException`'s class hierarchy. (Objective 2.6)

4. Which are true? (Choose all that apply.)

A. It is appropriate to use assertions to validate arguments to methods marked `public`

B. It is appropriate to catch and handle assertion errors

C. It is NOT appropriate to use assertions to validate command-line arguments

D. It is appropriate to use assertions to generate alerts when you reach code that should not be reachable

E. It is NOT appropriate for assertions to change a program's state

Answer:

☑ **C, D,** and **E** are correct statements.

☒ **A** is incorrect. It *is* acceptable to use assertions to test the arguments of `private` methods. **B** is incorrect. While assertion errors can be caught, Sun discourages you from doing so. (Objective 2.3)

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7. } }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

Answer:

☑ **A, D,** and **F** are correct. **A** is an example of the enhanced `for` loop. **D** and **F** are examples of the basic `for` loop.

☒ **B** is incorrect because its operands are swapped. **C** is incorrect because the enhanced `for` must declare its first operand. **E** is incorrect syntax to declare two variables in a `for` statement. (Objective 2.2)

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

Answer:

☑ **E** is correct. There is no problem nesting `try / catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, then the code in the `finally` block runs.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 2.5)

7. Given:

```
3. class SubException extends Exception { }
4. class SubSubException extends SubException { }
5.
6. public class CC { void doStuff() throws SubException { } }
7.
8. class CC2 extends CC { void doStuff() throws SubSubException { } }
9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

Answer:

☑ **C** is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objectives 1.5, 2.4)

8. Given:

```
3. public class Ebb {
4.   static int x = 7;
5.   public static void main(String[] args) {
6.     String s = "";
```

```
 7.      for(int y = 0; y < 3; y++) {
 8.        x++;
 9.        switch(x) {
10.          case 8: s += "8 ";
11.          case 9: s += "9 ";
12.          case 10: { s+= "10 "; break; }
13.          default: s += "d ";
14.          case 13: s+= "13 ";
15.        }
16.      }
17.      System.out.println(s);
18.    }
19.    static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

Answer:

☑ **D** is correct. Did you catch the static initializer block? Remember that switches work on "fall-thru" logic, and that fall-thru logic also applies to the default case, which is used when no other case matches.

☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (Objective 2.1)

9. Given:

```
3. class Infinity { }
4. public class Beyond extends Infinity {
5.   static Integer i;
6.   public static void main(String[] args) {
7.     int sw = (int)(Math.random() * 3);
8.     switch(sw) {
9.       case 0: {  for(int x = 10; x > 5; x++)
```

```
10.                   if(x > 10000000) x = 10;
11.                 break; }
12.       case 1: {  int y = 7 * i;  break;  }
13.       case 2: {  Infinity inf = new Beyond();
14.                  Beyond b = (Beyond)inf;  }
15.     }
16.   }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A ClassCastException might be thrown

C. A StackOverflowError might be thrown

D. A NullPointerException might be thrown

E. An IllegalStateException might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

Answer:

☑ **D** and **F** are correct. Because `i` was not initialized, case 1 will throw an NPE. Case 0 will initiate an endless loop, not a stack overflow. Case 2's downcast will *not* cause an exception.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (Objective 2.6)

10. Given:

```
3. public class Circles {
4.   public static void main(String[] args) {
5.     int[] ia = {1,3,5,7,9};
6.     for(int x : ia) {
7.       for(int j = 0; j < 3; j++) {
8.         if(x > 4 && x < 8) continue;
9.         System.out.print(" " + x);
10.        if(j == 1) break;
11.        continue;
12.      }
13.      continue;
14.    }
15.  }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

Answer:

☑ **D** is correct. The basic rule for unlabeled continue statements is that the current iteration stops early and execution jumps to the next iteration. The last two continue statements are redundant!

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 2.2)

11. Given:

```
3. public class OverAndOver {
4.   static String s = "";
5.   public static void main(String[] args) {
6.     try {
7.       s += "1";
8.       throw new Exception();
9.     } catch (Exception e) { s += "2";
10.    } finally { s += "3"; doStuff(); s += "4";
11.    }
12.    System.out.println(s);
13.  }
14.  static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

Answer:

☑ **H** is correct. It's true that the value of `String s` is `123` at the time that the divide-by-zero exception is thrown, but `finally()` is *not* guaranteed to complete, and in this case `finally()` never completes, so the System.out.println (S.O.P.) never executes.

☒ **A, B, C, D, E, F,** and **G** are incorrect based on the above. (Objective 2.5)

12. Given:

```
3. public class Wind {
4.   public static void main(String[] args) {
5.     foreach:
6.     for(int j=0; j<5; j++) {
7.       for(int k=0; k< 3; k++) {
8.         System.out.print(" " + j);
9.         if(j==3 && k==1) break foreach;
10.        if(j==0 || j==2) break;
11.      }
12.    }
13.  }
14. }
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

Answer:

☑ **C** is correct. A `break` breaks out of the current innermost loop and continues. A labeled `break` breaks out of and terminates the current loops.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 2.2)

13. Given:

```
3. public class Gotcha {
4.   public static void main(String[] args) {
5.     // insert code here
6.
7.   }
8.   void go() {
9.     go();
10.  }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();
II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I - III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile

B. They will all compile

C. All will complete normally

D. None will complete normally

E. Only one will complete normally

F. Two of them will complete normally

Answer:

☑ **B** and **E** are correct. First off, `go()` is a badly designed recursive method, guaranteed to cause a StackOverflowError. Since `Exception` is not a superclass of `Error`, catching an `Exception` will not help handle an `Error`, so fragment III will not complete normally. Only fragment II will catch the `Error`.

☒ **A, C, D,** and **F** are incorrect based on the above. (Objective 2.5)

14. Given:

```
3. public class Clumsy {
4.   public static void main(String[] args) {
5.     int j = 7;
6.     assert(++j > 7);
7.     assert(++j > 8): "hi";
8.     assert(j > 10): j=12;
9.     assert(j==12): doStuff();
10.    assert(j==12): new Clumsy();
11.  }
12.  static void doStuff() { }
13. }
```

Which are true? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 6

C. Compilation fails due to an error on line 7

D. Compilation fails due to an error on line 8

E. Compilation fails due to an error on line 9

F. Compilation fails due to an error on line 10

Answer:

☑ **E** is correct. When an `assert` statement has two expressions, the second expression must return a value. The only two-expression `assert` statement that doesn't return a value is on line 9.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 2.3)

15. Given:

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 4, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a try/catch block around line 6 will cause compilation to fail

Answer:

☑ **D** is correct. This is kind of sneaky, but remember that we're trying to toughen you up for the real exam. If you're going to throw an IOException, you have to import the java.io package or declare the exception with a fully qualified name.

☒ **E** is incorrect because it's okay to both handle and declare an exception. **A, B,** and **C** are incorrect based on the above. (Objective 2.4)

16. Given:

```
2. class MyException extends Exception { }
3. class Tire {
4.   void doStuff() {  }
5. }
6. public class Retread extends Tire {
7.   public static void main(String[] args) {
8.     new Retread().doStuff();
9.   }
10.  // insert code here
11.    System.out.println(7/0);
12.  }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I - IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All of those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

Answer:

☑ **C** and **D** are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However an overriding method *can* throw RuntimeExceptions not thrown by the overridden method.

☒ **A, B, E,** and **F** are incorrect based on the above. (Objective 2.4)

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Using String, StringBuffer, and StringBuilder (Objective 3.1)

- ❑ String objects are immutable, and String reference variables are not.
- ❑ If you create a new String without assigning it, it will be lost to your program.
- ❑ If you redirect a String reference to a new String, the old String can be lost.
- ❑ String methods use zero-based indexes, except for the second argument of `substring()`.
- ❑ The String class is `final`—its methods can't be overridden.
- ❑ When the JVM finds a String literal, it is added to the String literal pool.
- ❑ Strings have a method: `length()`; arrays have an attribute named `length`.
- ❑ The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
- ❑ StringBuilder methods should run faster than StringBuffer methods.
- ❑ All of the following bullets apply to both StringBuffer and StringBuilder:
  - ❑ They are mutable—they can change without creating a new object.
  - ❑ StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
  - ❑ StringBuffer `equals()` is not overridden; it doesn't compare values.
- ❑ Remember that chained methods are evaluated from left to right.
- ❑ String methods to remember: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- ❑ StringBuffer methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

File I/O (Objective 3.2)

- ❑ The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, PrintWriter, and Console.
- ❑ A new File object doesn't mean there's a new file on your hard drive.
- ❑ File objects can represent either a file or a directory.

- The File class lets you manage (add, rename, and delete) files and directories.
- The methods `createNewFile()` and `mkdir()` add entries to your file system.
- FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- Classes in java.io are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)
- It's very common to "wrap" a BufferedReader around a FileReader or a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
- PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
- Java 5 PrintWriters have new `append()`, `format()`, and `printf()` methods.
- Console objects can read non-echoed input and are instantiated using System.console().

Serialization (Objective 3.3)

- The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
- A class must implement Serializable before its objects can be serialized.
- The `ObjectOutputStream.writeObject()` method serializes objects, and the `ObjectInputStream.readObject()` method deserializes objects.
- If you mark an instance variable `transient`, it will not be serialized even thought the rest of the object's state will be.
- You can supplement a class's automatic serialization process by implementing the `writeObject()` and `readObject()` methods. If you do this, embedding calls to `defaultWriteObject()` and `defaultReadObject()`, respectively, will handle the part of serialization that happens normally.
- If a superclass implements Serializable, then its subclasses do automatically.
- If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its superconstructor(s).
- DataInputStream and DataOutputStream aren't actually on the exam, in spite of what the Sun objectives say.

Dates, Numbers, and Currency (Objective 3.4)

- The classes you need to understand are java.util.Date, java.util.Calendar, java.text.DateFormat, java.text.NumberFormat, and java.util.Locale.
- Most of the Date class's methods have been deprecated.
- A Date is stored as a `long`, the number of milliseconds since January 1, 1970.
- Date objects are go-betweens the Calendar and Locale classes.
- The Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.
- Create Calendar instances using static factory methods (`getInstance()`).
- The Calendar methods you should understand are `add()`, which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and `roll()`, which works like `add()` but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
- DateFormat instances are created using static factory methods (`getInstance()` and `getDateInstance()`).
- There are several format "styles" available in the DateFormat class.
- DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
- The `DateFormat.format()` method is used to create Strings containing properly formatted dates.
- The Locale class is used in conjunction with DateFormat and NumberFormat.
- Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.
- For the exam you should understand creating Locales using language, or a combination of language and country.

Parsing, Tokenizing, and Formatting (Objective 3.5)

- regex is short for regular expressions, which are the patterns used to search for data within large data sources.
- regex is a sub-language that exists in Java and other languages (such as Perl).
- regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".

- Study the `\d`, `\s`, `\w`, and . metacharacters
- regex provides for *quantifiers* which allow you to specify concepts like: "look for one or more digits in a row."
- Study the `?`, `*`, and `+` greedy quantifiers.
- Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance `String s = "\\d";`
- The Pattern and Matcher classes have Java's most powerful regex capabilities.
- You should understand the Pattern `compile()` method and the Matcher `matches()`, `pattern()`, `find()`, `start()`, and `group()` methods.
- You WON'T need to understand Matcher's replacement-oriented methods.
- You can use java.util.Scanner to do simple regex searches, but it is primarily intended for tokenizing.
- Tokenizing is the process of splitting delimited data into small pieces.
- In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- Tokenizing can be done with the Scanner class, or with `String.split()`.
- Delimiters are single characters like commas, or complex regex expressions.
- The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.
- The Scanner class allows you to tokenize Strings or streams or files.
- The `String.split()` method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
- New to Java 5 are two methods used to format data for output. These methods are `format()` and `printf()`. These methods are found in the PrintStream class, an instance of which is the `out` in `System.out`.
- The `format()` and `printf()` methods have identical functionality.
- Formatting data with `printf()` (or `format()`) is accomplished using *formatting strings* that are associated with primitive or string arguments.
- The `format()` method allows you to mix literals in with your format strings.
- The format string values you should know are
  - Flags: `-`, `+`, `0`, `,` , and `(`
  - Conversions: `b`, `c`, `d`, `f`, and `s`
- If your conversion character doesn't match your argument type, an exception will be thrown.

SELF TEST

1. Given:

```
import java.util.regex.*;
class Regex2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + m.group());
    }
  }
}
```

And the command line:

```
java Regex2 "\d*" ab34ef
```

What is the result?

A. `234`
B. `334`
C. `2334`
D. `0123456`
E. `01234456`
F. `12334567`
G. Compilation fails

2. Given:

```
import java.io.*;
class Player {
  Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
  CardPlayer() { System.out.print("c"); }
```

```
  public static void main(String[] args) {
    CardPlayer c1 = new CardPlayer();
    try {
      FileOutputStream fos = new FileOutputStream("play.txt");
      ObjectOutputStream os = new ObjectOutputStream(fos);
      os.writeObject(c1);
      os.close();
      FileInputStream fis = new FileInputStream("play.txt");
      ObjectInputStream is = new ObjectInputStream(fis);
      CardPlayer c2 = (CardPlayer) is.readObject();
      is.close();
    } catch (Exception x ) { }
  }
}
```

What is the result?

A. `pc`

B. `pcc`

C. `pcp`

D. `pcpc`

E. Compilation fails

F. An exception is thrown at runtime

3. Given:

```
class TKO {
  public static void main(String[] args) {
    String s = "-";
    Integer x = 343;
    long L343 = 343L;
    if(x.equals(L343))  s += ".e1 ";
    if(x.equals(343))  s += ".e2 ";
    Short s1 = (short)((new Short((short)343)) / (new Short((short)49)));
    if(s1 == 7)        s += "=s ";
    if(s1 < new Integer(7+1))  s += "fly ";
    System.out.println(s);
  }
}
```

Which of the following will be included in the output String `s`? (Choose all that apply.)

A. `.e1`
B. `.e2`
C. `=s`
D. `fly`
E. None of the above
F. Compilation fails
G. An exception is thrown at runtime

4. Given:

```
import java.io.*;

class Keyboard { }
public class Computer implements Serializable {
  private Keyboard k = new Keyboard();
  public static void main(String[] args) {
    Computer c = new Computer();
    c.storeIt(c);
  }
  void storeIt(Computer c) {
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(c);
      os.close();
      System.out.println("done");
    } catch (Exception x) {System.out.println("exc"); }
  }
}
```

What is the result? (Choose all that apply.)

A. `exc`
B. `done`
C. Compilation fails
D. Exactly one object is serialized
E. Exactly two objects are serialized

5. Using the fewest **fragments** possible (and filling the fewest slots possible), complete the **code** below so that the class builds a directory named "`dir3`" and creates a file named "`file3`" inside "`dir3`". Note you can use each fragment either zero or one times.

Code:

```
import java.io._____________

class Maker {
  public static void main(String[] args) {

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________
  }
}
```

Fragments:

```
File;          FileDescriptor;      FileWriter;     Directory;
try {          .createNewDir();     File dir        File
{ }            (Exception x)        ("dir3");       file
file           .createNewFile();    = new File      = new File
dir            (dir, "file3");      (dir, file);    .createFile();
} catch        ("dir3", "file3");   .mkdir();       File file
```

6. Given that 1119280000000L is roughly the number of milliseconds from Jan 1, 1970, to June 20, 2005, and that you want to print that date in German, using the `LONG` style such that "`June`" will be displayed as "`Juni`", complete the code using the fragments below. Note: you can use each fragment either zero or one times, and you might not need to fill all of the slots.

Code:

```
import java._____________

import java._____________

class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);

    DateFormat df = ________________________________

                    _________________, __________________ );

    System.out.println(_________________
  }
}
```

Fragments:

```
io.*;      new DateFormat(                 Locale.LONG
nio.*;     DateFormat.getInstance(         Locale.GERMANY
util.*;    DateFormat.getDateInstance(     DateFormat.LONG
text.*;    util.regex;                     DateFormat.GERMANY
date.*;    df.format(d));                  d.format(df));
```

7. Given:

```
import java.io.*;
class Directories {
  static String [] dirs = {"dir1", "dir2"};
  public static void main(String [] args) {
    for (String d : dirs) {

      // insert code 1 here

      File file = new File(path, args[0]);

      // insert code 2 here
    }
  }
}
```

and that the invocation

```
java Directories file2.txt
```

is issued from a directory that has two subdirectories, "dir1" and "dir2", and that "dir1" has a file "file1.txt" and "dir2" has a file "file2.txt", and the output is "false true"; which set(s) of code fragments must be inserted? (Choose all that apply.)

A.
```
String path = d;
System.out.print(file.exists() + " ");
```

B.
```
String path = d;
System.out.print(file.isFile() + " ");
```

C.
```
String path = File.separator + d;
System.out.print(file.exists() + " ");
```

D.
```
String path = File.separator + d;
System.out.print(file.isFile() + " ");
```

8. Given:

```
import java.io.*;

public class TestSer {
  public static void main(String[] args) {
    SpecialSerial s = new SpecialSerial();
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(s);  os.close();
      System.out.print(++s.z + " ");

      ObjectInputStream is = new ObjectInputStream(
         new FileInputStream("myFile"));
      SpecialSerial s2 = (SpecialSerial)is.readObject();
      is.close();
```

```
      System.out.println(s2.y + " " + s2.z);
    } catch (Exception x) {System.out.println("exc"); }
  }
}
class SpecialSerial implements Serializable {
  transient int y = 7;
  static int z = 9;
}
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output is `10 0 9`

C. The output is `10 0 10`

D. The output is `10 7 9`

E. The output is `10 7 10`

F. In order to alter the standard deserialization process you would implement the `readObject()` method in `SpecialSerial`

G. In order to alter the standard deserialization process you would implement the `defaultReadObject()` method in `SpecialSerial`

9. Given:

```
3. public class Theory {
4.   public static void main(String[] args) {
5.     String s1 = "abc";
6.     String s2 = s1;
7.     s1 += "d";
8.     System.out.println(s1 + " " + s2 + " " + (s1==s2));
9.
10.    StringBuffer sb1 = new StringBuffer("abc");
11.    StringBuffer sb2 = sb1;
12.    sb1.append("d");
13.    System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The first line of output is `abc abc true`

C. The first line of output is `abc abc false`

D. The first line of output is `abcd abc false`

E. The second line of output is `abcd abc false`

F. The second line of output is `abcd abcd true`

G. The second line of output is `abcd abcd false`

10. Given:

```
 3. import java.io.*;
 4. public class ReadingFor {
 5.   public static void main(String[] args) {
 6.     String s;
 7.     try {
 8.       FileReader fr = new FileReader("myfile.txt");
 9.       BufferedReader br = new BufferedReader(fr);
10.       while((s = br.readLine()) != null)
11.         System.out.println(s);
12.       br.flush();
13.     } catch (IOException e) { System.out.println("io error"); }
16.   }
17. }
```

And given that myfile.txt contains the following two lines of data:

```
ab
cd
```

What is the result?

A. `ab`

B. `abcd`

C. `ab`
`cd`

D. `a`
`b`
`c`
`d`

E. Compilation fails

11. Given:

```
3. import java.io.*;
4. public class Talker {
5.   public static void main(String[] args) {
6.     Console c = System.console();
7.     String u = c.readLine("%s", "username: ");
8.     System.out.println("hello " + u);
9.     String pw;
10.    if(c != null && (pw = c.readPassword("%s", "password: ")) != null)
11.      // check for valid password
12.  }
13. }
```

If line 6 creates a valid Console object, and if the user enters *fred* as a username and *1234* as a password, what is the result? (Choose all that apply.)

A. `username:`
`password:`

B. `username: fred`
`password:`

C. `username: fred`
`password: 1234`

D. Compilation fails

E. An exception is thrown at runtime

12. Given:

```
3. import java.io.*;
4. class Vehicle { }
5. class Wheels { }
6. class Car extends Vehicle implements Serializable {  }
7. class Ford extends Car { }
8. class Dodge extends Car {
9.   Wheels w = new Wheels();
10. }
```

Instances of which class(es) can be serialized? (Choose all that apply.)

A. Car

B. Ford

C. Dodge
D. Wheels
E. Vehicle

13. Given:

```
3. import java.text.*;
4. public class Slice {
5.   public static void main(String[] args) {
6.     String s = "987.123456";
7.     double d  = 987.123456d;
8.     NumberFormat nf = NumberFormat.getInstance();
9.     nf.setMaximumFractionDigits(5);
10.    System.out.println(nf.format(d) + " ");
11.    try {
12.      System.out.println(nf.parse(s));
13.    } catch (Exception e) { System.out.println("got exc"); }
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. The output is `987.12345 987.12345`
B. The output is `987.12346 987.12345`
C. The output is `987.12345 987.123456`
D. The output is `987.12346 987.123456`
E. The try/catch block is unnecessary
F. The code compiles and runs without exception
G. The invocation of `parse()` must be placed within a try/catch block

14. Given:

```
3. import java.util.regex.*;
4. public class Archie {
5.   public static void main(String[] args) {
6.     Pattern p = Pattern.compile(args[0]);
7.     Matcher m = p.matcher(args[1]);
8.     int count = 0;
9.     while(m.find())
10.      count++;
```

```
11.      System.out.print(count);
12.    }
13. }
```

And given the command line invocation:

```
java Archie "\d+" ab2c4d67
```

What is the result?

A. 0

B. 3

C. 4

D. 8

E. 9

F. Compilation fails

15. Given:

```
 3. import java.util.*;
 4. public class Looking {
 5.   public static void main(String[] args) {
 6.     String input = "1 2 a 3 45 6";
 7.     Scanner sc = new Scanner(input);
 8.     int x = 0;
 9.     do {
10.       x = sc.nextInt();
11.       System.out.print(x + " ");
12.     } while (x!=0);
13.   }
14. }
```

What is the result?

A. `1 2`

B. `1 2 3 45 6`

C. `1 2 3 4 5 6`

D. `1 2 a 3 45 6`

E. Compilation fails

F. `1 2` followed by an exception

SELF TEST ANSWERS

1. Given:

```
import java.util.regex.*;
class Regex2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + m.group());
    }
  }
}
```

And the command line:

```
java Regex2 "\d*" ab34ef
```

What is the result?

A. `234`

B. `334`

C. `2334`

D. `0123456`

E. `01234456`

F. `12334567`

G. Compilation fails

Answer:

☑ **E** is correct. The `\d` is looking for digits. The `*` is a quantifier that looks for `0` to many occurrences of the pattern that precedes it. Because we specified `*`, the `group()` method returns empty Strings until consecutive digits are found, so the only time `group()` returns a value is when it returns 34 when the matcher finds digits starting in position 2. The `start()` method returns the starting position of the previous match because, again, we said find `0` to many occurrences.

☒ **A, B, C, D, F,** and **G** are incorrect based on the above. (Objective 3.5)

2. Given:

```
import java.io.*;
class Player {
  Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
  CardPlayer() { System.out.print("c"); }
  public static void main(String[] args) {
    CardPlayer c1 = new CardPlayer();
    try {
      FileOutputStream fos = new FileOutputStream("play.txt");
      ObjectOutputStream os = new ObjectOutputStream(fos);
      os.writeObject(c1);
      os.close();
      FileInputStream fis = new FileInputStream("play.txt");
      ObjectInputStream is = new ObjectInputStream(fis);
      CardPlayer c2 = (CardPlayer) is.readObject();
      is.close();
    } catch (Exception x ) { }
  }
}
```

What is the result?

A. `pc`

B. `pcc`

C. `pcp`

D. `pcpc`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

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

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 3.3)

3. Given:

```
class TKO {
  public static void main(String[] args) {
    String s = "-";
    Integer x = 343;
    long L343 = 343L;
    if(x.equals(L343))  s += ".e1 ";
    if(x.equals(343))  s += ".e2 ";
    Short s1 = (short)((new Short((short)343)) / (new Short((short)49)));
    if(s1 == 7)         s += "=s ";
    if(s1 < new Integer(7+1))  s += "fly ";
    System.out.println(s);
} }
```

Which of the following will be included in the output String `s`? (Choose all that apply.)

A. `.e1`
B. `.e2`
C. `=s`
D. `fly`
E. None of the above
F. Compilation fails
G. An exception is thrown at runtime

Answer:

☑ **B, C,** and **D** are correct. Remember, that the `equals()` method for the integer wrappers will only return `true` if the two primitive types and the two values are equal. With **C**, it's okay to unbox and use `==`. For **D**, it's okay to create a wrapper object with an expression, and unbox it for comparison with a primitive.

☒ **A, E, F,** and **G** are incorrect based on the above. (Remember that **A** is using the `equals()` method to try to compare two different types.) (Objective 3.1)

4. Given:

```
import java.io.*;

class Keyboard { }
public class Computer implements Serializable {
```

```
  private Keyboard k = new Keyboard();
  public static void main(String[] args) {
    Computer c = new Computer();
    c.storeIt(c);
  }
  void storeIt(Computer c) {
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(c);
      os.close();
      System.out.println("done");
    } catch (Exception x) {System.out.println("exc"); }
  }
}
```

What is the result? (Choose all that apply.)

A. `exc`

B. `done`

C. Compilation fails

D. Exactly one object is serialized

E. Exactly two objects are serialized

Answer:

☑ **A** is correct. An instance of type Computer Has-a Keyboard. Because Keyboard doesn't implement Serializable, any attempt to serialize an instance of Computer will cause an exception to be thrown.

☒ **B, C, D,** and **E** are incorrect based on the above. If Keyboard did implement Serializable then two objects would have been serialized. (Objective 3.3)

5. Using the fewest fragments possible (and filling the fewest slots possible), complete the code below so that the class builds a directory named "`dir3`" and creates a file named "`file3`" inside "`dir3`". Note you can use each fragment either zero or one times.

Code:

```
import java.io._____________

class Maker {
  public static void main(String[] args) {

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________
} }
```

Fragments:

```
File;       FileDescriptor;     FileWriter;     Directory;
try {       .createNewDir();    File dir        File
{ }         (Exception x)       ("dir3");       file
file        .createNewFile();   = new File      = new File
dir         (dir, "file3");     (dir, file);    .createFile();
} catch     ("dir3", "file3");  .mkdir();       File file
```

Answer:

```
import java.io.File;
class Maker {
  public static void main(String[] args) {
    try {
      File dir = new File("dir3");
      dir.mkdir();
      File file = new File(dir, "file3");
      file.createNewFile();
    } catch (Exception x) { }
} }
```

Notes: The `new File` statements don't make actual files or directories, just objects. You need the `mkdir()` and `createNewFile()` methods to actually create the directory and the file. (Objective 3.2)

6. Given that 1119280000000L is roughly the number of milliseconds from Jan. 1, 1970, to June 20, 2005, and that you want to print that date in German, using the `LONG` style such that "`June`" will be displayed as "`Juni`", complete the code using the fragments below. Note: you can use each fragment either zero or one times, and you might not need to fill all of the slots.

Code:

```
import java._____________
import java._____________

class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);
    DateFormat df = _______________________________

                    _________________ , _________________ );

    System.out.println(_________________
  }
}
```

Fragments:

```
io.*;      new DateFormat(                Locale.LONG
nio.*;     DateFormat.getInstance(        Locale.GERMANY
util.*;    DateFormat.getDateInstance(    DateFormat.LONG
text.*;    util.regex;                    DateFormat.GERMANY
date.*;    df.format(d));                 d.format(df));
```

Answer:

```
import java.util.*;
import java.text.*;
class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);
  DateFormat df = DateFormat.getDateInstance(
                    DateFormat.LONG, Locale.GERMANY);
    System.out.println(df.format(d));
  }
}
```

Notes: Remember that you must build `DateFormat` objects using static methods. Also remember that you must specify a Locale for a `DateFormat` object at the time of instantiation. The `getInstance()` method does not take a Locale. (Objective 3.4)

7. Given:

```
import java.io.*;

class Directories {
  static String [] dirs = {"dir1", "dir2"};
  public static void main(String [] args) {
    for (String d : dirs) {

      // insert code 1 here

      File file = new File(path, args[0]);

      // insert code 2 here
    }
  }
}
```

and that the invocation

```
java Directories file2.txt
```

is issued from a directory that has two subdirectories, "`dir1`" and "`dir2`", and that "`dir1`" has a file "`file1.txt`" and "`dir2`" has a file "`file2.txt`", and the output is "`false true`", which set(s) of code fragments must be inserted? (Choose all that apply.)

A.
```
String path = d;

System.out.print(file.exists() + " ");
```

B.
```
String path = d;

System.out.print(file.isFile() + " ");
```

C.
```
String path = File.separator + d;

System.out.print(file.exists() + " ");
```

D.
```
String path = File.separator + d;

System.out.print(file.isFile() + " ");
```

Answer:

☑ **A** and **B** are correct. Because you are invoking the program from the directory whose direct subdirectories are to be searched, you don't start your path with a `File.separator` character. The `exists()` method tests for either files or directories; the `isFile()` method tests only for files. Since we're looking for a file, both methods work.

☒ **C** and **D** are incorrect based on the above. (Objective 3.2)

8. Given:

```
import java.io.*;

public class TestSer {
  public static void main(String[] args) {
    SpecialSerial s = new SpecialSerial();
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(s);  os.close();
      System.out.print(++s.z + " ");

      ObjectInputStream is = new ObjectInputStream(
         new FileInputStream("myFile"));
      SpecialSerial s2 = (SpecialSerial)is.readObject();
      is.close();
      System.out.println(s2.y + " " + s2.z);
    } catch (Exception x) {System.out.println("exc"); }
  }
}
class SpecialSerial implements Serializable {
  transient int y = 7;
  static int z = 9;
}
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output is `10 0 9`

C. The output is `10 0 10`

D. The output is `10 7 9`

E. The output is `10 7 10`

F. In order to alter the standard deserialization process you would implement the `readObject()` method in `SpecialSerial`

G. In order to alter the standard deserialization process you would implement the `defaultReadObject()` method in `SpecialSerial`

Answer:

☑ **C** and **F** are correct. **C** is correct because `static` and `transient` variables are not serialized when an object is serialized. **F** is a valid statement.

☒ **A, B, D,** and **E** are incorrect based on the above. **G** is incorrect because you don't implement the `defaultReadObject()` method, you call it from within the `readObject()` method, along with any custom read operations your class needs. (Objective 3.3)

9. Given:

```
3. public class Theory {
4.   public static void main(String[] args) {
5.     String s1 = "abc";
6.     String s2 = s1;
7.     s1 += "d";
8.     System.out.println(s1 + " " + s2 + " " + (s1==s2));
9.
10.    StringBuffer sb1 = new StringBuffer("abc");
11.    StringBuffer sb2 = sb1;
12.    sb1.append("d");
13.    System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. The first line of output is `abc abc true`
C. The first line of output is `abc abc false`
D. The first line of output is `abcd abc false`
E. The second line of output is `abcd abc false`
F. The second line of output is `abcd abcd true`
G. The second line of output is `abcd abcd false`

Answer:

☑ **D** and **F** are correct. While String objects are immutable, references to Strings are mutable. The code `s1 += "d";` creates a new String object. StringBuffer objects are mutable, so the `append()` is changing the single StringBuffer object to which both StringBuffer references refer.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (Objective 3.1)

10. Given:

```
 3. import java.io.*;
 4. public class ReadingFor {
 5.   public static void main(String[] args) {
 6.     String s;
 7.     try {
 8.       FileReader fr = new FileReader("myfile.txt");
 9.       BufferedReader br = new BufferedReader(fr);
10.       while((s = br.readLine()) != null)
11.         System.out.println(s);
12.       br.flush();
13.     } catch (IOException e) { System.out.println("io error"); }
16.   }
17. }
```

And given that myfile.txt contains the following two lines of data:

```
ab
cd
```

What is the result?

A. `ab`

B. `abcd`

C.
```
ab
cd
```

D.
```
a
b
c
d
```

E. Compilation fails

Answer:

☑ **E** is correct. You need to call `flush()` only when you're writing data. Readers don't have `flush()` methods. If not for the call to `flush()`, answer C would be correct.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 3.2)

11. Given:

```
3. import java.io.*;
4. public class Talker {
5.   public static void main(String[] args) {
6.     Console c = System.console();
7.     String u = c.readLine("%s", "username: ");
8.     System.out.println("hello " + u);
9.     String pw;
10.    if(c != null && (pw = c.readPassword("%s", "password: ")) != null)
11.      // check for valid password
12.  }
13. }
```

If line 6 creates a valid Console object, and if the user enters *fred* as a username and *1234* as a password, what is the result? (Choose all that apply.)

A.
```
username:
password:
```

B.
```
username: fred
password:
```

C. `username: fred`
`password: 1234`

D. Compilation fails

E. An exception is thrown at runtime

Answer:

☑ **D** is correct. The `readPassword()` method returns a `char[]`. If a `char[]` were used, answer B would be correct.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 3.2)

12. Given:

```
3. import java.io.*;
4. class Vehicle { }
5. class Wheels { }
6. class Car extends Vehicle implements Serializable {  }
7. class Ford extends Car { }
8. class Dodge extends Car {
9.   Wheels w = new Wheels();
10. }
```

Instances of which class(es) can be serialized? (Choose all that apply.)

A. Car

B. Ford

C. Dodge

D. Wheels

E. Vehicle

Answer:

☑ **A** and **B** are correct. Dodge instances cannot be serialized because they "have" an instance of Wheels, which is not serializable. Vehicle instances cannot be serialized even though the subclass Car can be.

☒ **C, D,** and **E** are incorrect based on the above. (Objective 3.3)

13. Given:

```
3. import java.text.*;
4. public class Slice {
5.   public static void main(String[] args) {
6.     String s = "987.123456";
7.     double d  = 987.123456d;
8.     NumberFormat nf = NumberFormat.getInstance();
9.     nf.setMaximumFractionDigits(5);
10.    System.out.println(nf.format(d) + " ");
11.    try {
12.      System.out.println(nf.parse(s));
13.    } catch (Exception e) { System.out.println("got exc"); }
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. The output is `987.12345 987.12345`

B. The output is `987.12346 987.12345`

C. The output is `987.12345 987.123456`

D. The output is `987.12346 987.123456`

E. The try/catch block is unnecessary

F. The code compiles and runs without exception

G. The invocation of `parse()` must be placed within a try/catch block

Answer:

☑ **D, F,** and **G** are correct. The `setMaximumFractionDigits()` applies to the formatting but not the parsing. The try/catch block is placed appropriately. This one might scare you into thinking that you'll need to memorize more than you really do. If you can remember that you're formatting the number and parsing the string you should be fine for the exam.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 3.4)

14. Given:

```
3. import java.util.regex.*;
4. public class Archie {
5.   public static void main(String[] args) {
6.     Pattern p = Pattern.compile(args[0]);
```

```
 7.      Matcher m = p.matcher(args[1]);
 8.      int count = 0;
 9.      while(m.find())
10.        count++;
11.      System.out.print(count);
12.    }
13. }
```

And given the command line invocation:

```
java Archie "\d+" ab2c4d67
```

What is the result?

A. 0

B. 3

C. 4

D. 8

E. 9

F. Compilation fails

Answer:

☑ **B** is correct. The "\d" metacharacter looks for digits, and the + quantifier says look for "one or more" occurrences. The `find()` method will find three sets of one or more consecutive digits: 2, 4, and 67.

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objective 3.5)

15. Given:

```
 3. import java.util.*;
 4. public class Looking {
 5.   public static void main(String[] args) {
 6.     String input = "1 2 a 3 45 6";
 7.     Scanner sc = new Scanner(input);
 8.     int x = 0;
 9.     do {
10.       x = sc.nextInt();
11.       System.out.print(x + " ");
12.     } while (x!=0);
13.   }
14. }
```

What is the result?

A. `1 2`

B. `1 2 3 45 6`

C. `1 2 3 4 5 6`

D. `1 2 a 3 45 6`

E. Compilation fails

F. `1 2` followed by an exception

Answer:

☑ **F** is correct. The `nextXxx()` methods are typically invoked after a call to a `hasNextXxx()`, which determines whether the next token is of the correct type.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 3.5)

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Overriding hashCode() and equals() (Objective 6.2)

- ❑ `equals()`, `hashCode()`, and `toString()` are `public`.
- ❑ Override `toString()` so that `System.out.println()` or other methods can see something useful, like your object's state.
- ❑ Use == to determine if two reference variables refer to the same object.
- ❑ Use `equals()` to determine if two objects are meaningfully equivalent.
- ❑ If you don't override `equals()`, your objects won't be useful hashing keys.
- ❑ If you don't override `equals()`, different objects can't be considered equal.
- ❑ Strings and wrappers override `equals()` and make good hashing keys.
- ❑ When overriding `equals()`, use the `instanceof` operator to be sure you're evaluating an appropriate class.
- ❑ When overriding `equals()`, compare the objects' significant attributes.
- ❑ Highlights of the `equals()` contract:
  - ❑ Reflexive: `x.equals(x)` is `true`.
  - ❑ Symmetric: If `x.equals(y)` is `true`, then `y.equals(x)` must be `true`.
  - ❑ Transitive: If `x.equals(y)` is `true`, and `y.equals(z)` is `true`, then `z.equals(x)` is `true`.
  - ❑ Consistent: Multiple calls to `x.equals(y)` will return the same result.
  - ❑ Null: If x is not `null`, then `x.equals(null)` is false.
- ❑ If `x.equals(y)` is true, then `x.hashCode() == y.hashCode()` is true.
- ❑ If you override `equals()`, override `hashCode()`.
- ❑ HashMap, HashSet, Hashtable, LinkedHashMap, & LinkedHashSet use hashing.
- ❑ An appropriate `hashCode()` override sticks to the `hashCode()` contract.
- ❑ An efficient `hashCode()` override distributes keys evenly across its buckets.
- ❑ An overridden `equals()` must be at least as precise as its `hashCode()` mate.
- ❑ To reiterate: if two objects are equal, their hashcodes must be equal.
- ❑ It's legal for a `hashCode()` method to return the same value for all instances (although in practice it's very inefficient).

- Highlights of the `hashCode()` contract:
  - Consistent: multiple calls to `x.hashCode()` return the same integer.
  - If `x.equals(y)` is `true`, `x.hashCode() == y.hashCode()` is `true`.
  - If `x.equals(y)` is `false`, then `x.hashCode() == y.hashCode()` can be either `true` or `false`, but `false` will tend to create better efficiency.
- `transient` variables aren't appropriate for `equals()` and `hashCode()`.

Collections (Objective 6.1)

- Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- Three meanings for "collection":
  - **collection** Represents the data structure in which objects are stored
  - **Collection** java.util interface from which Set and List extend
  - **Collections** A class that holds static collection utility methods
- Four basic flavors of collections include Lists, Sets, Maps, Queues:
  - **Lists of things** Ordered, duplicates allowed, with an index.
  - **Sets of things** May or may not be ordered and/or sorted; duplicates not allowed.
  - **Maps of things with keys** May or may not be ordered and/or sorted; duplicate keys are not allowed.
  - **Queues of things to process** Ordered by FIFO or by priority.
- Four basic sub-flavors of collections Sorted, Unsorted, Ordered, Unordered.
  - **Ordered** Iterating through a collection in a specific, non-random order.
  - **Sorted** Iterating through a collection in a sorted order.
- Sorting can be alphabetic, numeric, or programmer-defined.

Key Attributes of Common Collection Classes (Objective 6.1)

- ArrayList: Fast iteration and fast random access.
- Vector: It's like a slower ArrayList, but it has synchronized methods.
- LinkedList: Good for adding elements to the ends, i.e., stacks and queues.
- HashSet: Fast access, assures no duplicates, provides no ordering.
- LinkedHashSet: No duplicates; iterates by insertion order.
- TreeSet: No duplicates; iterates in sorted order.

- ❑ HashMap: Fastest updates (key/values); allows one `null` key, many `null` values.
- ❑ Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No `null` values or `null` keys allowed.
- ❑ LinkedHashMap: Faster iterations; iterates by insertion order or last accessed; allows one `null` key, many `null` values.
- ❑ TreeMap: A sorted map.
- ❑ PriorityQueue: A to-do list ordered by the elements' priority.

Using Collection Classes (Objective 6.3)

- ❑ Collections hold only Objects, but primitives can be autoboxed.
- ❑ Iterate with the enhanced `for`, or with an Iterator via `hasNext()` & `next()`.
- ❑ `hasNext()` determines if more elements exist; the Iterator does NOT move.
- ❑ `next()` returns the next element AND moves the Iterator forward.
- ❑ To work correctly, a Map's keys must override `equals()` and `hashCode()`.
- ❑ Queues use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.
- ❑ As of Java 6 TreeSets and TreeMaps have new navigation methods like floor() and higher().
- ❑ You can create/extend "backed" sub-copies of TreeSets and TreeMaps.

Sorting and Searching Arrays and Lists (Objective 6.5)

- ❑ Sorting can be in natural order, or via a Comparable or many Comparators.
- ❑ Implement Comparable using `compareTo()`; provides only one sort order.
- ❑ Create many Comparators to sort a class many ways; implement `compare()`.
- ❑ To be sorted and searched, a List's elements must be *comparable*.
- ❑ To be searched, an array or List must first be sorted.

Utility Classes: Collections and Arrays (Objective 6.5)

- ❑ Both of these java.util classes provide
  - ❑ A `sort()` method. Sort using a Comparator or sort using natural order.
  - ❑ A `binarySearch()` method. Search a pre-sorted array or List.

- `Arrays.asList()` creates a List from an array and links them together.
- `Collections.reverse()` reverses the order of elements in a List.
- `Collections.reverseOrder()` returns a Comparator that sorts in reverse.
- Lists and Sets have a `toArray()` method to create arrays.

Generics (Objective 6.4)

- Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- An ArrayList<Animal> can accept references of type Dog, Cat, or any other subtype of Animal (subclass, or if Animal is an interface, implementation).
- When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();
List list = new ArrayList();
// more code
String s = gList.get(0);           // no cast needed
String s = (String)list.get(0);    // cast required
```

- You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
- If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a List<String> into a method declared as

```
void foo(List aList) { aList.add(anInteger); }
```

  You'll get a warning because add() is potentially "unsafe".
- "Compiles without error" is not the same as "compiles without warnings." A compilation *warning* is not considered a compilation *error* or *failure*.
- Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
- Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

```
List<Animal> aList = new ArrayList<Animal>();   // yes
```

  You can't say

```
List<Animal> aList = new ArrayList<Dog>();      // no
```

- The polymorphic assignment rule applies everywhere an assignment can be made. The following are NOT allowed:

```
void foo(List<Animal> aList) { }  // cannot take a List<Dog>
List<Animal> bar() { }            // cannot return a List<Dog>
```

- Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

```
void addD(List<Dog> d) {}  // can take only <Dog>
void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>
```

- The wildcard keyword `extends` is used to mean either "extends" or "implements." So in `<? extends Dog>`, `Dog` can be a class or an interface.
- When using a wildcard, `List<? extends Dog>`, the collection can be accessed but not modified.
- When using a wildcard, `List<?>`, any generic type can be assigned to the reference, but for access only, no modifications.
- `List<Object>` refers only to a `List<Object>`, while `List<?>` or `List<? extends Object>` can hold any type of object, but for access only.
- Declaration conventions for generics use T for type and E for element:

```
public interface List<E>  // API declaration for List
boolean add(E o)          // List.add() declaration
```

- The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { }      // a class
T anInstance;         // an instance variable
Foo(T aRef) {}        // a constructor argument
void bar(T aRef) {}   // a method argument
T baz() {}            // a return type
```

  The compiler will substitute the actual type.

- You can use more than one parameterized type in a declaration:

```
public class UseTwo<T, X> { }
```

- You can declare a generic method using a type not defined in the class:

```
public <T> void makeList(T t) { }
```

  is NOT using T as the return type. This method has a `void` return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

SELF TEST

1. Given:

```
public static void main(String[] args) {

  // INSERT DECLARATION HERE
  for (int i = 0; i <= 10; i++) {
    List<Integer> row = new ArrayList<Integer>();
    for (int j = 0; j <= 10; j++)
      row.add(i * j);
    table.add(row);
  }
  for (List<Integer> row : table)
    System.out.println(row);
  }
```

Which statements could be inserted at `// INSERT DECLARATION HERE` to allow this code to compile and run? (Choose all that apply.)

A. `List<List<Integer>> table = new List<List<Integer>>();`

B. `List<List<Integer>> table = new ArrayList<List<Integer>>();`

C. `List<List<Integer>> table = new ArrayList<ArrayList<Integer>>();`

D. `List<List, Integer> table = new List<List, Integer>();`

E. `List<List, Integer> table = new ArrayList<List, Integer>();`

F. `List<List, Integer> table = new ArrayList<ArrayList, Integer>();`

G. None of the above

2. Which statements are true about comparing two instances of the same class, given that the `equals()` and `hashCode()` methods have been properly overridden? (Choose all that apply.)

A. If the `equals()` method returns true, the `hashCode()` comparison `==` might return false

B. If the `equals()` method returns false, the `hashCode()` comparison `==` might return true

C. If the `hashCode()` comparison `==` returns true, the `equals()` method must return true

D. If the `hashCode()` comparison `==` returns true, the `equals()` method might return true

E. If the `hashCode()` comparison `!=` returns true, the `equals()` method might return true

3. Given:

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
  System.out.print(it.next() + " ");
}
```

Which statements are true?

A. The `before()` method will print `1 2`

B. The `before()` method will print `1 2 3`

C. The `before()` method will print three numbers, but the order cannot be determined

D. The `before()` method will not compile

E. The `before()` method will throw an exception at runtime

4. Given:

```
import java.util.*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos t1 = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(t1, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.println(m.size());
  }
}
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() { return 9; }
}
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile

B. As the code stands the output will be `2`

C. As the code stands the output will be `3`

D. If the `hashCode()` method is uncommented the output will be `2`

E. If the `hashCode()` method is uncommented the output will be `3`

F. If the `hashCode()` method is uncommented the code will not compile

5. Given:

```
12. public class AccountManager {
13.     private Map accountTotals = new HashMap();
14.     private int retirementFund;
15.
16.     public int getBalance(String accountName) {
17.         Integer total = (Integer) accountTotals.get(accountName);
18.         if (total == null)
19.             total = Integer.valueOf(0);
20.         return total.intValue();
21.     }
23.     public void setBalance(String accountName, int amount) {
24.         accountTotals.put(accountName, Integer.valueOf(amount));
25.     }
26. }
```

This class is to be updated to make use of appropriate generic types, with no changes in behavior (for better or worse). Which of these steps could be performed? (Choose three.)

A. Replace line 13 with
```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

B. Replace line 13 with
```
private Map<String, Integer> accountTotals = new HashMap<String, Integer>();
```

C. Replace line 13 with
```
private Map<String<Integer>> accountTotals = new HashMap<String<Integer>>();
```

D. Replace lines 17–20 with
```
int total = accountTotals.get(accountName);
if (total == null)
    total = 0;
return total;
```

E. Replace lines 17–20 with
```
Integer total = accountTotals.get(accountName);
if (total == null)
    total = 0;
return total;
```

F. Replace lines 17–20 with
```
return accountTotals.get(accountName);
```

G. Replace line 24 with
```
accountTotals.put(accountName, amount);
```

H. Replace line 24 with
```
accountTotals.put(accountName, amount.intValue());
```

6. Given:

```
interface Hungry<E> { void munch(E x); }
interface Carnivore<E extends Animal> extends Hungry<E> {}
interface Herbivore<E extends Plant> extends Hungry<E> {}
abstract class Plant {}
class Grass extends Plant {}
abstract class Animal {}
class Sheep extends Animal implements Herbivore<Sheep> {
  public void munch(Sheep x) {}
}
class Wolf extends Animal implements Carnivore<Sheep> {
  public void munch(Sheep x) {}
}
```

Which of the following changes (taken separately) would allow this code to compile? (Choose all that apply.)

A. Change the Carnivore interface to
```
interface Carnivore<E extends Plant> extends Hungry<E> {}
```

B. Change the Herbivore interface to
```
interface Herbivore<E extends Animal> extends Hungry<E> {}
```

C. Change the Sheep class to
```
class Sheep extends Animal implements Herbivore<Plant> {
    public void munch(Grass x) {}
}
```

D. Change the Sheep class to
```
class Sheep extends Plant implements Carnivore<Wolf> {
    public void munch(Wolf x) {}
}
```

E. Change the Wolf class to

```
class Wolf extends Animal implements Herbivore<Grass> {
    public void munch(Grass x) {}
}
```

F. No changes are necessary

7. Which collection class(es) allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized? (Choose all that apply.)

A. `java.util.HashSet`

B. `java.util.LinkedHashSet`

C. `java.util.List`

D. `java.util.ArrayList`

E. `java.util.Vector`

F. `java.util.PriorityQueue`

8. Given a method declared as

```
public static <E extends Number> List<E> process(List<E> nums)
```

A programmer wants to use this method like this

```
// INSERT DECLARATIONS HERE

output = process(input);
```

Which pairs of declarations could be placed at `// INSERT DECLARATIONS HERE` to allow the code to compile? (Choose all that apply.)

A.
```
ArrayList<Integer> input = null;
ArrayList<Integer> output = null;
```

B.
```
ArrayList<Integer> input = null;
List<Integer> output = null;
```

C.
```
ArrayList<Integer> input = null;
List<Number> output = null;
```

D.
```
List<Number> input = null;
ArrayList<Integer> output = null;
```

E.
```
List<Number> input = null;
List<Number> output = null;
```

F.
```
List<Integer> input = null;
List<Integer> output = null;
```

G. None of the above

9. Given the proper import statement(s), and

```
13.    PriorityQueue<String> pq = new PriorityQueue<String>();
14.    pq.add("2");
15.    pq.add("4");
16.    System.out.print(pq.peek() + " ");
17.    pq.offer("1");
18.    pq.add("3");
19.    pq.remove("1");
20.    System.out.print(pq.poll() + " ");
21.    if(pq.remove("2")) System.out.print(pq.poll() + " ");
22.    System.out.println(pq.poll() + " " + pq.peek());
```

What is the result?

A. 2 2 3 3

B. 2 2 3 4

C. 4 3 3 4

D. 2 2 3 3 3

E. 4 3 3 3 3

F. 2 2 3 3 4

G. Compilation fails

H. An exception is thrown at runtime

10. Given:

```
3. import java.util.*;
4. public class Mixup {
5.   public static void main(String[] args) {
```

```
 6.     Object o = new Object();
 7.     // insert code here
 8.     s.add("o");
 9.     s.add(o);
10.   }
11. }
```

And these three fragments:

```
I.   Set s = new HashSet();
II.  TreeSet s = new TreeSet();
III. LinkedHashSet s = new LinkedHashSet();
```

When fragments I, II, or III are inserted, independently, at line 7, which are true? (Choose all that apply.)

A. Fragment I compiles

B. Fragment II compiles

C. Fragment III compiles

D. Fragment I executes without exception

E. Fragment II executes without exception

F. Fragment III executes without exception

11. Given:

```
3. import java.util.*;
4. class Turtle {
5.   int size;
6.   public Turtle(int s) { size = s; }
7.   public boolean equals(Object o) { return (this.size == ((Turtle)o).size); }
8.   // insert code here
9. }
10. public class TurtleTest {
11.   public static void main(String[] args) {
12.     LinkedHashSet<Turtle> t = new LinkedHashSet<Turtle>();
13.     t.add(new Turtle(1));   t.add(new Turtle(2));   t.add(new Turtle(1));
14.     System.out.println(t.size());
15.   }
16. }
```

And these two fragments:

```
I.   public int hashCode() { return size/5; }
II.  // no hashCode method declared
```

If fragment I or II is inserted, independently, at line 8, which are true? (Choose all that apply.)

A. If fragment I is inserted, the output is 2
B. If fragment I is inserted, the output is 3
C. If fragment II is inserted, the output is 2
D. If fragment II is inserted, the output is 3
E. If fragment I is inserted, compilation fails
F. If fragment II is inserted, compilation fails

12. Given the proper import statement(s), and:

```
13.    TreeSet<String> s = new TreeSet<String>();
14.    TreeSet<String> subs = new TreeSet<String>();
15.    s.add("a"); s.add("b"); s.add("c"); s.add("d"); s.add("e");
16.
17.    subs = (TreeSet)s.subSet("b", true, "d", true);
18.    s.add("g");
19.    s.pollFirst();
20.    s.pollFirst();
21.    s.add("c2");
22.    System.out.println(s.size() +" "+ subs.size());
```

Which are true? (Choose all that apply.)

A. The size of `s` is 4
B. The size of `s` is 5
C. The size of `s` is 7
D. The size of `subs` is 1
E. The size of `subs` is 2
F. The size of `subs` is 3
G. The size of `subs` is 4
H. An exception is thrown at runtime

13. Given:

```
 3. import java.util.*;
 4. public class Magellan {
 5.   public static void main(String[] args) {
 6.     TreeMap<String, String> myMap = new TreeMap<String, String>();
 7.     myMap.put("a", "apple"); myMap.put("d", "date");
 8.     myMap.put("f", "fig"); myMap.put("p", "pear");
 9.     System.out.println("1st after mango: " +    // sop 1
10.       myMap.higherKey("f"));
11.     System.out.println("1st after mango: " +    // sop 2
12.       myMap.ceilingKey("f"));
13.     System.out.println("1st after mango: " +    // sop 3
14.       myMap.floorKey("f"));
15.     SortedMap<String, String> sub = new TreeMap<String, String>();
16.     sub = myMap.tailMap("f");
17.     System.out.println("1st after mango: " +    // sop 4
18.       sub.firstKey());
19.   }
20. }
```

Which of the System.out.println statements will produce the output `1st after mango: p`? (Choose all that apply.)

A. `sop 1`

B. `sop 2`

C. `sop 3`

D. `sop 4`

E. None; compilation fails

F. None; an exception is thrown at runtime

14. Given:

```
3. import java.util.*;
4. class Business { }
5. class Hotel extends Business { }
6. class Inn extends Hotel { }
7. public class Travel {
8.   ArrayList<Hotel> go() {
9.     // insert code here
10.  }
11. }
```

Which, inserted independently at line 9, will compile? (Choose all that apply.)

A. `return new ArrayList<Inn>();`

B. `return new ArrayList<Hotel>();`

C. `return new ArrayList<Object>();`

D. `return new ArrayList<Business>();`

15. Given:

```
3. import java.util.*;
4. class Dog { int size; Dog(int s) { size = s; } }
5. public class FirstGrade {
6.   public static void main(String[] args) {
7.     TreeSet<Integer> i = new TreeSet<Integer>();
8.     TreeSet<Dog> d = new TreeSet<Dog>();
9.
10.    d.add(new Dog(1));   d.add(new Dog(2));   d.add(new Dog(1));
11.    i.add(1);            i.add(2);            i.add(1);
12.    System.out.println(d.size() + " " + i.size());
13.  }
14. }
```

What is the result?

A. 1 2

B. 2 2

C. 2 3

D. 3 2

E. 3 3

F. Compilation fails

G. An exception is thrown at runtime

16. Given:

```
3. import java.util.*;
4. public class GeoCache {
5.   public static void main(String[] args) {
6.     String[] s = {"map", "pen", "marble", "key"};
7.     Othello o = new Othello();
```

```
 8.      Arrays.sort(s,o);
 9.      for(String s2: s) System.out.print(s2 + " ");
10.      System.out.println(Arrays.binarySearch(s, "map"));
11.    }
12.    static class Othello implements Comparator<String> {
13.      public int compare(String a, String b) { return b.compareTo(a); }
14.    }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output will contain a 1

C. The output will contain a 2

D. The output will contain a –1

E. An exception is thrown at runtime

F. The output will contain `"key map marble pen"`

G. The output will contain `"pen marble map key"`

SELF TEST ANSWERS

1. Given:

```
public static void main(String[] args) {
  // INSERT DECLARATION HERE
  for (int i = 0; i <= 10; i++) {
    List<Integer> row = new ArrayList<Integer>();
    for (int j = 0; j <= 10; j++)
      row.add(i * j);
    table.add(row);
  }
  for (List<Integer> row : table)
    System.out.println(row);
  }
```

Which statements could be inserted at `// INSERT DECLARATION HERE` to allow this code to compile and run? (Choose all that apply.)

A. `List<List<Integer>> table = new List<List<Integer>>();`

B. `List<List<Integer>> table = new ArrayList<List<Integer>>();`

C. `List<List<Integer>> table = new ArrayList<ArrayList<Integer>>();`

D. `List<List, Integer> table = new List<List, Integer>();`

E. `List<List, Integer> table = new ArrayList<List, Integer>();`

F. `List<List, Integer> table = new ArrayList<ArrayList, Integer>();`

G. None of the above

Answer:

☑ **B** is correct.

☒ **A** is incorrect because List is an interface, so you can't say `new List()` regardless of any generic types. **D, E,** and **F** are incorrect because List only takes one type parameter (a Map would take two, not a List). **C** is tempting, but incorrect. The type argument `<List<Integer>>` must be the same for both sides of the assignment, even though the constructor `new ArrayList()` on the right side is a subtype of the declared type List on the left. (Objective 6.4)

2. Which statements are true about comparing two instances of the same class, given that the `equals()` and `hashCode()` methods have been properly overridden? (Choose all that apply.)

A. If the `equals()` method returns true, the `hashCode()` comparison `==` might return false

B. If the `equals()` method returns false, the `hashCode()` comparison `==` might return true

C. If the `hashCode()` comparison `==` returns true, the `equals()` method must return true

D. If the `hashCode()` comparison `==` returns true, the `equals()` method might return true

E. If the `hashCode()` comparison `!=` returns true, the `equals()` method might return true

Answer:

☑ **B** and **D**. **B** is true because often two dissimilar objects can return the same hashcode value. **D** is true because if the `hashCode()` comparison returns ==, the two objects might or might not be equal.

☒ **A, C,** and **E** are incorrect. **C** is incorrect because the `hashCode()` method is very flexible in its return values, and often two dissimilar objects can return the same hash code value. **A** and **E** are a negation of the `hashCode()` and `equals()` contract. (Objective 6.2)

3. Given:

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
  System.out.print(it.next() + " ");
}
```

Which statements are true?

A. The `before()` method will print `1 2`

B. The `before()` method will print `1 2 3`

C. The `before()` method will print three numbers, but the order cannot be determined

D. The `before()` method will not compile

E. The `before()` method will throw an exception at runtime

Answer:

☑ **E** is correct. You can't put both Strings and `int`s into the same TreeSet. Without generics, the compiler has no way of knowing what type is appropriate for this TreeSet, so it allows everything to compile. At runtime, the TreeSet will try to sort the elements as they're added, and when it tries to compare an Integer with a String it will throw a ClassCastException. Note that although the `before()` method does not use generics, it does use autoboxing. Watch out for code that uses some new features and some old features mixed together.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 6.5)

4. Given:

```
import java.util.*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos t1 = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(t1, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.println(m.size());
} }
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() { return 9; }
}
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile

B. As the code stands the output will be `2`

C. As the code stands the output will be `3`

D. If the `hashCode()` method is uncommented the output will be `2`

E. If the `hashCode()` method is uncommented the output will be `3`

F. If the `hashCode()` method is uncommented the code will not compile

Answer:

☑ **C** and **D** are correct. If `hashCode()` is not overridden then every entry will go into its own bucket, and the overridden `equals()` method will have no effect on determining equivalency. If `hashCode()` is overridden, then the overridden `equals()` method will view `t1` and `t2` as duplicates.

☒ **A, B, E,** and **F** are incorrect based on the above. (Objective 6.2)

5. Given:

```
12. public class AccountManager {
13.     private Map accountTotals = new HashMap();
14.     private int retirementFund;
15.
16.     public int getBalance(String accountName) {
17.         Integer total = (Integer) accountTotals.get(accountName);
18.         if (total == null)
19.             total = Integer.valueOf(0);
20.         return total.intValue();
21.     }
23.     public void setBalance(String accountName, int amount) {
24.         accountTotals.put(accountName, Integer.valueOf(amount));
25.  }  }
```

This class is to be updated to make use of appropriate generic types, with no changes in behavior (for better or worse). Which of these steps could be performed? (Choose three.)

A. Replace line 13 with
```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

B. Replace line 13 with
```
private Map<String, Integer> accountTotals = new HashMap<String, Integer>();
```

C. Replace line 13 with
```
private Map<String<Integer>> accountTotals = new HashMap<String<Integer>>();
```

D. Replace lines 17–20 with
```
int total = accountTotals.get(accountName);
if (total == null)  total = 0;
return total;
```

E. Replace lines 17–20 with
```
Integer total = accountTotals.get(accountName);
if (total == null)  total = 0;
return total;
```

F. Replace lines 17–20 with
```
return accountTotals.get(accountName);
```
G. Replace line 24 with
```
accountTotals.put(accountName, amount);
```
H. Replace line 24 with
```
accountTotals.put(accountName, amount.intValue());
```

Answer:

☑ **B, E,** and **G** are correct.

☒ **A** is wrong because you can't use a primitive type as a type parameter. **C** is wrong because a Map takes two type parameters separated by a comma. **D** is wrong because an int can't autobox to a null, and **F** is wrong because a `null` can't unbox to 0. **H** is wrong because you can't autobox a primitive just by trying to invoke a method with it. (Objective 6.4)

6. Given:
```
interface Hungry<E> { void munch(E x); }
interface Carnivore<E extends Animal> extends Hungry<E> {}
interface Herbivore<E extends Plant> extends Hungry<E> {}
abstract class Plant {}
class Grass extends Plant {}
abstract class Animal {}
class Sheep extends Animal implements Herbivore<Sheep> {
  public void munch(Sheep x) {}
}
class Wolf extends Animal implements Carnivore<Sheep> {
  public void munch(Sheep x) {}
}
```

Which of the following changes (taken separately) would allow this code to compile? (Choose all that apply.)

A. Change the Carnivore interface to
```
interface Carnivore<E extends Plant> extends Hungry<E> {}
```
B. Change the Herbivore interface to
```
interface Herbivore<E extends Animal> extends Hungry<E> {}
```
C. Change the Sheep class to
```
class Sheep extends Animal implements Herbivore<Plant> {
    public void munch(Grass x) {}
}
```

D. Change the Sheep class to

```
class Sheep extends Plant implements Carnivore<Wolf> {
    public void munch(Wolf x) {}
}
```

E. Change the Wolf class to

```
class Wolf extends Animal implements Herbivore<Grass> {
    public void munch(Grass x) {}
}
```

F. No changes are necessary

Answer:

☑ **B** is correct. The problem with the original code is that Sheep tries to implement `Herbivore<Sheep>` and Herbivore declares that its type parameter `E` can be any type that extends Plant. Since a Sheep is not a Plant, `Herbivore<Sheep>` makes no sense—the type Sheep is outside the allowed range of Herbivore's parameter `E`. Only solutions that either alter the definition of a Sheep or alter the definition of Herbivore will be able to fix this. So **A, E,** and **F** are eliminated. **B** works, changing the definition of an Herbivore to allow it to eat Sheep solves the problem. **C** doesn't work because an `Herbivore<Plant>` must have a `munch(Plant)` method, not `munch(Grass)`. And **D** doesn't work, because in **D** we made Sheep extend Plant, now the Wolf class breaks because its `munch(Sheep)` method no longer fulfills the contract of Carnivore. (Objective 6.4)

7. Which collection class(es) allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized? (Choose all that apply.)

A. `java.util.HashSet`
B. `java.util.LinkedHashSet`
C. `java.util.List`
D. `java.util.ArrayList`
E. `java.util.Vector`
F. `java.util.PriorityQueue`

Answer:

☑ **D** is correct. All of the collection classes allow you to grow or shrink the size of your collection. `ArrayList` provides an index to its elements. The newer collection classes tend not to have synchronized methods. Vector is an older implementation of `ArrayList` functionality and has synchronized methods; it is slower than `ArrayList`.

☒ **A, B, C, E,** and **F** are incorrect based on the logic described above; Notes: **C**, List is an interface, and **F**, `PriorityQueue` does not offer access by index. (Objective 6.1)

8. Given a method declared as

```
public static <E extends Number> List<E> process(List<E> nums)
```

A programmer wants to use this method like this

```
// INSERT DECLARATIONS HERE
output = process(input);
```

Which pairs of declarations could be placed at `// INSERT DECLARATIONS HERE` to allow the code to compile? (Choose all that apply.)

A.
```
ArrayList<Integer> input = null;
ArrayList<Integer> output = null;
```

B.
```
ArrayList<Integer> input = null;
List<Integer> output = null;
```

C.
```
ArrayList<Integer> input = null;
List<Number> output = null;
```

D.
```
List<Number> input = null;
ArrayList<Integer> output = null;
```

E.
```
List<Number> input = null;
List<Number> output = null;
```

F.
```
List<Integer> input = null;
List<Integer> output = null;
```

G. None of the above

Answer:

☑ **B, E,** and **F** are correct.

☒ The return type of process is definitely declared as a `List`, not an `ArrayList`, so **A** and **D** are wrong. **C** is wrong because the return type evaluates to `List<Integer>`, and that can't be assigned to a variable of type `List<Number>`. Of course all these would probably cause a `NullPointerException` since the variables are still null—but the question only asked us to get the code to compile. (Objective 6.4)

9. Given the proper import statement(s), and

```
13.    PriorityQueue<String> pq = new PriorityQueue<String>();
14.    pq.add("2");
15.    pq.add("4");
```

```
16.     System.out.print(pq.peek() + " ");
17.     pq.offer("1");
18.     pq.add("3");
19.     pq.remove("1");
20.     System.out.print(pq.poll() + " ");
21.     if(pq.remove("2")) System.out.print(pq.poll() + " ");
22.     System.out.println(pq.poll() + " " + pq.peek());
```

What is the result?

A. 2 2 3 3

B. 2 2 3 4

C. 4 3 3 4

D. 2 2 3 3 3

E. 4 3 3 3 3

F. 2 2 3 3 4

G. Compilation fails

H. An exception is thrown at runtime

Answer:

☑ **B** is correct. For the sake of the exam, `add()` and `offer()` both add to (in this case), naturally sorted queues. The calls to `poll()` both return and then remove the first item from the queue, so the if test fails.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (Objective 6.1)

10. Given:

```
3. import java.util.*;
4. public class Mixup {
5.   public static void main(String[] args) {
6.     Object o = new Object();
7.     // insert code here
8.     s.add("o");
9.     s.add(o);
10.  }
11. }
```

And these three fragments:

```
I.   Set s = new HashSet();
II.  TreeSet s = new TreeSet();
III. LinkedHashSet s = new LinkedHashSet();
```

When fragments I, II, or III are inserted, independently, at line 7, which are true? (Choose all that apply.)

A. Fragment I compiles

B. Fragment II compiles

C. Fragment III compiles

D. Fragment I executes without exception

E. Fragment II executes without exception

F. Fragment III executes without exception

Answer:

☑ **A, B, C, D,** and **F** are all correct.

☒ Only **E** is incorrect. Elements of a TreeSet must in some way implement `Comparable`. (Objective 6.1)

11. Given:

```
3. import java.util.*;
4. class Turtle {
5.   int size;
6.   public Turtle(int s) { size = s; }
7.   public boolean equals(Object o) { return (this.size == ((Turtle)o).size); }
8.   // insert code here
9. }
10. public class TurtleTest {
11.   public static void main(String[] args) {
12.     LinkedHashSet<Turtle> t = new LinkedHashSet<Turtle>();
13.     t.add(new Turtle(1));   t.add(new Turtle(2));   t.add(new Turtle(1));
14.     System.out.println(t.size());
15.   }
16. }
```

And these two fragments:

```
I.   public int hashCode() { return size/5; }
II.  // no hashCode method declared
```

If fragment I or II is inserted, independently, at line 8, which are true? (Choose all that apply.)

A. If fragment I is inserted, the output is 2

B. If fragment I is inserted, the output is 3

C. If fragment II is inserted, the output is 2

D. If fragment II is inserted, the output is 3

E. If fragment I is inserted, compilation fails

F. If fragment II is inserted, compilation fails

Answer:

☑ **A** and **D** are correct. While fragment II wouldn't fulfill the `hashCode()` contract (as you can see by the results), it is legal Java. For the purpose of the exam, if you don't override `hashCode()`, every object will have a unique hashcode.

☒ **B, C, E,** and **F** are incorrect based on the above. (Objective 6.2)

12. Given the proper import statement(s), and:

```
13.    TreeSet<String> s = new TreeSet<String>();
14.    TreeSet<String> subs = new TreeSet<String>();
15.    s.add("a"); s.add("b"); s.add("c"); s.add("d"); s.add("e");
16.
17.    subs = (TreeSet)s.subSet("b", true, "d", true);
18.    s.add("g");
19.    s.pollFirst();
20.    s.pollFirst();
21.    s.add("c2");
22.    System.out.println(s.size() +" "+ subs.size());
```

Which are true? (Choose all that apply.)

A. The size of `s` is 4

B. The size of `s` is 5

C. The size of `s` is 7

D. The size of `subs` is 1

E. The size of `subs` is 2

F. The size of `subs` is 3

G. The size of `subs` is 4

H. An exception is thrown at runtime

Answer:

☑ **B** and **F** are correct. After `"g"` is added, TreeSet `s` contains six elements and TreeSet `subs` contains three (`b`, `c`, `d`), because `"g"` is out of the range of `subs`. The first `pollFirst()` finds and removes only the `"a"`. The second `pollFirst()` finds and removes the `"b"` from *both* TreeSets (remember they are backed). The final `add()` is in range of both TreeSets. The final contents are [c,c2,d,e,g] and [c,c2,d].

☒ **A, C, D, E, G,** and **H** are incorrect based on the above. (Objective 6.3)

13. Given:

```
 3. import java.util.*;
 4. public class Magellan {
 5.   public static void main(String[] args) {
 6.     TreeMap<String, String> myMap = new TreeMap<String, String>();
 7.     myMap.put("a", "apple"); myMap.put("d", "date");
 8.     myMap.put("f", "fig"); myMap.put("p", "pear");
 9.     System.out.println("1st after mango: " +   // sop 1
10.       myMap.higherKey("f"));
11.     System.out.println("1st after mango: " +   // sop 2
12.       myMap.ceilingKey("f"));
13.     System.out.println("1st after mango: " +   // sop 3
14.       myMap.floorKey("f"));
15.     SortedMap<String, String> sub = new TreeMap<String, String>();
16.     sub = myMap.tailMap("f");
17.     System.out.println("1st after mango: " +   // sop 4
18.       sub.firstKey());
19.   }
20. }
```

Which of the System.out.println statements will produce the output `1st after mango: p`? (Choose all that apply.)

A. `sop 1`

B. `sop 2`

C. `sop 3`

D. `sop 4`

E. None; compilation fails

F. None; an exception is thrown at runtime

Answer:

☑ **A** is correct. The `ceilingKey()` method's argument is inclusive. The `floorKey()` method would be used to find keys before the specified key. The `firstKey()` method's argument is also inclusive.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 6.3)

14. Given:

```
3. import java.util.*;
4. class Business { }
5. class Hotel extends Business { }
6. class Inn extends Hotel { }
7. public class Travel {
8.   ArrayList<Hotel> go() {
9.     // insert code here
10.  }
11. }
```

Which, inserted independently at line 9, will compile? (Choose all that apply.)

A. `return new ArrayList<Inn>();`

B. `return new ArrayList<Hotel>();`

C. `return new ArrayList<Object>();`

D. `return new ArrayList<Business>();`

Answer:

☑ **B** is correct.

☒ **A** is incorrect because polymorphic assignments don't apply to generic type parameters. **C** and **D** are incorrect because they don't follow basic polymorphism rules. (Objective 6.4)

15. Given:

```
3. import java.util.*;
4. class Dog { int size; Dog(int s) { size = s; } }
5. public class FirstGrade {
6.   public static void main(String[] args) {
7.     TreeSet<Integer> i = new TreeSet<Integer>();
8.     TreeSet<Dog> d = new TreeSet<Dog>();
9.
10.    d.add(new Dog(1));   d.add(new Dog(2));   d.add(new Dog(1));
11.    i.add(1);            i.add(2);            i.add(1);
12.    System.out.println(d.size() + " " + i.size());
13.  }
14. }
```

What is the result?

A. 1 2

B. 2 2

C. 2 3

D. 3 2

E. 3 3

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **G** is correct. Class `Dog` needs to implement `Comparable` in order for a TreeSet (which keeps its elements sorted) to be able to contain `Dog` objects.

☒ **A, B, C, D, E,** and **F** are incorrect based on the above. (Objective 6.5)

16. Given:

```
3. import java.util.*;
4. public class GeoCache {
5.   public static void main(String[] args) {
6.     String[] s = {"map", "pen", "marble", "key"};
7.     Othello o = new Othello();
8.     Arrays.sort(s,o);
```

```
 9.      for(String s2: s) System.out.print(s2 + " ");
10.      System.out.println(Arrays.binarySearch(s, "map"));
11.    }
12.    static class Othello implements Comparator<String> {
13.      public int compare(String a, String b) { return b.compareTo(a); }
14.    }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output will contain a 1

C. The output will contain a 2

D. The output will contain a –1

E. An exception is thrown at runtime

F. The output will contain `"key map marble pen"`

G. The output will contain `"pen marble map key"`

Answer:

☑ **D** and **G** are correct. First, the `compareTo()` method will reverse the normal sort. Second, the `sort()` is valid. Third, the `binarySearch()` gives –1 because it needs to be invoked using the same `Comparator (o)`, as was used to sort the array. Note that when the `binarySearch()` returns an "undefined result" it doesn't officially have to be a -1, but it usually is, so if you selected only G, you get full credit!

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 6.5)

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Inner Classes

- A "regular" inner class is declared *inside* the curly braces of another class, but *outside* any method or other code block.
- An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the `abstract` or `final` modifiers. (Never both `abstract` and `final` together— remember that `abstract` *must* be subclassed, whereas `final` *cannot* be subclassed).
- An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to *all* of the outer class's members, including those marked `private`.
- To instantiate an inner class, you must have a reference to an instance of the outer class.
- From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

- From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

```
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
```

- From code within the inner class, the keyword `this` holds a reference to the inner class instance. To reference the *outer* `this` (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword `this` with the outer class name as follows: `MyOuter.this;`

Method-Local Inner Classes

- A method-local inner class is defined within a method of the enclosing class.
- For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but *after* the class definition code.
- A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked `final`.

- The only modifiers you can apply to a method-local inner class are `abstract` and `final`. (Never both at the same time, though.)

Anonymous Inner Classes

- Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
- An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
- Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
- An anonymous inner class can extend one subclass *or* implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class *and* implement an interface, nor can it implement more than one interface.
- An argument-defined inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: `});`

Static Nested Classes

- Static nested classes are inner classes marked with the `static` modifier.
- A static nested class is *not* an inner class, it's a top-level nested class.
- Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
- Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

- A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an *outer* `this` reference).

SELF TEST

The following questions will help you measure your understanding of the dynamic and life-altering material presented in this chapter. Read all of the choices carefully. Take your time. Breathe.

1. Which are true about a static nested class? (Choose all that apply.)

A. You must have a reference to an instance of the enclosing class in order to instantiate it
B. It does not have access to non-`static` members of the enclosing class
C. Its variables and methods must be `static`
D. If the outer class is named `MyOuter`, and the nested class is named `MyInner`, it can be instantiated using `new MyOuter.MyInner();`
E. It must extend the enclosing class

2. Given:

```
class Boo {
   Boo(String s) { }
   Boo() { }
}
class Bar extends Boo {
   Bar() { }
   Bar(String s) {super(s);}
   void zoo() {
   // insert code here
   }
}
```

Which create an anonymous inner class from within class Bar? (Choose all that apply.)

A. `Boo f = new Boo(24) { };`
B. `Boo f = new Bar() { };`
C. `Boo f = new Boo() {String s; };`
D. `Bar f = new Boo(String s) { };`
E. `Boo f = new Boo.Bar(String s) { };`

3. Which are true about a method-local inner class? (Choose all that apply.)

A. It must be marked `final`
B. It can be marked `abstract`

C. It can be marked `public`
D. It can be marked `static`
E. It can access private members of the enclosing class

4. Given:

```
1. public class TestObj {
2.   public static void main(String[] args) {
3.     Object o = new Object() {
4.       public boolean equals(Object obj) {
5.         return true;
6.       }
7.     }
8.     System.out.println(o.equals("Fred"));
9.   }
10. }
```

What is the result?

A. An exception occurs at runtime
B. `true`
C. `Fred`
D. Compilation fails because of an error on line 3
E. Compilation fails because of an error on line 4
F. Compilation fails because of an error on line 8
G. Compilation fails because of an error on a line other than 3, 4, or 8

5. Given:

```
1. public class HorseTest {
2.   public static void main(String[] args) {
3.     class Horse {
4.       public String name;
5.       public Horse(String s) {
6.         name = s;
7.       }
8.     }
9.     Object obj = new Horse("Zippo");
10.    System.out.println(obj.name);
11.  }
12. }
```

What is the result?

A. An exception occurs at runtime at line 10
B. `Zippo`
C. Compilation fails because of an error on line 3
D. Compilation fails because of an error on line 9
E. Compilation fails because of an error on line 10

6. Given:

```
public abstract class AbstractTest {
   public int getNum() {
      return 45;
   }
   public abstract class Bar {
     public int getNum() {
       return 38;
     }
   }
   public static void main(String[] args) {
      AbstractTest t = new AbstractTest() {
         public int getNum() {
           return 22;
         }
      };
      AbstractTest.Bar f = t.new Bar() {
         public int getNum() {
           return 57;
         }
      };
      System.out.println(f.getNum() + " " + t.getNum());
   }
}
```

What is the result?

A. `57 22`
B. `45 38`
C. `45 57`
D. An exception occurs at runtime
E. Compilation fails

7. Given:

```
3. public class Tour {
4.   public static void main(String[] args) {
5.     Cathedral c = new Cathedral();
6.     // insert code here
7.     s.go();
8.   }
9. }
10. class Cathedral {
11.   class Sanctum {
12.     void go() { System.out.println("spooky"); }
13.   }
14. }
```

Which, inserted independently at line 6, compile and produce the output "spooky"? (Choose all that apply.)

A. `Sanctum s = c.new Sanctum();`
B. `c.Sanctum s = c.new Sanctum();`
C. `c.Sanctum s = Cathedral.new Sanctum();`
D. `Cathedral.Sanctum s = c.new Sanctum();`
E. `Cathedral.Sanctum s = Cathedral.new Sanctum();`

8. Given:

```
5. class A { void m() { System.out.println("outer"); } }
6.
7. public class TestInners {
8.   public static void main(String[] args) {
9.     new TestInners().go();
10.   }
11.   void go() {
12.     new A().m();
13.     class A { void m() { System.out.println("inner"); } }
14.   }
15.   class A { void m() { System.out.println("middle"); } }
16. }
```

What is the result?

A. `inner`
B. `outer`

C. `middle`
D. Compilation fails
E. An exception is thrown at runtime

9. Given:

```
3. public class Car {
4.   class Engine {
5.     // insert code here
6.   }
7.   public static void main(String[] args) {
8.     new Car().go();
9.   }
10.  void go() {
11.    new Engine();
12.  }
13.  void drive() { System.out.println("hi"); }
14. }
```

Which, inserted independently at line 5, produce the output "hi"? (Choose all that apply.)

A. `{ Car.drive(); }`
B. `{ this.drive(); }`
C. `{ Car.this.drive(); }`
D. `{ this.Car.this.drive(); }`
E. `Engine() { Car.drive(); }`
F. `Engine() { this.drive(); }`
G. `Engine() { Car.this.drive(); }`

10. Given:

```
3. public class City {
4.   class Manhattan {
5.     void doStuff() throws Exception { System.out.print("x "); }
6.   }
7.   class TimesSquare extends Manhattan {
8.     void doStuff() throws Exception { }
9.   }
10.  public static void main(String[] args) throws Exception {
11.    new City().go();
12.  }
13.  void go() throws Exception { new TimesSquare().doStuff(); }
14. }
```

What is the result?

A. `x`
B. `x x`
C. No output is produced
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 7
G. Compilation fails due only to an error on line 10
H. Compilation fails due only to an error on line 13

11. Given:

```
3. public class Navel {
4.   private int size = 7;
5.   private static int length = 3;
6.   public static void main(String[] args) {
7.     new Navel().go();
8.   }
9.   void go() {
10.    int size = 5;
11.    System.out.println(new Gazer().adder());
12.  }
13.  class Gazer {
14.    int adder() { return size * length; }
15.  }
16. }
```

What is the result?

A. `15`
B. `21`
C. An exception is thrown at runtime
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 5

12. Given:

```
3. import java.util.*;
4. public class Pockets {
5.   public static void main(String[] args) {
6.     String[] sa = {"nickel", "button", "key", "lint"};
7.     Sorter s = new Sorter();
8.     for(String s2: sa) System.out.print(s2 + " ");
9.     Arrays.sort(sa,s);
10.    System.out.println();
11.    for(String s2: sa) System.out.print(s2 + " ");
12.  }
13.  class Sorter implements Comparator<String> {
14.    public int compare(String a, String b) {
15.      return b.compareTo(a);
16.    }
17.  }
18. }
```

What is the result?

A. Compilation fails

B. `button key lint nickel`
`nickel lint key button`

C. `nickel button key lint`
`button key lint nickel`

D. `nickel button key lint`
`nickel button key lint`

E. `nickel button key lint`
`nickel lint key button`

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. Which are true about a static nested class? (Choose all that apply.)
   A. You must have a reference to an instance of the enclosing class in order to instantiate it
   B. It does not have access to non-`static` members of the enclosing class
   C. Its variables and methods must be `static`
   D. If the outer class is named `MyOuter`, and the nested class is named `MyInner`, it can be instantiated using `new MyOuter.MyInner();`
   E. It must extend the enclosing class

   Answer:

   ☑ **B** and **D. B** is correct because a static nested class is not tied to an instance of the enclosing class, and thus can't access the non-`static` members of the class (just as a `static` method can't access non-static members of a class). **D** uses the correct syntax for instantiating a static nested class.

   ☒ **A** is incorrect because static nested classes do not need (and can't use) a reference to an instance of the enclosing class. **C** is incorrect because static nested classes can declare and define non-static members. **E** is wrong because…it just is. There's no rule that says an inner or nested class has to extend anything.

2. Given:

```
class Boo {
   Boo(String s) { }
   Boo() { }
}
class Bar extends Boo {
   Bar() { }
   Bar(String s) {super(s);}
   void zoo() {
   // insert code here
   }
}
```

   Which create an anonymous inner class from within class Bar? (Choose all that apply.)
   A. `Boo f = new Boo(24) { };`
   B. `Boo f = new Bar() { };`

C. `Boo f = new Boo() {String s; };`
D. `Bar f = new Boo(String s) { };`
E. `Boo f = new Boo.Bar(String s) { };`

Answer:

☑ **B** and **C. B** is correct because anonymous inner classes are no different from any other class when it comes to polymorphism. That means you are always allowed to declare a reference variable of the superclass type and have that reference variable refer to an instance of a subclass type, which in this case is an anonymous subclass of Bar. Since Bar is a subclass of Boo, it all works. **C** uses correct syntax for creating an instance of Boo.

☒ **A** is incorrect because it passes an `int` to the Boo constructor, and there is no matching constructor in the Boo class. **D** is incorrect because it violates the rules of polymorphism; you cannot refer to a superclass type using a reference variable declared as the subclass type. The superclass doesn't have everything the subclass has. **E** uses incorrect syntax.

3. Which are true about a method-local inner class? (Choose all that apply.)

A. It must be marked `final`
B. It can be marked `abstract`
C. It can be marked `public`
D. It can be marked `static`
E. It can access private members of the enclosing class

Answer:

☑ **B** and **E. B** is correct because a method-local inner class can be `abstract`, although it means a subclass of the inner class must be created if the `abstract` class is to be used (so an `abstract` method-local inner class is probably not useful). **E** is correct because a method-local inner class works like any other inner class—it has a special relationship to an instance of the enclosing class, thus it can access all members of the enclosing class.

☒ **A** is incorrect because a method-local inner class does not have to be declared `final` (although it is legal to do so). **C** and **D** are incorrect because a method-local inner class cannot be made `public` (remember—local variables can't be `public`) or `static`.

4. Given:

```
1. public class TestObj {
2.   public static void main(String[] args) {
3.     Object o = new Object() {
```

```
4.        public boolean equals(Object obj) {
5.          return true;
6.        }
7.      }
8.      System.out.println(o.equals("Fred"));
9.    }
10. }
```

What is the result?

A. An exception occurs at runtime
B. `true`
C. `fred`
D. Compilation fails because of an error on line 3
E. Compilation fails because of an error on line 4
F. Compilation fails because of an error on line 8
G. Compilation fails because of an error on a line other than 3, 4, or 8

Answer:

☑ **G**. This code would be legal if line 7 ended with a semicolon. Remember that line 3 is a statement that doesn't end until line 7, and a statement needs a closing semicolon!

☒ **A, B, C, D, E,** and **F** are incorrect based on the program logic described above. If the semicolon were added at line 7, then answer **B** would be correct—the program would print `true`, the return from the `equals()` method overridden by the anonymous subclass of `Object`.

5. Given:

```
1. public class HorseTest {
2.   public static void main(String[] args) {
3.     class Horse {
4.       public String name;
5.       public Horse(String s) {
6.         name = s;
7.       }
8.     }
9.     Object obj = new Horse("Zippo");
10.    System.out.println(obj.name);
11.  }
12. }
```

What is the result?

A. An exception occurs at runtime at line 10
B. `Zippo`
C. Compilation fails because of an error on line 3
D. Compilation fails because of an error on line 9
E. Compilation fails because of an error on line 10

Answer:

☑ **E.** If you use a reference variable of type `Object`, you can access only those members defined in class `Object`.

☒ **A, B, C,** and **D** are incorrect based on the program logic described above.

6. Given:

```
public abstract class AbstractTest {
   public int getNum() {
      return 45;
   }
   public abstract class Bar {
     public int getNum() {
       return 38;
     }
   }
   public static void main(String[] args) {
      AbstractTest t = new AbstractTest() {
         public int getNum() {
           return 22;
         }
      };
      AbstractTest.Bar f = t.new Bar() {
         public int getNum() {
           return 57;
         }
      };
      System.out.println(f.getNum() + " " + t.getNum());
} }
```

What is the result?

A. `57 22`
B. `45 38`
C. `45 57`
D. An exception occurs at runtime
E. Compilation fails

Answer:

☑ **A**. You can define an inner class as `abstract`, which means you can instantiate only concrete subclasses of the abstract inner class. The object referenced by the variable `t` is an instance of an anonymous subclass of AbstractTest, and the anonymous class overrides the `getNum()` method to return `22`. The variable referenced by `f` is an instance of an anonymous subclass of `Bar`, and the anonymous Bar subclass also overrides the `getNum()` method (to return `57`). Remember that to create a Bar instance, we need an instance of the enclosing AbstractTest class to tie to the new `Bar` inner class instance. AbstractTest can't be instantiated because it's `abstract`, so we created an anonymous subclass (non-`abstract`) and then used the instance of that anonymous subclass to tie to the new Bar subclass instance.

☒ **B, C, D,** and **E** are incorrect based on the program logic described above.

7. Given:

```
3. public class Tour {
4.   public static void main(String[] args) {
5.     Cathedral c = new Cathedral();
6.     // insert code here
7.     s.go();
8.   }
9. }
10. class Cathedral {
11.   class Sanctum {
12.     void go() { System.out.println("spooky"); }
13.   }
14. }
```

Which, inserted independently at line 6, compile and produce the output "spooky"? (Choose all that apply.)

A. `Sanctum s = c.new Sanctum();`
B. `c.Sanctum s = c.new Sanctum();`
C. `c.Sanctum s = Cathedral.new Sanctum();`
D. `Cathedral.Sanctum s = c.new Sanctum();`
E. `Cathedral.Sanctum s = Cathedral.new Sanctum();`

Answer:

☑ **D** is correct. It is the only code that uses the correct inner class instantiation syntax.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 1.1)

8. Given:

```
 5. class A { void m() { System.out.println("outer"); } }
 6.
 7. public class TestInners {
 8.   public static void main(String[] args) {
 9.     new TestInners().go();
10.   }
11.   void go() {
12.     new A().m();
13.     class A { void m() { System.out.println("inner"); } }
14.   }
15.   class A { void m() { System.out.println("middle"); } }
16. }
```

What is the result?

A. `inner`
B. `outer`
C. `middle`
D. Compilation fails
E. An exception is thrown at runtime

Answer:

☑ **C** is correct. The `"inner"` version of `class A` isn't used because its declaration comes after the instance of `class A` is created in the `go()` method.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 1.1)

9. Given:

```
 3. public class Car {
 4.   class Engine {
 5.     // insert code here
 6.   }
 7.   public static void main(String[] args) {
 8.     new Car().go();
 9.   }
10.   void go() {
11.     new Engine();
12.   }
13.   void drive() { System.out.println("hi"); }
14. }
```

Which, inserted independently at line 5, produce the output "hi"? (Choose all that apply.)

A. `{ Car.drive(); }`
B. `{ this.drive(); }`
C. `{ Car.this.drive(); }`
D. `{ this.Car.this.drive(); }`
E. `Engine() { Car.drive(); }`
F. `Engine() { this.drive(); }`
G. `Engine() { Car.this.drive(); }`

Answer:

☑ **C** and **G** are correct. **C** is the correct syntax to access an inner class's outer instance method from an initialization block, and **G** is the correct syntax to access it from a constructor.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objectives 1.1, 1.4)

10. Given:

```
3. public class City {
4.   class Manhattan {
5.     void doStuff() throws Exception { System.out.print("x "); }
6.   }
7.   class TimesSquare extends Manhattan {
8.     void doStuff() throws Exception { }
9.   }
10.  public static void main(String[] args) throws Exception {
11.    new City().go();
12.  }
13.  void go() throws Exception { new TimesSquare().doStuff(); }
14. }
```

What is the result?

A. `x`
B. `x x`
C. No output is produced
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 7
G. Compilation fails due only to an error on line 10
H. Compilation fails due only to an error on line 13

Answer:

☑ **C** is correct. The inner classes are valid, and all the methods (including `main()`), correctly throw an Exception, given that `doStuff()` throws an Exception. The `doStuff()` in class `TimesSquare` overrides class `Manhattan`'s `doStuff()` and produces no output.

☒ **A, B, D, E, F, G,** and **H** are incorrect based on the above. (Objectives 1.1, 2.4)

11. Given:

```
3. public class Navel {
4.   private int size = 7;
5.   private static int length = 3;
6.   public static void main(String[] args) {
7.     new Navel().go();
8.   }
9.   void go() {
10.    int size = 5;
11.    System.out.println(new Gazer().adder());
12.  }
13.  class Gazer {
14.    int adder() { return size * length; }
15.  }
16. }
```

What is the result?

A. `15`
B. `21`
C. An exception is thrown at runtime
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 5

Answer:

☑ **B** is correct. The inner class `Gazer` has access to `Navel`'s `private static` and private instance variables.

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objectives 1.1, 1.4)

12. Given:

```
 3. import java.util.*;
 4. public class Pockets {
 5.   public static void main(String[] args) {
 6.     String[] sa = {"nickel", "button", "key", "lint"};
 7.     Sorter s = new Sorter();
 8.     for(String s2: sa) System.out.print(s2 + " ");
 9.     Arrays.sort(sa,s);
10.     System.out.println();
11.     for(String s2: sa) System.out.print(s2 + " ");
12.   }
13.   class Sorter implements Comparator<String> {
14.     public int compare(String a, String b) {
15.       return b.compareTo(a);
16.     }
17.   }
18. }
```

What is the result?

A. Compilation fails

B.
```
button key lint nickel
nickel lint key button
```

C.
```
nickel button key lint
button key lint nickel
```

D.
```
nickel button key lint
nickel button key lint
```

E.
```
nickel button key lint
nickel lint key button
```

F. An exception is thrown at runtime

Answer:

☑ **A** is correct, the inner class `Sorter` must be declared `static` to be called from the `static` method `main()`. If `Sorter` had been `static`, answer E would be correct.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objectives 1.1, 1.4, 6.5)

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. Photocopy it and sleep with it under your pillow for complete absorption.

Defining, Instantiating, and Starting Threads (Objective 4.1)

- Threads can be created by extending Thread and overriding the `public void run()` method.
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target* of the thread.
- You can call `start()` on a Thread object only once. If `start()` is called more than once on a Thread object, it will throw a RuntimeException.
- It is legal to create many Thread objects using the same Runnable object as the target.
- When a Thread object is created, it does not become a *thread of execution* until its `start()` method is invoked. When a Thread object exists but hasn't been started, it is in the *new* state and is not considered *alive*.

Transitioning Between Thread States (Objective 4.2)

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.
- For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run.
- There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the `sleep()` method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, `yield()` works well enough to encourage your threads to play together nicely.)
- A running thread may enter a blocked/waiting state by a `wait()`, `sleep()`, or `join()` call.

- ❑ A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
- ❑ When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will *go* directly from waiting to running (well, for all practical purposes anyway).
- ❑ A dead thread cannot be started again.

Sleep, Yield, and Join (Objective 4.2)

- ❑ Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- ❑ A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the `sleep()` method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- ❑ The `sleep()` method is a `static` method that sleeps the currently executing thread's state. One thread *cannot* tell another thread to sleep.
- ❑ The `setPriority()` method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels—some levels may be treated as effectively equal.
- ❑ If not explicitly set, a thread's priority will have the same priority as the priority of the thread that created it.
- ❑ The `yield()` method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. A thread might yield and then immediately reenter the running state.
- ❑ The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
- ❑ When one thread calls the `join()` method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the `join()` method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Concurrent Access Problems and Synchronized Threads (Objective 4.3)

- `synchronized` methods prevent more than one thread from accessing an object's critical method code simultaneously.
- You can use the `synchronized` keyword as a method modifier, or to start a synchronized block of code.
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's *un*synchronized code.
- When a thread goes to sleep, its locks will be unavailable to other threads.
- `static` methods can be `synchronized`, using the lock from the java.lang.Class instance representing that class.

Communicating with Objects by Waiting and Notifying (Objective 4.4)

- The `wait()` method lets a thread say, "there's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about." Basically, a `wait()` call means "wait me in your pool," or "add me to your waiting list."
- The `notify()` method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- The `notify()` method can NOT specify which waiting thread to notify.
- The method `notifyAll()` works in the same way as `notify()`, only it sends the signal to *all* of the threads waiting on the object.
- All three methods—`wait()`, `notify()`, and `notifyAll()`—must be called from within a `synchronized` context! A thread invokes `wait()` or `notify()` on a particular object, and the thread must currently hold the lock on that object.

Deadlocked Threads (Objective 4.3)

- Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will *never* be released!
- Deadlocking is bad. Don't do it.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. If you have a rough time with some of these at first, don't beat yourself up. Some of these questions are long and intricate, expect long and intricate questions on the real exam too!

1. The following block of code creates a Thread using a Runnable target:

```
Runnable target = new MyRunnable();
Thread myThread = new Thread(target);
```

Which of the following classes can be used to create the target, so that the preceding code compiles correctly?

A. `public class MyRunnable extends Runnable{public void run(){}}`
B. `public class MyRunnable extends Object{public void run(){}}`
C. `public class MyRunnable implements Runnable{public void run(){}}`
D. `public class MyRunnable implements Runnable{void run(){}}`
E. `public class MyRunnable implements Runnable{public void start(){}}`

2. Given:

```
3.  class MyThread extends Thread {
4.     public static void main(String [] args) {
5.        MyThread t = new MyThread();
6.        Thread x = new Thread(t);
7.        x.start();
8.     }
9.     public void run() {
10.       for(int i=0;i<3;++i) {
11.          System.out.print(i + "..");
12.       }
13.    }
14. }
```

What is the result of this code?

A. Compilation fails
B. 1..2..3..
C. 0..1..2..3..
D. 0..1..2..
E. An exception occurs at runtime

3. Given:

```
3.  class Test {
4.     public static void main(String [] args) {
5.        printAll(args);
6.     }
7.     public static void printAll(String[] lines) {
8.        for(int i=0;i<lines.length;i++){
9.           System.out.println(lines[i]);
10.          Thread.currentThread().sleep(1000);
11.       }
12.    }
13. }
```

The `static` method `Thread.currentThread()` returns a reference to the currently executing Thread object. What is the result of this code?

A. Each String in the array `lines` will output, with a 1-second pause between lines

B. Each String in the array `lines` will output, with no pause in between because this method is not executed in a Thread

C. Each String in the array `lines` will output, and there is no guarantee there will be a pause because `currentThread()` may not retrieve this thread

D. This code will not compile

E. Each String in the `lines` array will print, with at least a one-second pause between lines

4. Assume you have a class that holds two `private` variables: `a` and `b`. Which of the following pairs can prevent concurrent access problems in that class? (Choose all that apply.)

A.
```
public int read(){return a+b;}
public void set(int a, int b){this.a=a;this.b=b;}
```

B.
```
public synchronized int read(){return a+b;}
public synchronized void set(int a, int b){this.a=a;this.b=b;}
```

C.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(a){this.a=a;this.b=b;}}
```

D.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(b){this.a=a;this.b=b;}}
```

E.
```
public synchronized(this) int read(){return a+b;}
public synchronized(this) void set(int a, int b){this.a=a;this.b=b;}
```

F.
```
public int read(){synchronized(this){return a+b;}}
public void set(int a, int b){synchronized(this){this.a=a;this.b=b;}}
```

5. Given:

```
1.  public class WaitTest {
2.     public static void main(String [] args) {
3.        System.out.print("1 ");
4.        synchronized(args){
5.           System.out.print("2 ");
6.           try {
7.              args.wait();
8.           }
9.           catch(InterruptedException e){}
10.       }
11.       System.out.print("3 ");
12.    }
13. }
```

What is the result of trying to compile and run this program?

A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7

B. `1 2 3`

C. `1 3`

D. `1 2`

E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait

F. It will fail to compile because it has to be synchronized on the `this` object

6. Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will the thread A become a candidate to get another turn at the CPU?

A. After object B is notified, or after two seconds

B. After the lock on B is released, or after two seconds

C. Two seconds after object B is notified

D. Two seconds after lock B is released

7. Which are true? (Choose all that apply.)

A. The `notifyAll()` method must be called from a synchronized context

B. To call `wait()`, an object must own the lock on the thread

C. The `notify()` method is defined in class `java.lang.Thread`

D. When a thread is waiting as a result of `wait()`, it releases its lock

E. The `notify()` method causes a thread to immediately release its lock

F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on

8. Given the scenario: This class is intended to allow users to write a series of messages, so that each message is identified with a timestamp and the name of the thread that wrote the message:

```
public class Logger {
    private StringBuilder contents = new StringBuilder();
    public void log(String message) {
        contents.append(System.currentTimeMillis());
        contents.append(": ");
        contents.append(Thread.currentThread().getName());
        contents.append(message);
        contents.append("\n");
    }
    public String getContents() { return contents.toString(); }
}
```

How can we ensure that instances of this class can be safely used by multiple threads?

A. This class is already thread-safe

B. Replacing `StringBuilder` with `StringBuffer` will make this class thread-safe

C. Synchronize the `log()` method only

D. Synchronize the `getContents()` method only

E. Synchronize both `log()` and `getContents()`

F. This class cannot be made thread-safe

9. Given:

```
public static synchronized void main(String[] args) throws
 InterruptedException {
    Thread t = new Thread();
    t.start();
    System.out.print("X");
    t.wait(10000);
    System.out.print("Y");
}
```

What is the result of this code?

A. It prints `X` and exits

B. It prints `X` and never exits

C. It prints `XY` and exits almost immeditately

D. It prints `XY` with a 10-second delay between `X` and `Y`

E. It prints `XY` with a 10000-second delay between `X` and `Y`

F. The code does not compile

G. An exception is thrown at runtime

10. Given:

```
class MyThread extends Thread {
   MyThread() {
     System.out.print(" MyThread");
   }
   public void run() {
     System.out.print(" bar");
   }
   public void run(String s) {
      System.out.print(" baz");
  }
}
public class TestThreads {
  public static void main (String [] args) {
    Thread t = new MyThread() {
      public void run() {
        System.out.print(" foo");
      }
    };
    t.start();
} }
```

What is the result?

A. `foo`
B. `MyThread foo`
C. `MyThread bar`
D. `foo bar`
E. `foo bar baz`
F. `bar foo`
G. Compilation fails
H. An exception is thrown at runtime

11. Given:

```
public class ThreadDemo {
    synchronized void a() { actBusy(); }
    static synchronized void b() { actBusy(); }
    static void actBusy() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
    public static void main(String[] args) {
        final ThreadDemo x = new ThreadDemo();
        final ThreadDemo y = new ThreadDemo();
        Runnable runnable = new Runnable() {
            public void run() {
                int option = (int) (Math.random() * 4);
                switch (option) {
                    case 0: x.a(); break;
                    case 1: x.b(); break;
                    case 2: y.a(); break;
                    case 3: y.b(); break;
                }
            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
    }
}
```

Which of the following pairs of method invocations could NEVER be executing at the same time? (Choose all that apply.)

A. `x.a()` in `thread1`, and `x.a()` in `thread2`

B. `x.a()` in `thread1`, and `x.b()` in `thread2`

C. `x.a()` in `thread1`, and `y.a()` in `thread2`

D. `x.a()` in `thread1`, and `y.b()` in `thread2`

E. `x.b()` in `thread1`, and `x.a()` in `thread2`

F. `x.b()` in `thread1`, and `x.b()` in `thread2`

G. `x.b()` in `thread1`, and `y.a()` in `thread2`

H. `x.b()` in `thread1`, and `y.b()` in `thread2`

12. Given:

```
public class TwoThreads {
    static Thread laurel, hardy;
    public static void main(String[] args) {
        laurel = new Thread() {
            public void run() {
                System.out.println("A");
                try {
                    hardy.sleep(1000);
                } catch (Exception e) {
                    System.out.println("B");
                }
                System.out.println("C");
            }
        };
        hardy = new Thread() {
            public void run() {
                System.out.println("D");
                try {
                    laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
            }
        };
        laurel.start();
        hardy.start();
    }
}
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. `A`
B. `B`
C. `C`
D. `D`
E. `E`
F. `F`
G. The answer cannot be reliably determined
H. The code does not compile

13. Given:

```
3. public class Starter implements Runnable {
4.   void go(long id) {
5.     System.out.println(id);
6.   }
7.   public static void main(String[] args) {
8.     System.out.print(Thread.currentThread().getId() + " ");
9.     // insert code here
10.  }
11.  public void run() { go(Thread.currentThread().getId()); }
12. }
```

And given the following five fragments:

```
I.   new Starter().run();
II.  new Starter().start();
III. new Thread(new Starter());
IV.  new Thread(new Starter()).run();
V.   new Thread(new Starter()).start();
```

When the five fragments are inserted, one at a time at line 9, which are true? (Choose all that apply.)

A. All five will compile
B. Only one might produce the output `4 4`
C. Only one might produce the output `4 2`
D. Exactly two might produce the output `4 4`
E. Exactly two might produce the output `4 2`
F. Exactly three might produce the output `4 4`
G. Exactly three might produce the output `4 2`

14. Given:

```
 3. public class Leader implements Runnable {
 4.   public static void main(String[] args) {
 5.     Thread t = new Thread(new Leader());
 6.     t.start();
 7.     System.out.print("m1 ");
 8.     t.join();
 9.     System.out.print("m2 ");
10.   }
11.   public void run() {
12.     System.out.print("r1 ");
13.     System.out.print("r2 ");
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. The output could be `r1 r2 m1 m2`
C. The output could be `m1 m2 r1 r2`
D. The output could be `m1 r1 r2 m2`
E. The output could be `m1 r1 m2 r2`
F. An exception is thrown at runtime

15. Given:

```
 3. class Dudes {
 4.   static long flag = 0;
 5.   // insert code here
 6.     if(flag == 0) flag = id;
 7.     for(int x = 1; x < 3; x++) {
 8.       if(flag == id) System.out.print("yo ");
 9.       else System.out.print("dude ");
10.     }
11.   }
12. }
13. public class DudesChat implements Runnable {
14.   static Dudes d;
15.   public static void main(String[] args) {
16.     new DudesChat().go();
17.   }
18.   void go() {
19.     d = new Dudes();
```

```
20.      new Thread(new DudesChat()).start();
21.      new Thread(new DudesChat()).start();
22.    }
23.    public void run() {
24.      d.chat(Thread.currentThread().getId());
25.    }
26. }
```

And given these two fragments:

```
I.  synchronized void chat(long id) {
II. void chat(long id) {
```

When fragment I or fragment II is inserted at line 5, which are true? (Choose all that apply.)

A. An exception is thrown at runtime
B. With fragment I, compilation fails
C. With fragment II, compilation fails
D. With fragment I, the output could be `yo dude dude yo`
E. With fragment I, the output could be `dude dude yo yo`
F. With fragment II, the output could be `yo dude dude yo`

16. Given:

```
 3. class Chicks {
 4.   synchronized void yack(long id) {
 5.     for(int x = 1; x < 3; x++) {
 6.       System.out.print(id + " ");
 7.       Thread.yield();
 8.     }
 9.   }
10. }
11. public class ChicksYack implements Runnable {
12.   Chicks c;
13.   public static void main(String[] args) {
14.     new ChicksYack().go();
15.   }
16.   void go() {
17.     c = new Chicks();
18.     new Thread(new ChicksYack()).start();
19.     new Thread(new ChicksYack()).start();
20.   }
21.   public void run() {
```

```
22.      c.yack(Thread.currentThread().getId());
23.    }
24. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. The output could be `4 4 2 3`
C. The output could be `4 4 2 2`
D. The output could be `4 4 4 2`
E. The output could be `2 2 4 4`
F. An exception is thrown at runtime

17. Given:

```
3. public class Chess implements Runnable {
4.   public void run() {
5.     move(Thread.currentThread().getId());
6.   }
7.   // insert code here
8.     System.out.print(id + " ");
9.     System.out.print(id + " ");
10.  }
11.  public static void main(String[] args) {
12.    Chess ch = new Chess();
13.    new Thread(ch).start();
14.    new Thread(new Chess()).start();
15.  }
16. }
```

And given these two fragments:

```
I.  synchronized void move(long id) {
II.  void move(long id) {
```

When either fragment I or fragment II is inserted at line 7, which are true? (Choose all that apply.)

A. Compilation fails
B. With fragment I, an exception is thrown
C. With fragment I, the output could be `4 2 4 2`
D. With fragment I, the output could be `4 4 2 3`
E. With fragment II, the output could be `2 4 2 4`

SELF TEST ANSWERS

1. The following block of code creates a Thread using a Runnable target:

```
Runnable target = new MyRunnable();
Thread myThread = new Thread(target);
```

Which of the following classes can be used to create the target, so that the preceding code compiles correctly?

A. `public class MyRunnable extends Runnable{public void run(){}}`
B. `public class MyRunnable extends Object{public void run(){}}`
C. `public class MyRunnable implements Runnable{public void run(){}}`
D. `public class MyRunnable implements Runnable{void run(){}}`
E. `public class MyRunnable implements Runnable{public void start(){}}`

Answer:

☑ **C** is correct. The class implements the Runnable interface with a legal `run()` method.

☒ **A** is incorrect because interfaces are implemented, not extended. **B** is incorrect because even though the class has a valid `public void run()` method, it does not implement the Runnable interface. **D** is incorrect because the `run()` method must be `public`. **E** is incorrect because the method to implement is `run()`, not `start()`. (Objective 4.1)

2. Given:

```
3.  class MyThread extends Thread {
4.     public static void main(String [] args) {
5.        MyThread t = new MyThread();
6.        Thread x = new Thread(t);
7.        x.start();
8.     }
9.     public void run() {
10.       for(int i=0;i<3;++i) {
11.          System.out.print(i + "..");
12.  }  }  }
```

What is the result of this code?

A. Compilation fails

B. 1..2..3..

C. 0..1..2..3..

D. 0..1..2..

E. An exception occurs at runtime

Answer:

☑ **D** is correct. The thread `MyThread` will start and loop three times (from 0 to 2).

☒ **A** is incorrect because the Thread class implements the Runnable interface; therefore, in line 5, Thread can take an object of type Thread as an argument in the constructor (this is NOT recommended). **B** and **C** are incorrect because the variable `i` in the `for` loop starts with a value of 0 and ends with a value of 2. **E** is incorrect based on the above. (Objective 4.1)

3. Given:

```
3.  class Test {
4.     public static void main(String [] args) {
5.        printAll(args);
6.     }
7.     public static void printAll(String[] lines) {
8.        for(int i=0;i<lines.length;i++){
9.           System.out.println(lines[i]);
10.          Thread.currentThread().sleep(1000);
11. }  }  }
```

The `static` method `Thread.currentThread()` returns a reference to the currently executing Thread object. What is the result of this code?

A. Each String in the array `lines` will print, with exactly a 1-second pause between lines

B. Each String in the array `lines` will print, with no pause in between because this method is not executed in a Thread

C. Each String in the array `lines` will print, and there is no guarantee there will be a pause because `currentThread()` may not retrieve this thread

D. This code will not compile

E. Each String in the `lines` array will print, with at least a one-second pause between lines

Answer:

☑ **D** is correct. The `sleep()` method must be enclosed in a `try/catch` block, or the method `printAll()` must declare it throws the `InterruptedException`.

☒ **E** is incorrect, but it would be correct if the `InterruptedException` was dealt with (**A** is too precise). **B** is incorrect (even if the `InterruptedException` was dealt with) because all Java code, including the `main()` method, runs in threads. **C** is incorrect. The `sleep()` method is `static`, it always affects the currently executing thread. (Objective 4.2)

4. Assume you have a class that holds two `private` variables: `a` and `b`. Which of the following pairs can prevent concurrent access problems in that class? (Choose all that apply.)

A.
```
public int read(){return a+b;}
public void set(int a, int b){this.a=a;this.b=b;}
```

B.
```
public synchronized int read(){return a+b;}
public synchronized void set(int a, int b){this.a=a;this.b=b;}
```

C.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(a){this.a=a;this.b=b;}}
```

D.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(b){this.a=a;this.b=b;}}
```

E.
```
public synchronized(this) int read(){return a+b;}
public synchronized(this) void set(int a, int b){this.a=a;this.b=b;}
```

F.
```
public int read(){synchronized(this){return a+b;}}
public void set(int a, int b){synchronized(this){this.a=a;this.b=b;}}
```

Answer:

☑ **B** and **F** are correct. By marking the methods as `synchronized`, the threads will get the lock of the `this` object before proceeding. Only one thread will be setting or reading at any given moment, thereby assuring that `read()` always returns the addition of a valid pair.

☒ **A** is incorrect because it is not synchronized; therefore, there is no guarantee that the values added by the `read()` method belong to the same pair. **C** and **D** are incorrect; only objects can be used to synchronize on. **E** fails—it is not possible to select other objects (even `this`) to synchronize on when declaring a method as `synchronized`. (Objective 4.3)

5. Given:

```
1.  public class WaitTest {
2.     public static void main(String [] args) {
3.        System.out.print("1 ");
4.        synchronized(args){
```

```
 5.            System.out.print("2 ");
 6.            try {
 7.               args.wait();
 8.            }
 9.            catch(InterruptedException e){}
10.         }
11.         System.out.print("3 ");
12.   }  }
```

What is the result of trying to compile and run this program?

A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7

B. `1 2 3`

C. `1 3`

D. `1 2`

E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait

F. It will fail to compile because it has to be synchronized on the `this` object

Answer:

☑ **D** is correct. 1 and 2 will be printed, but there will be no return from the `wait` call because no other thread will notify the main thread, so 3 will never be printed. It's frozen at line 7.

☒ **A** is incorrect; `IllegalMonitorStateException` is an unchecked exception. **B** and **C** are incorrect; 3 will never be printed, since this program will wait forever. **E** is incorrect because `IllegalMonitorStateException` will never be thrown because the `wait()` is done on `args` within a block of code synchronized on `args`. **F** is incorrect because any object can be used to synchronize on and `this` and `static` don't mix. (Objective 4.4)

6. Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will the thread A become a candidate to get another turn at the CPU?

A. After object B is notified, or after two seconds

B. After the lock on B is released, or after two seconds

C. Two seconds after object B is notified

D. Two seconds after lock B is released

Answer:

☑ **A** is correct. Either of the two events will make the thread a candidate for running again.

☒ **B** is incorrect because a waiting thread will not return to runnable when the lock is released, unless a notification occurs. **C** is incorrect because the thread will become a candidate immediately after notification. **D** is also incorrect because a thread will not come out of a waiting pool just because a lock has been released. (Objective 4.4)

7. Which are true? (Choose all that apply.)

A. The `notifyAll()` method must be called from a synchronized context

B. To call `wait()`, an object must own the lock on the thread

C. The `notify()` method is defined in class `java.lang.Thread`

D. When a thread is waiting as a result of `wait()`, it releases its lock

E. The `notify()` method causes a thread to immediately release its lock

F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on

Answer:

☑ **A** is correct because `notifyAll()` (and `wait()` and `notify()`) must be called from within a synchronized context. **D** is a correct statement.

☒ **B** is incorrect because to call `wait()`, the thread must own the lock on the object that `wait()` is being invoked on, not the other way around. **C** is wrong because `notify()` is defined in `java.lang.Object`. **E** is wrong because `notify()` will not cause a thread to release its locks. The thread can only release its locks by exiting the synchronized code. **F** is wrong because `notifyAll()` notifies all the threads waiting on a particular locked object, not all threads waiting on *any* object. (Objective 4.4)

8. Given the scenario: This class is intended to allow users to write a series of messages, so that each message is identified with a timestamp and the name of the thread that wrote the message:

```
public class Logger {
    private StringBuilder contents = new StringBuilder();
    public void log(String message) {
        contents.append(System.currentTimeMillis());
        contents.append(": ");
        contents.append(Thread.currentThread().getName());
```

```
            contents.append(message);
            contents.append("\n");
        }
        public String getContents() { return contents.toString(); }
    }
```

How can we ensure that instances of this class can be safely used by multiple threads?

A. This class is already thread-safe
B. Replacing `StringBuilder` with `StringBuffer` will make this class thread-safe
C. Synchronize the `log()` method only
D. Synchronize the `getContents()` method only
E. Synchronize both `log()` and `getContents()`
F. This class cannot be made thread-safe

Answer:

☑ **E** is correct. Synchronizing the `public` methods is sufficient to make this safe, so **F** is false. This class is not thread-safe unless some sort of synchronization protects the changing data.

☒ **B** is not correct because although a `StringBuffer` is synchonized internally, we call `append()` multiple times, and nothing would prevent two simultaneous `log()` calls from mixing up their messages. **C** and **D** are not correct because if one method remains unsynchronized, it can run while the other is executing, which could result in reading the contents while one of the messages is incomplete, or worse. (You don't want to call `getString()` on the `StringBuffer` as it's resizing its internal character array.) (Objective 4.3)

9. Given:

```
public static synchronized void main(String[] args) throws
 InterruptedException {
    Thread t = new Thread();
    t.start();
    System.out.print("X");
    t.wait(10000);
    System.out.print("Y");
}
```

What is the result of this code?

A. It prints `X` and exits
B. It prints `X` and never exits
C. It prints `XY` and exits almost immeditately

D. It prints `XY` with a 10-second delay between `X` and `Y`
E. It prints `XY` with a 10000-second delay between `X` and `Y`
F. The code does not compile
G. An exception is thrown at runtime

Answer:

☑ **G** is correct. The code does not acquire a lock on `t` before calling `t.wait()`, so it throws an `IllegalMonitorStateException`. The method is `synchronized`, but it's not synchronized on `t` so the exception will be thrown. If the wait were placed inside a `synchronized(t)` block, then the answer would have been **D**.

☒ **A, B, C, D, E,** and **F** are incorrect based the logic described above. (Objective 4.2)

10. Given:

```
class MyThread extends Thread {
   MyThread() {
     System.out.print(" MyThread");
   }
   public void run() { System.out.print(" bar"); }
   public void run(String s) { System.out.print(" baz"); }
}
public class TestThreads {
  public static void main (String [] args) {
    Thread t = new MyThread() {
      public void run() { System.out.print(" foo"); }
    };
    t.start();
} }
```

What is the result?

A. `foo`
B. `MyThread foo`
C. `MyThread bar`
D. `foo bar`
E. `foo bar baz`
F. `bar foo`
G. Compilation fails
H. An exception is thrown at runtime

Answer:

☑ **B** is correct. The first line of main we're constructing an instance of an anonymous inner class extending from `MyThread`. So the `MyThread` constructor runs and prints `MyThread`. Next, `main()` invokes `start()` on the new thread instance, which causes the overridden `run()` method (the `run()` method in the anonymous inner class) to be invoked.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the logic described above. (Objective 4.1)

11. Given:

```
public class ThreadDemo {
    synchronized void a() { actBusy(); }
    static synchronized void b() { actBusy(); }
    static void actBusy() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {}
    }
    public static void main(String[] args) {
        final ThreadDemo x = new ThreadDemo();
        final ThreadDemo y = new ThreadDemo();
        Runnable runnable = new Runnable() {
            public void run() {
                int option = (int) (Math.random() * 4);
                switch (option) {
                    case 0: x.a(); break;
                    case 1: x.b(); break;
                    case 2: y.a(); break;
                    case 3: y.b(); break;
            }  }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
}   }
```

Which of the following pairs of method invocations could NEVER be executing at the same time? (Choose all that apply.)

A. `x.a()` in `thread1`, and `x.a()` in `thread2`

B. `x.a()` in `thread1`, and `x.b()` in `thread2`

C. `x.a()` in `thread1`, and `y.a()` in `thread2`

D. `x.a()` in `thread1`, and `y.b()` in `thread2`

E. `x.b()` in `thread1`, and `x.a()` in `thread2`

F. `x.b()` in `thread1`, and `x.b()` in `thread2`

G. `x.b()` in `thread1`, and `y.a()` in `thread2`

H. `x.b()` in `thread1`, and `y.b()` in `thread2`

Answer:

☑ **A, F,** and **H**. **A** is a right answer because when `synchronized` instance methods are called on the same *instance*, they block each other. **F** and **H** can't happen because `synchronized static` methods in the same class block each other, regardless of which instance was used to call the methods. (An instance is not required to call `static` methods; only the class.)

☒ **C** could happen because `synchronized` instance methods called on different instances do not block each other. **B, D, E,** and **G** could all happen because instance methods and `static` methods lock on different objects, and do not block each other. (Objective 4.3)

12. Given:

```
public class TwoThreads {
    static Thread laurel, hardy;
    public static void main(String[] args) {
        laurel = new Thread() {
            public void run() {
                System.out.println("A");
                try {
                    hardy.sleep(1000);
                } catch (Exception e) {
                    System.out.println("B");
                }
                System.out.println("C");
            }
        };
        hardy = new Thread() {
            public void run() {
                System.out.println("D");
                try {
                    laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
```

```
                }
            };
            laurel.start();
            hardy.start();
        }
    }
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. `A`
B. `B`
C. `C`
D. `D`
E. `E`
F. `F`
G. The answer cannot be reliably determined
H. The code does not compile

Answer:

☑ **A, C, D, E,** and **F** are correct. This may look like `laurel` and `hardy` are battling to cause the other to `sleep()` or `wait()`—but that's not the case. Since `sleep()` is a `static` method, it affects the current thread, which is `laurel` (even though the method is invoked using a reference to `hardy`). That's misleading but perfectly legal, and the Thread `laurel` is able to sleep with no exception, printing `A` and `C` (after at least a 1-second delay). Meanwhile `hardy` tries to call `laurel.wait()`—but hardy has not `synchronized` on `laurel`, so calling `laurel.wait()` immediately causes an `IllegalMonitorStateException`, and so `hardy` prints `D`, `E`, and `F`. Although the *order* of the output is somewhat indeterminate (we have no way of knowing whether `A` is printed before `D`, for example) it is guaranteed that `A`, `C`, `D`, `E`, and `F` will all be printed in some order, eventually—so **G** is incorrect.

☒ **B, G,** and **H** are incorrect based on the above. (Objective 4.4)

13. Given:

```
3. public class Starter implements Runnable {
4.   void go(long id) {
5.     System.out.println(id);
6.   }
7.   public static void main(String[] args) {
8.     System.out.print(Thread.currentThread().getId() + " ");
9.     // insert code here
```

```
10.   }
11.   public void run() { go(Thread.currentThread().getId()); }
12. }
```

And given the following five fragments:

```
I.   new Starter().run();
II.  new Starter().start();
III. new Thread(new Starter());
IV.  new Thread(new Starter()).run();
V.   new Thread(new Starter()).start();
```

When the five fragments are inserted, one at a time at line 9, which are true? (Choose all that apply.)

A. All five will compile
B. Only one might produce the output `4 4`
C. Only one might produce the output `4 2`
D. Exactly two might produce the output `4 4`
E. Exactly two might produce the output `4 2`
F. Exactly three might produce the output `4 4`
G. Exactly three might produce the output `4 2`

Answer:

☑ **C** and **D** are correct. Fragment I doesn't start a new thread. Fragment II doesn't compile. Fragment III creates a new thread but doesn't start it. Fragment IV creates a new thread and invokes `run()` directly, but it doesn't start the new thread. Fragment V creates *and* starts a new thread.

☒ **A, B, E, F,** and **G** are incorrect based on the above. (Objective 4.1)

14. Given:

```
3. public class Leader implements Runnable {
4.   public static void main(String[] args) {
5.     Thread t = new Thread(new Leader());
6.     t.start();
7.     System.out.print("m1 ");
8.     t.join();
9.     System.out.print("m2 ");
10.  }
```

```
11.   public void run() {
12.     System.out.print("r1 ");
13.     System.out.print("r2 ");
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. The output could be `r1 r2 m1 m2`
C. The output could be `m1 m2 r1 r2`
D. The output could be `m1 r1 r2 m2`
E. The output could be `m1 r1 m2 r2`
F. An exception is thrown at runtime

Answer:

☑ **A** is correct. The `join()` must be placed in a try/catch block. If it were, answers B and D would be correct. The `join()` causes the main thread to pause and join the end of the other thread, meaning `"m2"` must come last.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 4.2)

15. Given:

```
3. class Dudes {
4.   static long flag = 0;
5.   // insert code here
6.     if(flag == 0) flag = id;
7.     for(int x = 1; x < 3; x++) {
8.       if(flag == id) System.out.print("yo ");
9.       else System.out.print("dude ");
10.    }
11.  }
12. }
13. public class DudesChat implements Runnable {
14.   static Dudes d;
15.   public static void main(String[] args) {
16.     new DudesChat().go();
17.   }
18.   void go() {
19.     d = new Dudes();
```

```
20.      new Thread(new DudesChat()).start();
21.      new Thread(new DudesChat()).start();
22.    }
23.    public void run() {
24.      d.chat(Thread.currentThread().getId());
25.    }
26. }
```

And given these two fragments:

```
I.  synchronized void chat(long id) {
II. void chat(long id) {
```

When fragment I or fragment II is inserted at line 5, which are true? (Choose all that apply.)

A. An exception is thrown at runtime
B. With fragment I, compilation fails
C. With fragment II, compilation fails
D. With fragment I, the output could be `yo dude dude yo`
E. With fragment I, the output could be `dude dude yo yo`
F. With fragment II, the output could be `yo dude dude yo`

Answer:

☑ **F** is correct. With fragment I, the `chat` method is synchronized, so the two threads can't swap back and forth. With either fragment, the first output must be `yo`.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 4.3)

16. Given:

```
3. class Chicks {
4.   synchronized void yack(long id) {
5.     for(int x = 1; x < 3; x++) {
6.       System.out.print(id + " ");
7.       Thread.yield();
8.     }
9.   }
10. }
11. public class ChicksYack implements Runnable {
12.   Chicks c;
13.   public static void main(String[] args) {
14.     new ChicksYack().go();
15.   }
```

```
16.   void go() {
17.     c = new Chicks();
18.     new Thread(new ChicksYack()).start();
19.     new Thread(new ChicksYack()).start();
20.   }
21.   public void run() {
22.     c.yack(Thread.currentThread().getId());
23.   }
24. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. The output could be `4 4 2 3`
C. The output could be `4 4 2 2`
D. The output could be `4 4 4 2`
E. The output could be `2 2 4 4`
F. An exception is thrown at runtime

Answer:

☑ **F** is correct. When `run()` is invoked, it is with a new instance of `ChicksYack` and `c` has not been assigned to an object. If `c` were static, then because `yack` is synchronized, answers C and E would have been correct.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 4.3)

17. Given:

```
3. public class Chess implements Runnable {
4.   public void run() {
5.     move(Thread.currentThread().getId());
6.   }
7.   // insert code here
8.     System.out.print(id + " ");
9.     System.out.print(id + " ");
10.  }
11.  public static void main(String[] args) {
12.    Chess ch = new Chess();
13.    new Thread(ch).start();
14.    new Thread(new Chess()).start();
15.  }
16. }
```

And given these two fragments:

```
I.  synchronized void move(long id) {
II.  void move(long id) {
```

When either fragment I or fragment II is inserted at line 7, which are true? (Choose all that apply.)

A. Compilation fails
B. With fragment I, an exception is thrown
C. With fragment I, the output could be `4 2 4 2`
D. With fragment I, the output could be `4 4 2 3`
E. With fragment II, the output could be `2 4 2 4`

Answer:

☑ **C** and **E** are correct. **E** should be obvious. **C** is correct because even though `move()` is synchronized, it's being invoked on two different objects.

☒ **A, B,** and **D** are incorrect based on the above. (Objective 4.3)

EXERCISE ANSWERS

Exercise 9-1: Creating a Thread and Putting It to Sleep

The final code should look something like this:

```
class TheCount extends Thread {
   public void run() {
      for(int i = 1;i<=100;++i) {
         System.out.print(i + "  ");
         if(i % 10 == 0)  System.out.println("Hahaha");
         try { Thread.sleep(1000); }
         catch(InterruptedException e) {}
      }
   }
   public static void main(String [] args) {
      new TheCount().start();
   }
}
```

Exercise 9-2: Synchronizing a Block of Code

Your code might look something like this when completed:

```
class InSync extends Thread {
  StringBuffer letter;
  public InSync(StringBuffer letter) { this.letter = letter; }
  public void run() {
    synchronized(letter) {      // #1
      for(int i = 1;i<=100;++i) System.out.print(letter);
      System.out.println();
      char temp = letter.charAt(0);
      ++temp;         // Increment the letter in StringBuffer:
      letter.setCharAt(0, temp);
    }     // #2
  }
  public static void main(String [] args) {
    StringBuffer sb = new StringBuffer("A");
    new InSync(sb).start();  new InSync(sb).start();
    new InSync(sb).start();
  }
}
```

Just for fun, try removing lines 1 and 2 then run the program again. It will be unsynchronized—watch what happens.

TWO-MINUTE DRILL

Here are the key points from this chapter.

Using javac and java (Objective 7.2)

- Use `-d` to change the destination of a class file when it's first generated by the `javac` command.
- The `-d` option can build package-dependent destination classes on-the-fly if the *root* package directory already exists.
- Use the `-D` option in conjunction with the `java` command when you want to set a system property.
- System properties consist of name=value pairs that must be appended directly behind the `-D`, for example, `java -Dmyproperty=myvalue`.
- Command-line arguments are always treated as Strings.
- The `java` command-line argument 1 is put into array element 0, argument 2 is put into element 1, and so on.

Searching with java and javac (Objective 7.5)

- Both `java` and `javac` use the same algorithms to search for classes.
- Searching begins in the locations that contain the classes that come standard with J2SE.
- Users can define secondary search locations using classpaths.
- Default classpaths can be defined by using OS environment variables.
- A classpath can be declared at the command line, and it overrides the default classpath.
- A single classpath can define many different search locations.
- In Unix classpaths, forward slashes (`/`) are used to separate the directories that make up a path. In Windows, backslashes (`\`) are used.

- In Unix, colons (:) are used to separate the paths within a classpath. In Windows, semicolons (;) are used.
- In a classpath, to specify the current directory as a search location, use a dot (.)
- In a classpath, once a class is found, searching stops, so the order of locations to search is important.

Packages and Searching (Objective 7.5)

- When a class is put into a package, its fully qualified name must be used.
- An `import` statement provides an alias to a class's fully qualified name.
- In order for a class to be located, its fully qualified name must have a tight relationship with the directory structure in which it resides.
- A classpath can contain both relative and absolute paths.
- An absolute path starts with a `/` or a `\`.
- Only the final directory in a given path will be searched.

JAR Files (Objective 7.5)

- An entire directory tree structure can be archived in a single JAR file.
- JAR files can be searched by `java` and `javac`.
- When you include a JAR file in a classpath, you must include not only the directory in which the JAR file is located, but the name of the JAR file too.
- For testing purposes, you can put JAR files into `. . ./jre/lib/ext`, which is somewhere inside the Java directory tree on your machine.

Static Imports (Objective 7.1)

- You must start a static import statement like this: `import static`
- You can use static imports to create shortcuts for `static` members (static variables, constants, and methods) of any class.

SELF TEST

1. Given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

2. Given:

```
import static java.lang.System.*;
class _ {
  static public void main(String... __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `_A.`

E. `_-A.`

F. Compilation fails

G. An exception is thrown at runtime

3. Given the default classpath:

```
/foo
```

And this directory structure:

```
foo
  |
  test
     |
     xcom
        |--A.class
        |--B.java
```

And these two files:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Which allows `B.java` to compile? (Choose all that apply.)

A. Set the current directory to `xcom` then invoke
```
javac B.java
```

B. Set the current directory to `xcom` then invoke
```
javac -classpath . B.java
```

C. Set the current directory to `test` then invoke
```
javac -classpath . xcom/B.java
```

D. Set the current directory to `test` then invoke
```
javac -classpath xcom B.java
```

E. Set the current directory to `test` then invoke
```
javac -classpath xcom:. B.java
```

4. Given two files:

```
a=b.java
c_d.class
```

Are in the current directory, which command-line invocation(s) could complete without error? (Choose all that apply.)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

5. If three versions of `MyClass.class` exist on a file system:

Version 1 is in `/foo/bar`
Version 2 is in `/foo/bar/baz`
Version 3 is in `/foo/bar/baz/bing`

And the system's classpath includes

```
/foo/bar/baz
```

And this command line is invoked from `/foo`

```
java -classpath /foo/bar/baz/bing:/foo/bar MyClass
```

Which version will be used by `java`?

A. `/foo/MyClass.class`

B. `/foo/bar/MyClass.class`

C. `/foo/bar/baz/MyClass.class`

D. `/foo/bar/baz/bing/MyClass.class`

E. The result is not predictable

6. Given two files:

```
1. package pkgA;
2. public class Foo {
3.   int a = 5;
4.   protected int b = 6;
5. }
```

```
1. package pkgB;
2. import pkgA.*;
3. public class Fiz extends Foo {
4.   public static void main(String[] args) {
5.     Foo f = new Foo();
6.     System.out.print("  " + f.a);
7.     System.out.print(" " + f.b);
8.     System.out.print(" " + new Fiz().a);
9.     System.out.println(" " + new Fiz().b);
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. `5 6 5 6`

B. `5 6` followed by an exception

C. Compilation fails with an error on line 6

D. Compilation fails with an error on line 7

E. Compilation fails with an error on line 8

F. Compilation fails with an error on line 9

7. Given:

```
3. import java.util.*;
4. public class Antique {
5.   public static void main(String[] args) {
6.     List<String> myList = new ArrayList<String>();
7.     assert (args.length > 0);
8.     System.out.println("still static");
9.   }
10. }
```

Which sets of commands (`javac` followed by `java`) will compile and run without exception or error? (Choose all that apply.)

A.
```
javac Antique.java
java Antique
```

B.
```
javac Antique.java
java -ea Antique
```

C.
```
javac -source 6 Antique.java
java Antique
```

D.
```
javac -source 1.4 Antique.java
java Antique
```

E.
```
javac -source 1.6 Antique.java
java -ea Antique
```

8. Given:

```
 3. import java.util.*;
 4. public class Values {
 5.   public static void main(String[] args) {
 6.     Properties p = System.getProperties();
 7.     p.setProperty("myProp", "myValue");
 8.     System.out.print(p.getProperty("cmdProp") + " ");
 9.     System.out.print(p.getProperty("myProp") + " ");
10.     System.out.print(p.getProperty("noProp") + " ");
11.     p.setProperty("cmdProp", "newValue");
12.     System.out.println(p.getProperty("cmdProp"));
13.   }
14. }
```

And given the command line invocation:

```
java -DcmdProp=cmdValue Values
```

What is the result?

A. `null myValue null null`

B. `cmdValue null null cmdValue`

C. `cmdValue null null newValue`

D. `cmdValue myValue null cmdValue`

E. `cmdValue myValue null newValue`

F. An exception is thrown at runtime

9. Given the following directory structure:

```
x-|
  |- FindBaz.class
  |
  |- test-|
          |- Baz.class
          |
          |- myApp-|
                   |- Baz.class
```

And given the contents of the related .java files:

```
1. public class FindBaz {
2.   public static void main(String[] args) { new Baz(); }
3. }
```

In the test directory:

```
1. public class Baz {
2.   static { System.out.println("test/Baz"); }
3. }
```

In the myApp directory:

```
1. public class Baz {
2.   static { System.out.println("myApp/Baz"); }
3. }
```

If the current directory is x, which invocations will produce the output `"test/Baz"`? (Choose all that apply.)

A. `java FindBaz`

B. `java -classpath test FindBaz`

C. `java -classpath .:test FindBaz`

D. `java -classpath .:test/myApp FindBaz`

E. `java -classpath test:test/myApp FindBaz`

F. `java -classpath test:test/myApp:. FindBaz`

G. `java -classpath test/myApp:test:. FindBaz`

10. Given the following directory structure:

```
test-|
     |- Test.java
     |
     |- myApp-|
              |- Foo.java
              |
              |- myAppSub-|
                          |- Bar.java
```

If the current directory is test, and you create a .jar file by invoking this,

```
jar -cf MyJar.jar myApp
```

then which path names will find a file in the .jar file? (Choose all that apply.)

A. `Foo.java`

B. `Test.java`

C. `myApp/Foo.java`

D. `myApp/Bar.java`

E. `META-INF/Foo.java`

F. `META-INF/myApp/Foo.java`

G. `myApp/myAppSub/Bar.java`

11. Given the following directory structure:

```
test-|
     |- GetJar.java
     |
     |- myApp-|
              |-Foo.java
```

And given the contents of GetJar.java and Foo.java:

```
3. public class GetJar {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }
```

```
3. package myApp;
4. public class Foo { public static int d = 8; }
```

If the current directory is "test", and myApp/Foo.class is placed in a JAR file called MyJar.jar located in test, which set(s) of commands will compile GetJar.java and produce the output `8`? (Choose all that apply.)

A.
```
javac -classpath MyJar.jar GetJar.java
java GetJar
```

B.
```
javac MyJar.jar GetJar.java
java GetJar
```

C.
```
javac -classpath MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

D.
```
javac MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

12. Given the following directory structure:

```
x-|
  |- GoDeep.class
  |
  |- test-|
         |- MyJar.jar
         |
         |- myApp-|
                  |-Foo.java
                  |-Foo.class
```

And given the contents of GoDeep.java and Foo.java:

```
3. public class GoDeep {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }
```

```
3. package myApp;
4. public class Foo { public static int d = 8; }
```

And MyJar.jar contains the following entry:

```
myApp/Foo.class
```

If the current directory is x, which commands will successfully execute GoDeep.class and produce the output `8`? (Choose all that apply.)

A. `java GoDeep`

B. `java -classpath . GoDeep`

C. `java -classpath test/MyJar.jar GoDeep`

D. `java GoDeep -classpath test/MyJar.jar`

E. `java GoDeep -classpath test/MyJar.jar:.`

F. `java -classpath .:test/MyJar.jar GoDeep`

G. `java -classpath test/MyJar.jar:. GoDeep`

SELF TEST ANSWERS

1. Given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

Answer:

☑ **C** and **E** are correct syntax for static imports. Line 4 isn't making use of `static imports`, so the code will also compile with none of the imports.

☒ **A, B, D,** and **F** are incorrect based on the above. (Objective 7.1)

2. Given:

```
import static java.lang.System.*;
class _ {
  static public void main(String... __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `_A.`

E. `_-A.`

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **B** is correct. This question is using valid (but inappropriate and weird) identifiers, static imports, var-args in `main()`, and pre-incrementing logic.

☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (Objective 7.2)

3. Given the default classpath:

```
/foo
```

And this directory structure:

```
foo
  |
  test
     |
     xcom
       |--A.class
       |--B.java
```

And these two files:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Which allows `B.java` to compile? (Choose all that apply.)

A. Set the current directory to `xcom` then invoke
```
javac B.java
```

B. Set the current directory to `xcom` then invoke
```
javac -classpath . B.java
```

C. Set the current directory to `test` then invoke
```
javac -classpath . xcom/B.java
```

D. Set the current directory to `test` then invoke
```
javac -classpath xcom B.java
```

E. Set the current directory to `test` then invoke
```
javac -classpath xcom:. B.java
```

Answer:

☑ **C** is correct. In order for `B.java` to compile, the compiler first needs to be able to find `B.java`. Once it's found `B.java` it needs to find `A.class`. Because `A.class` is in the `xcom` package the compiler won't find `A.class` if it's invoked from the `xcom` directory. Remember that the `-classpath` isn't looking for `B.java`, it's looking for whatever classes `B.java` needs (in this case `A.class`).

☒ **A, B,** and **D** are incorrect based on the above. **E** is incorrect because the compiler can't find `B.java`. (Objective 7.2)

4. Given two files:

```
a=b.java
c_d.class
```

Are in the current directory, which command-line invocation(s) could complete without error? (Choose all that apply.)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

Answer:

☑ **A** is correct. The `-D` flag is NOT a compiler flag, and the name=value pair that is associated with the `-D` must follow the `-D` with no spaces.

☒ **B, C,** and **D** are incorrect based on the above. (Objective 7.2)

5. If three versions of `MyClass.class` exist on a file system:

Version 1 is in `/foo/bar`
Version 2 is in `/foo/bar/baz`
Version 3 is in `/foo/bar/baz/bing`

And the system's classpath includes

```
/foo/bar/baz
```

And this command line is invoked from `/foo`

```
java -classpath /foo/bar/baz/bing:/foo/bar MyClass
```

Which version will be used by `java`?

A. `/foo/MyClass.class`

B. `/foo/bar/MyClass.class`

C. `/foo/bar/baz/MyClass.class`

D. `/foo/bar/baz/bing/MyClass.class`

E. The result is not predictable.

Answer:

☒ **D** is correct. A -classpath included with a `java` invocation overrides a system classpath. When `java` is using any classpath, it reads the classpath from left to right, and uses the first match it finds.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 7.5)

6. Given two files:

```
1. package pkgA;
2. public class Foo {
3.   int a = 5;
4.   protected int b = 6;
5. }
```

```
1. package pkgB;
2. import pkgA.*;
3. public class Fiz extends Foo {
4.   public static void main(String[] args) {
5.     Foo f = new Foo();
6.     System.out.print("  " + f.a);
7.     System.out.print(" " + f.b);
8.     System.out.print(" " + new Fiz().a);
9.     System.out.println(" " + new Fiz().b);
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. `5 6 5 6`

B. `5 6` followed by an exception

C. Compilation fails with an error on line 6

D. Compilation fails with an error on line 7

E. Compilation fails with an error on line 8

F. Compilation fails with an error on line 9

Answer:

☑ **C, D,** and **E** are correct. Variable `a` (default access) cannot be accessed from outside the package. Since variable `b` is protected, it can be accessed only through inheritance.

☒ **A, B,** and **F** are incorrect based on the above. (Objectives 1.1, 7.1)

7. Given:

```
3. import java.util.*;
4. public class Antique {
5.   public static void main(String[] args) {
6.     List<String> myList = new ArrayList<String>();
```

```
 7.     assert (args.length > 0);
 8.     System.out.println("still static");
 9.   }
10. }
```

Which sets of commands (`javac` followed by `java`) will compile and run without exception or error? (Choose all that apply.)

A.
```
javac Antique.java
java Antique
```

B.
```
javac Antique.java
java -ea Antique
```

C.
```
javac -source 6 Antique.java
java Antique
```

D.
```
javac -source 1.4 Antique.java
java Antique
```

E.
```
javac -source 1.6 Antique.java
java -ea Antique
```

Answer:

☑ **A** and **C** are correct. If assertions (which were first available in Java 1.4) are enabled, an AssertionError will be thrown at line 7.

☒ **D** is incorrect because the code uses generics, and generics weren't introduced until Java 5. **B** and **E** are incorrect based on the above. (Objective 7.2)

8. Given:

```
 3. import java.util.*;
 4. public class Values {
 5.   public static void main(String[] args) {
 6.     Properties p = System.getProperties();
 7.     p.setProperty("myProp", "myValue");
 8.     System.out.print(p.getProperty("cmdProp") + " ");
 9.     System.out.print(p.getProperty("myProp") + " ");
10.     System.out.print(p.getProperty("noProp") + " ");
11.     p.setProperty("cmdProp", "newValue");
12.     System.out.println(p.getProperty("cmdProp"));
13.   }
14. }
```

And given the command line invocation:

```
java -DcmdProp=cmdValue Values
```

What is the result?

A. `null myValue null null`

B. `cmdValue null null cmdValue`

C. `cmdValue null null newValue`

D. `cmdValue myValue null cmdValue`

E. `cmdValue myValue null newValue`

F. An exception is thrown at runtime

Answer:

☑ **E** is correct. System properties can be set at the command line, as indicated correctly in the example. System properties can also be set and overridden programmatically.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 7.2)

9. Given the following directory structure:

```
x-|
  |- FindBaz.class
  |
  |- test-|
          |- Baz.class
          |
          |- myApp-|
                   |- Baz.class
```

And given the contents of the related .java files:

```
1. public class FindBaz {
2.   public static void main(String[] args) { new Baz(); }
3. }
```

In the test directory:

```
1. public class Baz {
2.   static { System.out.println("test/Baz"); }
3. }
```

In the myApp directory:

```
1. public class Baz {
2.   static { System.out.println("myApp/Baz"); }
3. }
```

If the current directory is x, which invocations will produce the output `"test/Baz"`? (Choose all that apply.)

A. `java FindBaz`

B. `java -classpath test FindBaz`

C. `java -classpath .:test FindBaz`

D. `java -classpath .:test/myApp FindBaz`

E. `java -classpath test:test/myApp FindBaz`

F. `java -classpath test:test/myApp:. FindBaz`

G. `java -classpath test/myApp:test:. FindBaz`

Answer:

☑ **C** and **F** are correct. The `java` command must find both `FindBaz` and the version of `Baz` located in the test directory. The `"."` finds `FindBaz`, and `"test"` must come before `"test/myApp"` or `java` will find the other version of `Baz`. Remember the real exam will default to using the Unix path separator.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (Objective 7.2)

10. Given the following directory structure:

```
test-|
     |- Test.java
     |
     |- myApp-|
              |- Foo.java
              |
              |- myAppSub-|
                          |- Bar.java
```

If the current directory is test, and you create a .jar file by invoking this,

```
jar -cf MyJar.jar myApp
```

then which path names will find a file in the .jar file? (Choose all that apply.)

A. `Foo.java`

B. `Test.java`

C. `myApp/Foo.java`

D. `myApp/Bar.java`

E. `META-INF/Foo.java`

F. `META-INF/myApp/Foo.java`

G. `myApp/myAppSub/Bar.java`

Answer:

☑ **C** and **G** are correct. The files in a .jar file will exist within the same exact directory tree structure in which they existed when the .jar was created. Although a .jar file will contain a META-INF directory, none of your files will be in it. Finally, if any files exist in the directory from which the `jar` command was invoked, they won't be included in the .jar file by default.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.5)

11. Given the following directory structure:

```
test-|
     |- GetJar.java
     |
     |- myApp-|
              |-Foo.java
```

And given the contents of GetJar.java and Foo.java:

```
3. public class GetJar {
4.   public static void main(String[] args) {
5.    System.out.println(myApp.Foo.d);
6.   }
7. }

3. package myApp;
4. public class Foo { public static int d = 8; }
```

If the current directory is "test", and myApp/Foo.class is placed in a JAR file called MyJar.jar located in test, which set(s) of commands will compile GetJar.java and produce the output `8`? (Choose all that apply.)

A.
```
javac -classpath MyJar.jar GetJar.java
java GetJar
```

B.
```
javac MyJar.jar GetJar.java
java GetJar
```

C.
```
javac -classpath MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

D.
```
javac MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

Answer:

☑ **A** is correct. Given the current directory and where the necessary files are located, these are the correct command line statements.

☒ **B** and **D** are wrong because javac MyJar.jar GetJar.java is incorrect syntax. **C** is wrong because the -classpath MyJar.java in the java invocation does not include the test directory. (Objective 7.5)

12. Given the following directory structure:

```
x-|
  |- GoDeep.class
  |
  |- test-|
          |- MyJar.jar
          |
          |- myApp-|
                   |-Foo.java
                   |-Foo.class
```

And given the contents of GoDeep.java and Foo.java:

```
3. public class GoDeep {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }
```

```
3. package myApp;
4. public class Foo { public static int d = 8; }
```

And MyJar.jar contains the following entry:

```
myApp/Foo.class
```

If the current directory is x, which commands will successfully execute GoDeep.class and produce the output `8`? (Choose all that apply.)

A. `java GoDeep`

B. `java -classpath . GoDeep`

C. `java -classpath test/MyJar.jar GoDeep`

D. `java GoDeep -classpath test/MyJar.jar`

E. `java GoDeep -classpath test/MyJar.jar:.`

F. `java -classpath .:test/MyJar.jar GoDeep`

G. `java -classpath test/MyJar.jar:. GoDeep`

Answer:

☑ **F** and **G** are correct. The `java` command must find both `GoDeep` and `Foo`, and the `-classpath` option must come before the class name. Note, the current directory (.), in the `classpath` can be searched first or last.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 7.5)

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/cceesept2023

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project