Some Standard Classes 5

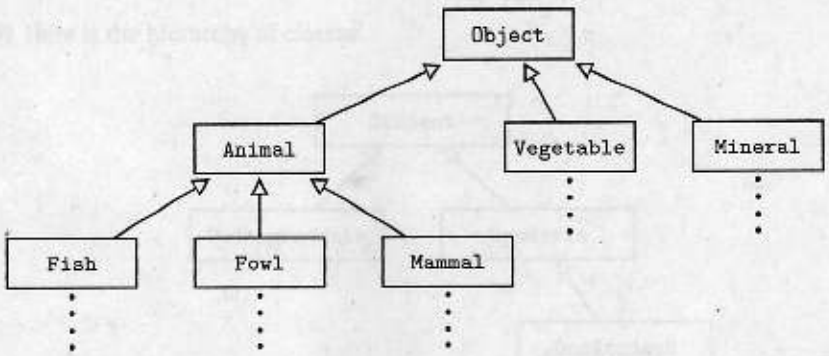
Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
—John von Neumann (1951)

- → **The `Object` class**
- → **The `String` class**
- → **Wrapper classes**
- → **The `Math` class**
- → **Random numbers**

THE `Object` CLASS

The Universal Superclass

Think of `Object` as the superclass of the universe. Every class automatically extends `Object`, which means that `Object` is a direct or indirect superclass of every other class. In a class hierarchy tree, `Object` is at the top:



Methods in `Object`

There are many methods in `Object`, all of them inherited by every other class. Since `Object` is not an abstract class, all of its methods have implementations. The expectation is that these methods will be overridden in any class where the default implementation is not suitable. The methods of `Object` in the AP Java subset are `toString` and `equals`.

THE `toString` METHOD

```
public String toString()
```

This method returns a version of your object in `String` form.

When you attempt to print an object, the inherited default `toString` method is invoked, and what you will see is the class name followed by an `@` followed by a meaningless number (the address in memory of the object). For example,

```
SavingsAccount s = new SavingsAccount(500);
System.out.println(s);
```

produces something like

```
SavingsAccount@fea485c4
```

To have more meaningful output, you need to override the `toString` method for your own classes. Even if your final program doesn't need to output any objects, you should define a `toString` method for each class to help in debugging.

➥ Example 1

```
public class OrderedPair
{
    private double x;
    private double y;

    //constructors and other methods ...

    /** @return this OrderedPair in String form */
    public String toString()
    {
        return "(" + x + "," + y + ")";
    }
}
```

Now the statements

```
OrderedPair p = new OrderedPair(7,10);
System.out.println(p);
```

will invoke the overridden `toString` method and produce output that looks like an ordered pair:

```
(7,10)
```

➥ Example 2

For a `BankAccount` class the overridden `toString` method may look something like this:

```
/** @return this  BankAccount in String form */
public String toString()
{
    return "Bank Account: balance = $" + balance;
}
```

The statements

```
BankAccount b = new BankAccount(600);
System.out.println(b);
```

will produce output that looks like this:

```
Bank Account: balance = $600
```

NOTE

1. The + sign is a concatenation operator for strings (see p. 192).
2. Array objects are unusual in that they do not have a `toString` method. To print the elements of an array, the array must be traversed and each element must explicitly be printed.

THE equals METHOD

```
public boolean equals(Object other)
```

All classes inherit this method from the `Object` class. It returns `true` if this object and `other` are the same object, `false` otherwise. Being the same object means referencing the same memory slot. For example,

Do not use == to test objects for equality. Use the `equals` method.

```
Date d1 = new Date("January", 14, 2001);
Date d2 = d1;
Date d3 = new Date("January", 14, 2001);
```

The test `if (d1.equals(d2))` returns `true`, but the test `if (d1==d3)` returns `false`, since `d1` and `d3` do not refer to the same object. Often, as in this example, you may want two objects to be considered equal if their *contents* are the same. In that case, you have to override the `equals` method in your class to achieve this. Some of the standard classes described later in this chapter have overridden `equals` in this way. You will not be required to write code that overrides `equals` on the AP exam.

NOTE

1. The default implementation of `equals` is equivalent to the == relation for objects: In the `Date` example above, the test `if (d1 == d2)` returns `true`; the test `if (d1 == d3)` returns `false`.
2. The operators <, >, and so on, are not overloaded in Java. To compare objects, one must use either the `equals` method or define a `compareTo` method for the class.

Optional topic

THE hashCode METHOD

Every class inherits the `hashCode` method from `Object`. The value returned by `hashCode` is an integer produced by some formula that maps your object to an address in a hash table. A given object must always produce the same hash code. Also, two objects that are equal should produce the same hash code; that is, if `obj1.equals(obj2)` is true, then `obj1` and `obj2` should have the same hash code. Note that the opposite is not necessarily true. Hash codes do not have to be unique—two objects with the same hash code are not necessarily equal.

To maintain the condition that `obj1.equals(obj2)` is true implies that `obj1` and `obj2` have the same hash code, overriding `equals` means that you should override `hashCode` at the same time. You will not be required to do this on the AP exam. You should, however, understand that every object is associated with an integer value called its hash code, and that objects that are equal have the same hash code.

THE String CLASS

String Objects

An object of type String is a sequence of characters. All *string literals*, such as "yikes!", are implemented as instances of this class. A string literal consists of zero or more characters, including escape sequences, surrounded by double quotes. (The quotes are not part of the String object.) Thus, each of the following is a valid string literal:

```
""                //empty string
"2468"
"I must\n go home"
```

String objects are *immutable*, which means that there are no methods to change them after they've been constructed. You can, however, always create a new String that is a mutated form of an existing String.

Constructing String Objects

A String object is unusual in that it can be initialized like a primitive type:

```
String s = "abc";
```

This is equivalent to

```
String s = new String("abc");
```

in the sense that in both cases s is a reference to a String object with contents "abc" (see Box on p. 193).

It is possible to reassign a String reference:

```
String s = "John";
s = "Harry";
```

This is equivalent to

```
String s = new String("John");
s = new String("Harry");
```

Notice that this is consistent with the immutable feature of String objects. "John" has not been changed; he has merely been discarded! The fickle reference s now refers to a new String, "Harry". It is also OK to reassign s as follows:

```
s = s + " Windsor";
```

s now refers to the object "Harry Windsor".

Here are other ways to initialize String objects:

```
String s1 = null;            //s1 is a null reference
String s2 = new String();    //s2 is an empty character sequence

String state = "Alaska";
String dessert = "baked " + state;  //dessert has value "baked Alaska"
```

The Concatenation Operator

The `dessert` declaration above uses the *concatenation operator*, `+`, which operates on `String` objects. Given two `String` operands `lhs` and `rhs`, `lhs + rhs` produces a single `String` consisting of `lhs` followed by `rhs`. If either `lhs` or `rhs` is an object other than a `String`, the `toString` method of the object is invoked, and `lhs` and `rhs` are concatenated as before. If one of the operands is a `String` and the other is a primitive type, then the non-`String` operand is converted to a `String`, and concatenation occurs as before. If neither `lhs` nor `rhs` is a `String` object, an error occurs. Here are some examples:

```
int five = 5;
String state = "Hawaii-";
String tvShow = state + five + "-0";  //tvShow has value
                                      //"Hawaii-5-0"
int x = 3, y = 4;
String sum = x + y;          //error: can't assign int 7 to String
```

Suppose a `Date` class has a `toString` method that outputs dates that look like this: `2/17/1948`.

```
Date d1 = new Date(8, 2, 1947);
Date d2 = new Date(2, 17, 1948);
String s = "My birthday is " + d2;  //s has value
                                    //"My birthday is 2/17/1948"
String s2 = d1 + d2;     //error: + not defined for objects
String s3 = d1.toString() + d2.toString();  //s3 has value
                                            //8/2/19472/17/1948
```

Comparison of `String` Objects

There are two ways to compare `String` objects:

1. Use the `equals` method that is inherited from the `Object` class and overridden to do the correct thing:

   ```
   if (string1.equals(string2)) ...
   ```

   This returns `true` if `string1` and `string2` are identical strings, `false` otherwise.
2. Use the `compareTo` method. The `String` class has a `compareTo` method:

   ```
   int compareTo(String otherString)
   ```

   It compares strings in dictionary (lexicographical) order:

   - If `string1.compareTo(string2) < 0`, then `string1` precedes `string2` in the dictionary.
   - If `string1.compareTo(string2) > 0`, then `string1` follows `string2` in the dictionary.
   - If `string1.compareTo(string2) == 0`, then `string1` and `string2` are identical. (This test is an alternative to `string1.equals(string2)`.)

Be aware that Java is case-sensitive. Thus, if `s1` is `"cat"` and `s2` is `"Cat"`, `s1.equals(s2)` will return `false`.

Characters are compared according to their position in the ASCII chart. All you need to know is that all digits precede all capital letters, which precede all lowercase letters. Thus `"5"`

comes before "R", which comes before "a". Two strings are compared as follows: Start at the left end of each string and do a character-by-character comparison until you reach the first character in which the strings differ, the *k*th character, say. If the *k*th character of s1 comes before the *k*th character of s2, then s1 will come before s2, and vice versa. If the strings have identical characters, except that s1 terminates before s2, then s1 comes before s2. Here are some examples:

```
String s1 = "HOT", s2 = "HOTEL", s3 = "dog";
if (s1.compareTo(s2) < 0))      //true, s1 terminates first
   ...
if (s1.compareTo(s3) > 0))      //false, "H" comes before "d"
```

Don't Use == to Test Strings!

The expression `if (string1 == string2)` tests whether `string1` and `string2` are the same reference. It does not test the actual strings. Using == to compare strings may lead to unexpected results.

➥ Example 1

```
String s = "oh no!";
String t = "oh no!";
if (s == t) ...
```

The test returns `true` even though it appears that `s` and `t` are different references. The reason is that for efficiency Java makes only one `String` object for equivalent string literals. This is safe in that a `String` cannot be altered.

➥ Example 2

```
String s = "oh no!";
String t = new String("oh no!");
if (s == t) ...
```

The test returns `false` because use of `new` creates a new object, and `s` and `t` *are* different references in this example!

The moral of the story? Use `equals` not == to test strings. It always does the right thing.

Other `String` Methods

The Java `String` class provides many methods, only a small number of which are in the AP Java subset. In addition to the constructors, comparison methods, and concatenation operator + discussed so far, you should know the following methods:

`int length()`

Returns the length of this string.

`String substring(int startIndex)`

Returns a new string that is a substring of this string. The substring starts with the character at `startIndex` and extends to the end of the string. The first character is at index zero. The method throws an `IndexOutOfBoundsException` if `startIndex` is negative or larger than the length of the string. Note that if you're using Java 7 or above, you will see the error `StringIndexOutOfBoundsException`. However, the AP Java subset lists only `IndexOutOfBoundsException`, which is what they will use on the AP exam.

`String substring(int startIndex, int endIndex)`

Returns a new string that is a substring of this string. The substring starts at index `startIndex` and extends to the character at `endIndex-1`. (Think of it this way: `startIndex` is the first character that you want; `endIndex` is the first character that you *don't* want.) The method throws a `StringIndexOutOfBoundsException` if `startIndex` is negative, or `endIndex` is larger than the length of the string, or `startIndex` is larger than `endIndex`.

`int indexOf(String str)`

Returns the index of the first occurrence of `str` within this string. If `str` is not a substring of this string, `-1` is returned. The method throws a `NullPointerException` if `str` is null.

Here are some examples:

```
"unhappy".substring(2)        //returns "happy"
"cold".substring(4)           //returns "" (empty string)
"cold".substring(5)           //StringIndexOutOfBoundsException
"strawberry".substring(5,7)   //returns "be"
"crayfish".substring(4,8)     //returns "fish"
"crayfish".substring(4,9)     //StringIndexOutOfBoundsException
"crayfish".substring(5,4)     //StringIndexOutOfBoundsException

String s = "funnyfarm";
int x = s.indexOf("farm");  //x has value 5
x = s.indexOf("farmer");    //x has value -1
int y = s.length();         //y has value 9
```

WRAPPER CLASSES

A *wrapper class* takes either an existing object or a value of primitive type, "wraps" or "boxes" it in an object, and provides a new set of methods for that type. The point of a wrapper class is to provide extended capabilities for the boxed quantity:

- It can be used in generic Java methods that require objects as parameters.
- It can be used in Java container classes that require the items be objects (see p. 256).

In each case, the wrapper class allows

1. Construction of an object from a single value (wrapping or boxing the primitive in a wrapper object).

2. Retrieval of the primitive value (unwrapping or unboxing from the wrapper object).

Java provides a wrapper class for each of its primitive types. The two that you should know for the AP exam are the `Integer` and `Double` classes.

The `Integer` Class

The `Integer` class wraps a value of type `int` in an object. An object of type `Integer` contains just one instance variable whose type is `int`.

Here are the `Integer` methods you should know for the AP exam:

`Integer(int value)`

Constructs an `Integer` object from an `int`. (Boxing.)

`int compareTo(Integer other)`

Returns `0` if the value of this `Integer` is equal to the value of `other`, a negative integer if it is less than the value of `other`, and a positive integer if it is greater than the value of `other`.

`int intValue()`

Returns the value of this `Integer` as an `int`. (Unboxing.)

`boolean equals(Object obj)`

Returns `true` if and only if this `Integer` has the same `int` value as `obj`.

NOTE

This method overrides `equals` in class `Object`.

`String toString()`

Returns a `String` representing the value of this `Integer`.

Here are some examples to illustrate the `Integer` methods:

```
Integer intObj = new Integer(6);  //boxes 6 in Integer object
int j = intObj.intValue();        //unboxes 6 from Integer object

System.out.println("Integer value is " + intObj);
//calls toString() for intObj
//output is: Integer value is 6

Object object = new Integer(5);   //Integer is a subclass of Object

Integer intObj2 = new Integer(3);
int k = intObj2.intValue();
if (intObj.equals(intObj2))       //OK, evaluates to false
    ...
if (intObj.intValue() == intObj2.intValue())
    ...              //OK, since comparing primitive types
```

```
if (k.equals(j))      //error, k and j not objects
    ...
if ((intObj.intValue()).compareTo(intObj2.intValue()) < 0)
    ...               //error, can't use compareTo on primitive types

if (intObj.compareTo(object) < 0)  //Error. Parameter needs Integer cast
if (intObj.compareTo((Integer) object) < 0)  //OK
    ...
if (object.compareTo(intObj) < 0)  //error, no compareTo in Object
    ...
if (((Integer) object).compareTo(intObj) < 0)  //OK
    ...
```

The Double Class

The Double class wraps a value of type double in an object. An object of type Double contains just one instance variable whose type is double.

The methods you should know for the AP exam are analogous to those for type Integer.

`Double(double value)`

Constructs a Double object from a double. (Boxing.)

`double doubleValue()`

Returns the value of this Double as a double. (Unboxing.)

Remember: Integer, Double, and String all have a compareTo method.

`int compareTo(Double other)`

Returns 0 if the value of this Double is equal to the value of other, a negative integer if it is less than the value of other, and a positive integer if it is greater than the value of other.

`boolean equals(Object obj)`

This method overrides equals in class Object. It returns true if and only if this Double has the same double value as obj.

`String toString()`

Returns a String representing the value of this Double.

Here are some examples:

```
Double dObj = new Double(2.5);      //boxes 2.5 in Double object
double d = dObj.doubleValue();      //unboxes 2.5 from Double object

Object object = new Double(7.3);    //Double is a subclass of Object
Object intObj = new Integer(4);
if (dObj.compareTo(object) > 0) //Error. Parameter needs cast to Double
if (dObj.compareTo((Double) object) > 0)    //OK
    ...
if (dObj.compareTo(intObj) > 0)     //Error:
    ...                             //can't compare Integer to Double
```

NOTE

1. `Integer` and `Double` objects are immutable: There are no mutator methods in the classes.
2. See p. 257 for a discussion of auto-boxing and -unboxing. This useful feature will *not* be tested on the AP exam.
3. Class casting will not explicitly be tested on the AP exam. You should, however, understand that if the parameter object for `compareTo` fails the *is-a* test for the calling object, an error will occur.

THE `Math` CLASS

This class implements standard mathematical functions such as absolute value, square root, trigonometric functions, the log function, the power function, and so on. It also contains mathematical constants such as π and e.

Here are the functions you should know for the AP exam:

```
static int abs(int x)
```

Returns the absolute value of integer x.

```
static double abs(double x)
```

Returns the absolute value of real number x.

```
static double pow(double base, double exp)
```

Returns $\texttt{base}^{\texttt{exp}}$. Assumes `base` > 0, or `base` = 0 and `exp` > 0, or `base` < 0 and `exp` is an integer.

```
static double sqrt(double x)
```

Returns $\sqrt{x}$, $x \geq 0$.

```
static double random()
```

Returns a random number r, where $0.0 \leq r < 1.0$. (See the next section, **Random Numbers**.)

All of the functions and constants are implemented as static methods and variables, which means that there are no instances of `Math` objects. The methods are invoked using the class name, `Math`, followed by the dot operator.

Here are some examples of mathematical formulas and the equivalent Java statements.

1. The relationship between the radius and area of a circle:

$$r = \sqrt{A/\pi}$$

In code:

```
radius = Math.sqrt(area / Math.PI);
```

2. The amount of money A in an account after ten years, given an original deposit of P and an interest rate of 5% compounded annually, is

$$A = P(1.05)^{10}$$

In code:

```
a = p * Math.pow(1.05, 10);
```

3. The distance D between two points $P(x_P, y)$ and $Q(x_Q, y)$ on the same horizontal line is

$$D = |x_P - x_Q|$$

In code:

```
d = Math.abs(xp - xq);
```

NOTE

The static import construct allows you to use the static members of a class without the class name prefix. For example, the statement

```
import static java.lang.Math.*;
```

allows use of all `Math` methods and constants without the `Math` prefix. Thus, the statement in formula 1 above could be written

```
radius = sqrt(area / PI);
```

Static imports are not part of the AP subset.

Random Numbers

RANDOM REALS

The statement

```
double r = Math.random();
```

produces a random real number in the range 0.0 to 1.0, where 0.0 is included and 1.0 is not.

This range can be scaled and shifted. On the AP exam you will be expected to write algebraic expressions involving `Math.random()` that represent linear transformations of the original interval $0.0 \le x < 1.0$.

➥ Example 1

Produce a random real value x in the range $0.0 \le x < 6.0$.

```
double x = 6 * Math.random();
```

➥ Example 2

Produce a random real value x in the range $2.0 \le x < 3.0$.

```
double x = Math.random() + 2;
```

➥ Example 3

Produce a random real value x in the range $4.0 \le x < 6.0$.

```
double x = 2 * Math.random() + 4;
```

In general, to produce a random real value in the range `lowValue` $\le x <$ `highValue`:

```
double x = (highValue - lowValue) * Math.random() + lowValue;
```

RANDOM INTEGERS

Using a cast to `int`, a scaling factor, and a shifting value, `Math.random()` can be used to produce random integers in any range.

➥ Example 1

Produce a random integer, from 0 to 99.

```
int num = (int) (Math.random() * 100);
```

In general, the expression

```
(int) .(Math.random() * k)
```

produces a random `int` in the range $0, 1, \ldots, k-1$, where k is called the scaling factor. Note that the cast to `int` truncates the real number `Math.random() * k`.

➥ Example 2

Produce a random integer, from 1 to 100.

```
int num = (int) (Math.random() * 100) + 1;
```

In general, if k is a scaling factor, and p is a shifting value, the statement

```
int n = (int) (Math.random() * k) + p;
```

produces a random integer n in the range $p, p+1, \ldots, p+(k-1)$.

➥ Example 3

Produce a random integer from 5 to 24.

```
int num = (int) (Math.random() * 20) + 5;
```

Note that there are 20 possible integers from 5 to 24, inclusive.

NOTE

There is further discussion of strings and random numbers, plus additional questions, in Chapter 10 (The AP Computer Science A Labs).

Chapter Summary

All students should know about overriding the `equals` and `toString` methods of the `Object` class and should be familiar with the `Integer` and `Double` wrapper classes.

Know the AP subset methods of the `Math` class, especially the use of `Math.random()` for generating random integers. Know the `String` methods `substring` and `indexOf` like the back of your hand, including knowing where exceptions are thrown in the `String` methods.

MULTIPLE-CHOICE QUESTIONS ON STANDARD CLASSES

1. Here is a program segment to find the quantity base^{exp}. Both base and exp are entered at the keyboard.

```
System.out.println("Enter base and exponent: ");
double base = ...;    //read user input
double exp = ...;     //read user input
/* code to find power, which equals base^exp */
System.out.print(base + " raised to the power " + exp);
System.out.println(" equals " + power);
```

Which is a correct replacement for
`/* code to find power, which equals base^exp */`?

I
```
double power;
Math m = new Math();
power = m.pow(base, exp);
```

II
```
double power;
power = Math.pow(base, exp);
```

III
```
int power;
power = Math.pow(base, exp);
```

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

2. Consider the `squareRoot` method defined below:

```
/** @param d a real number such that d >= 0
 *  Postcondition: Returns a Double whose value is the square
 *                 root of the value represented by d.
 */
public Double squareRoot(Double d)
{
   /* implementation code */
}
```

Which `/* implementation code */` satisfies the postcondition?

I
```
double x = d.doubleValue();
x = Math.sqrt(x);
return new Double(x);
```

II
```
return new Double(Math.sqrt(d.doubleValue()));
```

III
```
return Double(Math.sqrt(d.doubleValue()));
```

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

3. Here are some examples of negative numbers rounded to the nearest integer.

Negative real number	Rounded to nearest integer
−3.5	−4
−8.97	−9
−5.0	−5
−2.487	−2
−0.2	0

Refer to the declaration

```
double d = -4.67;
```

Which of the following correctly rounds `d` to the nearest integer?

(A)
```
int rounded = Math.abs(d);
```

(B)
```
int rounded = (int) (Math.random() * d);
```

(C)
```
int rounded = (int) (d - 0.5);
```

(D)
```
int rounded = (int) (d + 0.5);
```

(E)
```
int rounded = Math.abs((int) (d - 0.5));
```

4. A program is to simulate plant life under harsh conditions. In the program, plants die randomly according to some probability. Here is part of a `Plant` class defined in the program.

```
public class Plant
{
    /** probability that plant dies, a real number between 0 and 1 */
    private double probDeath;

    public Plant(double plantProbDeath, < other parameters >)
    {
        probDeath = plantProbDeath;
        < initialization of other instance variables >
    }

    /** Plant lives or dies. */
    public void liveOrDie()
    {
        /* statement to generate random number */
        if (/* test to determine if plant dies */)
            < code to implement plant's death >
        else
            < code to make plant continue living >
    }

    //Other variables and methods are not shown.
}
```

Which of the following are correct replacements for
(1) `/* statement to generate random number */` and
(2) `/* test to determine if plant dies */`?

(A)
```
(1) double x = Math.random();
(2) x == probDeath
```

(B)
```
(1) double x = (int) (Math.random());
(2) x > probDeath
```

(C)
```
(1) double x = Math.random();
(2) x < probDeath
```

(D)
```
(1) int x = (int) (Math.random() * 100);
(2) x < (int) probDeath
```

(E)
```
(1) int x = (int) (Math.random() * 100) + 1;
(2) x == (int) probDeath
```

5. A program simulates fifty slips of paper, numbered 1 through 50, placed in a bowl for a raffle drawing. Which of the following statements stores in `winner` a random integer from 1 to 50?

(A) `int winner = (int) (Math.random() * 50) + 1;`
(B) `int winner = (int) (Math.random() * 50);`
(C) `int winner = (int) (Math.random() * 51);`
(D) `int winner = (int) (Math.random() * 51) + 1;`
(E) `int winner = (int) (1 + Math.random() * 49);`

6. Consider the code segment

```
Integer i = new Integer(20);
/* more code */
```

Which of the following replacements for `/* more code */` correctly sets `i` to have an integer value of 25?

I `i = new Integer(25);`

II `i.intValue() = 25;`

III
```
Integer j = new Integer(25);
i = j;
```

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

7. Refer to these declarations:

```
Integer k = new Integer(8);
Integer m = new Integer(4);
```

Which test will *not* generate an error?

I `if (k.intValue() == m.intValue())...`

II `if ((k.intValue()).equals(m.intValue()))...`

III `if ((k.toString()).equals(m.toString()))...`

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

8. Consider the code fragment

```
Object intObj = new Integer(9);
System.out.println(intObj);
```

What will be output as a result of running the fragment?

(A) No output. An `IllegalArgumentException` will be thrown.
(B) No output. An `ArithmeticException` will be thrown.
(C) `9`
(D) `"9"`
(E) `nine`

9. Consider these declarations:

```
String s1 = "crab";
String s2 = new String("crab");
String s3 = s1;
```

Which expression involving these strings evaluates to `true`?

I `s1 == s2`

II `s1.equals(s2)`

III `s3.equals(s2)`

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

10. Suppose that `strA = "TOMATO"`, `strB = "tomato"`, and `strC = "tom"`. Given that `"A"` comes before `"a"` in dictionary order, which is true?

(A) `strA.compareTo(strB) < 0 && strB.compareTo(strC) < 0`
(B) `strB.compareTo(strA) < 0 || strC.compareTo(strA) < 0`
(C) `strC.compareTo(strA) < 0 && strA.compareTo(strB) < 0`
(D) `!(strA.equals(strB)) && strC.compareTo(strB) < 0`
(E) `!(strA.equals(strB)) && strC.compareTo(strA) < 0`

11. This question refers to the following declaration:

```
String line = "Some more silly stuff on strings!";
//the words are separated by a single space
```

What string will `str` refer to after execution of the following?

```
int x = line.indexOf("m");
String str = line.substring(10, 15) + line.substring(25, 25 + x);
```

(A) `"sillyst"`
(B) `"sillystr"`
(C) `"silly st"`
(D) `"silly str"`
(E) `"sillystrin"`

12. A program has a `String` variable `fullName` that stores a first name, followed by a space, followed by a last name. There are no spaces in either the first or last names. Here are some examples of `fullName` values: `"Anthony Coppola"`, `"Jimmy Carroll"`, and `"Tom DeWire"`. Consider this code segment that extracts the last name from a `fullName` variable, and stores it in `lastName` with no surrounding blanks:

```
int k = fullName.indexOf(" ");     //find index of blank
String lastName = /* expression */
```

Which is a correct replacement for `/* expression */`?

I `fullName.substring(k);`

II `fullName.substring(k + 1);`

III `fullName.substring(k + 1, fullName.length());`

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

13. One of the rules for converting English to Pig Latin states: If a word begins with a consonant, move the consonant to the end of the word and add "ay". Thus "dog" becomes "ogday," and "crisp" becomes "rispcay". Suppose `s` is a `String` containing an English word that begins with a consonant. Which of the following creates the correct corresponding word in Pig Latin? Assume the declarations

```
String ayString = "ay";
String pigString;
```

(A)
```
pigString = s.substring(0, s.length()) + s.substring(0,1)
          + ayString;
```

(B)
```
pigString = s.substring(1, s.length()) + s.substring(0,0)
          + ayString;
```

(C)
```
pigString = s.substring(0, s.length()-1) + s.substring(0,1)
          + ayString;
```

(D)
```
pigString = s.substring(1, s.length()-1) + s.substring(0,0)
          + ayString;
```

(E)
```
pigString = s.substring(1, s.length()) + s.substring(0,1)
          + ayString;
```

14. This question refers to the `getString` method shown below:

```
public static String getString(String s1, String s2)
{
    int index = s1.indexOf(s2);
    return s1.substring(index, index + s2.length());
}
```

Which is true about `getString`? It may return a string that

I Is equal to `s2`.
II Has no characters in common with `s2`.
III Is equal to `s1`.

(A) I and III only
(B) II and III only
(C) I and II only
(D) I, II, and III
(E) None is true.

15. Consider this method:

```
public static String doSomething(String s)
{
    final String BLANK = " ";   //BLANK contains a single space
    String str = "";            //empty string
    String temp;
    for (int i = 0; i < s.length(); i++)
    {
        temp = s.substring(i, i + 1);
        if (!(temp.equals(BLANK)))
            str += temp;
    }
    return str;
}
```

Which of the following is the most precise description of what `doSomething` does?

(A) It returns `s` unchanged.

(B) It returns `s` with all its blanks removed.

(C) It returns a `String` that is equivalent to `s` with all its blanks removed.

(D) It returns a `String` that is an exact copy of `s`.

(E) It returns a `String` that contains `s.length()` blanks.

Questions 16 and 17 refer to the classes `Position` and `PositionTest` below.

```
public class Position
{
    /** row and col are both >= 0 except in the default
     *  constructor where they are initialized to -1.
     */
    private int row, col;

    public Position()          //constructor
    {
       row = -1;
       col = -1;
    }

    public Position(int r, int c)        //constructor
    {
        row = r;
        col = c;
    }

    /** @return row of Position */
    public int getRow()
    { return row; }

    /** @return column of Position */
    public int getCol()
    { return col; }

    /** @return Position north of (up from) this position */
    public Position north()
    { return new Position(row - 1, col); }

    //Similar methods south, east, and west
          ...

    /** Compares this Position to another Position object.
     *  @param p a Position object
     *  @return -1 (less than), 0 (equals), or 1 (greater than)
     */
    public int compareTo(Position p)
    {
        if (this.getRow() < p.getRow() || this.getRow() == p.getRow()
             && this.getCol() < p.getCol())
                return -1;
        if (this.getRow() > p.getRow() || this.getRow() == p.getRow()
             && this.getCol() > p.getCol())
                return 1;
        return 0;              //row and col both equal
    }

    /** @return string form of Position */
    public String toString()
    { return "(" + row + "," + col + ")"; }
}
```

```
public class PositionTest
{
    public static void main(String[] args)
    {
        Position p1 = new Position(2, 3);
        Position p2 = new Position(4, 1);
        Position p3 = new Position(2, 3);

        //tests to compare positions
            ...
    }
}
```

16. Which is true about the value of `p1.compareTo(p2)`?
    (A) It equals `true`.
    (B) It equals `false`.
    (C) It equals `0`.
    (D) It equals `1`.
    (E) It equals `-1`.

17. Which boolean expression about `p1` and `p3` is true?

    I `p1 == p3`

    II `p1.equals(p3)`

    III `p1.compareTo(p3) == 0`

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

Questions 18 and 19 deal with the problem of swapping two integer values. Three methods are proposed to solve the problem, using primitive `int` types, `Integer` objects, and `IntPair` objects, where `IntPair` is defined as follows:

```
public class IntPair
{
    private int firstValue;
    private int secondValue;

    public IntPair(int first, int second)
    {
        firstValue = first;
        secondValue = second;
    }

    public int getFirst()
    { return firstValue; }

    public int getSecond()
    { return secondValue; }

    public void setFirst(int a)
    { firstValue = a; }

    public void setSecond(int b)
    { secondValue = b;}
}
```

18. Here are three different `swap` methods, each intended for use in a client program.

I
```
public static void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

II
```
public static void swap(Integer obj_a, Integer obj_b)
{
    Integer temp = new Integer(obj_a.intValue());
    obj_a = obj_b;
    obj_b = temp;
}
```

III
```
public static void swap(IntPair pair)
{
    int temp = pair.getFirst();
    pair.setFirst(pair.getSecond());
    pair.setSecond(temp);
}
```

When correctly used in a client program with appropriate parameters, which method will swap two integers, as intended?

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

19. Consider the following program that uses the `IntPair` class:

```
public class TestSwap
{
    public static void swap(IntPair pair)
    {
        int temp = pair.getFirst();
        pair.setFirst(pair.getSecond());
        pair.setSecond(temp);
    }

    public static void main(String[] args)
    {
        int x = 8, y = 6;
        /* code to swap x and y */
    }
}
```

Which is a correct replacement for `/* code to swap x and y */`?

I
```
IntPair iPair = new IntPair(x, y);
swap(x, y);
x = iPair.getFirst();
y = iPair.getSecond();
```

II
```
IntPair iPair = new IntPair(x, y);
swap(iPair);
x = iPair.getFirst();
y = iPair.getSecond();
```

III
```
IntPair iPair = new IntPair(x, y);
swap(iPair);
x = iPair.setFirst();
y = iPair.setSecond();
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) None is correct.

Refer to the Name class below for Questions 20 and 21.

```
public class Name
{
    private String firstName;
    private String lastName;

    public Name(String first, String last)  //constructor
    {
        firstName = first;
        lastName = last;
    }

    public String toString()
    { return firstName + " " + lastName; }

    public boolean equals(Object obj)
    {
        Name n = (Name) obj;
        return n.firstName.equals(firstName) &&
               n.lastName.equals(lastName);
    }

    public int hashCode()
    { /* implementation not shown */ }

    public int compareTo(Name n)
    {
        /* more code */
    }
}
```

20. The `compareTo` method implements the standard name-ordering algorithm where last names take precedence over first names. Lexicographic or dictionary ordering of `Strings` is used. For example, the name Scott Dentes comes before Nick Elser, and Adam Cooper comes before Sara Cooper.

    Which of the following is a correct replacement for `/* more code */`?

    I
    ```
    int lastComp = lastName.compareTo(n.lastName);
    if (lastComp != 0)
        return lastComp;
    else
        return firstName.compareTo(n.firstName);
    ```

    II
    ```
    if (lastName.equals(n.lastName))
        return firstName.compareTo(n.firstName);
    else
        return 0;
    ```

    III
    ```
    if (!(lastName.equals(n.lastName)))
        return firstName.compareTo(n.firstName);
    else
        return lastName.compareTo(n.lastName);
    ```

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

21. Which statement about the `Name` class is *false*?
    (A) `Name` objects are immutable.
    (B) It is possible for the methods in `Name` to throw a `NullPointerException`.
    (C) If `n1` and `n2` are `Name` objects in a client class, then the expressions `n1.equals(n2)` and `n1.compareTo(n2) == 0` must have the same value.
    (D) The `compareTo` method throws a run-time exception if the parameter is null.
    (E) Since the `Name` class has a `compareTo` method, it *must* provide an implementation for an `equals` method.

ANSWER KEY

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

ANSWERS EXPLAINED

1. **(B)** All the `Math` class methods are static methods, which means you can't use a `Math` object that calls the method. The method is invoked using the class name, `Math`, followed by the dot operator. Thus segment II is correct, and segment I is incorrect. Segment III will cause an error: Since the parameters of `pow` are of type `double`, the result should be stored in a `double`.

2. **(B)** The `Math.sqrt` method must be invoked on a primitive type `double`, which is the reason `d.doubleValue()` is used. (Note that auto-unboxing would apply if you failed to use `d.doubleValue()`, and used just `d` instead.) Segment III fails because you can't use the `Double` constructor to create a new object without using the keyword `new`.

3. **(C)** The value −4.67 must be rounded to −5. Subtracting 0.5 gives a value of −5.17. Casting to `int` truncates the number (chops off the decimal part) and leaves a value of −5. None of the other choices produces −5. Choice A gives the absolute value of `d`: 4.67. Choice B is an incorrect use of `Random`. The parameter for `nextInt` should be an integer n, $n \geq 2$. The method then returns a random `int` k, where $0 \leq k < n$. Choice D is the way to round a *positive* real number to the nearest integer. In the actual case it produces −4. Choice E gives the absolute value of −5, namely 5.

4. **(C)** The statement `double x = Math.random();` generates a random `double` in the range $0 \leq x < 1$. Suppose `probDeath` is 0.67, or 67%. Assuming that random doubles are uniformly distributed in the interval, one can expect that 67% of the time `x` will be in the range $0 \leq x < 0.67$. You can therefore simulate the probability of death by testing if x is between 0 and 0.67, that is, if $x < 0.67$. Thus, `x < probDeath` is the desired condition for plant death, eliminating choices A and B. Choices D and E fail because `(int) probDeath` truncates `probDeath` to 0. The test `x < 0` will always be false, and the test `x == 0` will only be true if the random number generator returned exactly 0, an extremely unlikely occurrence! Neither of these choices correctly simulates the probability of death.

5. **(A)** The expression

```
(int) (Math.random() * 50);
```

returns an `int` from 0 to 49. Therefore, adding 1 shifts the range to be 1 to 50, which was required.

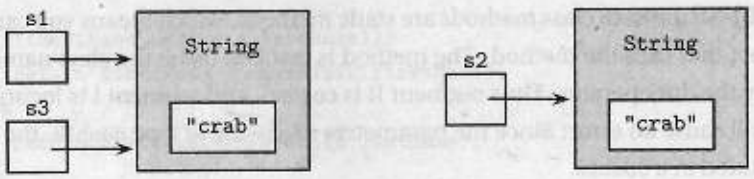
6. **(D)** The `Integer` class has no methods that can change the contents of `i`. However, `i` can be reassigned so that it refers to another object. This happens in both segments I and III. Segment II is wrong because `intValue` is an *accessor*—it cannot be used to change the value of object `i`.

7. **(D)** Test I is correct because it's OK to compare primitive types (in this case `int` values) using `==`. Test III works because `k.toString()` and `m.toString()` are `Strings`, which should be compared with `equals`. Test II is wrong because you can't invoke a method (in this case `equals`) on an `int`.

8. **(C)** The `toString` method of the `Integer` class is invoked, which returns a string representing the value of `intObj`:

```
System.out.println(intObj.toString());      //outputs 9
```

9. **(C)** Here are the memory slots:



Statements II and III are true because the contents of `s1` and `s2` are the same, and the contents of `s3` and `s2` are the same. Statement I is false because `s1` and `s2` are not the same reference. Note that the expression `s1 == s3` would be true since `s1` and `s3` *are* the same reference.

10. **(D)** Note that `"TOMATO"` precedes both `"tomato"` and `"tom"`, since `"T"` precedes `"t"`. Also, `"tom"` precedes `"tomato"` since the length of `"tom"` is less than the length of `"tomato"`. Therefore each of the following is true:

```
strA.compareTo(strB) < 0
strA.compareTo(strC) < 0
strC.compareTo(strB) < 0
```

So

Choice A is T and F which evaluates to F
Choice B is F or F which evaluates to F
Choice C is F and T which evaluates to F
Choice D is T and T which evaluates to T
Choice E is T and F which evaluates to F

11. **(A)** `x` contains the index of the first occurrence of `"m"` in `line`, namely 2. (Remember that `"S"` is at index 0.) The method call `line.substring(10,15)` returns `"silly"`, the substring starting at index 10 and extending though index 14. The method call `line.substring(25,27)` returns `"st"` (don't include the character at index 27!). The concatenation operator, `+`, joins these.

12. **(D)** The first character of the last name starts at the first character after the space. Thus, `startIndex` for `substring` must be `k+1`. This eliminates expression I. Expression II takes all the characters from position `k+1` to the end of the `fullName` string, which is correct. Expression III takes all the characters from position `k+1` to position `fullName.length()-1`, which is also correct.

13. **(E)** Suppose `s` contains `"cat"`. You want `pigString = "at" + "c" + "ay"`. Now the string `"at"` is the substring of `s` starting at position 1 and ending at position `s.length()-1`. The correct substring call for this piece of the word is `s.substring(1,s.length())`, which eliminates choices A, C, and D. (Recall that the first parameter is the starting position, and the second parameter is one position past the last index of the substring.) The first letter of the word—`"c"` in the example—starts at position 0 and ends at position 0. The correct expression is `s.substring(0,1)`, which eliminates choice B.

14. **(A)** Statement I is true whenever `s2` occurs in `s1`. For example, if strings `s1 = "catastrophe"` and `s2 = "cat"`, then `getString` returns `"cat"`. Statement II will never happen. If `s2` is not contained in `s1`, the `indexOf` call will return `-1`. Using a negative integer as the first parameter of `substring` will cause a `StringIndexOutOfBoundsException`. Statement III will be true whenever `s1` equals `s2`.

15. **(C)** The `String temp` represents a single-character substring of `s`. The method examines each character in `s` and, if it is a nonblank, appends it to `str`, which is initially empty. Each assignment `str += temp` assigns a new reference to `str`. Thus, `str` ends up as a copy of `s` but without the blanks. A reference to the final `str` object is returned. Choice A is correct in that `s` is left unchanged, but it is not the *best* characterization of what the method does. Choice B is not precise because an object parameter is never modified: Changes, if any, are performed on a copy. Choices D and E are wrong because the method removes blanks.

16. **(E)** The `compareTo` method returns an `int`, so eliminate choices A and B. In the implementation of `compareTo`, the code segment that applies to the particular example is

```
if (this.getRow() < p.getRow() || ...
    return -1;
```

Since 2 < 4, the value `-1` is returned.

17. **(C)** Expression III is true: The `compareTo` method is implemented to return `0` if two `Position` objects have the same row and column. Expression I is false because `object1 == object2` returns `true` only if `object1` and `object2` are the *same reference*. Expression II is tricky. One would like `p1` and `p3` to be equal since they have the same row and column values. This is not going to happen automatically, however. The `equals` method must explicitly be overridden for the `Position` class. If this hasn't been done, the default `equals` method, which is inherited from class `Object`, will return `true` only if `p1` and `p3` are the same reference, which is not true.

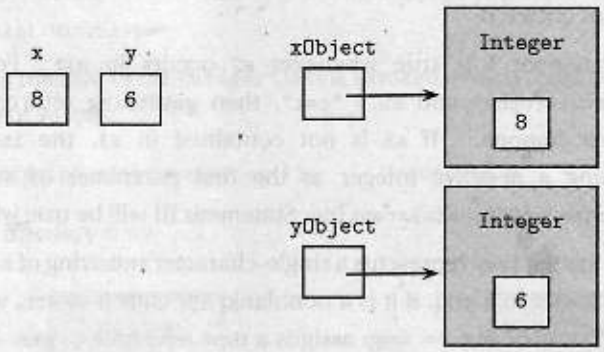
18. **(C)** Recall that primitive types and object references are passed by value. This means that copies are made of the actual arguments. Any changes that are made are made to the *copies*. The actual parameters remain unchanged. Thus, in methods I and II, the parameters will retain their original values and remain unswapped.

    To illustrate, for example, why method II fails, consider this piece of code that tests it:

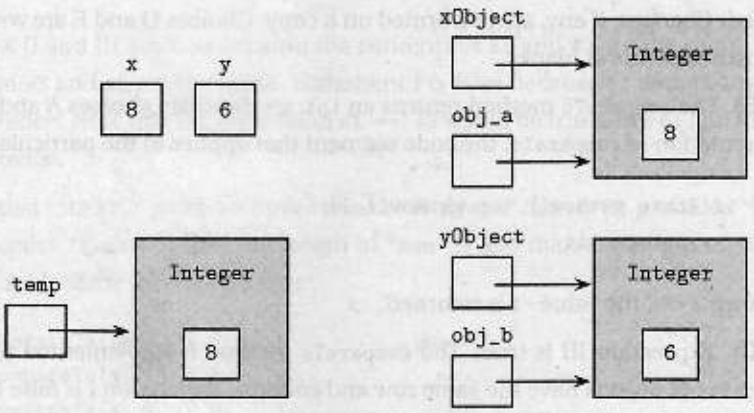
```
public static void main(String[] args)
{
    int x = 8, y = 6;
    Integer xObject = new Integer(x);
    Integer yObject = new Integer(y);
```

```
        swap(xObject, yObject);
        x = xObject.intValue();      //surprise! still has value 8
        y = yObject.intValue();      //surprise! still has value 6
        ...
    }
```

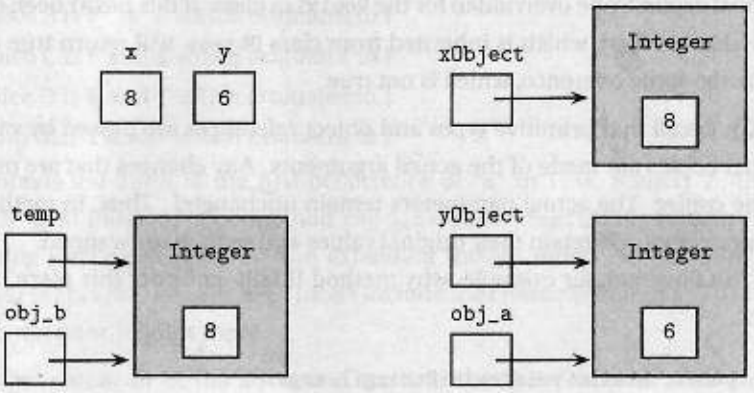
Here are the memory slots before `swap` is called:



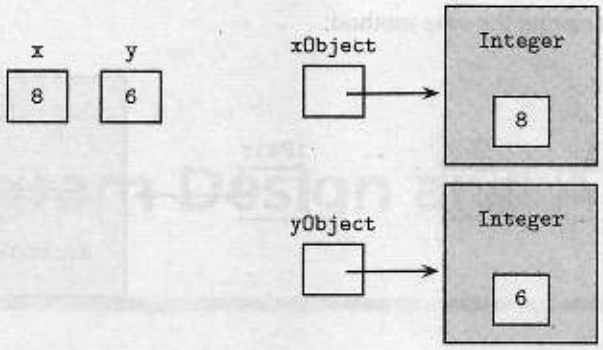
Here they are when `swap` is invoked:



Just before exiting the `swap` method:



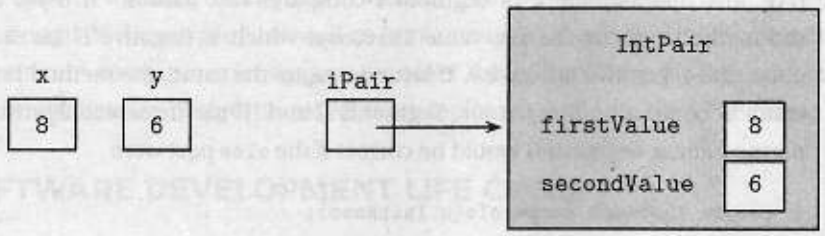
After exiting, `xObject` and `yObject` have retained their original values:



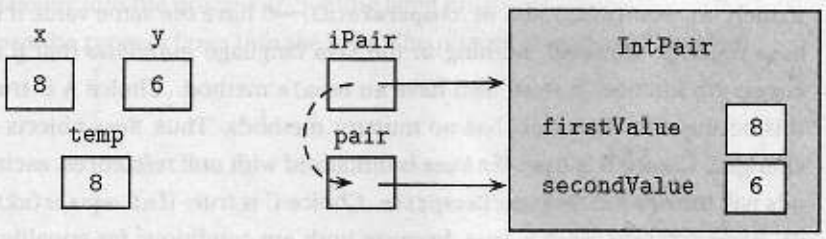
The reason method III works is that instead of the object references being changed, the object *contents* are changed. Thus, after exiting the method, the `IntPair` reference is as it was, but the first and second values have been interchanged. (See explanation to next question for diagrams of the memory slots.) In this question, `IntPair` is used as a wrapper class for a pair of integers whose values need to be swapped.

19. **(B)** The `swap` method has just a single `IntPair` parameter, which eliminates segment I. Segment III fails because `setFirst` and `setSecond` are used incorrectly. These are mutator methods that change an `IntPair` object. What is desired is to return the (newly swapped) first and second values of the pair: Accessor methods `getFirst` and `getSecond` do the trick. To see why this `swap` method works, look at the memory slots.

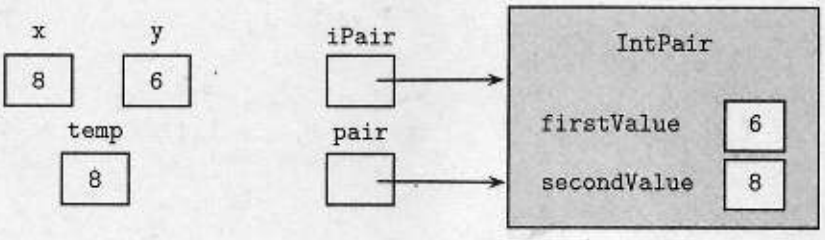
Before the `swap` method is called:



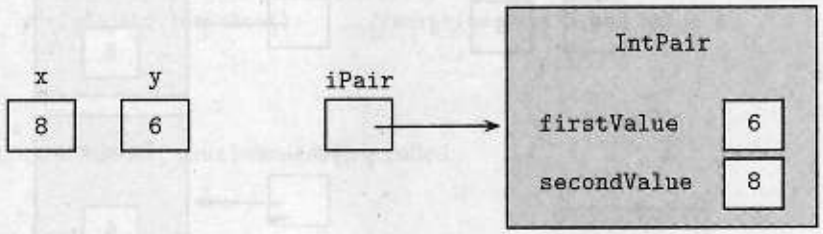
Just after the `swap` method is called:



Just before exiting the `swap` method:

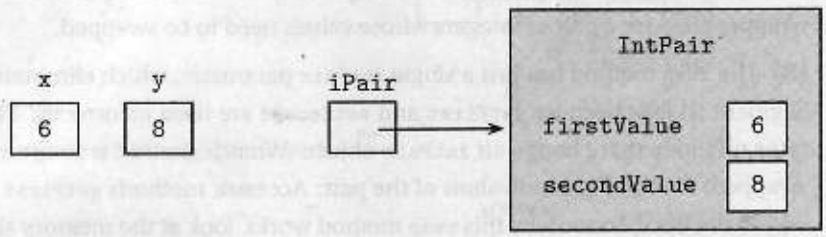


Just after exiting the `swap` method:



After the statements:

```
x = iPair.getFirst();
y = iPair.getSecond();
```



Notice that `x` and `y` have been swapped!

20. **(A)** The first statement of segment I compares last names. If these are different, the method returns the `int` value `lastComp`, which is negative if `lastName` precedes `n.lastName`, positive otherwise. If last names are the same, the method returns the `int` result of comparing first names. Segments II and III use incorrect algorithms for comparing names. Segment II would be correct if the `else` part were

```
return lastName.compareTo(n.lastName);
```

Segment III would be correct if the two `return` statements were interchanged.

21. **(E)** It is *wise* to have an `equals` method that is compatible with the `compareTo` method, namely, `n1.equals(n2)` and `n1.compareTo(n2)==0` have the same value if `n1` and `n2` are `Name` objects. However, nothing in the Java language *mandates* that if a class has a `compareTo` method, it must also have an `equals` method. Choice A is true. You know this because the `Name` class has no mutator methods. Thus, `Name` objects can never be changed. Choice B is true: If a `Name` is initialized with null references, each of the methods will throw a `NullPointerException`. Choice C is true: If `n1.equals(n2)` is true, then `n1.compareTo(n2) == 0` is true, because both are conditions for equality of `n1` and `n2` and should therefore be consistent. Choice D is true: If the parameter is null, the `compareTo` method will throw a `NullPointerException`.