Chapter 2. Java Features, Part 1

2.1. Variables; Arithmetic, Relational, and Logical Operators

Primitive data types included in the subset are `boolean`, `int`, and `double`. In Java, an `int` always takes four bytes, regardless of a particular computer or Java compiler, and its range is from -2^{31} to $2^{31} - 1$. The smallest and the largest integer values are defined in Java as symbolic constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`; use these symbolic constants if you need to refer to the limits of the `int` range. A `double` takes 8 bytes and has a huge range, but its precision is about 15 significant digits.

Remember to declare local variables.

If you declare a variable inside a nested block, make sure it is used only in that block. If you declare a variable in a `for` loop, it will be undefined outside that loop.

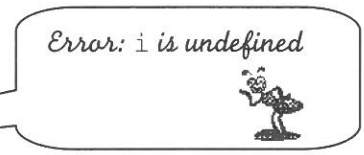
For example:

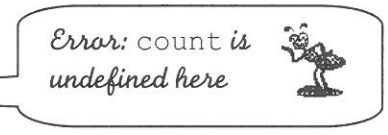
```
public int countMins(int[] a)
{
  if (a.length > 0)
  {
    int iMin = 0;

    for (int i = 1; i < a.length; i++)
    {
      if (a[i] < a[iMin])
        iMin = i;
    }

    int count = 0;

    for (i = 0; i < a.length; i++)
    {
      if (a[i] == a[iMin])
        count++;
    }
  }
  return count;
}
```





A safer version:

```
public int countMins(int[] a)
{
  int iMin = 0;
  int count = 0;

  if (a.length > 0)
  {
    iMin = 0;
    for (int i = 1; i < a.length; i++)
    {
      if (a[i] < a[iMin])
        iMin = i;
    }

    count = 0;

    for (int i = 0; i < a.length; i++)
    {
      if (a[i] == a[iMin])
        count++;
    }
  }
  return count;
}
```

Do not declare `count` *here:*
`int count = 0;`
would be a mistake

You won't be penalized for declarations inside blocks of code, but if you declare important variables near the top of the method's code, it makes it easier to read and may help you avoid mistakes.

1

Which of the following statements is true?

(A) In Java, data types in declarations of symbolic constants are needed only for documentation purposes.
(B) When a Java interpreter is running, variables of the `double` data type are represented in memory as strings of decimal digits with a decimal point and an optional sign.
(C) A variable's data type determines where it is stored in computer memory when the program is running.
(D) A variable's data type determines whether that variable may be passed as a parameter to a particular method.
(E) A variable of the `int` type cannot serve as an operand for the `/` operator.

This question gives us a chance to review what we know about data types.

Choice A is false: the data type of a symbolic constant is used by the compiler. In this sense, symbolic constants are not so different from variables. The difference is that a constant declaration includes the keyword `final`. Class constants are often declared as `static final` variables. For example:

```
public static final double LBS_IN_KG = 2.20462262;
public static final int maxNumSeats = 120;
```

Choice B is false, too. While real numbers may be written in programs in decimal notation, a Java compiler converts them into a special floating-point format that takes eight bytes and is convenient for computations.

Choice C is false. The data type by itself does not determine where the variable is stored. Its location in memory is determined by where and how the variable is declared: whether it is a local variable in a method or an instance variable of a class or a static variable.

Choice E is false, too. In Java, you can write `a/b`, where both `a` and `b` are of type `int`. The result is truncated to an integer.

Choice D is true. For example a value of the type `String` cannot be passed to a method that expects an `int` as a parameter. Sometimes, though, a parameter of a different type may be promoted to the type expected by the method (for example, an `int` can be promoted to a `double` when you call, say, `Math.sqrt(x)` for an `int x`). The answer is D.

Arithmetic operators

The most important thing to remember about Java's arithmetic operators is that the data type of the result, even each intermediate result, is the same as the data type of the operands. In particular, the result of division of one integer by another integer is truncated to an integer.

For example:

```
int n = 3;
double result;

result = (n + 1) * n / 2;          // result is 6.0
result = (n / 2) * (n + 1);        // result is 4.0
result = (1 / 2) * n * (n + 1);    // result is 0.0
```

To avoid truncation you have to watch the data types and sometimes use the *cast operator*. For example:

```
int a, b;
double ratio;
...
ratio = (double)a / b;          // Or: a / (double)b;
// But not ratio = (double)(a/b) -- this is a cast applied too late!
```

If at least one of the operands is a `double`, there is no need to cast the other one — it is promoted to a `double` automatically. For example:

```
int factor = 3;
double x = 2.0 / factor;  // Result: x = 0.6666...
```

2

Which of the following expressions does not evaluate to 0.4?

(A) `(int)4.5 / (double)10;`
(B) `(double)(4 / 10);`
(C) `4.0 / 10;`
(D) `4 / 10.0;`
(E) `(double)4 / (double)10;`

In B the cast to `double` is applied too late — after the ratio is truncated to 0 — so it evaluates to 0. The answer is B.

In the real world we have to worry about the range of values for different data types. For example, a method that calculates the factorial of n as an `int` may overflow the result, even for a relatively small n. In Java, arithmetic overflow is not reported in any way: there is no warning or exception. If an `int` result is greater than or equal to 2^{31}, the leftmost bit, which is the sign bit, may be set to 1, and the result becomes negative.

For the AP exam, you have to be aware of what overflow is, and you need to be aware of the `Integer.MIN_VALUE` and `Integer.MAX_VALUE` constants, but you don't have to know their specific values.

Modulo division

The `%` (modulo division) operator usually applies to two integers: it calculates the remainder when the first operand is divided by the second.

For example:

```
int r;
r = 17 % 3;  // r is set to 2
r = 8 % 2;   // r is set to 0
r = 4 % 5;   // r is set to 4
```

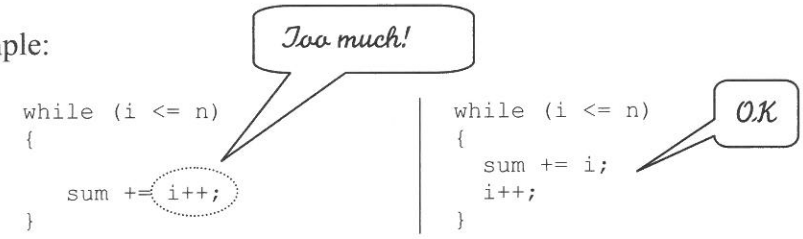
Compound assignments, ++ and --

Compound assignment operators are +=, -=, *= , /=, and %=. `x += y` is the same as `x = x + y`. Other operations follow the same pattern.

There are two forms of the ++ and -- operators in Java. The prefix form ("++k") increments (or decrements) the variable before its value is used in the rest of the expression; the postfix form increments (or decrements) it afterwards.

The AP CS Development Committee discourages the use of ++ and -- in expressions. Use ++ and -- only in separate statements and only in the postfix form, such as in k++ or k--.

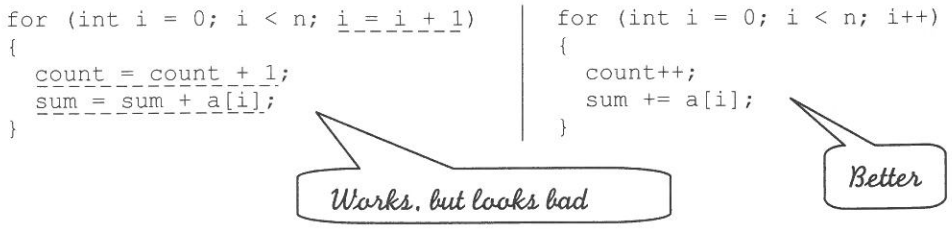
For example:



You won't lose points over ++ or -- in expressions if you use them correctly, but they won't earn you any extra credit, either.

It is bad style <u>not</u> to use increment or compound assignment operators where appropriate.

For example:



Again, this incurs no penalty but doesn't look good.

Arithmetic expressions are too easy to be tested alone. You may encounter them in questions that combine them with logic, iterations, recursion, and so on.

Relational operators

In the AP subset, the relational operators ==, !=, <, >, <=, >= apply to `int`s and `double`s. Remember that "is equal to" is represented by == (not to be confused with =, the assignment operator). `a != b` is equivalent to `!(a == b)`, but != is stylistically better.

The == and != operators can be also applied to two objects, but their meanings are different from what you might expect: they compare the addresses of the objects. The result of == is true if and only if the two variables refer to exactly the same object. You rarely care about that: most likely you want to compare the contents of two objects, for instance two strings. Then you need to use the `equals` or `compareTo` method.

For example:

```
if (str.equals("Stop")) ...
```

On the other hand, == or != must be used when you compare an object to `null`.

`null` is a Java reserved word that stands for a reference with a zero value. It is used to indicate that a variable currently does not refer to any valid object. For example:

```
if (str != null && str.equals("Stop")) ...
// str != null avoids NullPointerException --
// can't call a null's method
```

You can write instead:

```
if ("Stop".equals(str)) ...
```

This works because `"Stop"` is not `null`; it works even if `str` is `null`.

Logical operators

The logical operators `&&`, `||`, and `!` normally apply to Boolean values and expressions. For example:

```
boolean found = false;
...
while (i >= 0 && !found)
{
  ...
}
```

```
boolean found = false;
...
while (i >= 0 && found == false)
{
  ...
}
```

Works, but is more verbose

Do not write

```
while (... && !found == true)
```

— this works, but "`== true`" is redundant.

3

Assuming that `x`, `y`, and `z` are integer variables, which of the following three logical expressions are equivalent to each other, that is, have equal values for all possible values of `x`, `y`, and `z`?

I. `(x == y && x != z) || (x != y && x == z)`

II. `(x == y || x == z) && (x != y || x != z)`

III. `(x == y) != (x == z)`

(A) None of the three
(B) I and II only
(C) II and III only
(D) I and III only
(E) I, II, and III

Expression III is the key to the answer: all three expressions state the fact that exactly one out of the two equalities, `x == y` or `x == z`, is true. Expression I states that either the first and not the second or the second and not the first is true. Expression II states that one of the two is true and one of the two is false. Expression III simply states that they have different values. All three boil down to the same thing. The answer is E.

De Morgan's Laws

The exam is likely to include questions on De Morgan's Laws:

`!(a && b)` is the same as `!a || !b`
`!(a || b)` is the same as `!a && !b`

4

The expression `!((x <= y) && (y > 5))` is equivalent to which of the following?

(A) `(x <= y) && (y > 5)`
(B) `(x <= y) || (y > 5)`
(C) `(x >= y) || (y < 5)`
(D) `(x > y) || (y <= 5)`
(E) `(x > y) && (y <= 5)`

The given expression is pretty long, so if you try to plug in specific numbers you may lose a lot of time. Use De Morgan's Laws instead:

!((x <= y) && (y > 5))

!(x <= y) || !(y > 5)

(x > y) || (y <= 5)

When ! is distributed, && changes into || and vice-versa

The answer is D.

Short-circuit evaluation

An important thing to remember about the logical operators && and || is *short-circuit evaluation*. If the value of the first operand is sufficient to determine the result, then the second operand is <u>not</u> evaluated.

5

Consider the following code segment:

```
int x = 0, y = 3;
String op = "/";

if (op.equals("/") && (x != 0) && (y/x > 2))
{
  System.out.println("OK");
}
else
{
  System.out.println("Failed");
}
```

Which of the following statements about this code is true?

(A) There will be a compile error because `String` and `int` variables are intermixed in the same condition.
(B) There will be a run-time divide-by-zero error.
(C) The code will compile and execute without error; the output will be `OK`.
(D) The code will compile and execute without error; the output will be `Failed`.
(E) The code will compile and execute without error; there will be no output.

Choices A and E are just filler answers. Since `x` is equal to 0, the condition cannot be true, so C should be rejected, too. The question remains whether it crashes or executes. In Java, once `x != 0` fails, the rest of the condition, `y/x > 2`, won't be evaluated, and `y/x` won't be computed. The answer is D.

The relational expressions in the above question are parenthesized. This is not necessary because relational operators always take precedence over logical operators. If you are used to lots of parentheses, use them, but you can skip them as well. For example, the Boolean expression from Question 5 can be written with fewer parentheses:

```
if (op.equals("/") && x != 0 && y/x > 2) ...
```

`&&` also takes precedence over `||`, but it's clearer to use parentheses when `&&` and `||` appear in the same expression. For example:

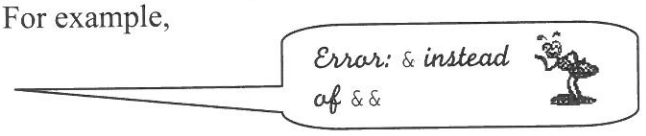
```
if ((0 < a && a < top) || (0 < b && b < top)) ...
```

Bitwise logical operators

The bitwise logical operators, &, |, ^, and ~, are <u>not</u> in the AP Java subset and are <u>not</u> tested on the AP exam. You don't have to worry about them.

Programmers use these operators to perform logical operations on individual bits, usually in `int` values. Unfortunately, Java also allows you to apply these operators to `boolean` values, and, when used that way, these operators do not comply with short-circuit evaluation. This may lead to a nasty bug if you inadvertently write `&` instead of `&&` or `|` instead of `||`. For example,

```
if (x != 0 & y/x > 2)
```



results in a division by 0 exception when `x = 0`.

2.2. Conditional Statements and Loops

You can use simplified indentation for `if-else-if` statements.

For example:

```
if (score >= 70)
  grade = 5;
else if (score >= 60)
  grade = 4;
...
else
  grade = 1;
```

But don't forget braces and proper indentation for nested `if`s. For example:

```
if (exam.equals("Calculus AB"))
{
  if (score >= 60)
    grade = 5;
  else if ...
    ...
}
else if (exam.equals("Calculus BC"))
{
  if (score >= 70)
    grade = 5;
  else if ...
    ...
}
```

6

Consider the following code segment, where `m` is a variable of the type `int`:

```
if (m > 0)
{
  if ((1000 / m) % 2 == 0)
    System.out.println("even");
  else
    System.out.println("odd");
}
else
  System.out.println("not positive");
```

Which of the following code segments are equivalent to the one above (that is, produce the same output as the one above regardless of the value of `m`)?

I.
```
if (m <= 0)
  System.out.println("not positive");
else if ((1000 / m) % 2 == 0)
  System.out.println("even");
else
  System.out.println("odd");
```

II.
```
if (m > 0 && (1000 / m) % 2 == 0)
  System.out.println("even");
else if (m <= 0)
  System.out.println("not positive");
else
  System.out.println("odd");
```

III.
```
if ((1000 / m) % 2 == 0)
{
  if (m <= 0)
    System.out.println("not positive");
  else
    System.out.println("even");
}
else
{
  if (m <= 0)
    System.out.println("not positive");
  else
    System.out.println("odd");
}
```

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

Segment I can actually be reformatted as:

```
if (m <= 0)
  System.out.println("not positive");
else
{
  if ((1000 / m) % 2 == 0)
    System.out.println("even");
  else
    System.out.println("odd");
}
```

So it's the same as the given segment with the condition negated and `if` and `else` swapped. Segment II restructures the sequence, but gives the same result. To see that, we can try different combinations of true/false for `m <= 0` and `(1000 / m) % 2 == 0`. Segment III would work, too, but it has a catch: it doesn't work when `m` is equal to 0. The answer is C.

`for` *and* `while` *loops*

The `for` loop,

```
for (initialize; condition; change)
{
  ...   // Do something
}
```

is equivalent to the `while` loop:

```
initialize;
while (condition)
{
  ...   // Do something
  change;
}
```

change can mean any change in the values of the variables that control the loop, such as incrementing or decrementing an index or a counter.

`for` loops are shorter and more idiomatic than `while` loops in many situations. For example:

```
int i = 0;
while (i < a.length)
{
  sum += a[i];
  i++;
}
```

OK

```
for (int i = 0; i < a.length; i++)
  sum += a[i];
```

Better, more idiomatic

In a `for` or `while` loop, the condition is evaluated at the beginning of the loop and the program does not go inside the loop if the condition is false. Thus, the body of the loop may be skipped entirely if the condition is false at the very beginning.

7

Consider the following methods:

```
public int fun1(int n)
{
  int product = 1;
  for (int k = 2; k <= n; k++)
  {
    product *= k;
  }
  return product;
}
```

```
public int fun2(int n)
{
  int product = 1;
  int k = 2;
  while (k <= n)
  {
    product *= k;
    k++;
  }
  return product;
}
```

For which integer values of `n` do `fun1(n)` and `fun2(n)` return the same result?

(A) Only `n` > 1
(B) Only `n` < 1
(C) Only `n` == 1
(D) Only `n` >= 1
(E) Any integer `n`

The best approach here is purely formal: since the initialization, condition, and increment in the `for` loop in `fun1` are the same as the ones used with the `while` loop in `fun2`, the two methods are equivalent. The answer is E.

8

Consider the following code segment:

```
while (x > y)
{
  x--;
  y++;
}
System.out.println(x - y);
```

Assume that `x` and `y` are `int` variables and their values satisfy the conditions $0 \le x \le 2$ and $0 \le y \le 2$. Which of the following describes the set of all possible outputs?

(A) 0
(B) -1, 1
(C) -1, -2
(D) 0, -1, -2
(E) 0, -1, 1, -2, 2

If $x \le y$ to begin with, then the `while` loop is never entered and the possible outputs are 0, -1, and -2 (for the pairs (0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)). If $x > y$, then the loop is entered and after the loop we must have $x \le y$, so $x - y$ cannot be positive. The answer is D.

OBOBs

When coding loops, beware of the so-called "off-by-one bugs" ("OBOBs"). These are mistakes of running through the loop one time too many or one time too few.

9

Suppose the `isPrime` method is defined:

```
// Returns true if p is a prime, false otherwise.
// Precondition: p >= 2
public static boolean isPrime(int p) { < implementation not shown > }
```

Given

```
int n = 101;
int sum = 0;
```

Which of the following code segments correctly computes the sum of all prime numbers from 2 to 101, inclusive?

(A)
```
while (n != 2)
{
  n--;
  if (isPrime(n)) sum += n;
}
```

(B)
```
while (n >= 2)
{
  n--;
  if (isPrime(n)) sum += n;
}
```

(C)
```
while (n != 2)
{
  if (isPrime(n)) sum += n;
  n--;
}
```

(D)
```
while (n >= 2)
{
  if (isPrime(n)) sum += n;
  n--;
}
```

(E)
```
while (n >= 2 && isPrime(n))
{
  sum += n;
  n--;
}
```

It is bad style to start the body of a loop with a decrement, so Choices A and B are most likely wrong. Indeed, both A and B miss 101 (which happens to be a prime) because `n` is decremented too early. In addition, Choice B eventually calls `isPrime(1)`, violating `isPrime`'s precondition. Choice C misses 2 — an OBOB on the other end. Choice E might look plausible for a moment, but it actually quits as soon at it encounters the first non-prime number. The answer is D.

The "for each" loop

The "for each" loop was first introduced in Java 5.0. This loop has the form

```
for (SomeType x : a)  // read: "for each x in a"
{
  ...   // do something
}
```

where `a` is an array or an `ArrayList` (or another `List` or "collection") that holds values of *SomeType*.

For example:

```
int[] scores = {87, 95, 76};
for (int score : scores)
  System.out.print(score + " ");
```

works the same way as

```
int[] scores = {87, 95, 76};
for (int i = 0; i < scores.length; i++)
{
  int score = scores[i];
  System.out.print(score + " ");
}
```

Both will display

```
87 95 76
```

Another example:

```
List<String> plants = new ArrayList<String>();
plants.add("Bougainvillea");
plants.add("Hibiscus");
plants.add("Poinciana");

for (String name : plants)
  System.out.print(name + " ");
```

will display

```
Bougainvillea Hibiscus Poinciana
```

Note that the "for each" loop traverses the array or list only in the forward direction and does not give you access to the indices of the values stored in the array or list. For example, a "for each" loop won't be very useful if you need to find the position of the first occurrence of a target value in a list.

If you need access to indices, use a `for` or `while` loop.

A "for each" loop does not allow you to set an element of an array to a new value, because it works with a copy of the element's value, and so the original values of an array will remain unchanged.

For example, given an `int` array `arr`,

```
for (int x : arr)
  x = 1;
```

will leave `arr` unchanged.

If you need to replace or delete elements in an array or list, use a `for` or `while` loop.

If an array or list holds objects, then you can call an object's method inside a "for each" loop, so you can potentially change object's state if the method changes it.

`break` *and* `return` *in loops*

In Java it is okay to use `break` and `return` inside loops. `return` immediately quits the method from any place inside or outside a loop. This may be a convenient shortcut, especially when you have to quit a method from within a nested loop. For example:

```
// Returns true if all values in list are different, false otherwise
public boolean allDifferent(int[] list)
{
  for (int i = 0; i < list.length; i++)
    for (int j = i + 1; j < list.length; j++)
      if (list[i] == list[j])
        return false;
  return true;
}
```

You can also use `break`, but it may be dangerous and is not in the AP subset. Remember that in a nested loop, `break` takes you out of the inner loop but not out of the outer loop. Avoid redundant, verbose, and incorrect code like this:

```
// Returns true if all values in list are different, false otherwise
public boolean allDifferent(int[] list)
{
  boolean foundDuplicates;

  for (int i = 0; i < list.length; i++)
  {
    for (int j = i + 1; j < list.length; j++)
    {
      if (list[i] == list[j])
      {
        foundDuplicates = true;
        break;
      }
      else
      {
        foundDuplicates = false;
      }
    }
  }
  if (foundDuplicates == true)
    return false;
  else
    return true;
}
```

Out of the inner `for` *but still in the outer* `for`.

If you insist on using Boolean flags, you need to be extra careful:

```
// Returns true if all values in list are different, false otherwise
public boolean allDifferent(int[] list)
{
  boolean foundDuplicates = false;

  for (int i = 0; i < list.length; i++)
  {
    for (int j = i + 1; j < list.length; j++)
    {
      if (list[i] == list[j])
      {
        foundDuplicates = true;
        break;
      }
    }
  }
  return !foundDuplicates;
}
```

The `continue` statement is not in the AP subset and should be avoided.

2.3. Strings

In Java, a string is an object of the type `String`, and, as with other types of objects, a `String` variable holds a reference to (the address of) the string.

Strings are immutable: no string methods can change the string.

An assignment statement

```
str1 = str2;
```

copies the reference from `str2` into `str1`, so they both refer to the same memory location.

A *literal string* is a string of characters within double quotes. A literal string may include "escape sequences" `\n` (newline), `\"` (a double quote), `\\` (one backslash), and other escapes. For example,

```
System.out.print("Hello\n");
```

has the same effect as

```
System.out.println("Hello");
```

The `String` class supports the + and += operators for concatenating strings.

> **`String` is the only class in Java that supports special syntax for using operators on its objects (only + and +=).**

The operator

```
s1 += s2;
```

appends `s2` to `s1`. In reality it creates a new string by concatenating `s1` and `s2` and then sets `s1` to refer to the new string. It is equivalent to

```
s1 = s1 + s2;
```

10

What is the output of the following code segment?

```
String str1 = "Happy ";
String str2 = str1;
str2 += "New Year! ";
str2.substring(6);
System.out.println(str1 + str2);
```

(A) `Happy New Year!`
(B) `Happy Happy New Year!`
(C) `Happy New Year! New Year!`
(D) `Happy New Year! Happy New Year!`
(E) `Happy New Year! Happy`

After `str2 = str1`, `str1` and `str2` point to the same memory location that contains `"Happy "`. But after `str2 += "New Year! "`, these variables point to different things: `str1` remains `"Happy "` (strings are immutable) while `str2` becomes `"Happy New Year! "`. `str2.substring(6)` does not change `str2` — it calls its `substring` method but does not use the returned value (a common beginner's mistake: again, strings are immutable). The answer is B.

`String` *methods*

The `String` methods included in the AP subset are:

```
int length()
boolean equals(Object other)
int compareTo(String other)
String substring(int from)
String substring(int from, int to)
int indexOf(String s)
```

Always use the `equals` method to compare a string to another string. The == and != operators, applied to two strings, compare their addresses, not their values.

`str1.equals(str2)` returns `true` if and only if `str1` and `str2` have the same values (that is, consist of the same characters in the same order).

`str1.compareTo(str2)` returns a positive number if `str1` is greater than `str2` (lexicographically), zero if they are equal, and a negative number if `str1` is less than `str2`.

`str.substring(from)` returns a substring of `str` starting at the `from` position to the end, and `str.substring(from, to)` returns `str`'s substring starting at the `from` position and up to but not including the `to` position (so the length of the returned substring is `to - from`). Positions are counted from 0. For example, `"Happy".substring(1, 4)` would return `"app"`.

`str.indexOf(s)` returns the starting position of the first occurrence of the substring `s` in `str`, or `-1` if not found.

11

Consider the following method:

```
public String process(String msg, String delim)
{
  int pos = msg.indexOf(delim);
  while (pos >= 0)
  {
    msg = msg.substring(0, pos) + " "
                          + msg.substring(pos + delim.length());
    pos = msg.indexOf(delim);
  }
  return msg;
}
```

What is the output of the following code segment?

```
String rhyme = "Twinkle\ntwinkle\nlittle star";
String rhyme2 = process(rhyme, "\n");
System.out.println(rhyme + "\n" + rhyme2);
```

(A)
```
little star
Twinkle twinkle little star
```

(B)
```
little star
Twinkle
twinkle
little star
```

(C)
```
Twinkle
twinkle
little star
Twinkle winkle ittle tar
```

(D)
```
Twinkle
twinkle
little star
Twinkle twinkle
little star
```

(E)
```
Twinkle
twinkle
little star
Twinkle twinkle little star
```

process receives and works with a copy of a reference to the original string (see Section 3.3). The method can reassign the copy, as it does here, but the original reference still refers to the same string. This consideration, combined with immutability of strings, assures us that rhyme remains unchanged after the call process(rhyme).

The rhyme string includes two newline characters, and, when printed, it produces

```
Twinkle
twinkle
little star
```

So the only possible answers are C, D or E. Note that we can come to this conclusion before we even look at the process method! This method repeatedly finds the first occurrence of delim in msg, cuts it out, and replaces it with a space. No other characters are replaced or lost. The resulting message prints on one line. The answer is E.

2.4. Integer and Double Classes

In Java, variables of primitive data types (int, double, etc.) are not objects. In some situations it is convenient to represent numbers as objects. For example, you might want to store numeric values in an ArrayList (see Section 2.6), but the elements of an ArrayList must be objects. The java.lang package provides several "wrapper" classes that represent primitive data types as objects. Two of these classes, Integer and Double, are in the AP subset.

The Integer class has a constructor that takes an int value and creates an Integer object representing that value. The intValue method of an Integer object returns the value represented by that object as an int. For example:

```
Integer obj = new Integer(123);
...
int num = obj.intValue();  // num gets the value of 123
```

Likewise, Double's constructor creates a Double object that represents a given double value. The method doubleValue returns the double represented by a Double object:

```
Double obj = new Double(123.45);
...
double x = obj.doubleValue();  // x gets the value of 123.45
```

The `Integer` class also provides two symbolic constants that describe the range for `int` values:

```
public static final int MIN_VALUE = -2147483648;
public static final int MAX_VALUE = 2147483647;
```

Use the `equals` method of the `Integer` or of the `Double` class if you want to compare two `Integer` or two `Double` variables, respectively.

For example:

```
Integer a = new Integer(...);
Integer b = new Integer(...);
...
if (a.equals(b))
  ...
```

If you apply a relational operator == or `!=` to two `Integer` or two `Double` variables, you will compare their addresses, not values. This is rarely, if ever, what you want to do.

The `Integer` and `Double` classes implement the `Comparable<Integer>` and `Comparable<Double>` interfaces, respectively (see Section 4.6), so each of these classes has a `compareTo` method. As usual, `obj1.compareTo(obj2)` returns a positive integer if `obj1` is greater than `obj2` (that is, `obj1`'s numeric value is greater than `obj2`'s numeric value), a negative integer if `obj1` is less than `obj2`, and zero if their numeric values are equal.

Autoboxing

Starting with Java 5.0, the compiler in certain situations automatically converts values of primitive data types, (`int`, `double`, etc.) into the corresponding wrapper types (`Integer`, `Double`, etc.). This feature is called *autoboxing* (or *autowrapping*). For example, in

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(5);
```

the second line is compiled as

```
numbers.add(new Integer(5));
```

Likewise, where appropriate, the compiler performs *autounboxing*. For example,

```
int num = numbers.get(0);
```

in effect compiles as

```
int num = numbers.get(0).intValue();
```

Autoboxing and autounboxing are not in the AP subset, but you won't be penalized for relying on them.

2.5. Arrays

There are two ways to declare and create a one-dimensional array:

```
SomeType[] a = new SomeType[size];
SomeType[] b = {value0, value1, ..., valuen-1};
```

For example:

```
double[] samples = new double[100];
int[] numbers = {1, 2, 3};
String[] cities = {"Atlanta", "Boston", "Cincinnati", "Dallas"};
```

The first declaration declares an array of `double`s of size 100. Its elements get default values (zeroes), but this fact is not in the AP subset. The second declaration creates an array of `int`s of size 3 with its elements initialized to the values 1, 2, and 3. The third declaration declares and initializes an array of four given strings.

We can refer to `a`'s elements as `a[i]`, where `a` is the name of the array and `i` is an index (subscript), which can be an integer constant, variable, or expression.

Indices start from 0.

`a.length` refers to the size of the array. (In an array, `length` is not a method, but rather works like a public instance variable, hence no parentheses.)

`a[a.length-1]` refers to the last element.

Once an array is created, its size cannot be changed. The only way to expand an array is to create a bigger array and copy the contents of the original array into the new one. The old array is discarded (or, more precisely, recycled by a process called "garbage collection"). For example: