# Object-Oriented Programming

CSCI-UA 0470-001
Class 11
Instructor: Randy Shepherd

# Arrays

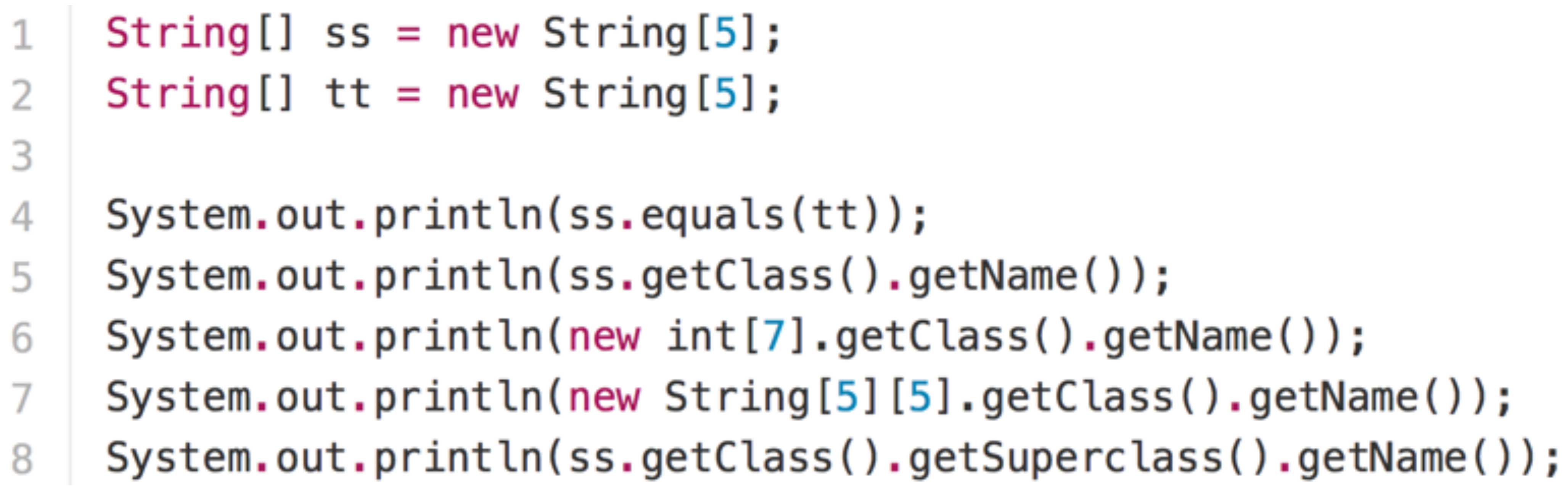
# Array Review

- 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. *****

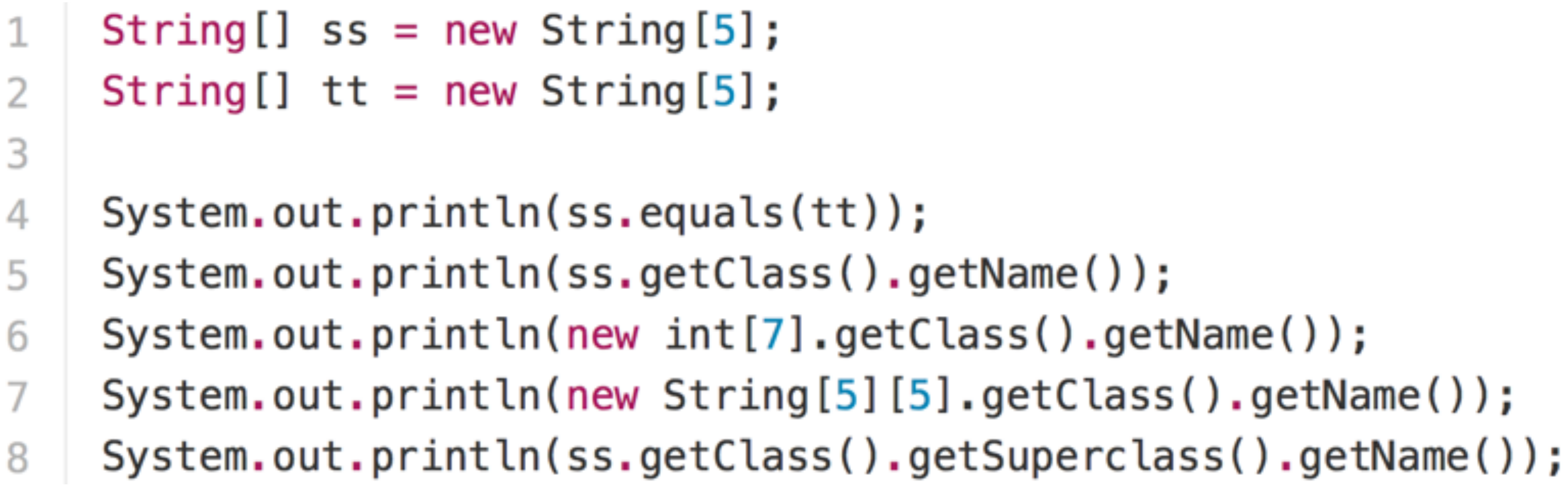
# Java Arrays



```
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?

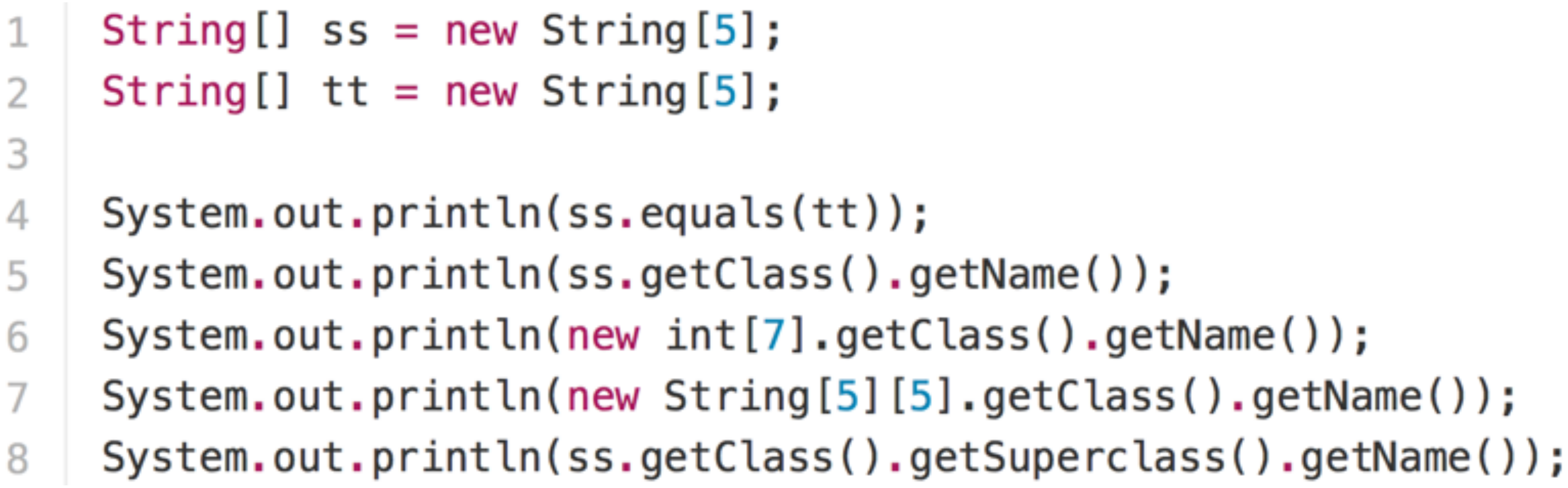
# Java Arrays



```
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



```java
String[] ss = new String[5];
String[] tt = new String[5];

System.out.println(ss.equals(tt));
System.out.println(ss.getClass().getName());
System.out.println(new int[7].getClass().getName());
System.out.println(new String[5][5].getClass().getName());
System.out.println(ss.getClass().getSuperclass().getName());
```

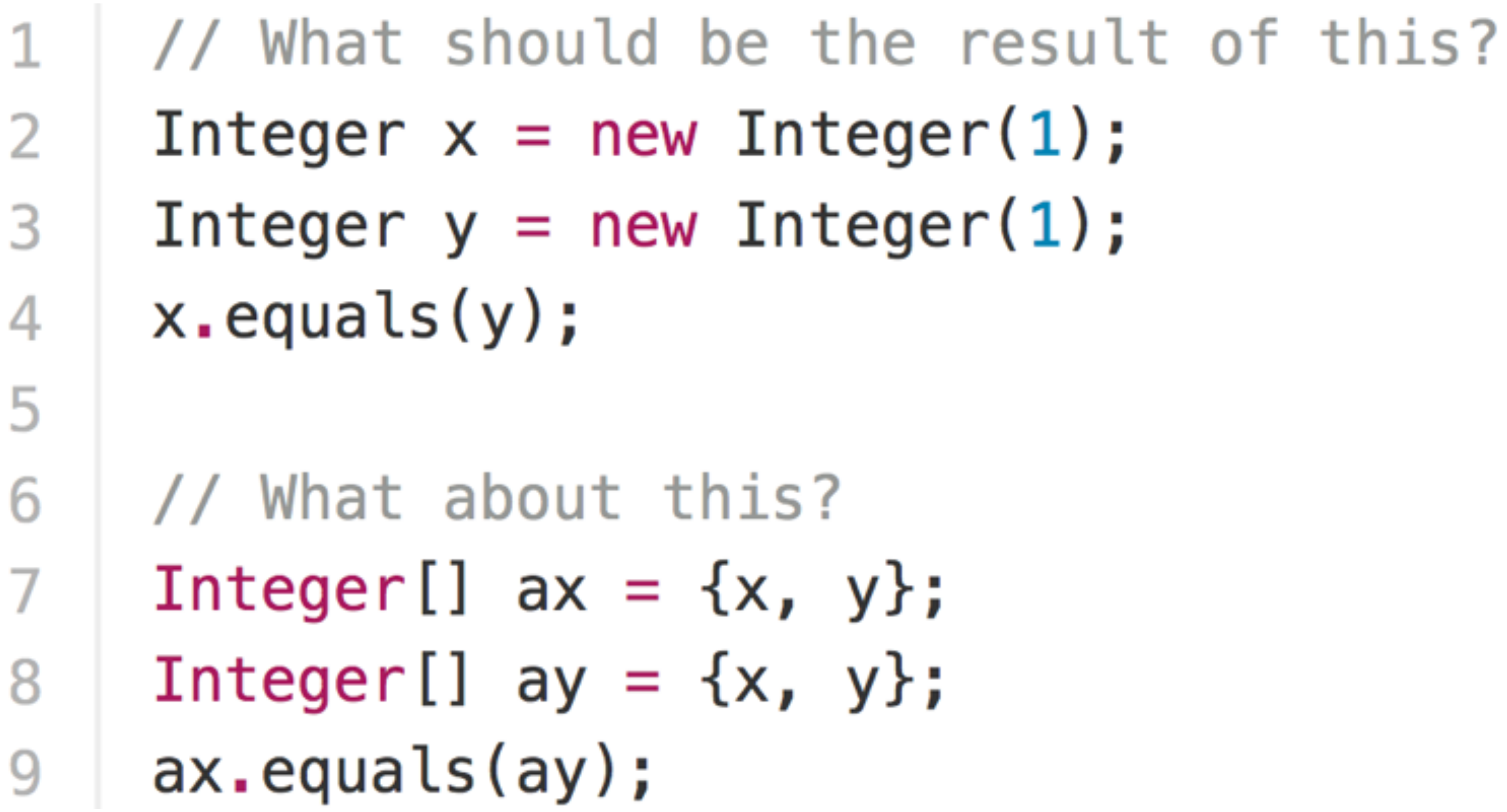
- Arrays, like any other non-primitive type in Java, are objects and extend Object.

- Arrays do not override any methods of class Object.

- Object *equals* compares the memory location. Remember?

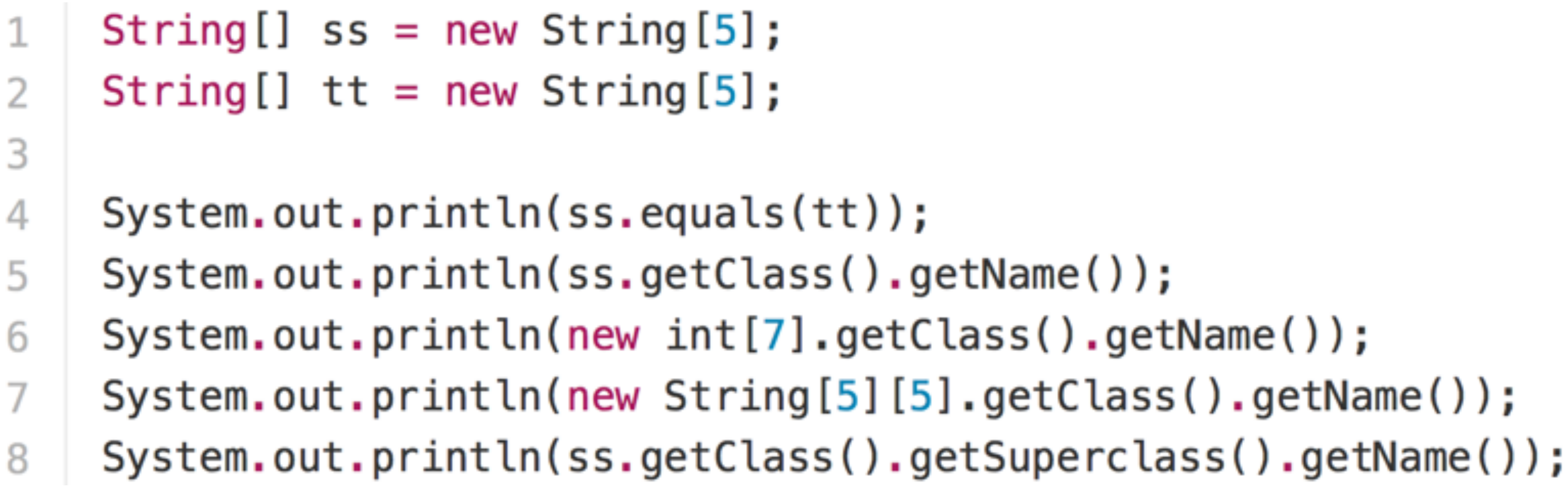
# Identity Vs Equality

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



```
// What should be the result of this?
Integer x = new Integer(1);
Integer y = new Integer(1);
x.equals(y);

// What about this?
Integer[] ax = {x, y};
Integer[] ay = {x, y};
ax.equals(ay);
```

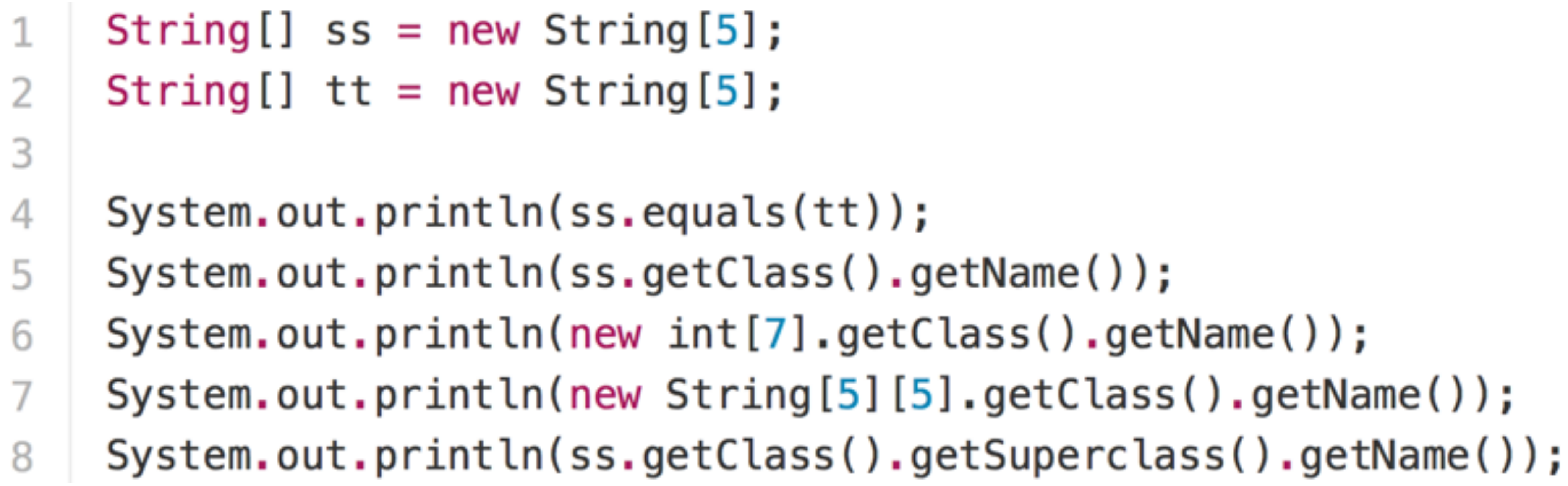
# Java Arrays



```
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

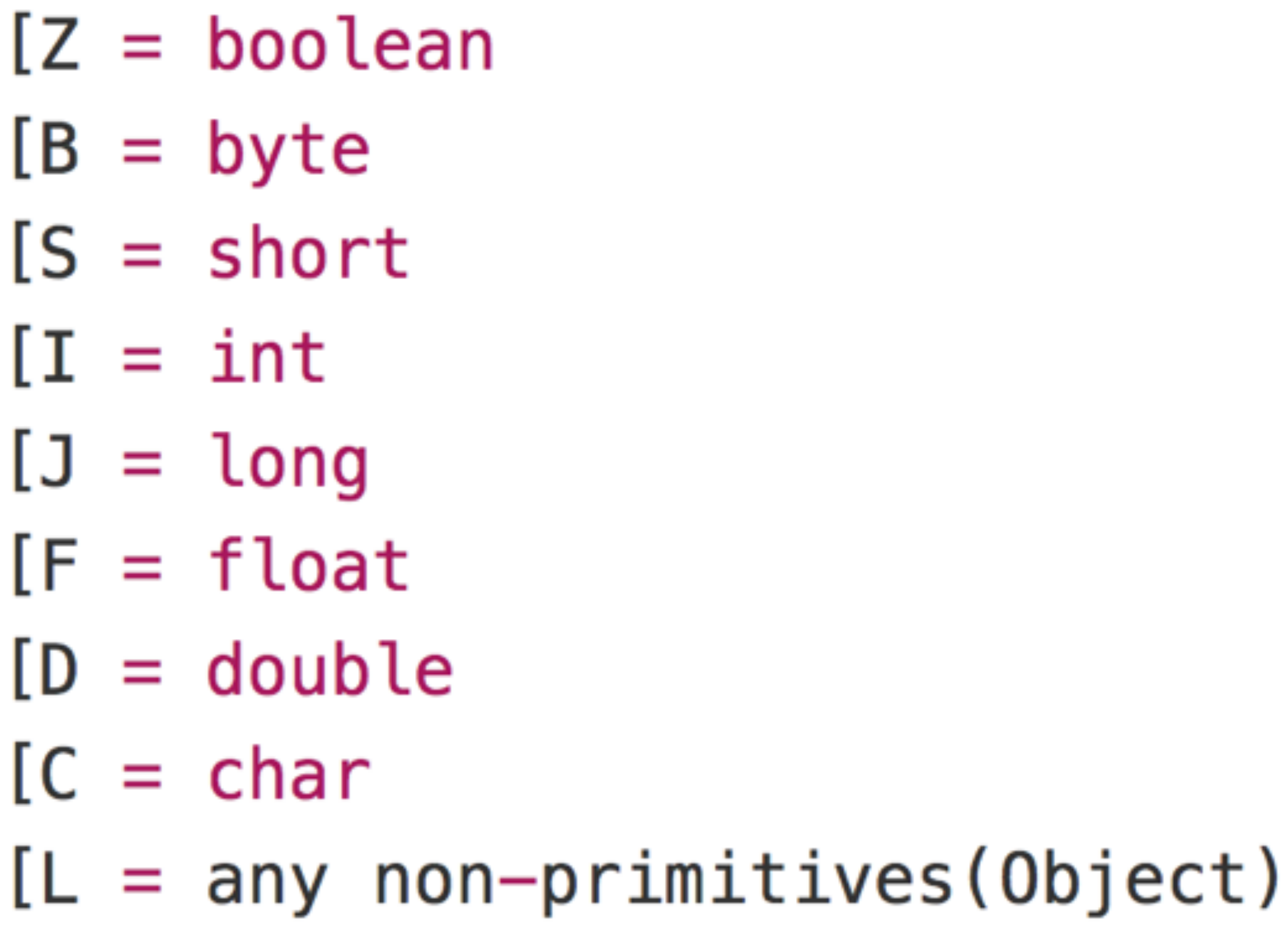


```
1  String[] ss = new String[5];
2  String[] tt = new String[5];
3
4  System.out.println(ss.equals(tt));
5  System.out.println(ss.getClass().getName());
6  System.out.println(new int[7].getClass().getName());
7  System.out.println(new String[5][5].getClass().getName());
8  System.out.println(ss.getClass().getSuperclass().getName());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

- 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?

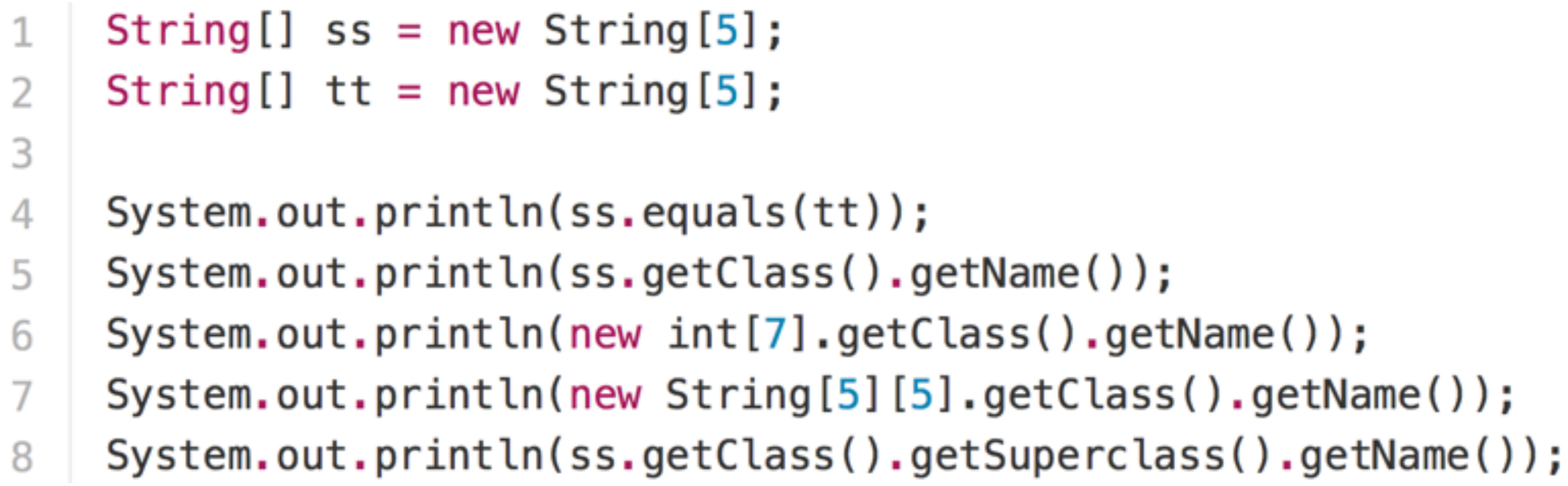
# Java Arrays

- The JVM uses this shorthand notation to indicate the type of the array.
- Primitives are denoted with a single letter
- [ indicates an array
- L is used for a class (terminated by a ;)
- Why no closing ‘]’?



```
[Z = boolean
[B = byte
[S = short
[I = int
[J = long
[F = float
[D = double
[C = char
[L = any non-primitives(Object)
```

# Java Arrays



```java
String[] ss = new String[5];
String[] tt = new String[5];

System.out.println(ss.equals(tt));
System.out.println(ss.getClass().getName());
System.out.println(new int[7].getClass().getName());
System.out.println(new String[5][5].getClass().getName());
System.out.println(ss.getClass().getSuperclass().getName());
```

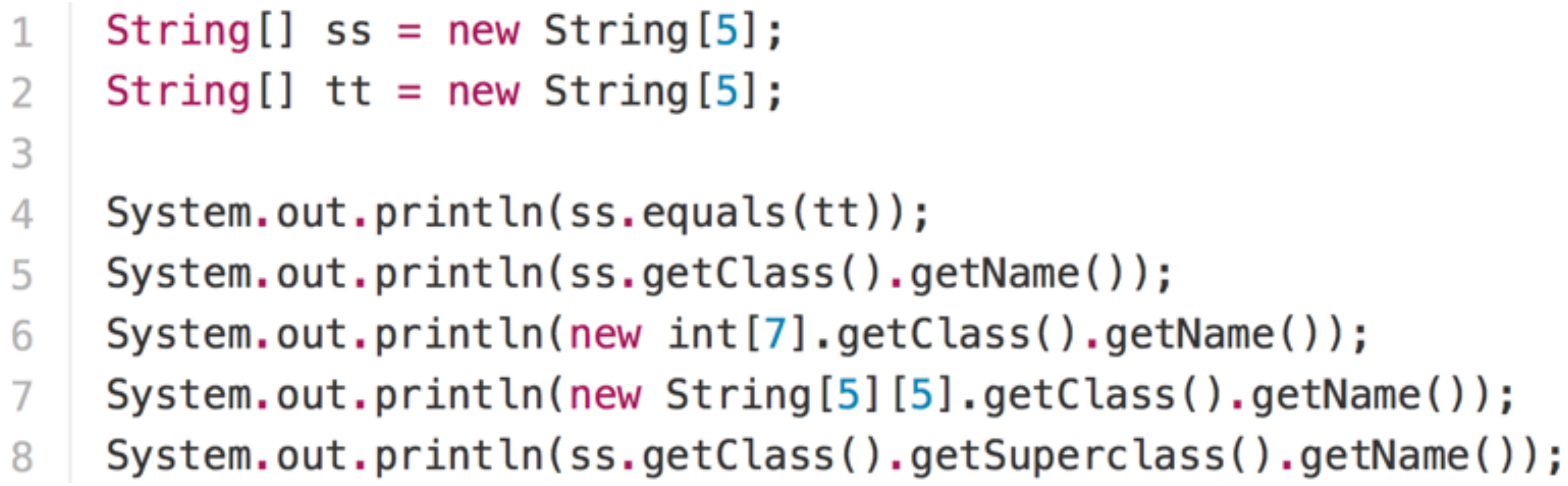
- Java arrays exhibit *external containment* for non-primitives.

- 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?

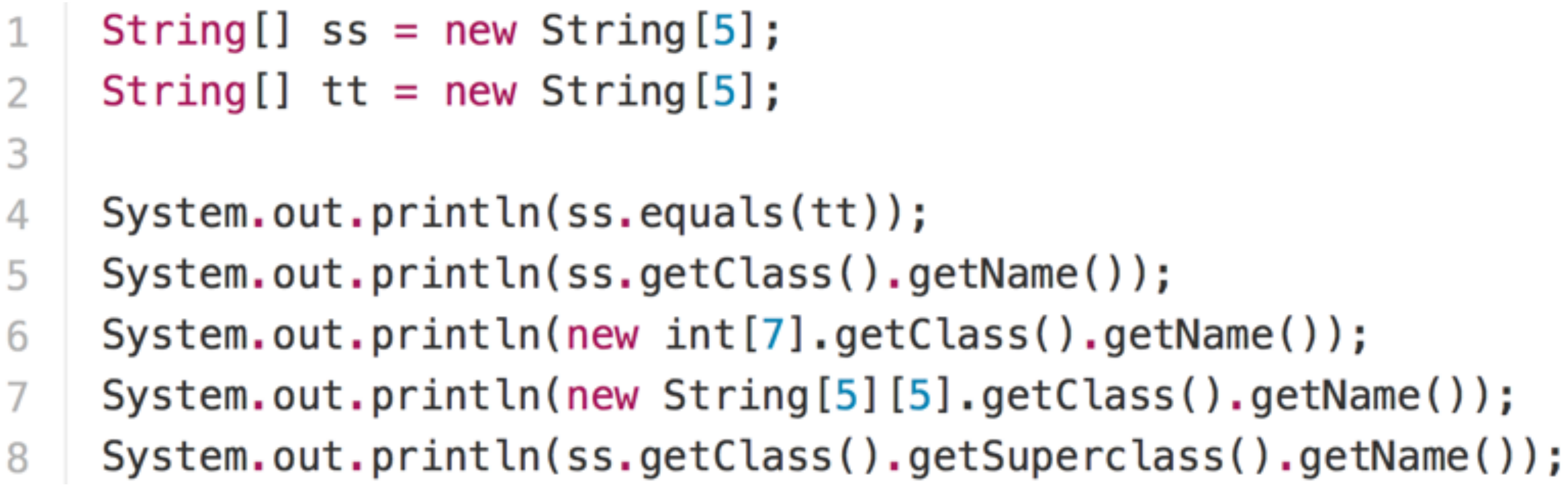
# Java Arrays



```
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?

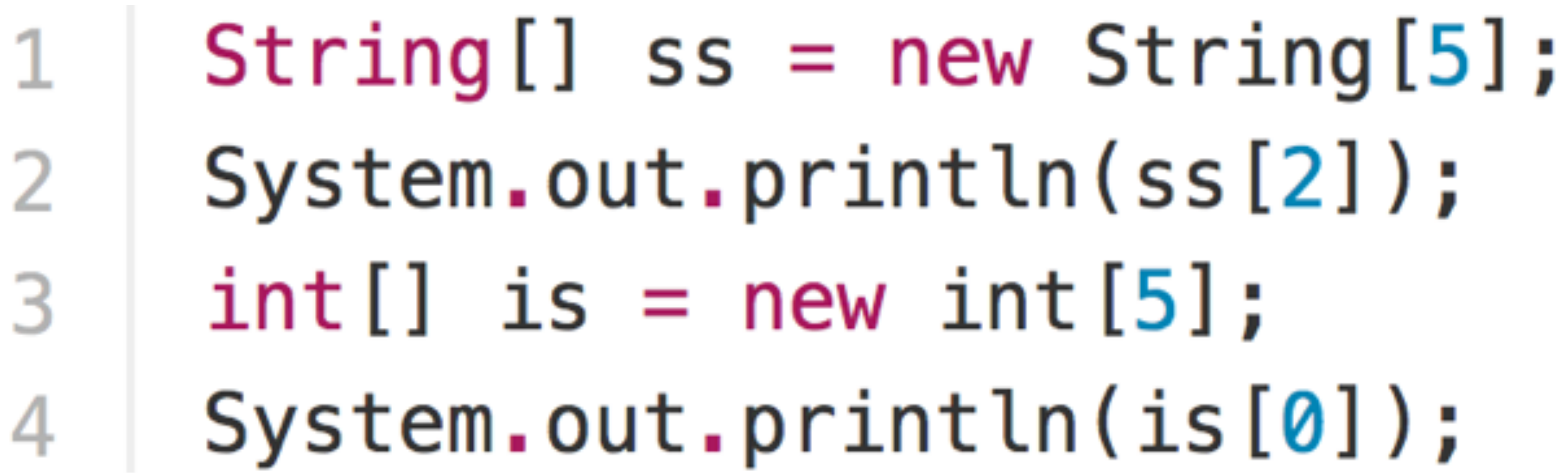
# Java Arrays



```
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.

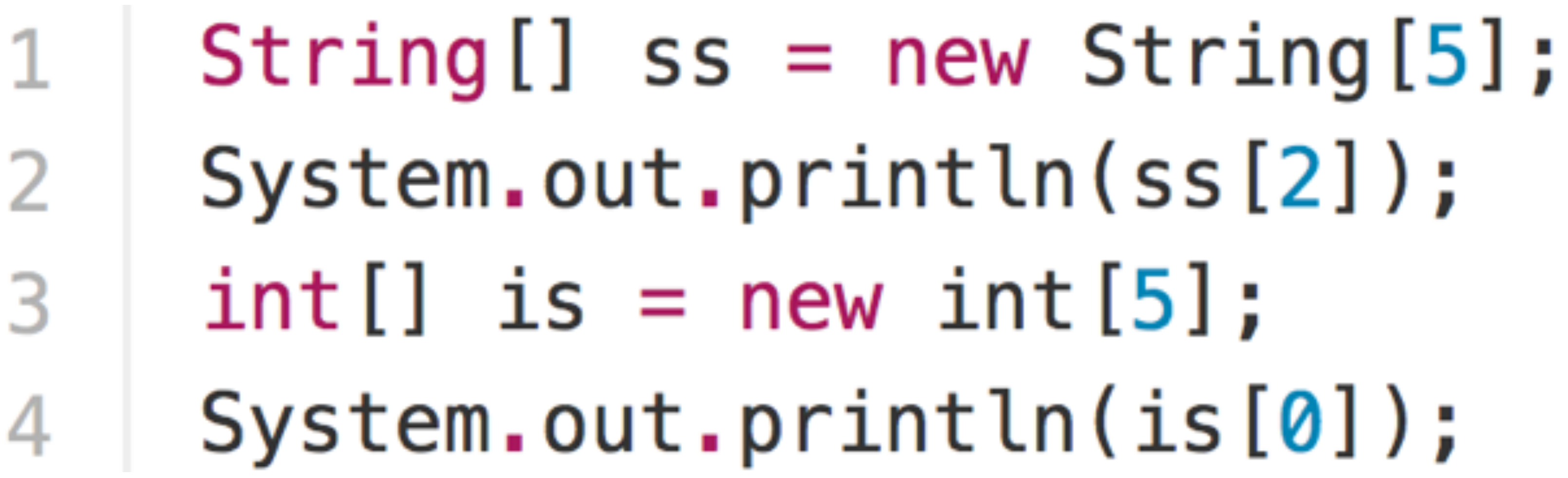
# Java Arrays



```
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?

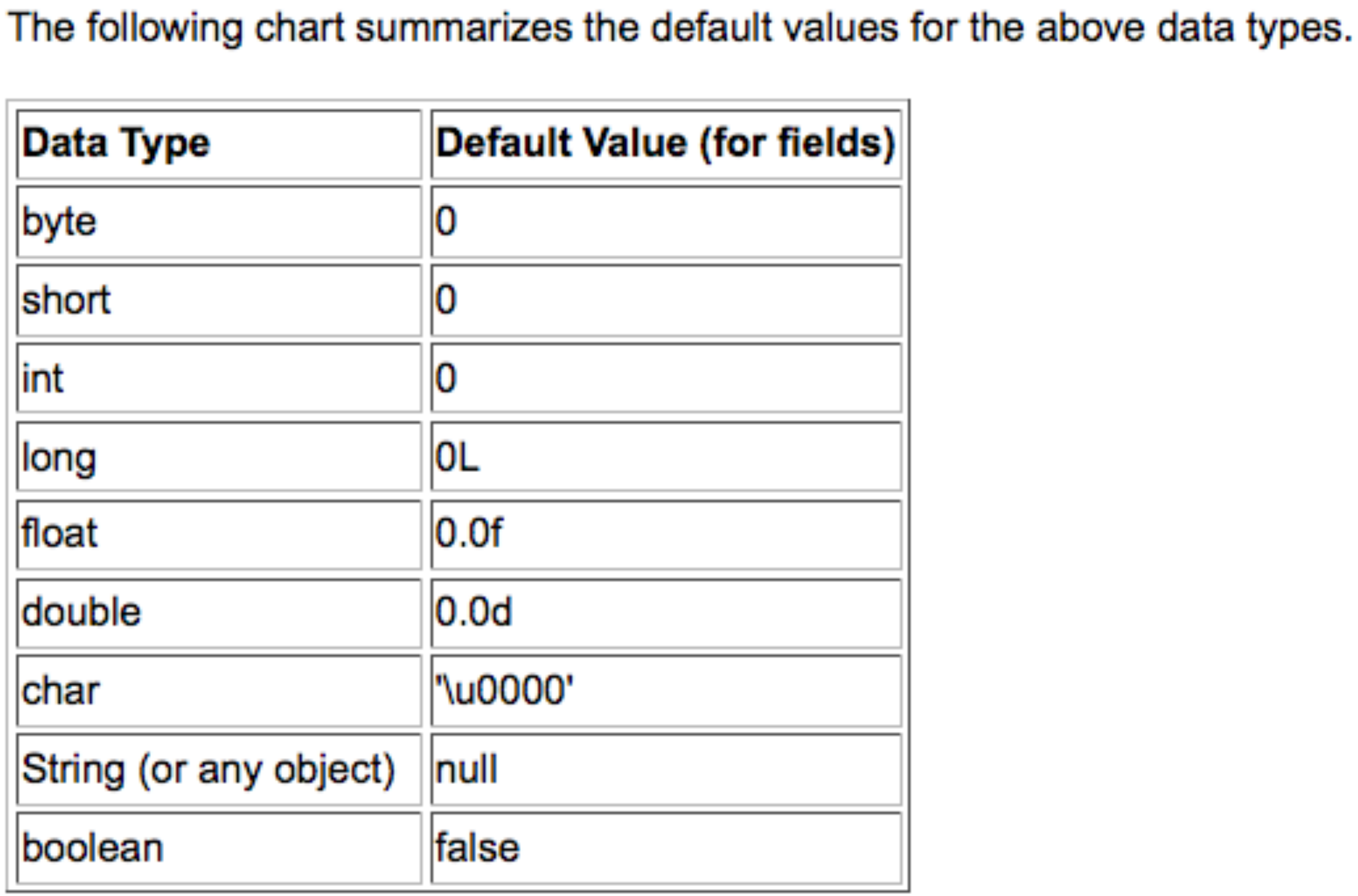
# Java Arrays



```
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.

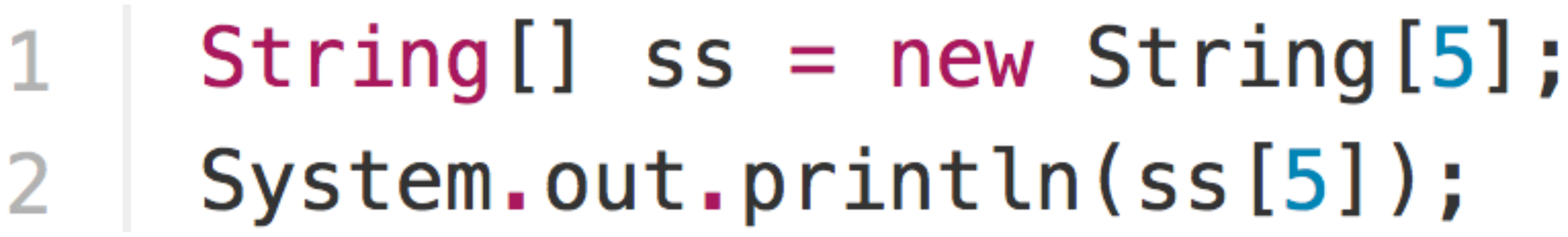
# Java Arrays



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 |

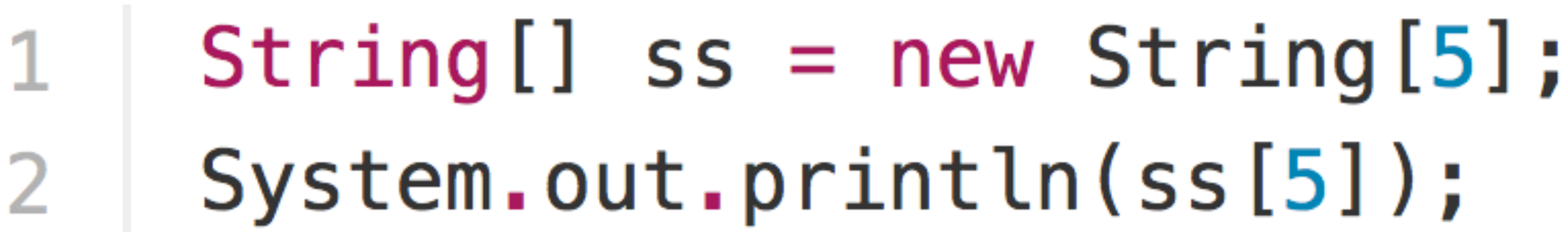
# Java Arrays



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

- What does line 2 print?

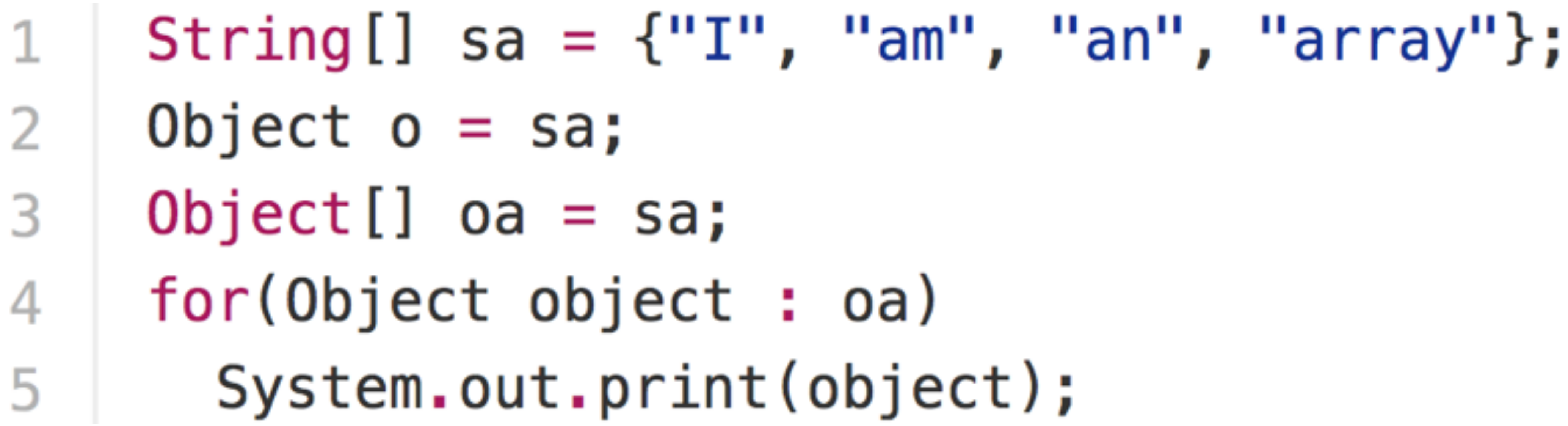
# Java Arrays



```
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?

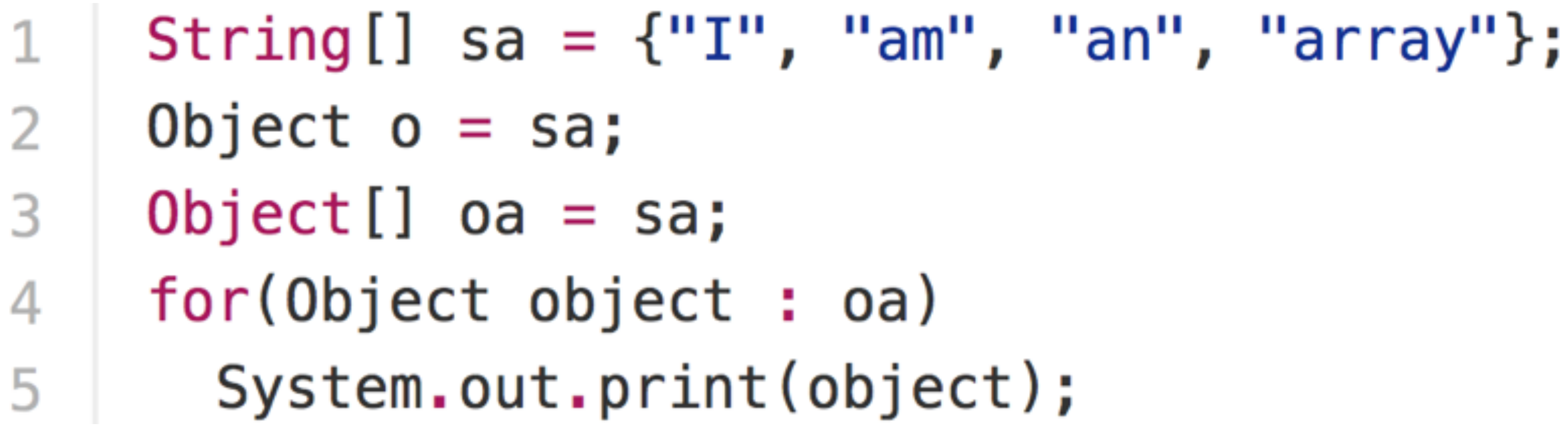
# Java Arrays



```
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?

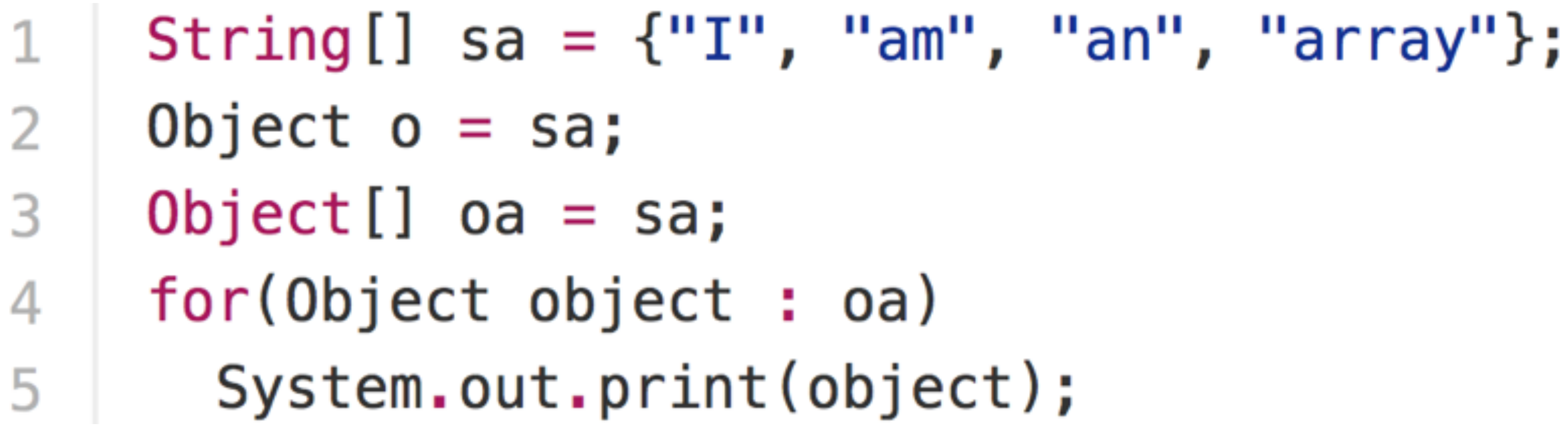
# Java Arrays



```
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.

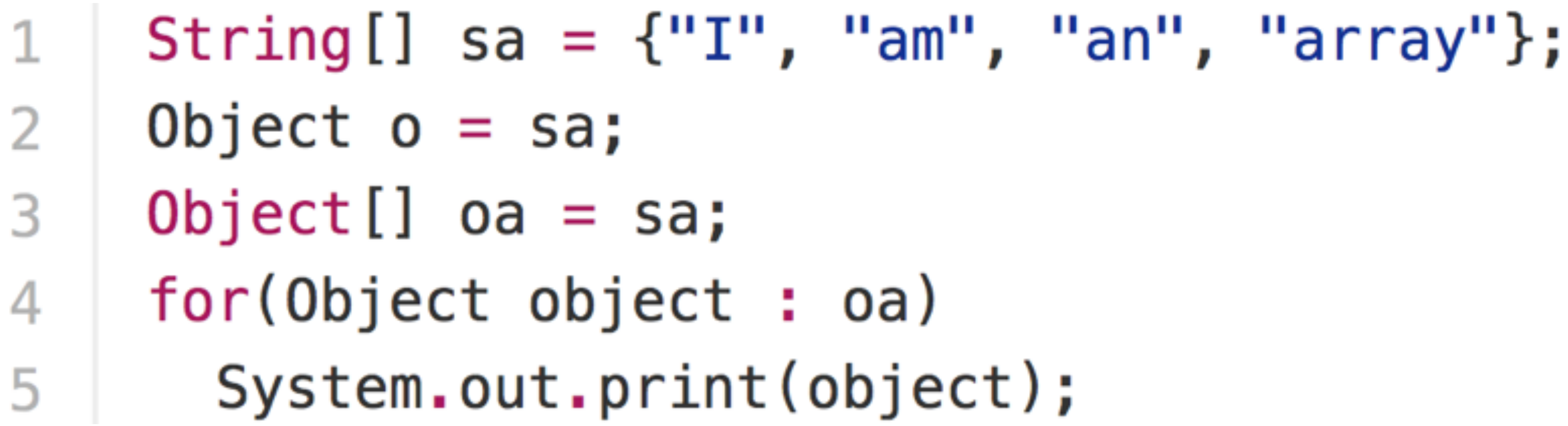
# Java Arrays



```
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?

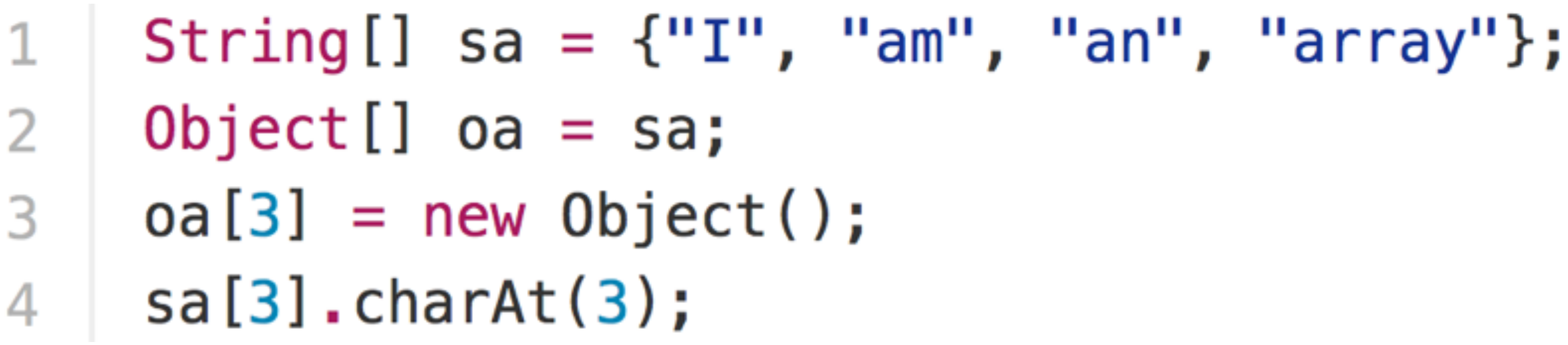
# Java Arrays



```
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.* *(What a great final exam question subject.)*
  - Moreover.. if S extends T, then S[] extends T[]

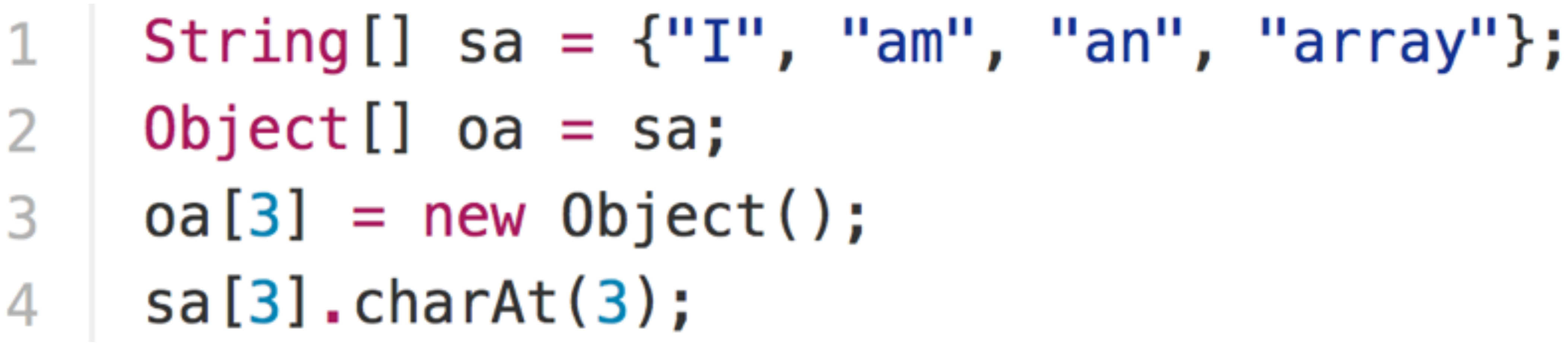
# Java Arrays



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

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

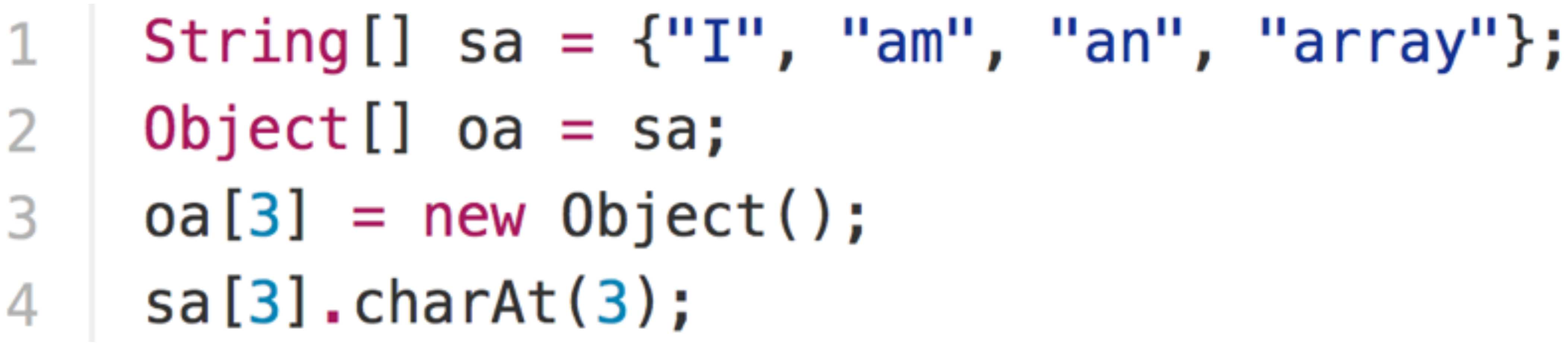
# Java Arrays



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

- Ok, if thats 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?

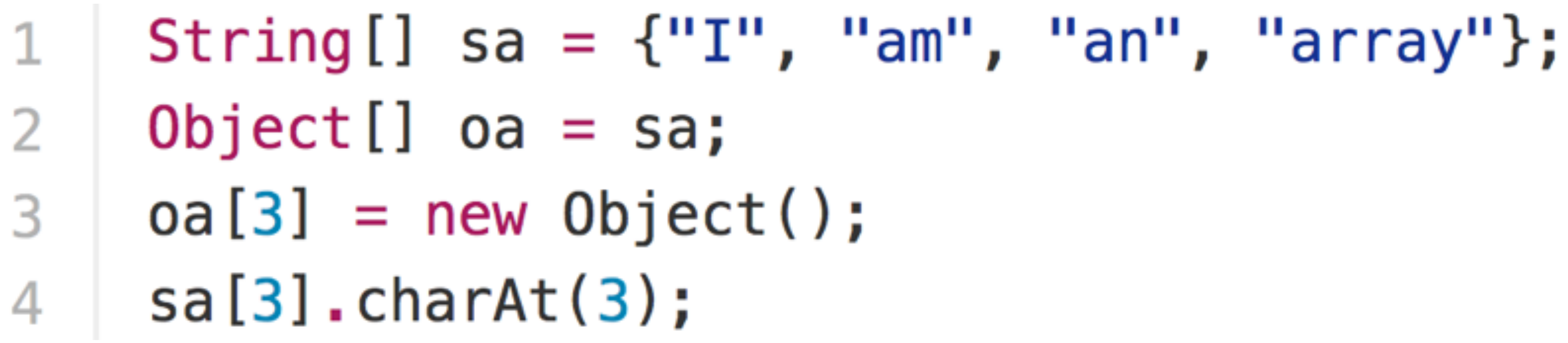
# Java Arrays



```
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, the second line 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 an ArrayStoreException before executing the second line.

- That is, because of covariant subtyping of arrays, all store operations on arrays incur an additional *dynamic type* check.

- As seen with the design decisions of vtables, Dynamic type checks are undesirable due to the performance cost.

# Array Translation

# Translation Strategy

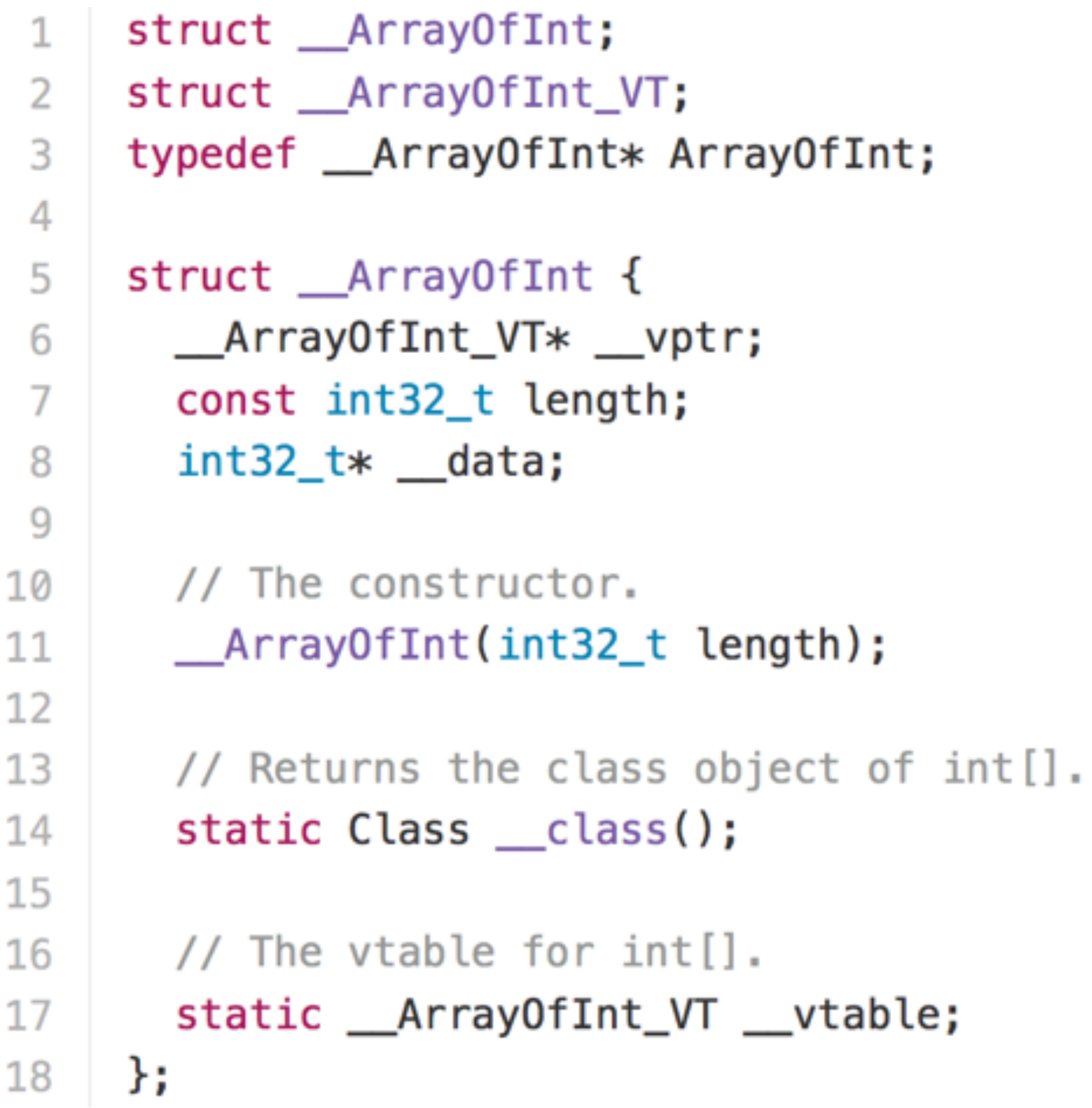
- Create a C++ type per array type contained.
- We could easily create these before hand for all Java primitive types.
- Let’s see what that looks like by starting with int.

# 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 if integers

# Array Data Layout

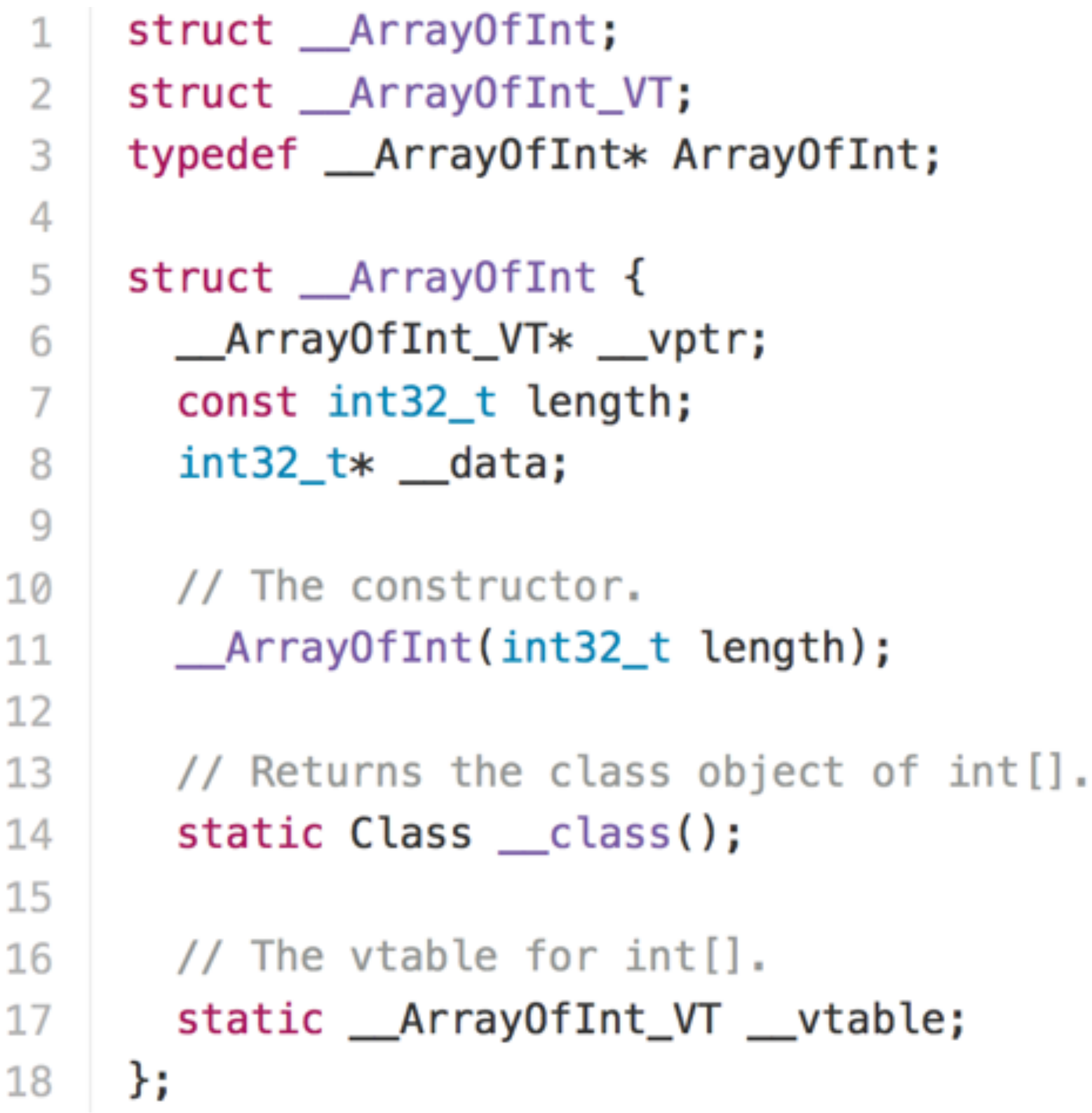
- Forward declarations and typedefs as we've seen before
- Add a field length and a field __data to the data layout.
  - length stores the size of the array
  - __data stores the actual content of the array



```
struct __ArrayOfInt;
struct __ArrayOfInt_VT;
typedef __ArrayOfInt* ArrayOfInt;

struct __ArrayOfInt {
  __ArrayOfInt_VT* __vptr;
  const int32_t length;
  int32_t* __data;

  // The constructor.
  __ArrayOfInt(int32_t length);

  // Returns the class object of int[].
  static Class __class();

  // The vtable for int[].
  static __ArrayOfInt_VT __vtable;
};
```

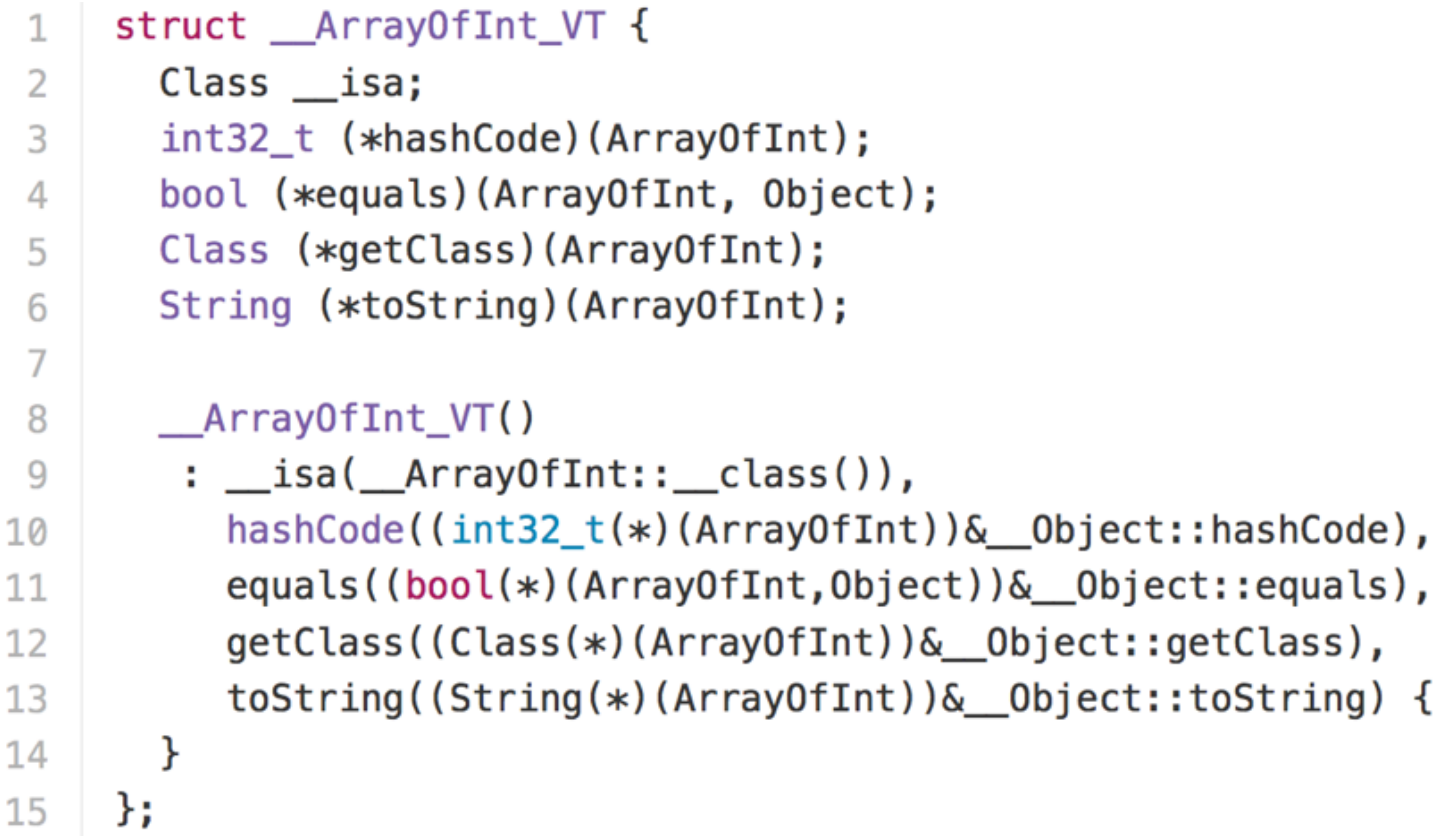
# Array Data Layout

- 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.



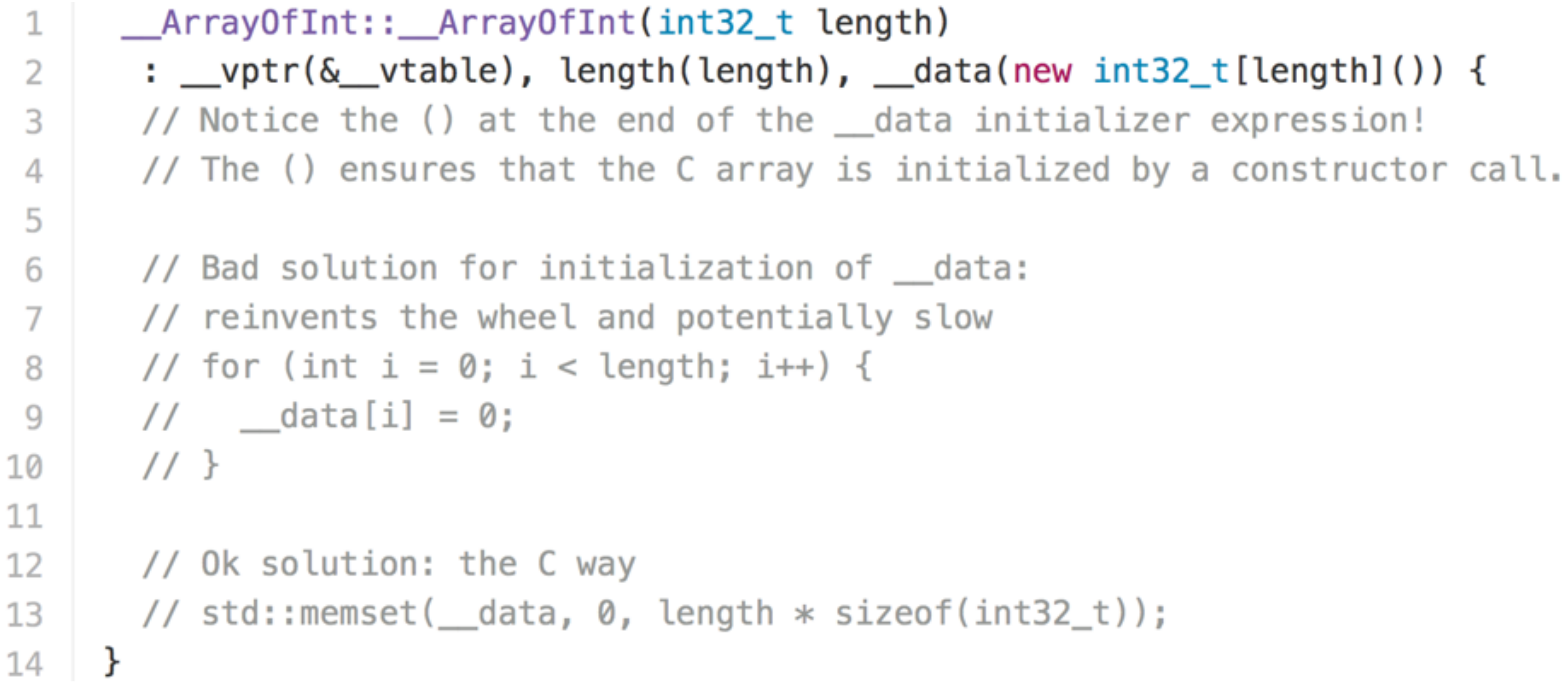
```
struct __ArrayOfInt;
struct __ArrayOfInt_VT;
typedef __ArrayOfInt* ArrayOfInt;

struct __ArrayOfInt {
  __ArrayOfInt_VT* __vptr;
  const int32_t length;
  int32_t* __data;

  // The constructor.
  __ArrayOfInt(int32_t length);

  // Returns the class object of int[].
  static Class __class();

  // The vtable for int[].
  static __ArrayOfInt_VT __vtable;
};
```

# Array Vtable



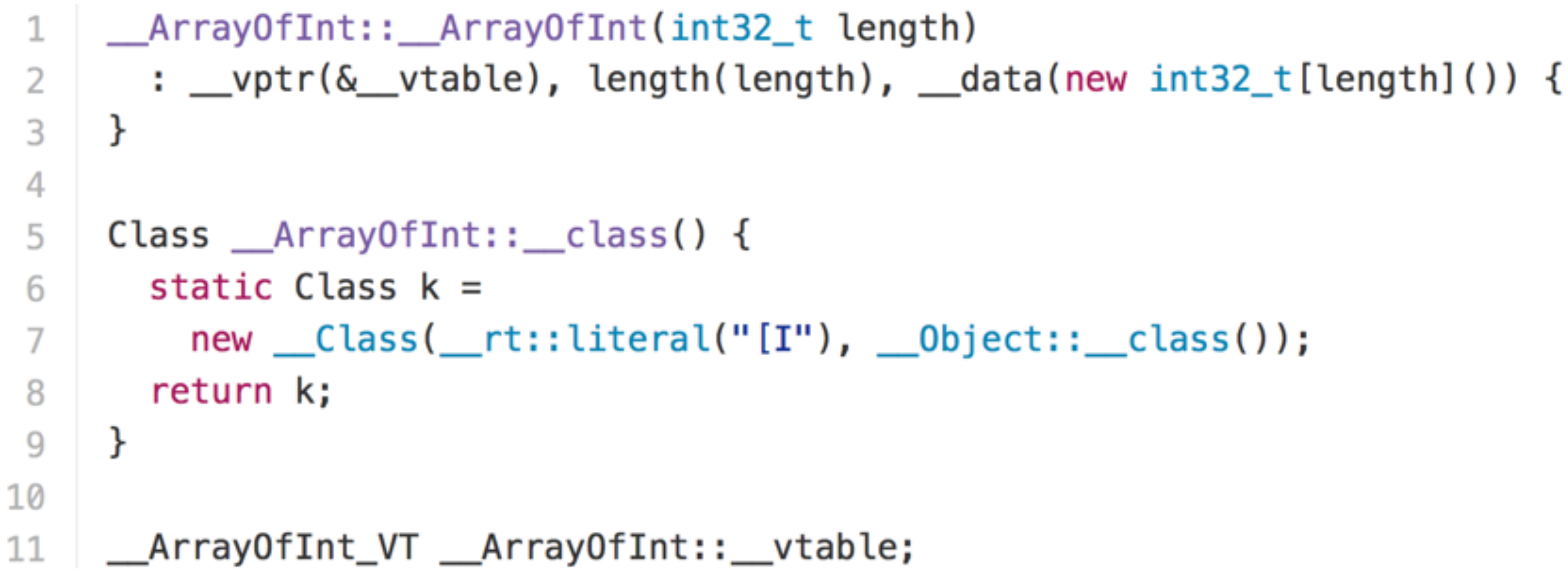
```
struct __ArrayOfInt_VT {
  Class __isa;
  int32_t (*hashCode)(ArrayOfInt);
  bool (*equals)(ArrayOfInt, Object);
  Class (*getClass)(ArrayOfInt);
  String (*toString)(ArrayOfInt);

  __ArrayOfInt_VT()
   : __isa(__ArrayOfInt::__class()),
     hashCode((int32_t(*)(ArrayOfInt))&__Object::hashCode),
     equals((bool(*)(ArrayOfInt,Object))&__Object::equals),
     getClass((Class(*)(ArrayOfInt))&__Object::getClass),
     toString((String(*)(ArrayOfInt))&__Object::toString) {
  }
};
```

# Array Constructor



```
 __ArrayOfInt::__ArrayOfInt(int32_t length)
  : __vptr(&__vtable), length(length), __data(new int32_t[length]()) {
  // Notice the () at the end of the __data initializer expression!
  // The () ensures that the C array is initialized by a constructor call.

  // Bad solution for initialization of __data:
  // reinvents the wheel and potentially slow
  // for (int i = 0; i < length; i++) {
  //   __data[i] = 0;
  // }

  // Ok solution: the C way
  // std::memset(__data, 0, length * sizeof(int32_t));
}
```

# Array Implementation

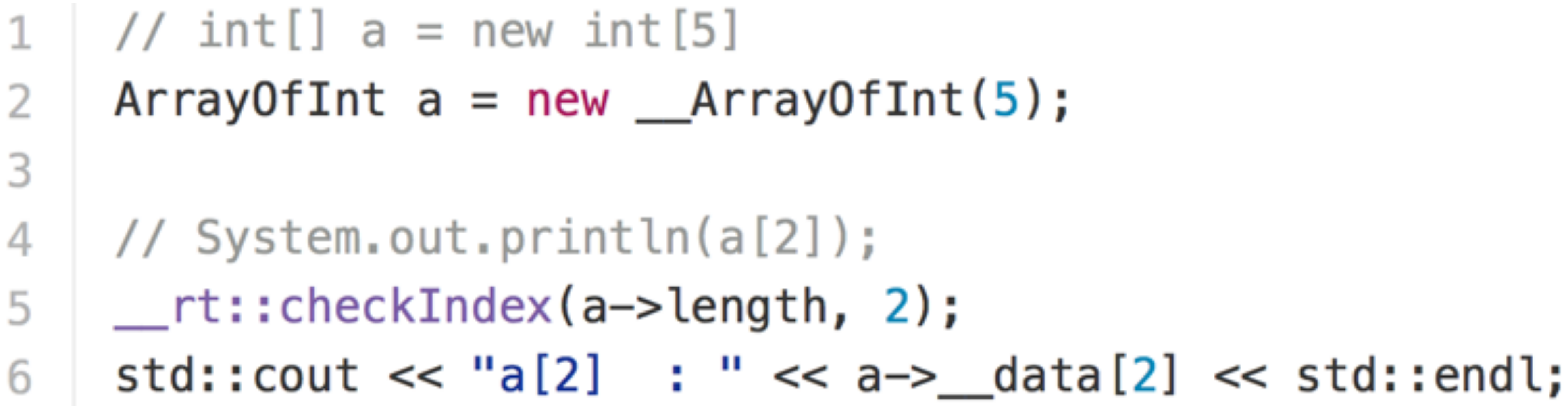


```
__ArrayOfInt::__ArrayOfInt(int32_t length)
  : __vptr(&__vtable), length(length), __data(new int32_t[length]()) {
}

Class __ArrayOfInt::__class() {
  static Class k =
    new __Class(__rt::literal("[I"), __Object::__class());
  return k;
}

__ArrayOfInt_VT __ArrayOfInt::__vtable;
```

- Constructor uses initializer lists to properly initialize each of the data members.

- Very similar to Object, String and Class

# Array Usage



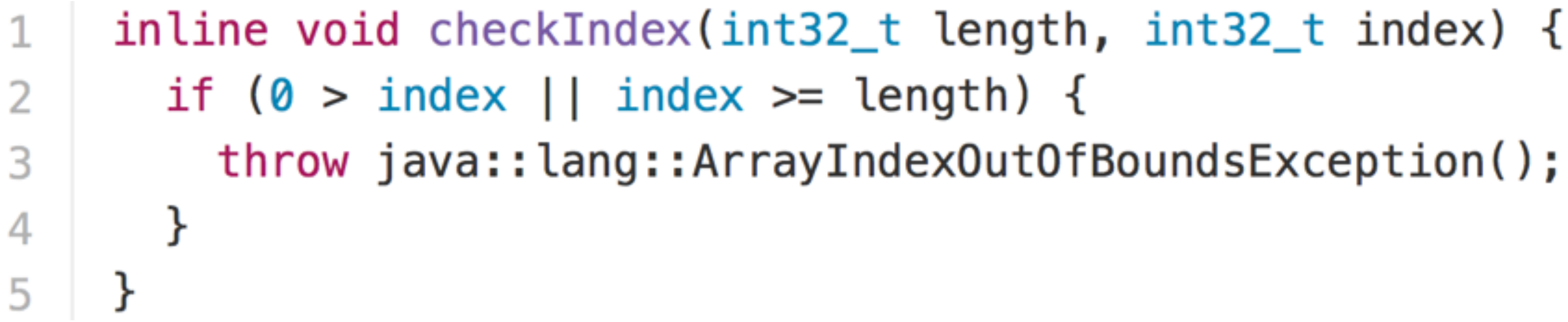
```
// int[] a = new int[5]
ArrayOfInt a = new __ArrayOfInt(5);

// System.out.println(a[2]);
__rt::checkIndex(a->length, 2);
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



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

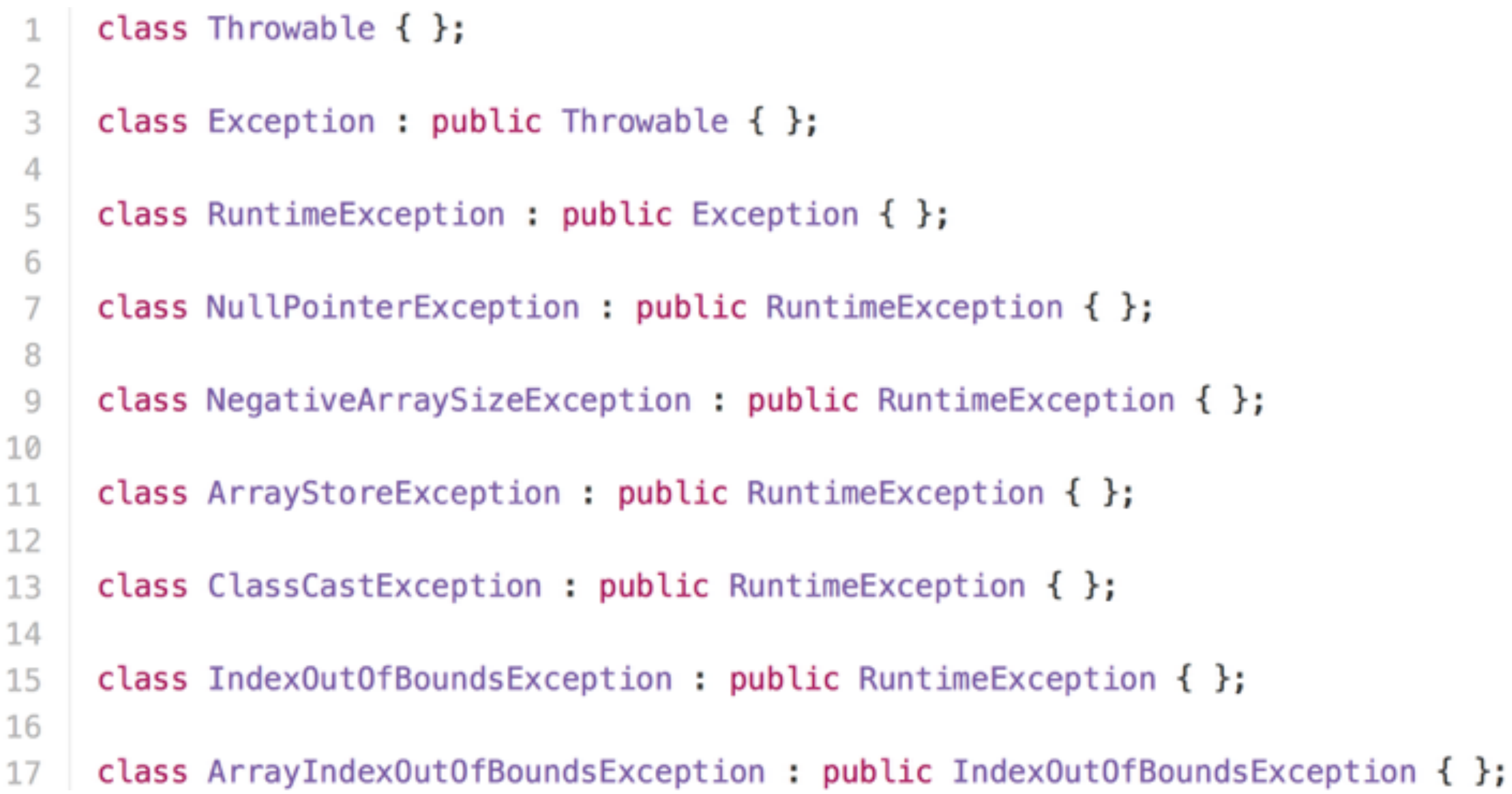
- Since C arrays are not bounds checked (and seg-faulting is not an option) we'll need to check bounds ourselves prior to each dereferencing of an array element.

- 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



```cpp
class Throwable { };

class Exception : public Throwable { };

class RuntimeException : public Exception { };

class NullPointerException : public RuntimeException { };

class NegativeArraySizeException : public RuntimeException { };

class ArrayStoreException : public RuntimeException { };

class ClassCastException : public RuntimeException { };

class IndexOutOfBoundsException : public RuntimeException { };

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/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?