CS 2100: Data Structures & Algorithms 1

References

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Primitives vs. Objects

Reference Type	Primitive Type
It is not pre-defined except the String. (Usually defined from **classes**.)	It is pre-defined in Java.
All reference type begins with Uppercase letter.	All primitive type begins with a lowercase letter.
Non-primitive types have all the same size.	The size of a primitive type depends on the data type.
It is used to invoke or call methods.	We cannot invoke the method with a primitive type.
It can be null.	It cannot be null. It always has value.
Examples of reference data types are class, Arrays, String, Interface, etc.	Examples of primitive data types are int, float, double, long, etc.
JVM allocates 8 bytes for each reference variable, by **default**.	Its size depends on the data type.
MUST be compared with special .equals() method.	May be compared with double equal sign "=="

Primitives vs. Objects

- **Primitives** in Java (e.g., int, double, etc.)
  - Store a value directly in memory
  - Variable refers directly to that memory address
  - Creates space on the stack (compile time)

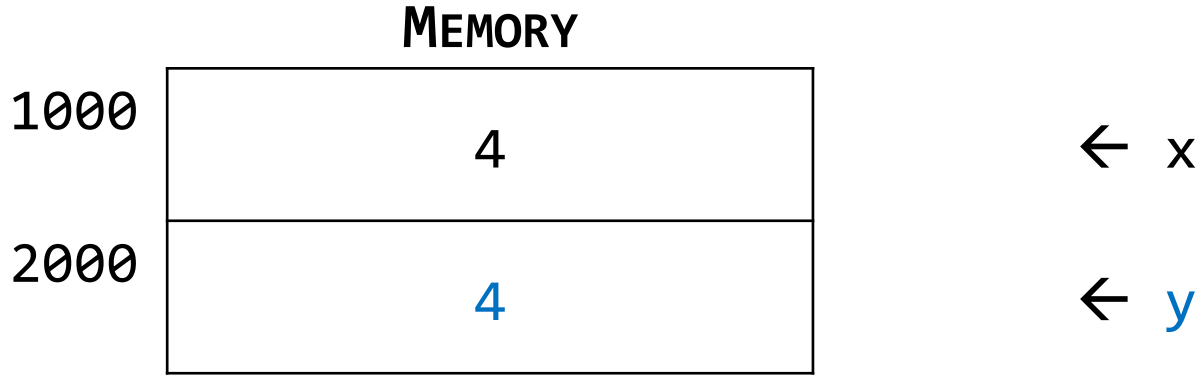
- Objects in Java are stored as **References**
  - Stores memory address of the variable
  - Uses space on the heap (run time)
  - Makes **parameter passing** and equality tricky *(examples coming up next class!)*

(Recall) Primitive Data Types

Primitive Types: ("non-reference types")

```
int x = 4;
int y = x;
```

Two copies of the data made



Actual values are stored in memory.

Primitive Types vs. Reference Types

Primitive type:
(Built-in to Java)

- A "box" or chunk of memory holding the data itself
  - Ex: int, double, …

Reference (class) type:

- All objects defined from **classes**
- The object "refers to" or "points to" the chunk of memory that actually holds the data

Reference type (cont'd):

- An **object-variable** must be made to refer to a chunk
  - Create chunks with "**new**" (which calls a **constructor**)
  - Use assignment ("=")
  - null value for an object-variable: not pointing to *anything*
- Example:

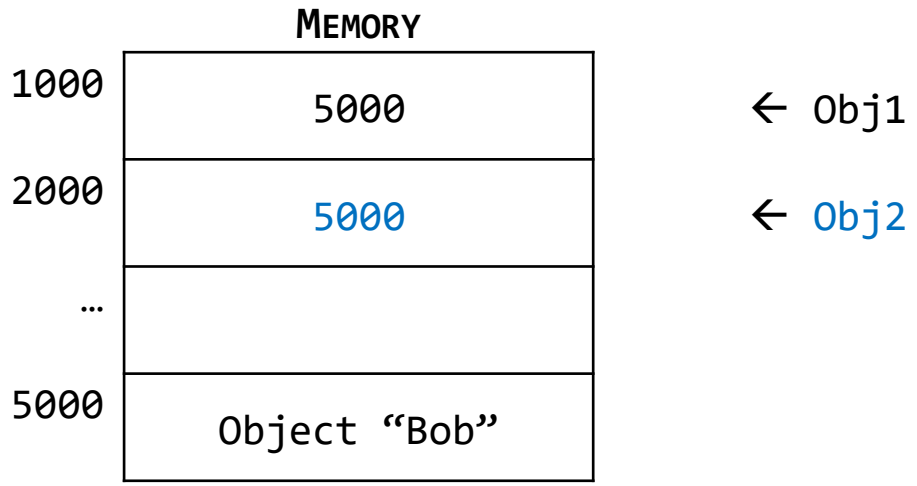
```
MyClass obj1 = new MyClass();
```

Reference Types

Reference Types: (more complicated declaration)

```
MyClass obj1 = new MyClass("Bob");
MyClass obj2 = obj1; // not copying data; new name for
                     // same data chunk in memory
```

Object 1 and Object 2 are referencing the class object. Their memory locations consist of an address (**5000**) to another memory location where the **object is located**.



Understanding the reference type declaration...

Declaration:

- A variable declaration associates a variable name with an object type (data type)

Instantiation:

- The "new" keyword is a Java operator that creates the object

Initialization:

- The "new" operator is followed by a call to a constructor, which initializes the new object (A constructor is a special kind of method in the object class)

Example (with the pieces color coded):

```
Point originOne
```

Understanding the reference type declaration...

Declaration:

- A variable declaration associates a variable name with an object type (data type)

Instantiation:

- The "new" keyword is a Java operator that creates the object

Initialization:

- The "new" operator is followed by a call to a constructor, which initializes the new object (A constructor is a special kind of method in the object class)

Example (with the pieces color coded):

```
Point originOne = new
```

Understanding the reference type declaration...

Declaration:

- A variable declaration associates a variable name with an object type (data type)

Instantiation:

- The "new" keyword is a Java operator that creates the object

Initialization:

- The "new" operator is followed by a call to a constructor, which initializes the new object (A constructor is a special kind of method in the object class)

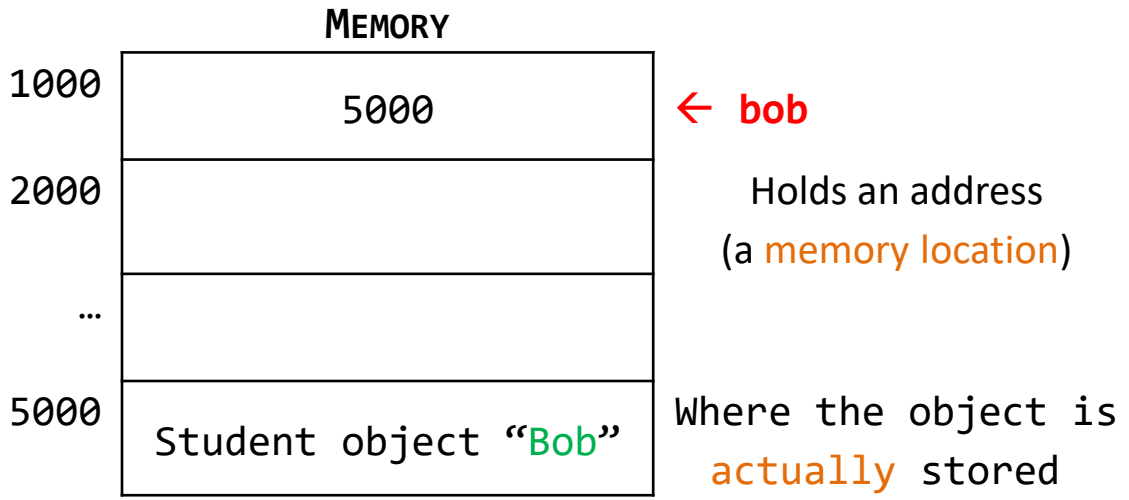
Example (with the pieces color coded):

```
Point originOne = new Point(23, 45); // (x and y coordinates)
```

Reference Types

- `Student bob`
  - Is `bob` a student? No!
  - Reference to student, not a REAL student object yet
- `Student bob = new Student("Bob");`
  - Is `bob` a student (now)? Yes!
- `Student("Bob")`
  - The **constructor** method

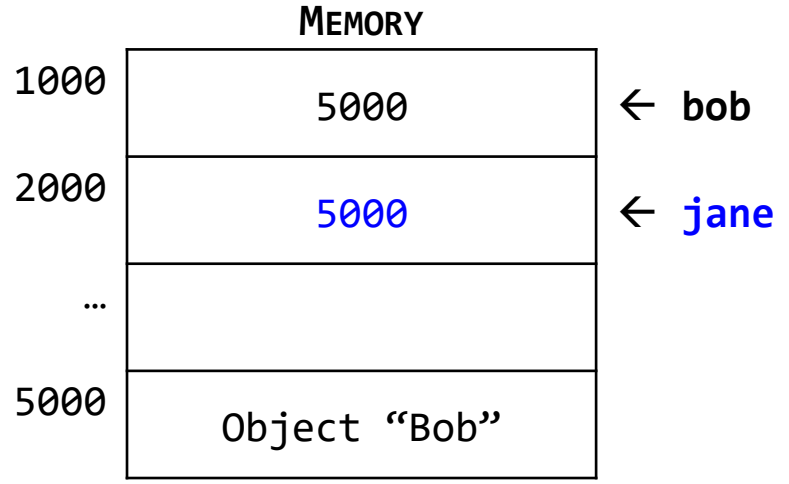
  Is called at the time the object is created



Reference Types

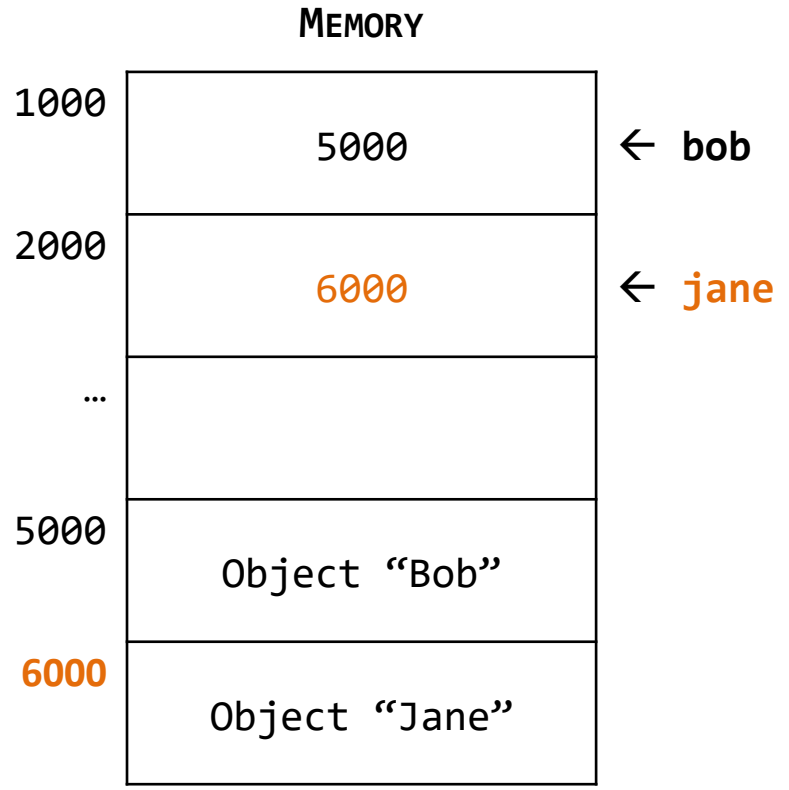
- Before:

```
Student bob = new Student("Bob");
Student jane = bob;
```

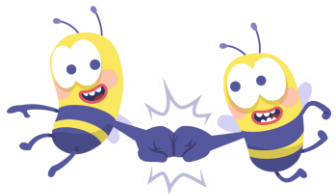


- Now:

```
Student bob = new Student("Bob");
Student jane = new Student("Jane");
```



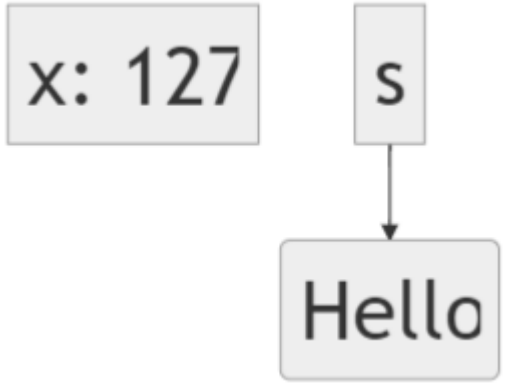
This now *breaks the alias* – breaks the old reference (`jane` doesn't point to 5000)
Uses new reference – `jane` pointing to own Student object



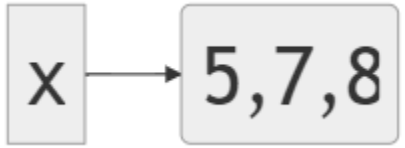
Examples

Reference Examples

```
/* This is a primative */
int x = 127;

/* This is a reference */
String s = new String ("Hello");
```




```
/* Arrays are Reference Types */

int[] arr = {4, 6, 2};
```



Using == vs .equals() on References

```
GregorianCalendar date1 = new GregorianCalendar(2018, 6, 14);
GregorianCalendar date2 = new GregorianCalendar(2018, 6, 14);
/* Is date1 “==” to date2? */
if(date1 == date2) {
        System.out.println("They are the same date!");
}
/* date1 and date2 are NOT == */
```

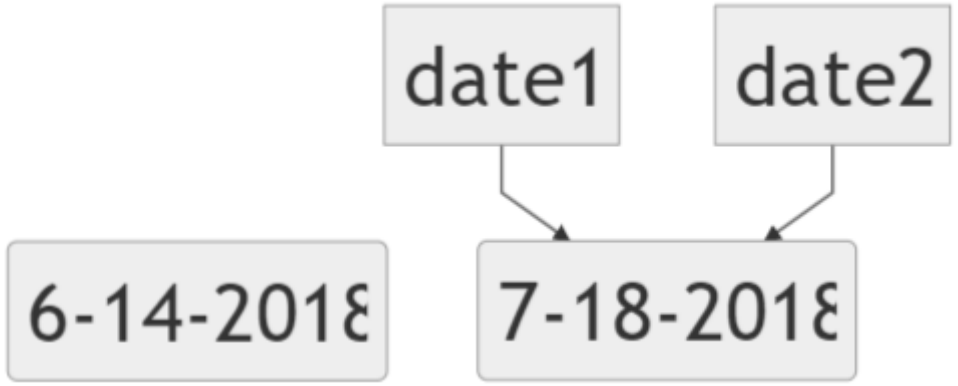

```
/* Use .equals() to compare references */
if(date1.equals(date2)) {
        System.out.println("They are the same!");
} /* It will print “They are the same!” */
```

Assignment of References

```
Date date1 = new Date(2018, 6, 14);

Date date2 = new Date(2018, 7, 18);

// Assignment

date1 = date2;


System.out.println(date1);
```

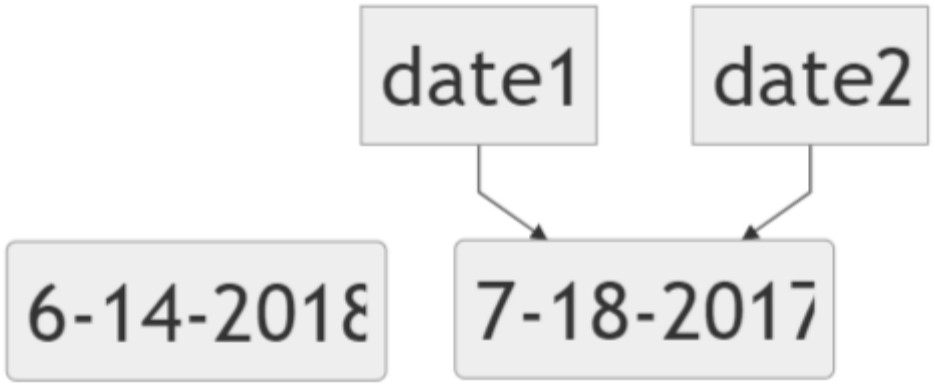
BEFORE THE ASSIGNMENT

AFTER THE ASSIGNMENT



Shared References

- Now if you executed something like: `date1.setYear(2017);`



If we print date1 or date2 the output will be exactly the same!

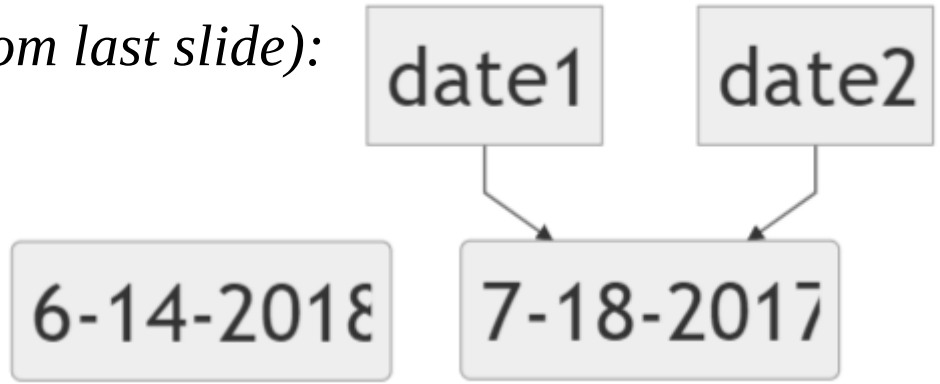
- Note that BOTH date1 and date2 are changed!
- This is because date1 and date2 are aliases to one another. They're both referred to the SAME memory location of the actual date.

Shared References

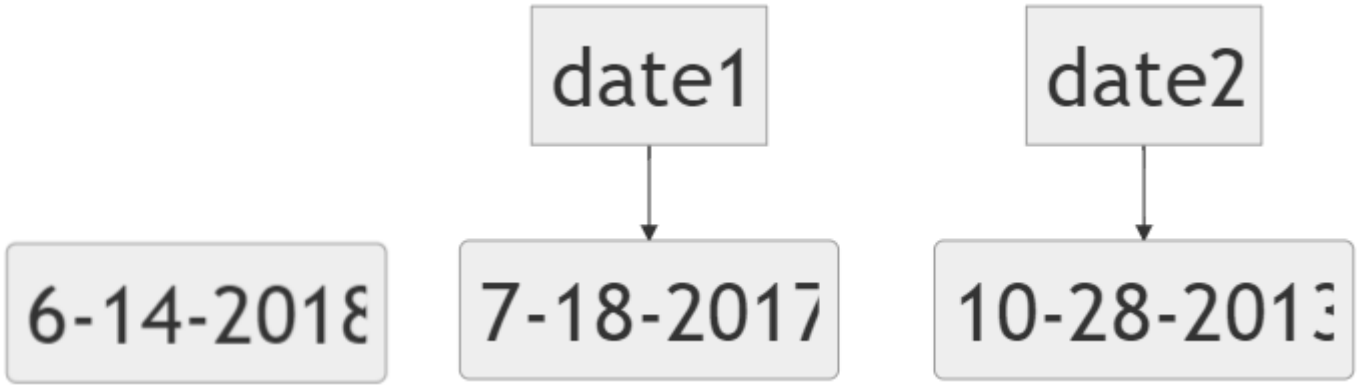
- Now if you **decouple** date1 and date2 by typing something like:

```
date2 = new Date(2013, 10, 28); // instantiating & initializing date2 (own object)
```

- We used to have *(from last slide):*



- Now we have:



EXTRA Slide // Clarification on Integer Division and Casting

- Casting and Integer Division
  - Dividing two integers will produce an **integer** ("*integer division*")
  - Example: `int m = 2 / 5;`
  - `value of m = 0`
    - Often this is **not** the value we want (or was expecting)!

  - To resolve this issue, we can **cast explicitly** (one or both operands):
  - Example: `System.out.println( (float) 2 / 5 );`
  - `value of result is 0.4`