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. B
13. E
14. C
15. D
16. E
17. D
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 `hrs`, `mins`, and `secs`. Choice E is wrong because the keyword `new` should be used to create a new object, not to implement the constructor!

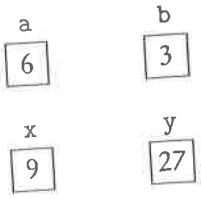
3. (C) Replacement III will automatically print time `t` in the required form since a `toString` method was defined for the `Time` class. Replacement I is wrong because it doesn't refer to the parameter, `t`, of the method. Replacement II is wrong because a client program may not access private data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



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

The next step outputs `6 3`.

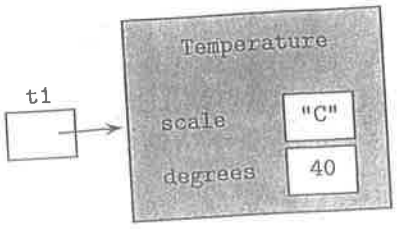
14. (C) The `reduce()` method will be used only in the implementation of the instance methods of the `Rational` class.

15. (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.

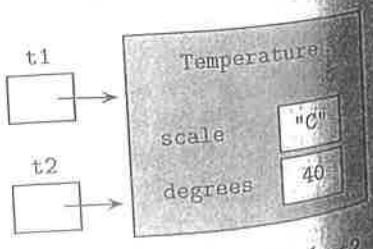
16. (E) A new `Rational` object must be created using the newly calculated `numer` and `denom`. Then it must be reduced before being returned. Choice A is wrong because it doesn't correctly create the new object. Choice B returns a correctly constructed object, but one that has not been reduced. Choice 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.

17. (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.

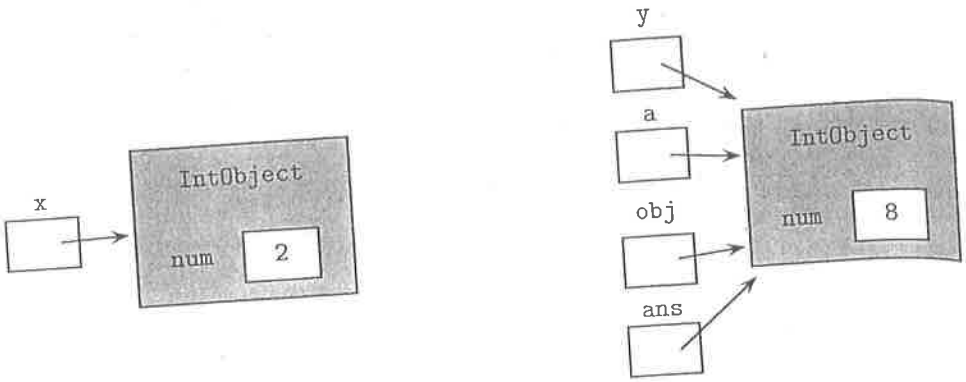
18. (B) This is an example of *aliasing*. The keyword `new` is used just once, which means that just one object is constructed. Here are the memory slots after each declaration:



After declaration for `t1`



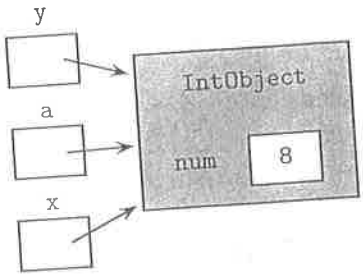
After declaration for `t2`



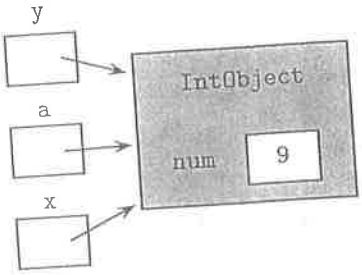
After exiting

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

x has been reassigned, so the object with num = 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.