Java Types & References	start time:

This activity will help you understand how Java uses references to memory locations, how Java represents arrays in memory, and some of the implications.
Before you start, complete the form below to assign a role to each member.
If you have 3 people, combine Speaker & Reflector.

Team	**Date**
Team Roles	**Team Member**
Recorder: records all answers & questions, and provides copies to team & facilitator.	
Speaker: talks to facilitator and other teams.	
Manager: keeps track of time and makes sure everyone contributes appropriately.	
Reflector: considers how the team could work and learn more effectively.	

Reminders:

1. *Note the time whenever your team starts a new section or question.*
2. *Write legibly & neatly so that everyone can read & understand your responses.*

(10 min) A. Variables	start time:

```
1 float   f1;
2 int     i1 = 13;
3 double  d1,  d2 = 3.1415;
4 char    c1,  c2 = 'a';
```

1. (3 min) Refer to the code above to answer the following questions.

a.	What is the **name** of the variable declared in line 1?	
b.	What is the **data type** of the variable declared in line 1?	
c.	What is the **name** of the variable declared in line 2?	
d.	What is the **data type** of the variable declared in line 2?	
e.	What is the **value** of the variable declared in line 2?	
f.	What are the **names** of the variables declared in line 3?	
g.	What is the **data type** of the variables declared in line 4?	
h.	How many variables are **declared** in the code above?	
i.	How many variables are **initialized** (given an initial value)?	
j.	How many different **data types** are used?	

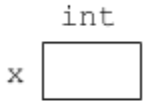
2. (1 min) In Java each variable must be **declared** before it can be used.

a.	What two pieces of information are **required** in a declaration?	
b.	What piece of information is **optional**?	

An **assignment** changes the value; we say "x **equals** y plus 2" but we should say "x **gets the value of** y plus 2". However, there are some subtle issues in how this works.

3. (4 min) Program code can be manually **traced** (simulated) by updating the value of each variable as each statement is executed. (This is an important skill, although it can be tedious.) There are several ways we can organize this information.

We can draw a **diagram** with a **box** for each variable (see right), and change the box's contents as needed.



We can also use a **table** with a row for each statement that is executed, and a **column** for each variable showing its after each statement.

Trace the code below using this approach.

#	code statement	x	y	z
a.	`int x=3, y=4, z=5;`	3	4	5
b.	`y = x + z;`			
c.	`z += x;  // same as z=z+x`			
d.	`x++;  // same as x=x+1`			
e.	`x *= 2;  // same as x=x*2`			
f.	`z -= x;`			

4. (2 min) Of the two tracing methods described above (diagram, table, or both), which:

a.	Is **faster** to set up.	
b.	Can show related variables near each other.	
c.	Shows how each variable **changes** over time.	
d.	Shows which variables tend to change at the same time.	

(15 min) B. Primitive Data Types	start time:

data type	range of values (approximate)	# bytes	examples & notes
`short`	-32,000 ... 32,000	2	`short s0 = 13,` `  s1 = 0b1101;  // binary`

`int`	$-2x10^9$ ... $2x10^9$	4	`int  i0 = 13, i1 = 0xFFAA; // hex`
			recommended default for integer values

`long`	-10^{18} ... 10^{18}	8	`long  m0 = 9876543210L,`
			`  m1 = 9_876_543_210L,`
			`  m2 = 0xFF_FF_FF_FFL; // hex`

`float`	$\pm10^{-40}$ ... 10^{40}	4	`float f0 = 2.718f,`
			`  f1 = 3.141_592F,`
			`  f2 = 299.79E6F;`

`double`	$\pm10^{-300}$ ... 10^{300}	8	recommended default for floating point values
`char`	'\u0000' ... '\uffff' (Unicode characters)	2	`char  c0 = 'a', c1 = '\u0000',` `  c2 = '\t'; // tab`

1. (2 min) **Primitive data types** are built into the language definition.
The table above lists some (not all) of Java's primitive types, the range of possible values, the number of bytes used by a value of each type, and some examples and notes.

a.	How much memory (in bytes) is required for one `int`?	
b.	How much memory (in bytes) is required for one `long`?	
c.	Which 3 data types represent **integer values**?	
d.	Which 2 data types represent **floating point values**?	
e.	What does the `0x` prefix indicate?	
f.	What does the `0b` prefix indicate?	
g.	What does the L suffix indicate?	
h.	What might the D suffix indicate?	
i.	What does E indicate in a float or double value?	

2. (2 min) An underscore (_) between digits in a numeric constant has no effect on the value. Use examples to explain how underscores can make code easier to read.

3. (3 min) Refer to the range of values in the table above to answer the following questions:

a.	What is the largest possible `short` value?	
b.	What is the largest possible `int` value?	
c.	What is the largest possible `long` value?	
d.	Can every `short` value be converted to an `int`?	
e.	Can every `int` value be converted to a `short`?	
f.	Can every `long` value be converted to an `int`?	
g.	Can every `float` value be converted to a `double`?	
h.	Can every `long` value be converted to a `float`?	
i.	How much memory (in bytes) is required for 10,000 `int`'s?	
j.	How much memory (in bytes) is required for 10,000 `short`'s?	

4. (2 min) Explain how a choice between `int` and `long` depends on two factors: **memory use** and the **range of possible values**.

5. (2 min) Java **automatically converts** between some types:

`short` to `int` `int` to `long` `long` to `float` `float` to `double`

but **not** between other types:

`int` to `short` `long` to `int` `float` to `long` `double` to `float`

In complete sentences, explain why the first set are automatic and the second are not, in terms of the range of values and memory requirements for the data types.

(10 min) C. Arrays	start time:

```
int []  ia,  ib = new  int[3],  ic = { 9, 8, 7, 6, 5 };
char[]  ca,  cb = new char[2],  cc = { 'd', 'o', 'g' };

ic[0] =  3 ;   // change 0th value in ic to be 3
cc[2] = 't';   // change 2nd (last) value in cc to be 't'
```

1. (2 min) An **array** variable contains **multiple values** of the **same data type**,
and a 0-based **integer index** is used to access specific values within the array.
Thus, an array of length 10 will have indices (index values) from 0 to 9.
To **create** an array, use the keyword **`new`**, the data type, and the array's length.
To create and **initialize** an array, specify a set of values in curly brackets;
this only works when the array is first declared.

a.	How many arrays are **declared** above?	
b.	How many arrays are **created**?	
c.	How many arrays are **initialized**?	

2. (4 min) **Trace** the array code below.

#	**code statement**	`x[0]`	`x[1]`	`x[2]`
a.	`int[] x = {4,5,6};`	4	5	6
b.	`x[0] = x.length;`			
c.	`x[2] -= x[0];`			
d.	`x[1]++;`			
e.	`x[1] /= 2;`			
f.	`x = new int[2];`			

3. (2 min) Within each **primitive type**, every value uses the **same** amount of memory,
but **arrays** of the same primitive type can use **different** amounts of memory.
In complete sentences, explain this difference between values and arrays.

(15 min) D. References	start time:

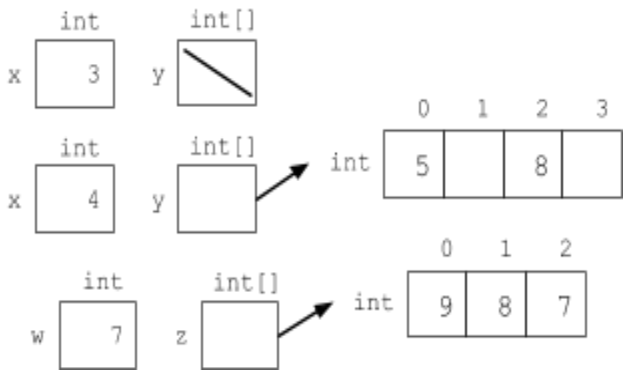
1. (2 min) The program compiler can always allocate the right amount of memory for a primitive type variable, but not for an array variable. Explain why this is true.

2. (4 min) Instead, the compiler allocates memory for a **reference** to where the array is stored, and allocates memory for the array separately (using the keyword **`new`**).

i	`int   x = 3;` `int[] y = null;`

ii	`x     = x + 1;`
	`y     = new int[x];`
	`y[0] = x + 1;`
	`y[2] = 8;`

iii	`int   w = y[2] - 1;` `int[] z = { 9,8,7 };`



a. What is the Java syntax to **declare** an array variable that does not (yet) contain a reference? How is this **shown** in the memory diagram above?

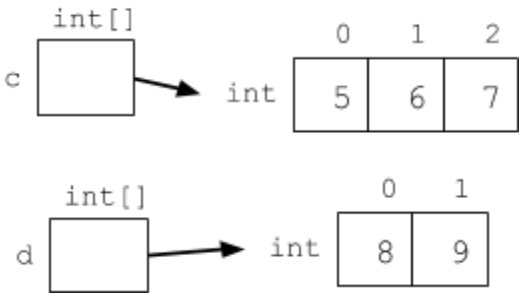
b. Although a reference is actually a numeric address in computer memory, how is it **shown** in the memory diagram and why might this be more **useful**?

c. When allocating memory for an array with the keyword **`new`**, why is it necessary to specify **both** the **data type** and the **array size**?

3. (5 min) The example below has 2 array variables, each initialized to refer to an array.

```
int[] c = {5,6,7};
int[] d = { 8,9 };

c = d;
```



However, the assignment statement **`"c = d;"`** could be interpreted in several ways:

i. Copy the **array values**, so c has the values { 8, 9, 7 }.
ii. Copy the **entire array**, so c refers to a new array with values { 8, 9 }.
iii. Copy the **reference**, so c and d both refer to the same array.

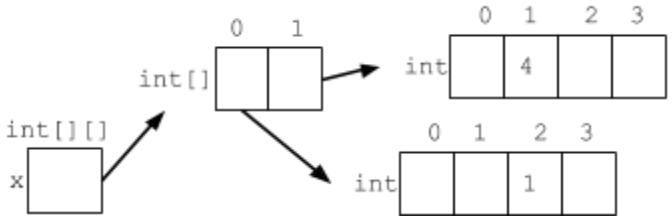
To understand which option is used in Java, consider:

a.	Which option could have problems if the two arrays are of **different sizes**?	
b.	Which option makes the **fewest changes in memory**, (and is likely the **fastest**) especially for large arrays? Why?	
c.	Which option is **most consistent** with the behavior of variables with primitive types? Why?	

4. (3 min) In complete sentences, explain why option iii is most appropriate and why.
Before you continue, review progress with the facilitator.

(16 min) E. Arrays of Arrays	start time:

```
int [][] x = new int[2][4];
x[0][2] = 1;
x[1][1] = 4;
```



1. (3 min) In Java, a 2 dimensional array is stored as an **array of arrays**. The variable **x** above references a 2-element array that contains two references, each to an int array. Explain why the values 1 and 4 appear where they do in the diagram.

2. (3 min) Draw a diagram (on paper) showing the variable **y** and the arrays it references, based on the code shown below.

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

3. (5 min) Draw and/or describe the result of each of the following statements, in order:

a.	`y[0][0] = 7;`	
b.	`y[1]    = new int[2];`	
c.	`y[1]    = y[0];`	
d.	`y[1][2] = 8;`	
e.	`y[1]    = null;`	

4. (5 min) Draw a diagram (on paper) showing the array produced by the following code:

```
int [][] z = new int[4][];
for (int i=0; i<z.length; i++) {
  z[i] = new int[i+1];
  for (int j=0; j<=i; j++) { z[i][j] = j; }
}
```

```
/**
 * Trace activity on reference types.
 * Do this INDIVIDUALLY, for HOMEWORK to check your understanding.
 * @author  Clif Kussmaul
 * @version 2012-02
 */
public class RefTrace {
  public static void main(String [] args) {
    int i = 3, j = 4, k ;
    k = i; i = j;
    System.out.println("i=" + i + " j=" + j + " k=" + k);
    // arrays of primitives
    int [] a = { 2, 3, 4 }, b = { 7, 8, 9 };
    int [] c = a;
    System.out.println(" a=" + a2s(a) + "\n c=" + a2s(c));
    a[0] = b[1]; c[2] = b[2];
    System.out.println(" a=" + a2s(a) + "\n c=" + a2s(c));
    // arrays of arrays
    int [][] x = { { 1, 2, 3}, {4, 5, 6} },
             y = { { 1, 3, 5}, {2, 4, 6} };
    int [][] z = x;
    System.out.println(" x=" + aa2s(x) + "\n z=" + aa2s(z));
    x[0] = y[1]; z[1] = x[0];
    System.out.println(" x=" + aa2s(x) + "\n z=" + aa2s(z));
  }

  public static String a2s(int [] arr) {
    String result = "";
    for (int i : arr) { result += i + " "; }
    return "{ " + result + "}";
  }
  public static String aa2s(int [][] arr) {
    String result = "";
    for (int [] a : arr) { result += a2s(a) + " "; }
    return "{ " + result + "}";
  }
}
```