

Object-Oriented Programming

CSCI-UA 0470-001

Class 11

Instructor: Randy Shepherd

Arrays

Array Review

- Defining characteristics in a statically-typed language?

Array Review

- Defining characteristics in a statically-typed language?
- Is a container object that holds a fixed number of values of a single type.
- Length of an array is established when the array is created.
 - i.e. after creation, its length is fixed.
- Elements are contiguous in memory.*

Java Vs C++

- In Java, all array entries are initialized to the default value of the element type.
- In Java, array accesses are guarded by bounds checks
- In Java, arrays are initialized to default values.
- More on all of these to follow...

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What does line 4 print?

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What does line 4 print?
 - false
- Why?

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- Arrays, like any other non-primitive type in Java, are objects and extend Object.
- Arrays do not override any methods of class Object.
- Object *equals* compares the memory location. Remember?

Identity Vs Equality

- This raises the question of *identity* vs *equality*

```
1 // What should be the result of this?
2 Integer x = new Integer(1);
3 Integer y = new Integer(1);
4 x.equals(y);
5
6 // What about this?
7 Integer[] ax = {x, y};
8 Integer[] ay = {x, y};
9 ax.equals(ay);
```

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What do lines 5, 6 and 7 print?

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What do lines 5, 6 and 7 print?
 - line 5: `[Ljava.lang.String;`
 - line 6: `[I`
 - line 7: `[[Ljava.lang.String;`
- Java initializes each element of an array to some sensible default value.
- What do all those brackets and characters mean though?

Java Arrays

- The JVM uses this shorthand notation to indicate the type of the array.
- [indicates an array
- Primitives are denoted with a single letter
- L is used for a class, concluded by a semicolon

[Z = boolean

[B = byte

[S = short

[I = int

[J = long

[F = float

[D = double

[C = char

[L = any non-primitives(Object)

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- Java arrays exhibit *external containment* for non-primitives.
- For non-primitive types, Java does not provide C-style arrays in which all array *values* are stored in a contiguous block of memory.
- This is also true for *multidimensional* arrays of *primitive* types. Why?
- Remember that asterisk back on the 'Array Review' slide?

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What does line 8 print?

Java Arrays

```
1 String[] ss = new String[5];
2 String[] tt = new String[5];
3
4 System.out.println(ss.equals(tt));
5 System.out.println(ss.getClass().getName());
6 System.out.println(new int[7].getClass().getName());
7 System.out.println(new String[5][5].getClass().getName());
8 System.out.println(ss.getClass().getSuperclass().getName());
```

- What does line 8 print?
 - `java.lang.Object`
- Again, in Java arrays are objects and extend `Object`.
- Again, arrays do not override any methods of class `Object`.

Java Arrays

```
1 String[] ss = new String[5];  
2 System.out.println(ss[2]);  
3 int[] is = new int[5];  
4 System.out.println(is[0]);
```

- What do lines 2 and 4 print?

Java Arrays

```
1 String[] ss = new String[5];  
2 System.out.println(ss[2]);  
3 int[] is = new int[5];  
4 System.out.println(is[0]);
```

- What do lines 2 and 4 print?
 - line 2: `null`
 - line 4: `0`
- Again, Java gives default values to objects based on contained type.

Java Arrays

The following chart summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Java Arrays

```
1 | String[] ss = new String[5];  
2 | System.out.println(ss[5]);
```

- What does line 2 print?

Java Arrays

```
1 | String[] ss = new String[5];  
2 | System.out.println(ss[5]);
```

- What does line 2 print?
 - Remember offsets/index vs cardinality?
 - All array accesses are guarded by bounds checks
 - throws *ArrayIndexOutOfBoundsException*
 - How will we deal with this in our translator?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object o = sa;
3 Object[] oa = sa;
4 for(Object object : oa)
5     System.out.print(object);
```

- Is line 2 legal?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object o = sa;
3 Object[] oa = sa;
4 for(Object object : oa)
5     System.out.print(object);
```

- Is line 2 legal?
- Yes of course. Arrays extend Object.

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object o = sa;
3 Object[] oa = sa;
4 for(Object object : oa)
5     System.out.print(object);
```

- Are lines 3 and 4 legal?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object o = sa;
3 Object[] oa = sa;
4 for(Object object : oa)
5     System.out.print(object);
```

- Are lines 3 and 4 legal?
- Yes!
- Java arrays are typed *covariantly*.
- Moreover.. if S extends T, then S[] extends T[]

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};  
2 Object[] oa = sa;  
3 oa[3] = new Object();  
4 sa[3].charAt(3);
```

- Ok, if thats the case then, is this legal Java?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object[] oa = sa;
3 oa[3] = new Object();
4 sa[3].charAt(3);
```

- Ok, if that's the case then, is this legal Java?
 - line 2 is ok, because of covariant array sub-typing
 - line 3 is ok because Object is a subtype of Object
 - line 4 is ok because sa is an array of Strings
- That doesn't seem right... does it?

Java Arrays

```
1 String[] sa = {"I", "am", "an", "array"};
2 Object[] oa = sa;
3 oa[3] = new Object();
4 sa[3].charAt(3);
```

- Lets look at that again..
 - After executing line 2, the arrays *sa* and *oa* reference the same array object in memory.
 - Hence, line 3 would store an *Object* into the 3rd position of the *ss* array.
 - The third line would then attempt to call the *charAt* method on an *Object*.
 - *charAt* can only be called safely on objects of class *String* and its subclasses.

Java Arrays

```
1  String[] sa = {"I", "am", "an", "array"};
2  Object[] oa = sa;
3  oa[3] = new Object();
4  sa[3].charAt(3);
```

- To prevent such unsafe behavior, the JVM will throw a runtime `ArrayStoreException` before executing line 3.
- Moreover, because of covariant subtyping of arrays, all store operations on arrays incur an additional *dynamic type* check.
- Many consider covariant arrays a bad design because dynamic type checks are undesirable due to the performance cost.

Array Translation

Translation Strategy

- Create a C++ type per array type contained.
- We could easily create these before hand for all Java primitive types.
- Let's see what that looks like by starting with int.

Integer Array

- The translation for Java's `int[]` type is similar to the translations of the `String` and `Object` classes.
- We provide the data layout and vtable structure for the array of integers

Array Data Layout

- Forward declarations and typedefs as we've seen before
- Add a field length and a field __data to the data layout.
- length stores the size of the array
- __data stores the actual content of the array

```
1  struct __ArrayOfInt;  
2  struct __ArrayOfInt_VT;  
3  typedef __ArrayOfInt* ArrayOfInt;  
4  
5  struct __ArrayOfInt {  
6      __ArrayOfInt_VT* __vptr;  
7      const int32_t length;  
8      int32_t* __data;  
9  
10     // The constructor.  
11     __ArrayOfInt(int32_t length);  
12  
13     // Returns the class object of int[].  
14     static Class __class();  
15  
16     // The vtable for int[].  
17     static __ArrayOfInt_VT __vtable;  
18 };
```


Array Data Layout

- We do not override any of Object's methods (or add any new ones).
- We use a C-style array to store the array content (i.e. just a pointer)
- This array will be allocated on the heap by the constructor and we'll store a pointer to its first element in `__data`.

```
1  struct __ArrayOfInt;  
2  struct __ArrayOfInt_VT;  
3  typedef __ArrayOfInt* ArrayOfInt;  
4  
5  struct __ArrayOfInt {  
6      __ArrayOfInt_VT* __vptr;  
7      const int32_t length;  
8      int32_t* __data;  
9  
10     // The constructor.  
11     __ArrayOfInt(int32_t length);  
12  
13     // Returns the class object of int[].  
14     static Class __class();  
15  
16     // The vtable for int[].  
17     static __ArrayOfInt_VT __vtable;  
18 };
```

Array Vtable

```
1 struct __Array0fInt_VT {
2     Class __isa;
3     int32_t (*hashCode)(Array0fInt);
4     bool (*equals)(Array0fInt, Object);
5     Class (*getClass)(Array0fInt);
6     String (*toString)(Array0fInt);
7
8     __Array0fInt_VT()
9         : __isa(__Array0fInt::__class()),
10           hashCode((int32_t (*)(Array0fInt))&__Object::hashCode),
11           equals((bool (*)(Array0fInt, Object))&__Object::equals),
12           getClass((Class (*)(Array0fInt))&__Object::getClass),
13           toString((String (*)(Array0fInt))&__Object::toString) {
14     }
15 };
```

Array Constructor

```
1  __Array0fInt::__Array0fInt(int32_t length)
2      : __vptr(&__vtable), length(length), __data(new int32_t[length]()) {
3      // Notice the () at the end of the __data initializer expression!
4      // The () ensures that the C array is initialized by a constructor call.
5
6      // Bad solution for initialization of __data:
7      // reinvents the wheel and potentially slow
8      // for (int i = 0; i < length; i++) {
9      //     __data[i] = 0;
10     // }
11
12     // Ok solution: the C way
13     // std::memset(__data, 0, length * sizeof(int32_t));
14 }
```

Array Implementation

```
1  __Array0fInt::__Array0fInt(int32_t length)
2      : __vptr(&__vtable), length(length), __data(new int32_t[length]()) {
3  }
4
5  Class __Array0fInt::__class() {
6      static Class k =
7          new __Class(__rt::literal("[I"], __Object::__class());
8      return k;
9  }
10
11 __Array0fInt_VT __Array0fInt::__vtable;
```

- Constructor uses initializer lists to properly initialize each of the data members.
- Very similar to Object, String and Class

Array Usage

```
1 // int[] a = new int[5]
2 ArrayOfInt a = new __ArrayOfInt(5);
3
4 // System.out.println(a[2]);
5 __rt::checkIndex(a->length, 2);
6 std::cout << "a[2] : " << a->__data[2] << std::endl;
```

- We can now create and use arrays as expected.
- Accesses to array elements are implemented via direct access to the underlying C-style array.
- C-style arrays are not bounds-checked, we have to add these checks explicitly.

Array Bounds Checking

```
1  inline void checkIndex(int32_t length, int32_t index) {  
2      if (0 > index || index >= length) {  
3          throw java::lang::ArrayIndexOutOfBoundsException();  
4      }  
5  }
```

- Since C arrays are not bounds checked (and seg-faulting is not an option) we'll need to check bounds ourselves prior to each dereferencing of an array element.
- Should we put this in our `__ArrayOfInt` class?
- We'll throw this in our general purpose namespace "rt"
- Oh wait... we're throwing exceptions??

Exceptions

```
1  class Throwable { };
2
3  class Exception : public Throwable { };
4
5  class RuntimeException : public Exception { };
6
7  class NullPointerException : public RuntimeException { };
8
9  class NegativeArraySizeException : public RuntimeException { };
10
11 class ArrayStoreException : public RuntimeException { };
12
13 class ClassCastException : public RuntimeException { };
14
15 class IndexOutOfBoundsException : public RuntimeException { };
16
17 class ArrayIndexOutOfBoundsException : public IndexOutOfBoundsException { };
```

- For simplicity, we use C++ inheritance for exceptions and throw them by value.
- In other words, the translator does not support user-defined exceptions and simply relies on a few built-in classes.

As usual...

- The code is available for you to review and experiment with at <https://github.com/nyu-oop-fall16/java-lang/tree/master/java-lang-2>
- Additions to the `java_lang.*` files are going to be
your responsibility to port to your translator project
- But, don't move this one yet.....

What's wrong with this?

