Primitive Types in C++ and Java and Their Input/Output
1. What is the recommended alternative to a Vector in Java 2 SDK?
2. A recommended programming style for constructing a container object in Java is to declare a maximally general type for the desired functionality.

For example, to construct an ArrayList object in Java, you would in most cases use the following declaration:

```
List mylist = new ArrayList();
```

If you did not need the ListIterator and other methods defined specifically for a List, you could even say

```
Collection mylist = new ArrayList();
```

Why is this a useful style of programming?
3. Let’s say you need a Set container, would you use a TreeSet or a HashSet?
4. Shown below are C++ and Java code examples for iterating through a container. What are the similarities and dissimilarities between the two? What sort of a diagram would you use to show the differences between how an iterator scans a container in C++ and in Java?

C++:

```cpp
set<string> animals;
// put items in animals
set<string>::iterator iter = animals.begin();
while( iter != animals.end() )
  cout << *iter++ << endl;
```

Java:

```java
Set animals = new TreeSet();
// put items in animals
Iterator iter = animals.iterator();
while ( iter.hasNext() )
  System.out.println( iter.next() );
```
Primitive Types in C++ and Java

The primitives types are the variables that are declared to hold integer, character, floating-point, and boolean values.

Let’s first see what names can be used for variables in general.
Tokens, Identifiers, and Variable Names

When source code is parsed for compilation, it is broken into tokens.

Tokens are sequences of characters that cannot be split further without violating the intent of the programmer.
A token can be

- an operator, such as +, *, etc.,
- a keyword, such as `main`, `#include`, etc.,
- a string literal,
- a punctuation,
- an identifier,
- etc.
In both C++ and Java, tokens are often delimited by white space (meaning, spaces, tabs, newline characters, and form-feed characters).

But that’s not always the case:

```cpp
char ch = ' ';  
string str = 'hi there';
```

Tokens can also be delimited by operators, punctuation marks, and other symbols that do not belong in identifiers, keywords, etc.

How many tokens in the following statement?

```cpp
cout<<"Height is: " + height << endl;
```
The set of all tokens (consists of identifiers, keywords, string literals, operators, punctuations, etc.)

The set of all identifiers (consists of variable names, function names, class names, labels, etc.)

The set of names of all variables
Identifiers

A variable name must be an identifier.

Identifiers in both C++ and Java are also used for naming constants, labels, functions, objects, object classes, etc.
Identifiers in C++

In C++, an identifier consists of a sequence of characters that must be letters or digits.

The first character of an identifier must be a letter. For the purpose of this definition, the underscore character (\_) is a letter.

Examples of identifiers in C++:

x y i j hello var0 var1 var_x var_y ....
The C++ standard itself – and therefore the C++ compiler – places no limits on the length of an identifier. However, parts of the overall compilation processing – such as linking – may impose constraints on the length of an identifier.

Identifiers in C++ are usually written using the 7-bit ASCII character set.
Identifiers in Java

An identifier in Java looks very much like an identifier in C++ except that the definition of a letter and a digit is now much broader because a 16-bit Unicode representation is used for characters in Java.

This means Java can use a character set containing 65,536 characters.

The first 256 characters of Unicode constitute the Latin-1 character set and the first 128 of these are equivalent to the 7-bit ASCII character set.

Current Java environments read ASCII or Latin-1 files, converting them to Unicode on the fly.
Converting an ASCII character to a Unicode character means in most cases extending the bit pattern of an ASCII character with a byte of zeros.
Primitive Types in C++ and Java

Primitive types (also known as the fundamental types) common to both C++ and Java are:

1. A boolean type: `bool` in C++ and `boolean` in Java.

2. Character types: `char` in both C++ and Java. Additionally, `signed char` and `unsigned char` in C++.

3. Integer types: `short`, `int`, `long` in both C++ and Java. Additionally, `byte` in Java. Additionally, `signed int` and `unsigned int` in C++. `signed int` in C++ is synonymous with `int`.

4. Floating-point types: `float` and `double` in both C++ and Java. Additionally, `long double` in C++. 
Boolean Types

A Boolean type is allowed to have only one of two values: true or false.

A Boolean is used to express the result of logical operations.

```java
int x;
int y;
...
...
bool b = x == y; in C++
boolean b = x==y; in Java
```
C++:  bool greater( int a, int b ) { return a > b; }
Java:  boolean greater( int a, int b ) { return a > b; }
Some of the older C++ compilers do not have `bool` as a built-in type. However, you can easily create the `bool` type with an enumeration:

```cpp
enum bool {false, true};
```
In Java, casting to a `boolean` type from any other type or casting from a `boolean` type to any other type is not permitted.

A point of difference between C++ and Java:

unlike the `bool` type in C++, `boolean` values are not integers.
Character Types

Almost universally, a C++ `char` has 8 bits so that it can hold one of 256 values.

Which of the 8-bit patterns correspond to what characters is not addressed by the ANSI standard.

Most C++ implementations that you are likely to see in this country will have the values from 0 through 127 reserved for the letters, digits, punctuations, etc., according to the the ASCII format.

All printable characters belong to this set of 128 values.
From the standpoint of writing portable code, note that some of the characters one usually sees in C++ source code may not be available in a particular character set available for C++ programming.

For example, some European character sets do not provide for {,},[,] etc.
In addition to `char`, C++ also provides:

```markdown
unsigned char
signed char
```

for the character type.

The type `signed char` holds values from -128 to 127, and the type `unsigned char` from 0 to 255.

Is a plain `char` in C++ an `unsigned char` or a `signed char`?

The answer to that depends on the implementation.
On SUN’s and PC’s, a plain \texttt{char} is the same as a \texttt{signed char}. On the other hand, on SGI’s, a \texttt{char} is unsigned.
A `char` variable can be initialized by either an integer whose value falls within a certain range or by what’s known as a `character literal`:

```java
char ch = 98; // ch is the character b
char x = 'b';
char y = '2';
```

Suppose with `y` set as above, the next statement in a computer program is

```java
int z = y + 8; (works for both C++ and Java)
```

What’s the value of `z`?
A `char` can also be initialized by an appropriately formatted *escape sequence*.

An escape sequence is a sequence of characters that begins with a back-slash.

All of the following C++ declarations are equivalent:

```cpp
char x = 'b';       // value of 'b' is 98
char x = '\x62';   // 62 is hex for 98
char x = '\142';   // 142 is octal for 98
```

By virtue of the fact that an escape sequence begins with a backslash, the entire sequence stands for a single character from the character set.
There are two kinds of escape sequences: \textit{character escapes} and \textit{numeric escapes}.

Character escapes, such as $\backslash n$, $\backslash t$, etc., represent ASCII’s more commonly used control characters that when sent to an output device can be used to, for example, move the cursor to a new line, or to move the cursor to the next horizontal tab, etc.

Since character escapes are few in number, a more general form of an escape sequence is the numeric escape.

A numeric escape, of the form $\backslash xdd$ in hex and $\backslash ddd$ in octal, can represent any individual character, printable or nonprintable.
But note that a backslash is also used for defining another kind of escape – an escape that tells the system to suspend the usual meaning of the next character.

Suppose you want to initialize the variable \texttt{x} with the single-quote character. Since a single quote is used for delimiting character literals, you obviously won’t be able to say

\begin{verbatim}
char x = '”;  \ERROR
\end{verbatim}

Instead, you will use the backslash as an escape, as in

\begin{verbatim}
char x = '\’';
\end{verbatim}
Another illustration of this would be if you wanted to initialize the character variable \texttt{x} to the backslash itself:

\begin{verbatim}
char x = '\\';
\end{verbatim}
Unlike octal numbers in general, octal numbers in escape sequences do not have to begin with a 0.

Also, a maximum of three digits is allowed in an octal escape sequence.

For hexadecimal escape sequences, there is no official limit on the number of hex digits after \x, but the integer represented by the hex digits cannot exceed 255 (FF in hex) for 8-bit characters.

These properties of escape sequences require care when they are used as characters in string literals.
string y1 = "a\x62";

string y2 = "a\0a";

string y3 = "a\nbcdef";

string y4 = "a\0awxyz";

string y5 = "a\0abcdef";

string y6 = "a\xef";

string w1 = "a\142";

string w2 = "a\142c";
string w3 = "a\142142";
Character Type in Java

A Java `char` has 2 bytes.

Any two contiguous bytes in the memory represent a legal Java `char`. Which 16-bit bit pattern corresponds to what character is determined by the Unicode representation.
In Java, all of the following declarations are equivalent:

```java
char x = 'b'; // value of 'b' is 98
char x = 98;
char x = '\u0062'; // 0062 is hex for 98
char x = '\142'; // 142 is octal for 98
```

The hex form on a character escape in Java must begin with the letter u. The letter u must be followed by exactly four hex characters.

\udddd
Integer Types

C++ and Java share the following integer types:

- short
- int
- long

In C++, typically a short has two bytes of memory, an int four bytes, and a long eight bytes.
It is important to note that the C++ standard says nothing about the exact number of bytes to be used for each.

All that is stipulated by the C++ standard is that the memory allocated to a `short` be at least as large as the memory assigned to a `char`, the memory assigned to an `int` be at least as large as the memory assigned to a `short`, and that the memory assigned to a `long` be at least as large as the memory assigned to an `int`. 
Java stipulates that a **short** have exactly two bytes, an **int** four, and a **long** eight.

Java also supports one additional integer type:

**byte**

which, as its name implies, has one byte for its storage.

**byte** represents a signed integer; its value ranges from -128 to 127.

C++ also supports signed and unsigned types. A **signed int** is synonymous with a plain **int**. The unsigned integer types are ideal for uses that treat storage as a bit array.
Constant integer values are referred to as integer literals.

For example, the integer 234 is an integer literal.

For both C++ and Java, integer literals come in three forms:

- decimal
- octal
- hexadecimal

the most common being the decimal form.
An integer literal starting with a zero followed by a digit between 0 and 7 is an integer in octal form.

An integer literal starting with a 0x followed by a hex digit is an integer in hexadecimal form.
Examples of integer literals:

<table>
<thead>
<tr>
<th></th>
<th>decimal:</th>
<th>0</th>
<th>7</th>
<th>81</th>
<th>2345</th>
</tr>
</thead>
<tbody>
<tr>
<td>octal:</td>
<td></td>
<td>00</td>
<td>07</td>
<td>0121</td>
<td>04451</td>
</tr>
<tr>
<td>hexadec:</td>
<td></td>
<td>0x0</td>
<td>0x7</td>
<td>0x51</td>
<td>0x929</td>
</tr>
</tbody>
</table>

Octal and hexadecimal notations are most useful for expressing bit patterns. The 4-byte bit pattern that corresponds to the integer 81 is

```
00000000 00000000 00000000 01010001
```

Its hex representation: ???
The hex representation of the integer 2345 is \( \text{0x929} \), we can conclude that the bit pattern for the integer 2345 must be

\[
00000000 \ 00000000 \ 00001001 \ 00101001
\]
Floating-point Types

C++ and Java have in common the floating-point types

float
double

In C++, typically four bytes are used for a float and eight for a double.

Although the C++ standard does not specify how many bytes exactly should be devoted to each type, it does stipulate that a double be allocated at least as many bytes as a float.
Java stipulates that float will have exactly 4 bytes assigned to it and a double 8 bytes.
In addition, C++ supports `long double` for extended precision which, on most machines today, would be represented by 16 bytes.

From the standpoint of portability, the exact meaning of `float`, `double`, and `long double` is implementation defined in C++. That is, the number of bits reserved for the exponent and the mantissa may vary from implementation to implementation. On the other hand, Java stipulates that the `float` and the `double` types conform to the IEEE 754 standard.
Integral Types

Boolean, character, and integer types are collectively called *integral types*.

The integral and floating-point types are collectively called *arithmetic types*.

For solving a majority of problems, one is likely to use **bool** for logical values, **char** for characters, **int** for integer values, and **double** for floating-point values.
Type Conversions for Primitive Types

Automatic type conversions are often called for during initialization, assignment, and matching of arguments with the corresponding parameters during function invocations.

One can also force a type conversion by using the `cast` operator.

With regard to automatic type conversion for primitive types, C++ makes a distinction between a *promotion* and a *standard conversion*. 
In a promotion, integral types stay integral and non-integral types stay non-integral. In addition, there must not be a loss of information.

Examples of integral promotions include `bool` to `int`, `char` to `int`, `short` to `int`, and their unsigned counterparts.

Examples of non-integral promotions `float` to `double`, and `double` to `long double`, etc.
This program illustrates promotions in initialization, assignment, and in function invocation.

```cpp
#include <iostream>

void g1( int x ) { cout << "int version invoked" << endl; }
void g2( long x ) { cout << "long version invoked" << endl; }
void g3( double x ) { cout << "double version invoked" << endl; }
void g4( long double x ) { cout << "long double version invoked" << endl; }

int main()
{
    char ch = 'a';
    int i = ch; // promotion in initialization

    long j;
    j = i; // promotion in assignment

    g1( ch ); // promote char to int
    g2( ch ); // promote char to long

    float f = 1.234f;
    g3( f ); // promote float to double
    g4( f ); // promote float to long double

    return 0;
}
```
The other kind of automatic type conversion for the primitive types in C++ – the standard conversion – can entail loss of information.
Examples of these are \textbf{int} to \textbf{double}, \textbf{double} to \textbf{int}, etc. Here is an example of a C++ program that depends on automatic standard conversions.

```cpp
#include <iostream>

void g1( short x ) { cout << "short version invoked x: " << x << endl; }
void g2( int x ) { cout << "int version invoked x: " << x << endl; }
void g3( float x ) { cout << "float version invoked x: " << x << endl; }

int main()
{
    char ch = 'a';
    float x = ch; // standard conversion in initialization
    long j;
    j = x; // standard conversion in assignment
    cout << j << endl;

    float y = 1e20f; // value too large to fit into an int
    double z = -1e100; // value too large to fit into a float

    g1( y ); // standard-convert float into a short
    g2( y ); // standard-convert float into an int
    g3( z ); // standard-convert double into a float

    return 0;
}
```
This program produces the following output

97
short version invoked  x: -1
int version invoked  x: 2147483647
float version invoked  x: -inf
Automatic Type Conversions in Java

The automatic type conversions in Java fall into two categories: *widening primitive conversions* and *narrowing primitive conversions*.

Widening primitive conversions are, for the most part, like the promotions in C++.
Here are the 19 widening conversions permitted automatically in Java for initialization, assignment, and for matching arguments with parameters during function invocation:

- from byte to short, int, long, float, or double
- from short to int, long, float, or double
- from char to int, long, float, or double
- from int to long, float, or double
- from long to float or double
- from float to double

These do not entail any loss of information when the conversions from an integral type to another integral type, and from a non-integral type to another non-integral type.
However, there may be a loss of precision (although the overall magnitude will be preserved, but some of the least significant bits may be lost) when converting from an integral type to a non-integral type, such as from `long` to `float`.

```java
class Test {
    public static void main( String[] args ) {
        int i = 1234567890; // (A)
        float f = i;
        System.out.println( i - (int) f );
    }
}
```

This program prints out -46, which is the error of precision between the `int` value shown in line (A) and its `float` representation. This error is caused by the fact that the precision of a `float` is limited to six significant digits.
Compared to a widening conversion, a narrowing conversion in Java (like many of the standard conversions in C++) can entail loss of information besides loss of precision.

A majority of the permissible narrowing conversions require an explicit use of the `cast` operator.

Java allows very few narrowing conversions to take place automatically for initialization and assignment.

Only conversions from an `int` to a `byte, short, or char` are allowed, and that too only for initialization, provided the value to be converted can be represented by the target types.
For example, the following is legal syntax

```java
byte b = 15;  // narrow conversion from int to byte
```

because the right hand side is an `int`, the type of the variable on the left a `byte` and the value 15 representable without error by a `byte`. 
In particular, Java does not allow any narrowing conversions to take place automatically when matching arguments with parameters in function invocations.
class Test {

    void g1( short x ) { System.out.println( "short version invoked" ); }
    void g2( int x ) { System.out.println( "int version invoked" ); }
    void g3( float x ) { System.out.println( "float version invoked" ); }

    public static void main( String[] args )
    {
        int i = 98;
        char c1 = i; // ERROR

        char c2 = 98; // but this is okay

        char ch = 'a';
        float x = ch; // ok, widening conversion char to float

        long j;
        j = x; // error, cannot automatically convert float to long

        float y = 1e20f;
        double z = -1e100;

        g1( y ); // error, cannot automatically convert float to short
        g2( y ); // error, cannot automatically convert float to int
        g3( z ); // error, cannot automatically convert double to float
    }
}