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.

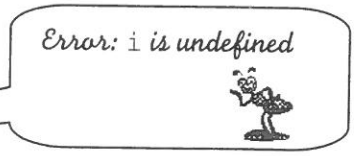
For example:

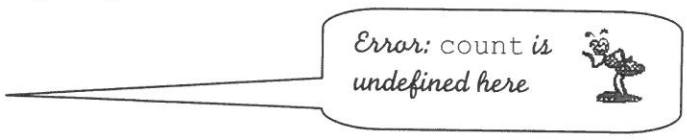
```
public int countMins(int[] a)
{
  if (a.length > 0)
  {
    int iMin = 0;

    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.

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:

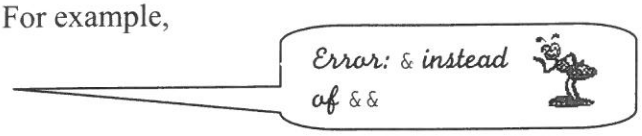
```
if ((0 < a && a < top) || (0 < b && b < top)) ...
```

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 `doubles` 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 `ints` 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:

```
int[] a = new int[100];
...
int[] temp = new int[a.length * 2];
for (int i = 0; i < a.length; i++)
  temp[i] = a[i];
a = temp;   // reassign a to the new array; the old array is discarded
```

If `a` and `b` are arrays, `a = b` does not copy elements from `b` into `a`: it just reassigns the reference `a` to `b`, so that both `a` and `b` refer to the same array.

The following method reverses the order of elements in an array of strings:

```
public void reverse(String[] words)
{
  int i = 0, j = words.length - 1;

  while (i < j)
  {
    String temp = words[i]; words[i] = words[j]; words[j] = temp;
    i++;
    j--;
  }
}
```

The Java Virtual Machine (the run-time interpreter) checks that an array index is within the valid range, from 0 to `array.length-1`. If an index value is invalid, the interpreter "throws" an `ArrayIndexOutOfBoundsException` — reports a run-time error, the line number for the offending program statement, and a trace of the method calls that led to it.

An *exception* is a run-time error, not a compile-time error.

The compiler cannot catch errors that will be caused by certain circumstances that occur during program execution, such as a variable used as an array index whose value has gone out of the allowed range.

12

Suppose the method `int sign(int x)` returns 1 if x is positive, -1 if x is negative, and 0 if x is 0. Given

```
int[] nums = {-2, -1, 0, 1, 2};
```

what are the values of the elements of `nums` after the following code is executed?

```
for (int k = 0; k < nums.length; k++)
{
  nums[k] -= sign(nums[k]);
  nums[k] += sign(nums[k]);
}
```

(A) -2, -1, 0, 1, 2
(B) -1, 0, 0, 0, 1
(C) 0, 0, 0, 0, 0
(D) -2, 0, 0, 2, 3
(E) -2, 0, 0, 0, 2

Remember that the first statement within the loop changes `nums[k]`, which may change the sign of `nums[k]`, too. Jot down a little table:

Before		*After* -=		*After* +=
a[k]	sign *of* a[k]	a[k]	sign *of* a[k]	a[k]
-2	*-1*	*-1*	*-1*	*-2*
-1	*-1*	*0*	*0*	*0*
0	*0*	*0*	*0*	*0*
1	*1*	*0*	*0*	*0*
2	*1*	*1*	*1*	*2*

The answer is E.

13

Consider the following method:

```
// Returns true if there are no two elements among
// counts[0], ... counts[n-1], whose values are the same
// or are consecutive integers; otherwise, returns false.
// Precondition: counts contains n values, n > 1
public boolean isSparse(int[] counts, int n)
{
  < missing code >
}
```

Which of the following code segments can be used to replace < *missing code* > so that the method `isSparse` works as specified?

I.
```
for (int j = 0; j < n; j++)
{
  for (int k = j + 1; k < n; k++)
  {
    int diff = counts[j] - counts[k];
    if (diff >= -1 && diff <= 1)
      return false;
  }
}
return true;
```

II.
```
for (int j = 1; j < n; j++)
{
  for (int k = 0; k < j; k++)
  {
    int diff = counts[j] - counts[k];
    if (diff >= -1 && diff <= 1)
      return false;
  }
}
return true;
```

III.
```
for (int j = 0; j < n; j++)
{
  for (int k = 1; k < n; k++)
  {
    int diff = counts[j] - counts[k];
    if (Math.abs(diff) <= 1)
      return false;
  }
}
return true;
```

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

Note that in this question not all `counts.length` elements of `counts` are used, just the first `n`. Their subscripts range from 0 to `n-1`. The precondition states that `n > 1`, so there is no need to worry about an empty array or an array of just one element. Looking at the inner loop in each segment you can quickly see that they work the same way. So the difference is how the loops are set up; more precisely, the limits in which the indices vary. In Option I the outer loop starts with the first item in the list; the inner loop compares it with each of the subsequent items. In Option II the outer loop starts with the second item in the list; the inner loop compares it with each of the preceding items. Both of these are correct and quite standard in similar algorithms. This eliminates A, B, and D. Option III at first seems harmless, too, but it has a catch: the inner loop doesn't set a limit for `k` that depends on `j`, so when `j` is greater than 0, `k` may eventually take the same value as `j` (such as `j = 1, k = 1`). The method will erroneously detect the same value in `counts` when it is actually comparing an item to itself. The answer is C.

Array return type

Occasionally you may need to return an array from a method. Suppose you want to restructure the `reverse` method above so that it returns a new array containing the values from a given array in reverse order. The original array remains unchanged. The method can be coded as follows:

```
public String[] reverse(String[] words)
{
  String[] result = new String[words.length];

  for (int i = 0; i < words.length; i++)
    result[i] = words[words.length - 1 - i];

  return result;
}
```

Two-dimensional arrays

The AP subset includes rectangular two-dimensional arrays. These are similar to one-dimensional arrays but use two indices, one for the row and one for the column. For example:

```
double[][] matrix = new double[3][5]; // 3 rows by 5 cols
int r, c;
...
matrix[r][c] = 1.23;
```

In fact a two dimensional array is an array of arrays, its rows. If `m` is a two-dimensional array, `m[0]` is its first row, `m[1]` is the second row, and so forth.

Therefore,

if `m` is a two-dimensional array, `m.length` represents the number of rows.

`m[k].length` represents the length of the *k*-th row.

Only rectangular 2-D arrays are considered in the AP subset; therefore, the lengths of all the rows are the same. `m[0].length` represents the number of columns.

The following method calculates the sums of the values in each column of a 2-D array, places these sums into a new 1-D array of sums, and returns that array:

```
// Returns a 1-D array containing sums of all the values
// in each column of table
public double[] totalsByColumn(double[][] table)
{
  int nRows = table.length;
  int nCols = table[0].length;
  double[] totals = new double[nCols];

  for (int c = 0; c < nCols; c++)
  {
    totals[c] = 0.0;   // optional: default
    for (int r = 0; r < nRows; r++)
      totals[c] += table[r][c];
  }
  return totals;
}
```

14

Consider the following code segment:

```
String[][] m = new String[6][3];

for (int k = 0; k < m.length; k++)
{
  m[k][m[0].length - 1] = "*";
}
```

Which of the following best describes the result when this code segment is executed?

(A) All elements in the first row of `m` are set to `"*"`
(B) All elements in the last row of `m` are set to `"*"`
(C) All elements in the last column of `m` are set to `"*"`
(D) The code has no effect
(E) `ArrayIndexOutOfBoundsException` is reported

The first index is the row, and the `for` loop is set up for all rows. The answer is C.

Since a two-dimensional array is treated as an array of one-dimensional arrays, a naive "for each" loop won't work for traversing a 2-D array.

To make a "for each" loop work on a 2-D array, you need nested loops. For example:

```
int[][] a = {{1, 2, 3}, {4, 5, 6}};

for (int[] row : a)
{
  for (int x : row)
    System.out.print(x + " ");
  System.out.println();
}
```

Java library class `Math` has static methods `abs`, `sqrt`, `pow`, `random`. For your convenience, it also includes the `public static final` "variables" `Math.PI`, which represents π, the ratio of a circle's circumference to its diameter, and `Math.E`, which represents e, the base of the natural logarithm. It might be useful (but not required) for you to know that `Math` also has static methods `min` and `max`, which return the smaller and the larger, respectively, of two numbers (`int`s or `double`s).

Outside the class, public static methods are called and public static constants are accessed using the dot notation, with the class's name as the prefix. For example:

```
double volume = 4.0 / 3.0 * Math.PI * Math.pow(r, 3);
```

3.3. Method Calls

In Java, all methods belong to classes.

It is universal Java style that all method names start with a lowercase letter.

An *instance* method is called for a particular object; then the object's name and a dot are used as a prefix in a call, as in `obj.someMethod(...)`. If an object's method calls another method of the same object, the prefix is not needed and you write simply `otherMethod(...)`, which is the same as `this.otherMethod(...)`.

Class (*static*) methods belong to the class as a whole and are called using the class's name with a dot as a prefix. For example: `Math.max(x, y)` or `School.getNationalEnrollment(...)`.

A method takes a specific number of parameters of specific data types. (Starting with Java 5.0, a method can be defined with a variable number of parameters, as in `System.out.printf`; however, this feature is not in the AP subset.) Some methods take no parameters, such as `List`'s `size()` or `String`'s `length()`. A method call may include a whole expression as a parameter; then the expression is evaluated first and the result is passed to the method. An expression may include calls to other methods. For example:

```
double x, y;
...
x = Math.sqrt(Math.abs(2*y - 1));
```

A method usually returns a value of the specified data type, but a `void` method does not return any value. The return type is specified in the method's header. The return value is specified in the `return` statement.

It is considered a serious error (1 point deduction) to read the new values for a method's parameters from `System.in` inside the method. It is also a serious error to print the return value to `System.out` inside a method (when it is not requested) and another serious error to omit a `return` statement in a non-`void` method.

For example:

```
// Returns the sum of all integers from 1 to n.
// Precondition: n >= 1
public int addNumbers(int n)
{
  int sum = 0;

  n = System.in.read();

  for (int k = 1; k <= n; k++)
  {
    sum += k;
  }

  System.out.println(sum);

  return sum;
}
```

Error: `n` is passed to this method from `main` or from another calling method

Error: Not specified in the method description

18

`Math`'s static method `min` returns the value of the smaller of two integers. If `a`, `b`, `c`, and `m` are integer variables, which of the following best describes the behavior of a program with the following statement?

```
m = Math.min(Math.min(a, c), Math.min(b, c));
```

(A) The statement has a syntax error and will not compile.
(B) The program will run but go into an infinite loop.
(C) `a` will get the smaller value of `a` and `c`; `b` will get the smaller value of `b` and `c`; `m` will get the smallest value of `a`, `b`, and `c`.
(D) `m` will be assigned the smallest of the values `a`, `b`, and `c`.
(E) None of the above

Any expression of the appropriate data type, including a method call that returns a value of the appropriate data type, may be used in a larger expression or as a parameter to a method. The code above is basically equivalent to:

```
int temp1 = Math.min(a, c);
int temp2 = Math.min(b, c);
m = Math.min(temp1, temp2);
```

So `m` gets the smallest of the three values. The answer is D.

Parameters of primitive data types

In Java, all parameters of primitive data types are passed to methods "by value."

When a parameter is passed by value, the method works with a copy of the variable passed to it, so it has no way of changing the value of the original.

19

Consider the following method:

```
public void fun(int a, int b)
{
  a += b;
  b += a;
}
```

What is the output from the following code?

```
int x = 3, y = 5;
fun(x, y);
System.out.println(x + " " + y);
```

(A) 3 5
(B) 3 8
(C) 3 13
(D) 8 8
(E) 8 13

`x` and `y` are `int`s, so they are passed to `fun` by value. `fun` works with copies of `x` and `y`, named `a` and `b`. What is happening inside `fun` is irrelevant here because `x` and `y` do not change after the method call. The answer is A.

Objects passed to methods

All objects are passed to methods as references. A method receives a copy of a reference to (the address of) the object.

When a variable gets an "object" as a value, what it actually holds is a reference to (the address of) that object. Likewise, when an object is passed to a method, the method receives a copy of the object's address, and therefore it potentially can change the original object. Usually all instance variables of an object are private, so to change the object, the method would have to call one of the object's *modifier* methods.

But note that the `String`, `Integer`, and `Double` classes represent *immutable* objects, that is, objects without modifier methods. Even though these objects are passed to methods as references, no method can change them. For example, there is no way in Java to write a method

```
// Converts s to upper case
public void toUpperCase(String s)
{
  ...
}
```

because the method has no way of changing the string passed to it. For immutable objects, the method has to create and return a new object with the desired properties:

```
// Returns s converted to upper case
public String toUpperCase(String s)
{
  ...
}
```

Arrays passed to methods

A one- or two-dimensional array is passed to a method as a copy of a reference to the array. So an array is treated like an object.

A method can change the values of the elements of an array passed to it as a parameter, but cannot change the size of the array.

(A method can change the size of an `ArrayList` passed to it as a parameter.)

20

Consider the following method:

```
public void accumulate(int[] a, int n)
{
  while (n < a.length)
  {
    a[n] += a[n-1];
    n++;
  }
}
```

What is the output from the following code?

```
int[] a = {1, 2, 3, 4, 5};
int n = 1;
accumulate(a, n);
for (int k = 0; k < a.length; k++)
  System.out.println(a[k] + " ");
System.out.println(n);
```

(A) 1 2 3 4 5 1
(B) 1 2 3 4 5 5
(C) 1 3 5 7 9 1
(D) 1 3 5 7 9 5
(E) 1 3 6 10 15 1

`n` is passed to `accumulate` by value, so `accumulate` cannot change it. This rules out Choices B and D. (Inside `accumulate`, `n` acts like a local variable.) `a` is passed to `accumulate` as a reference, so `accumulate` can change its values. Starting at `n = 1`, it adds the value of the previous element to the current one. As the name of the method implies, this sets `a[k]` to the sum of all the elements up to and including `a[k]` in the original array. The answer is E.

Aliasing

A more complicated concept is *aliasing*. It is good to understand in general, but it is very unlikely to come up on the exam. The following explanation is for a more inquisitive reader.

Suppose a class `Point` represents a point on the plane. Consider a method:

```
// point2 receives coordinates of point1 rotated 90 degrees
// counterclockwise around the origin
public void rotate90(Point point1, Point point2)
{
  point2.setY(point1.getX());
  point2.setX(-point1.getY());
}
```

In this example, `rotate90` takes two `Point` objects as parameters. Like all objects, these are passed to the method as references. Note that a `Point` object here is not immutable because it has the `setX` and `setY` methods. The code looks pretty harmless: it sets `point2`'s coordinates to the new values obtained from `point1`'s coordinates. However, suppose you call `rotate90(point, point)` hoping to change the coordinates of `point` appropriately. The compiler will not prevent you from doing that, but the result will be not what you expected. Inside the method, `point1` and `point2` actually both refer to `point`. The first statement will set `point`'s `y` equal to `x`, and the original value of `y` will be lost. If `point`'s coordinates are, say, `x` = 3, `y` = 5, instead of getting `x` = -5, `y` = 3, as intended, you will get `x` = -3, `y` = 3. This type of error is called an *aliasing* error.

In Java, aliasing may happen only when parameters are objects and they are mutable, or when parameters are arrays and the method moves values from one array to another. In the above example, it would be safer to make `rotate90` return a new value, as in

```
public Point rotate90(Point point)
{
  return new Point(-point.getY(), point.getX());
}
```

return

A method that is not `void` must return a value of the designated type using the `return` statement. `return` works with any expression, not just variables. For example:

```
return (-b + Math.sqrt(b*b - 4*a*c)) / (2*a);
```

An often overlooked fact is that a `boolean` method can return the value of a Boolean expression. For example, you can write simply

```
return x >= a && x <= b;
```

as opposed to the redundant and verbose

```
if (x >= a && x <= b)
  return true;
else
  return false;
```

A `void` method can use a `return` (within `if` or `else`) to quit early, but there is no need for a `return` at the end of the method. It is OK to have multiple `return` statements in a method, and often advisable. For example, the following recursive method is well-written:

```
// Returns the index of target in a sorted array
// among a[i], ..., a[j] or -1 if not found
public static int binarySearch(int[] a, int i, int j, int target)
{
  if (i > j)
    return -1;

  int m = (i + j) / 2;
  if (target == a[m])
    return m;

  if (target < a[m])
    return binarySearch(a, i, m-1, target);
  else
    return binarySearch(a, m+1, j, target);
}
```

Returning objects

A method's return type can be a class, and a method can return an object of that class. Often a new object is created in the method and then returned from the method. For example:

```
public String getFullName(String firstName, String lastName)
{
  return firstName + " " + lastName;
}
```

Or:

```
public Location adjacentSouth(Location loc)
{
  int r = loc.getRow(), c = loc.getCol();
  return new Location(r+1, c);
}
```

A method whose return type is a class can also return a `null` (a reference with a zero value, which indicates that it does not refer to any valid object). For example:

```
/** Precondition: listOfNames and listOfAddresses (instance variables
 *  of this class) hold valid data
 */
public String getAddress(String name)
{
  for (int i = 0; i < listOfNames.length; i++)
  {
    if (listOfNames[i].equals(name))
      return listOfAddresses[i];
  }
  return null; // not found
}
```

If a method returns an `ArrayList`, write the full `ArrayList` type, including its elements' type in angle brackets, as the method's return type. For example:

```
public ArrayList<Integer> getCourseNumbers()
{
  ArrayList<Integer> courseNumbers = new ArrayList<Integer>();
  ...
  return courseNumbers;
}
```

Overloaded methods

Methods of the same class with the same name but different numbers or types of parameters are called *overloaded* methods. (The order of different types of parameters is important, too.)

The compiler treats overloaded methods as different methods. It figures out which one to call depending on the number, the types, and the order of the parameters.

The `String` class, for example, has two forms of the `substring` method:

```
String substring(int from)
String substring(int from, int to)
```

If you call `"Happy".substring(2)`, then the first overloaded method will be called, but if you call `"Happy".substring(1, 3)` then the second overloaded method will be called. Another example of overloading is `Math.abs(x)`, which has different versions of the static method `abs`, including `abs(int)` and `abs(double)`. `System.out.print(x)` has overloaded versions for all primitive data types as well as for `String` and `Object`.

The `ArrayList` class has two overloaded `add` methods: `add(obj)`, which adds an object `obj` at the end of the list, and `add(index, obj)`, which inserts `obj` at a specified index.

Overloading methods is basically a stylistic device. You could instead give different names to different forms of a method, but it would be hard to remember them. Overloaded methods do not have to have the same return type, but often they do, because they perform similar tasks. The return type alone cannot distinguish between overloaded methods.

All constructors of a class have the same name, so they are overloaded by definition and must differ from each other in the number and/or types of their parameters.

21

Consider the following class declaration:

```
public class Date
{
  public Date()
  { < implementation not shown > }

  public Date(String monthName, int day, int year)
  { < implementation not shown > }

  public void setDate(int month, int day, int year)
  { < implementation not shown > }

    < fields and other constructors and methods not shown >
}
```

Consider modifying the `Date` class to make it possible to initialize variables of the type `Date` with month (given as a month name or number), day, and year information when they are declared, as well as to set their values later using the method `setDate`. For example, the following code segment should define and initialize three `Date` variables:

```
Date d1 = new Date();
d1.setDate("May", 11, 2006);
Date d2 = new Date("June", 30, 2010);
Date d3 = new Date(6, 30, 2010);
```

Which of the following best describes the additional features that should be present?

(A) An overloaded version of `setDate` with three `int` parameters
(B) An overloaded version of `setDate` with one `String` and two `int` parameters
(C) A constructor with three `int` parameters
(D) Both an overloaded version of `setDate` with three `int` parameters and a constructor with three `int` parameters
(E) Both an overloaded version of `setDate` with one `String` and two `int` parameters and a constructor with three `int` parameters

 This is a wordy but simple question. Just match the declarations against the provided class features:

```
Date d1 = new Date();                    ✓ Date()

Date d2 = new Date("June", 30, 2010);    ✓ Date(String, int, int)

Date d3 = new Date(6, 30, 2010);           Date(int, int, int)

d1.setDate("May", 11, 2004);               void setDate(String, int, int)

            not used                     ✓ void setDate(int, int, int)
```

As we can see, what's missing in the class definition is a constructor with three `int` parameters and a version of `setDate` with one `String` and two `int` parameters. The answer is E.

Three review questions

Questions 22-24 refer to the following partial class definition:

```
public class TicketSales
{
  private String name;
  private double[] sales;
        /** sales[0], ..., sales[51] hold sales totals for 52 weeks */

  public TicketSales(String movieName) { < implementation not shown > }

  /** Sets box office receipts for a given week.
   *  Precondition: 1 <= week <= 52
   */
  public void setWeekSales(int week, double dollars)
  { < implementation not shown > }

  /** Finds and returns the week with best sales
   */
  private int findBestWeek() { < implementation not shown > }

  < Other methods not shown >
}
```

22

The method `findBestWeek` is declared `private` because

(A) `findBestWeek` is not intended to be used by clients of the class.
(B) `findBestWeek` is intended to be used only by clients of the class.
(C) Methods that work with private instance variables of the `array` type cannot be public.
(D) Methods that have a loop in their code cannot be public.
(E) Methods that return a value cannot be public.

In this question only the first two choices deserve any consideration — the other three are fillers. You might get confused for a moment about what a "client" means, but common sense helps: a client is anyone who is not yourself, so if a client needs to use something of yours, you have to make it public. Private things are for your class, not for clients. The answer is A.

23

The constructor for the `TicketSales` class initializes the `sales` array to hold 52 values. Which of the following statements will do that?

(A) `double sales[52];`
(B) `double sales = new double[52];`
(C) `double[] sales = new double[52];`
(D) `sales = new double[52];`
(E) `sales.setSize(52);`

This is a syntax question. Choice A has invalid syntax. Choice E is absurd: an array does not have a `setSize` method (or any other methods). Choice B assigns an array to a `double` variable — a syntax error. Both Choices C and D appear syntactically plausible and in fact either one will compile with no errors. But C, instead of initializing an instance variable `sales`, will declare and initialize a local variable with the same name. This is a very common nasty bug in Java programs. The answer is D.

24

Given the declaration

```
TicketSales movie = new TicketSales("Monsters, Inc.");
```

which of the following statements sets the third week sales for that movie to 245,000?

(A) `movie = TicketSales(3, 245000.00);`
(B) `setWeekSales(movie, 3, 245000.00);`
(C) `movie.setWeekSales(3, 245000.00);`
(D) `movie(setWeekSales, 3, 245000.00);`
(E) `setWeekSales(3, 245000.00);`

This is another syntax question. The variable `movie` of the type `TicketSales` is defined outside the class, in a client of the class. The key word in this question is "sets." It indicates that a method, a modifier, is called, and the way to call a method from a client class is with dot notation. (Besides, Choice A assumes that there is a constructor with two parameters; Choices B and D look like calls to non-existing methods; Choice E forgets to mention `movie` altogether.) The answer is C.

3.4. Random Numbers

Random numbers simulate chance in computer programs. For example, if you want to simulate a roll of a die, you need to obtain a random number from 1 to 6 (with any one of these values appearing with the same probability). "Random" numbers are not truly random — their sequence is generated using a certain formula — but they are good enough for many applications.

One way to get random numbers in a Java program is to call the static method `random` of the `Math` class. It returns a random `double` from 0 (inclusive) to 1 (exclusive). To get a random integer from 1 to *n* use:

```
int r =  (int)(n * Math.random()) + 1;
```

25

Which of the following is a list of all possible outputs of the following code segment?

```
String memo = "MEMO";
System.out.print("[" +
      memo.substring((int)(3 * Math.random()),
                                        (int)(3 * Math.random()) + 2) +
                  "]");
```

(A) `[ME], [EM]`
(B) `[ME], [EM], [MO]`
(C) `[EM], [MO], [O], []`
(D) `[], [M], [ME], [EM], [MO]`
(E) `[], [M], [E], [ME], [EM], [MO], [MEM], [EMO], [MEMO]`

The two calls to `Math.random()` look the same, but they return different values — two successive values in the random number sequence. The "from" parameter of `memo.substring` can be 0, 1, or 2, and the "to" parameter can be 2, 3, or 4. Any combination of these from/to values results in a valid substring (including `substring(2,2)`, which returns an empty string). The answer is E.

Another way to get random numbers relies on the `java.util.Random` class, but it is not in the AP subset.

3.5. Input and Output

The AP subset does not include any classes or methods for data input. In particular, the `Scanner` class is not in the AP subset. If a question involves user input it may be stated as follows:

```
double x = < call to a method that reads a floating-point number >
```

or

```
int x = IO.readInt();  // Reads user input
```

Output is limited to `System.out.print` and `System.out.println` calls.

You do not have to worry about formatting numbers. Java converts an `int` or a `double` value passed to `System.out.print` or `System.out.println` into a string using default formatting. Starting with Java 5.0, `System.out.printf`, a new method with a variable number of parameters, can be used for more precisely formatted output of one or several numbers and strings. `printf` is not in the AP subset.

You can pass any object to `System.out.print`, `System.out.println`. These methods handle an object by calling its `toString` method. Both `Integer` and `Double` classes have reasonable `toString` methods defined. If you are designing a class, it is a good idea to supply a reasonable `toString` method for it. For example:

```
public class Fraction
{
  ...
  public String toString()
  {
    return num + "/" + denom;
  }
}
```

Otherwise, your class inherits a generic `toString` method from `Object`, which returns the object's class name followed by the object's address.

The `System.out.print` and `System.out.println` methods take only one parameter.

If you need to print several things, use the + operator for concatenating strings. You can also concatenate a string and an `int` or a `double`: the latter will be converted into a string. For example:

```
System.out.println(3 + " hours " + 15 + " minutes.");
```

The displayed result will be

```
3 hours 15 minutes.
```

You can also concatenate a string and an object: the object's `toString` method will be called to convert it into a string. For example:

```
int n = 3, d = 4;
Fraction f = new Fraction(n, d);
System.out.println(f + " = " + (double)n / (double)d);
```

The displayed result will be

```
3/4 = 0.75
```

Just be careful not to apply a + operator to two numbers or two objects other than strings: in the former case the numbers will be added rather than concatenated; the latter will cause a syntax error.

3.6. Exceptions and Assertions

An exception is a run-time event that signals an abnormal condition in the program. Some run-time errors, such as invalid user input or an attempt to read past the end of a file, are considered fixable. The `try-catch-finally` syntax allows the programmer to catch and process the exception and have the program recover. This type of exception is called a *checked exception*.

Checked exceptions and the `try-catch` statements are not in the AP subset.

Other errors, such as an array index out of bounds or an attempt to call a method of a non-existing object (null reference) are considered fatal: the program displays an error message with information about where the error occurred, then quits. This type of exception is called an *unchecked exception*.

In Java, an exception is an object of one of Java exception classes. The Java library implements many types of exceptions, and, if necessary, you can derive your own exception class from one of the library classes. For an AP CS exam, you are expected to understand what `ArithmeticException`, `IndexOutOfBoundsException`, `ArrayIndexOutOfBoundsException`, `NullPointerException`, and `IllegalArgumentException` mean.

We say that a program "throws" an exception. An `ArithmeticException` is thrown in case of an arithmetic error, such as integer division by zero. (You would expect `Math.sqrt(x)` to throw an `ArithmeticException` for a negative `x`, but it doesn't — it returns NaN, "not a number" instead. Java exception handling is inconsistent at times.)

`ArrayIndexOutOfBoundsException` is self-explanatory: it is thrown at run time when an array index is negative or is greater than `array.length-1`. `ArrayList` methods throw a similar `IndexOutOfBoundsException`.

`NullPointerException` is thrown when you forget to initialize an object-type instance variable or an element of an array and then try to call its method. For example:

```
public class MyClass
{
  private String name;  // name is set to null
  ...
    // the following statement is inside a method:
    int n = name.length();    // if name has not been initialized
                              // by MyClass's constructor, this statement
                              // will throw a NullPointerException
  ...
}
```

Another example:

```
    Integer[] a = new Integer[10];
    int x = a[0].intValue();      // a[0] is null -- Java interpreter
                                  // throws a NullPointerException
```

A third example:

```
public class DeckOfCards
{
  private Card[] cards;

  public DeckOfCards()
  {
    cards = new Card[52];     // all elements set to null
          // Forgot to initialize each element
  }

  public Card getCard(int k) { return cards[k]; }
}
```

In a client class,

```
    DeckOfCards deck = new DeckOfCards();
    int r = deck.getCard(0).getRank();
```

will throw a `NullPointerException`, because `deck.getCard(0)` returns `null`.

Throwing your own exceptions

Occasionally you need to "throw" your own exception. For example, you are implementing the `remove` method for a queue data structure. What is your method to do when the queue is empty? Throw a `NoSuchElementException`.

`throw` is a Java reserved word. The syntax for using it is

```
throw < exception >;
```

For example:

```
if (items.size() == 0)
  throw new NoSuchElementException();
```

Throw an `IllegalStateException` if an object is not ready for a particular method call. For example:

```
public void stop()
{
  if (!isMoving())
    throw new IllegalStateException();  // displays a message
                                        //   and quits
  speed = 0;
  ...
}
```

Throw an `IllegalArgumentException` if a constructor or method receives an unacceptable parameter. For example:

```
public Clock(int hours, int mins)
{
  if (hours < 0 || hours >= 24 || mins < 0 || mins >= 60)
    throw new IllegalArgumentException();
  ...
}
```

Your code does not need to explicitly throw an `ArithmeticException`, `NullPointerException`, or `ArrayIndexOutOfBoundsException` — Java does it automatically when a triggering condition occurs.

26

Consider the following class:

```
public class TestSample
{
  private ArrayList<Integer> samples;

  public TestSample(int n)
  {
    for (int k = 0; k < n; k++)
    {
      samples.add(k);
    }
  }

  public double getBestRatio()
  {
    double maxRatio = samples.get(1).intValue() /
                              samples.get(0).intValue();

    for (int k = 1; k < samples.size() - 1; k++)
    {
      double ratio = samples.get(k+1).intValue() /
                              samples.get(k).intValue();
      if (ratio > maxRatio)
      {
        maxRatio = ratio;
      }
    }
    return maxRatio;
  }
}
```

What is the result of the following code segment?

```
TestSample t = new TestSample(1);
System.out.println(t.getBestRatio());
```

(A) `NullPointerException`
(B) `ArithmeticException`
(C) `IndexOutOfBoundsException`
(D) `ArrayIndexOutOfBoundsException`
(E) `Infinity`

Luckily we don't have to look at the `getBestRatio` method. The programmer has forgotten to initialize `samples` and calls its `add` method in the constructor. The answer is A.

Now suppose we added

```
samples = new ArrayList<Integer>();
```

at the top of the constructor. What would happen then? `getBestRatio` would call `samples.get(1)`, but we would have added only one value to `samples`. The answer would be C, `IndexOutOfBoundsException`.

Now suppose we changed

```
TestSample t = new TestSample(1);
```

to

```
TestSample t = new TestSample(2);
```

What would happen then? The answer would be B, `ArithmeticException`, because we would have an integer division by 0.

Actually, the programmer probably meant to write

```
double maxRatio = (double)samples.get(1).intValue() /
                                samples.get(0).intValue();
```

What would happen if we added this cast to a `double`? We would still expect an `ArithmeticException` for floating-point division by 0, but Java actually prints "Infinity." The answer would be E (but you do not have to know that).

Note that the numbers and objects *are* cast correctly into `Integer`s automatically where necessary due to autoboxing.

Assertions

Assertions is a tool provided in many programming languages for debugging and verification of program correctness. An assertion is a statement in a program that specifies that a certain condition must be true when that statement is reached. The program checks the assertion condition at run time and throws an exception if the condition is false. The condition is given as a Boolean expression.

In Java, an assertion is written using the reserved word `assert`:

```
assert < boolean expression >;
```

For example:

```
assert a[i] >= a[i-1];
```

The `assert` statement may also include a message (a `String`) to be displayed if the assertion fails. For example:

```
assert a[i] >= a[i-1] : "Elements in a out of order";
```

The `assert` statement is not in the AP subset and won't be tested on AP exams.