18. A client method contains this code segment:

```
Temperature t1 = new Temperature(40, "C");
Temperature t2 = t1;
Temperature t3 = t2.lower(20);
Temperature t4 = t1.toFahrenheit();
```

Which statement is *true* following execution of this segment?

(A) `t1`, `t2`, `t3`, and `t4` all represent the identical temperature, in degrees Celsius.
(B) `t1`, `t2`, `t3`, and `t4` all represent the identical temperature, in degrees Fahrenheit.
(C) `t4` represents a Fahrenheit temperature, while `t1`, `t2`, and `t3` all represent degrees Celsius.
(D) `t1` and `t2` refer to the same `Temperature` object; `t3` refers to a `Temperature` object that is 20 degrees lower than `t1` and `t2`, while `t4` refers to an object that is `t1` converted to Fahrenheit.
(E) A `NullPointerException` was thrown.

19. Consider the following code:

```
public class TempTest
{
    public static void main(String[] args)
    {
        System.out.println("Enter temperature scale: ");
        String scale = IO.readString();     //read user input
        System.out.println("Enter number of degrees: ");
        double degrees = IO.readDouble();   //read user input
        /* code to construct a valid temperature from user input */
    }
}
```

Which is a correct replacement for `/* code to construct... */`?

I
```
Temperature t = new Temperature(degrees, scale);
if (!t.isValidTemp(degrees,scale))
    /* error message and exit program */
```

II
```
if (isValidTemp(degrees,scale))
    Temperature t = new Temperature(degrees, scale);
else
    /* error message and exit program */
```

III
```
if (Temperature.isValidTemp(degrees,scale))
    Temperature t = new Temperature(degrees, scale);
else
    /* error message and exit program */
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I and III only

20. The formula to convert degrees Celsius C to Fahrenheit F is

$$F = 1.8C + 32$$

For example, 30° C is equivalent to 86° F.

An `inFahrenheit()` accessor method is added to the `Temperature` class. Here is its implementation:

```
/* Precondition:  temperature is a valid temperature in
 *                degrees Celsius
 * Postcondition: an equivalent temperature in degrees
 *                Fahrenheit has been returned. Original
 *                temperature remains unchanged */
public Temperature inFahrenheit()
{
    Temperature result;
    /* more code */
    return result;
}
```

Which of the following correctly replaces `/* more code */` so that the postcondition is achieved?

I
```
result = new Temperature(myDegrees*1.8 + 32, "F");
```

II
```
result = new Temperature(myDegrees*1.8, "F");
result = result.raise(32);
```

III
```
myDegrees *= 1.8;
this = this.raise(32);
result = new Temperature(myDegrees, "F");
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

21. Consider this program:

```
public class CountStuff
{
    public static void doSomething()
    {
        int count = 0;
            ...
        //code to do something - no screen output produced
        count++;
    }

    public static void main(String[] args)
    {
        int count = 0;
        System.out.println("How many iterations?");
        int n = IO.readInt();     //read user input
        for (int  i = 1; i <= n; i++)
        {
            doSomething();
            System.out.println(count);
        }
    }
}
```

If the input value for n is 3, what screen output will this program subsequently produce?

(A)
```
0
0
0
```

(B)
```
1
2
3
```

(C)
```
3
3
3
```

(D)
```
?
?
?
```
where ? is some undefined value.

(E) No output will be produced.

22. This question refers to the following class:

```
public class IntObject
{
    private int myInt;

    public IntObject()        //default constructor
    { myInt = 0; }
    public IntObject(int n)   //constructor
    { myInt = n; }
    public void increment()   //increment by 1
    { myInt++; }
}
```

Here is a client program that uses this class:

```
public class IntObjectTest
{
    public static IntObject someMethod(IntObject obj)
    {
        IntObject ans = obj;
        ans.increment();
        return ans;
    }

    public static void main(String[] args)
    {
        IntObject x = new IntObject(2);
        IntObject y = new IntObject(7);
        IntObject a = y;
        x = someMethod(y);
        a = someMethod(x);
    }
}
```

Just before exiting this program, what are the object values of x, y, and a, respectively?

(A) 9, 9, 9
(B) 2, 9, 9
(C) 2, 8, 9
(D) 3, 8, 9
(E) 7, 8, 9

23. Consider the following program:

```
public class Tester
{
    public void someMethod(int a, int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class TesterMain
{
    public static void main(String[] args)
    {
        int x = 6, y = 8;
        Tester tester = new Tester();
        tester.someMethod(x, y);
    }
}
```

Just before the end of execution of this program, what are the values of `x`, `y`, and `temp`, respectively?

(A) 6, 8, 6
(B) 8, 6, 6
(C) 6, 8, ?, where ? means undefined
(D) 8, 6, ?, where ? means undefined
(E) 8, 6, 8

ANSWER KEY

1. D
2. B
3. C
4. C
5. B
6. C
7. E
8. E
9. A
10. A
11. C
12. C
13. D
14. E
15. D
16. B
17. E
18. B
19. C
20. D
21. A
22. A
23. C

ANSWERS EXPLAINED

1. (D) There are just two constructors. Constructors are recognizable by having the same name as the class, and no return type.

2. (B) Each of the private instance variables should be assigned the value of the matching parameter. Choice B is the only choice that does this. Choice D confuses the order of the assignment statements. Choice A gives the code for the *default* constructor, ignoring the parameters. Choice C would be correct if it were `resetTime(h, m, s)`. As written, it doesn't assign the parameter values `h`, `m`, and `s` to `myHrs`, `myMins`, and `mySecs`. Choice E is wrong because the keyword `new` should be used to create a new object, not to implement the constructor!

3. (C) Replacement III will automatically print time `t` in the required form since a `toString` method was defined for the `Time` class. Replacement I is wrong because it doesn't refer to the parameter, `t`, of the method. Replacement II is wrong because a client program may not access private data of the class.

4. (C) The parameter names can be the same—the *signatures* must be different. For example,

```
public void print(int x)        //prints x
public void print(double x)     //prints x
```

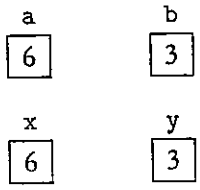
   The signatures (method name plus parameter types) here are `print(int)` and `print(double)`, respectively. The parameter name `x` is irrelevant. Choice A is true: All local variables and parameters go out of scope (are erased) when the method is exited. Choice B is true: Static methods apply to the whole class. Only instance methods have an implicit `this` parameter. Choice D is true even for object parameters: Their references are passed by value. Note that choice E is true because it's possible to have two different constructors with different signatures but the same number of parameters (e.g., one for an `int` argument and one for a `double`).

5. (B) Constructing an object requires the keyword `new` and a constructor of the `Date` class. Eliminate choices D and E since they omit `new`. The class name `Date` should appear on the right-hand side of the assignment statement, immediately following the keyword `new`. This eliminates choices A and C.

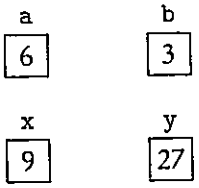
6. (C) Segment III will cause a `NullPointerException` to be thrown since `d` is a null reference. You cannot invoke a method for a null reference. Segment II has the effect of assigning `null` to both `d1` and `d2`—obscure but not incorrect. Segment I creates the object reference `d1` and then declares a second reference `d2` that refers to the same object as `d1`.
7. (E) A client program cannot access a private instance variable.
8. (E) All are correct. Since `write()` is a `Date` instance method, it is OK to use the private data members in its implementation code. Segment III prints `this`, the current `Date` object. This usage is correct since `write()` is part of the `Date` class. The `toString()` method guarantees that the date will be printed in the required format (see p. 164).
9. (A) The idea here is to read in three separate variables for month, day, and year and then to construct the required date using `new` and the `Date` class constructor with three parameters. Code segment II won't work because `month()`, `day()`, and `year()` are accessor methods that access existing values and may not be used to read new values into `bDate`. Segment III is wrong because it tries to access private instance variables from a client program.
10. (A) Segment I will not create a second object. It will simply cause `d2` to refer to the *same* object as `d1`, which is not what was required. The keyword `new` *must* be used to create a new object.
11. (C) When `recentDate` is declared in `main()`, its value is null. Recall that a method is not able to replace an object reference, so `recentDate` remains null. Note that the intent of the program is to change `recentDate` to refer to the updated `oldDate` object. The code, however, doesn't do this. Choice A is false: No methods are invoked with a null reference. Choice B is false because `addYears()` is a mutator method. Even though a method doesn't change the address of its object parameter, it can change the contents of the object, which is what happens here. Choices D and E are wrong because the `addCentury()` method cannot change the value of its `recentDate` argument.
12. (C) The `reduce()` method will be used only in the implementation of the instance methods of the `Rational` class.
13. (D) None of the constructors in the `Rational` class takes a real-valued parameter. Thus, the real-valued parameter in choice D will need to be converted to an integer. Since in general truncating a real value to an integer involves a loss of precision, it is not done automatically—you have to do it explicitly with a cast. Omitting the cast causes a compile-time error.
14. (E) A new `Rational` object must be created using the newly calculated `num` and `denom`. Then it must be reduced before being returned. Choice A is wrong because it doesn't correctly create the new object. Choice B returns a correctly constructed object, but one that has not been reduced. Choice C reduces the current object, `this`, instead of the new object, `rat`. Choice D is wrong because it invokes `reduce()` for the `Rational` class instead of the specific `rat` object.
15. (D) The `plus` method of the `Rational` class can only be invoked by `Rational` objects. Since `n` is an `int`, the statement in choice D will cause an error.
16. (B) The method `getPondTemperature` is the only method that applies to more than one frog. It should therefore be static. All of the other methods relate directly to one particular `Frog` object. So `f.swim()`, `f.die()`, `f.getWeight()`,

and `f.eat()` are all reasonable methods for a single instance `f` of a `Frog`. On the other hand, it doesn't make sense to say `f.getPondTemperature()`. It makes more sense to say `Frog.getPondTemperature()`, since the same value will apply to all frogs in the class.

17. (E) Here are the memory slots at the start of `strangeMethod(a, b)`:



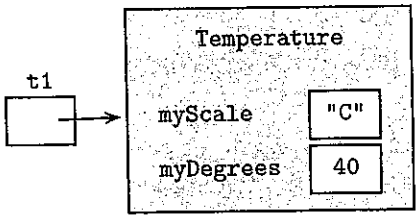
Before exiting `strangeMethod(a, b)`:



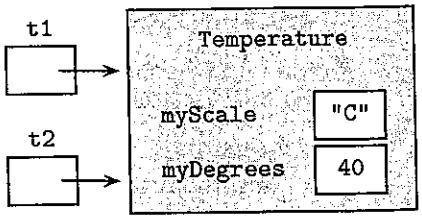
Note that `9 27` is output before exiting. After exiting `strangeMethod(a, b)`, the memory slots are

The next step outputs `6 3`.

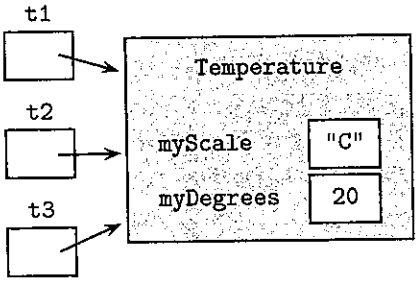
18. (B) This is an example of *aliasing*. The keyword `new` is used just once, which means that just one object is constructed. Here are the memory slots after each declaration:



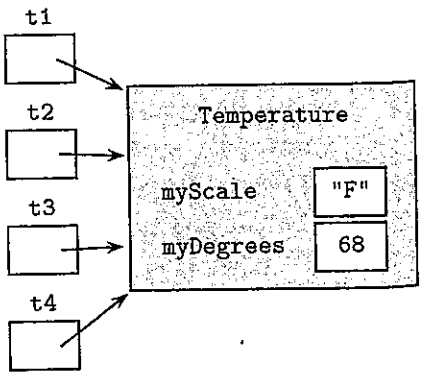
After declaration for `t1`



After declaration for `t2`



After declaration for `t3`

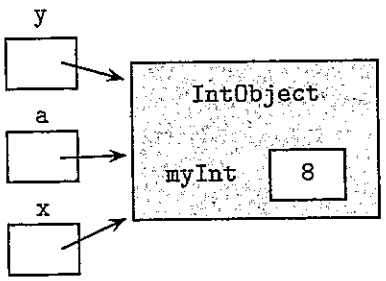


After declaration for `t4`

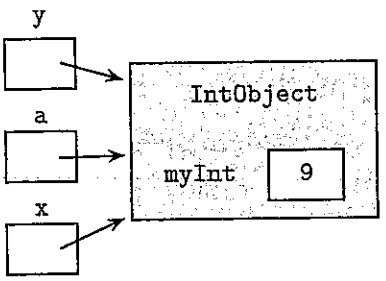
After exiting

```
x = someMethod(y);
```

x has been reassigned, so the object with `myInt = 2` has been recycled:



After exiting `a = someMethod(x)`:



23. (C) Recall that when primitive types are passed as parameters, copies are made of the actual arguments. All manipulations in the method are performed on the copies, and the arguments remain unchanged. Thus x and y retain their values of 6 and 8. The local variable `temp` goes out of scope as soon as `someMethod` is exited and is therefore undefined just before the end of execution of the program.