# DATA TYPES REITERATION

# OUTLINE

1. PRIMITIVE DATA TYPES
2. OBJECT REFFERENCES
3. DESTROYING OBJECTS

# 1) PRIMITIVE DATA TYPES

| TYPE | CONTAINS | DEFAULT | SIZE | RANGE |
|---|---|---|---|---|
| **`boolean`** | true or false | false | 1 bit | NA |
| **`char`** | Unicode character | \u0000 | 16 bits | \u0000 to \uFFFF |
| **`byte`** | Signed integer | 0 | 8 bits | -128 to 127 |
| **`short`** | Signed integer | 0 | 16 bits | -32768 to 32767 |
| **`int`** | Signed integer | 0 | 32 bits | -2147483648 to 2147483647 |
| **`long`** | Signed integer | 0 | 64 bits | -9223372036854775808 to 9223372036854775807 |
| **`float`** | IEEE 754 floating point | 0.0 | 32 bits | ±1.4E-45 to ±3.4028235E+38 |
| **`double`** | IEEE 754 floating point | 0.0 | 64 bits | ±4.9E-324 to ±1.7976931348623157E+308 |

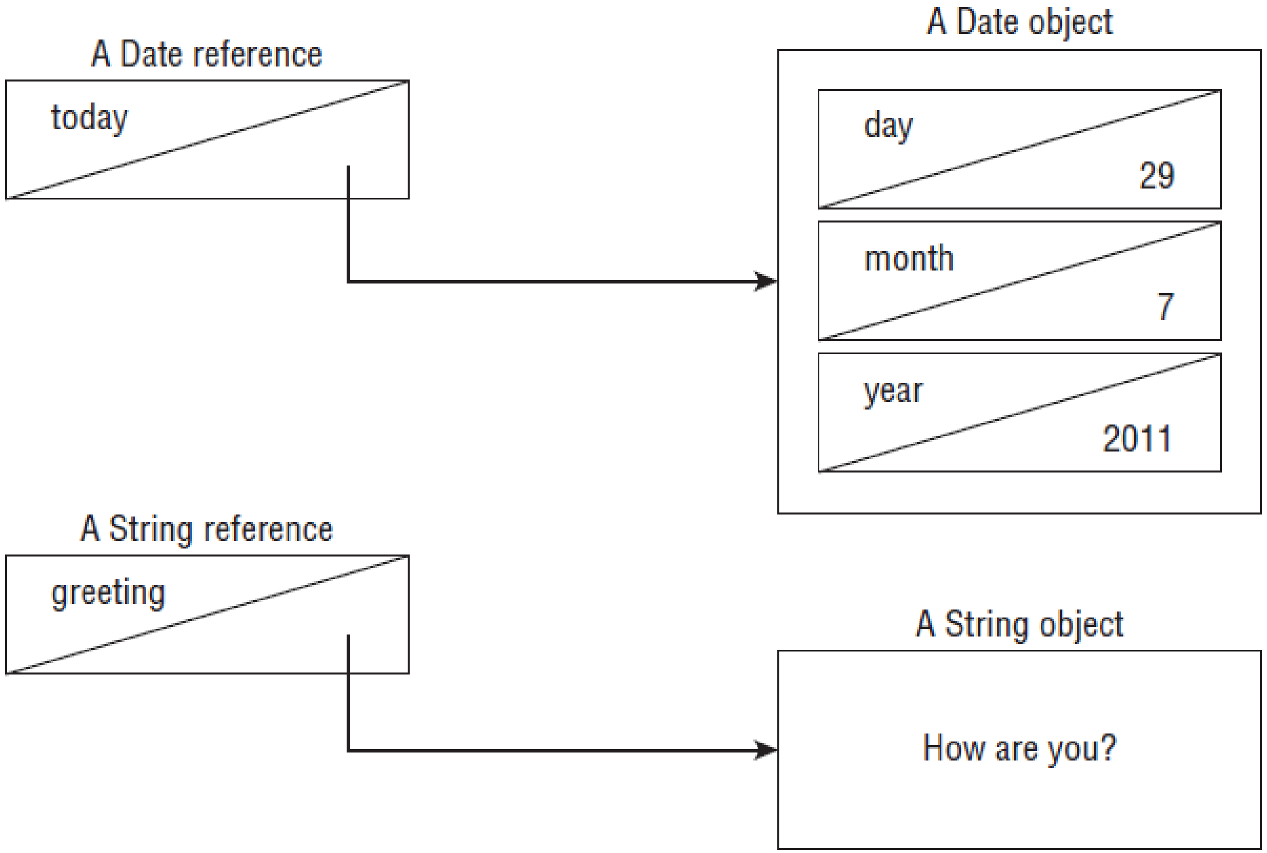
- **byte, short, int,** and **long** are used for numbers without decimal points.
- **`float`** and **`double`** are used for floating-point (decimal) values.
- A **`float`** requires the letter f following the number so Java knows it is a float.
- Each numeric type uses twice as many bits as the smaller similar type.
- For example: **`short`** uses twice as many bits as **`byte`** does.

# 2) OBJECT REFFERENCES

- A **reference** type refers to an object (an instance of a class).
- Primitive types that hold their values in the memory where the variable is allocated, references **do not hold** the value of the object they refer to.
- Instead, a reference “points” to an object by storing the memory address where the object is located, a concept referred to as a pointer.
- You can only use the reference to refer to the object.
- Suppose we declare a reference of type **`java.util.Date`** and a reference of type **`String`**:
  - `java.util.Date today;`
  - `String greeting;`
- The **`today`** variable is a reference of type Date and can only point to a Date object.
- The greeting variable is a reference that can only point to a String object.
- A value is assigned to a reference in one of two ways:
  - **A reference can be assigned to another object of the same type.**
  - **A reference can be assigned to a new object using the new keyword.**
- For example, the following statements assign these references to new objects:
  - `today = `**`new`**` java.util.Date();`
  - `greeting = "Hello!";`

# 2) OBJECT REFFERENCES

- The **`today`** reference now points to a new Date object in memory, and today can be used to access the various fields and methods of this Date object.
- Similarly, the **`greeting`** reference points to a new String object, “Hello!".
- The String and Date objects do not have names and can be accessed only via their corresponding reference.
- The figure shows how the reference types appear in memory.



# 3) DESTROYING OBJECTS

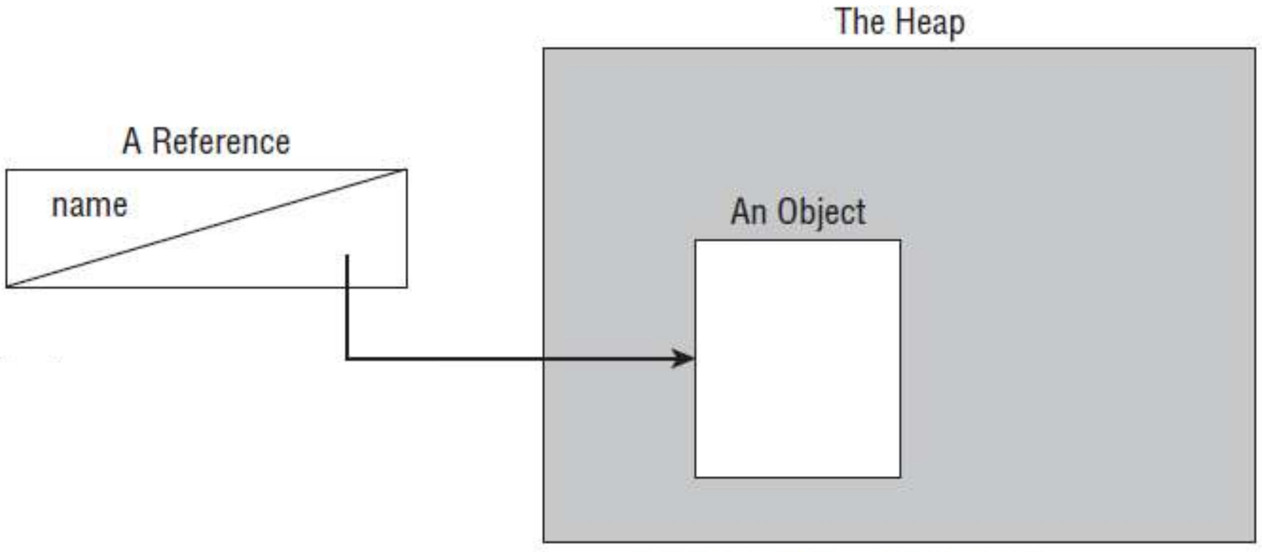
- Now that we’ve played with our objects, it is time to put them away.
- Luckily, Java automatically takes care of that for us.
- Java provides a **garbage collector** to automatically look for objects that aren’t needed anymore.
- All Java objects are stored in our program memory’s heap.
- The heap, which is also referred to as the free store, represents a large pool of unused memory allocated to our Java application.
- The heap may be quite large, depending on your environment, but there is always a limit to its size.
- If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory.
- In the following sections, we’ll look at garbage collection and the finalize() method.

# 3) DESTROYING OBJECTS
## 3.1) GARBAGE COLLECTION

- Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program.
- Java provides a method called **`System.gc().`**
- Now you might think from the name that this tells Java to run garbage collection. Nope!
- It meekly suggests that now might be a good time for Java to kick off a garbage collection run.
- Java is free to ignore the request.
- The more interesting part of garbage collection is when the memory belonging to an object can be reclaimed.
- Java waits patiently until the code no longer needs that memory.
- An object will remain on the heap until it is no longer reachable.
- An object is no longer reachable when one of two situations occurs:
  - The object no longer has any references pointing to it.
  - All references to the object have gone out of scope.

# 3) DESTROYING OBJECTS
## 3.1) GARBAGE COLLECTION

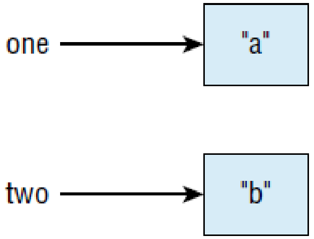
- Realizing the difference between a reference and an object goes a long way toward understanding garbage collection, the new operator, and many other facets of the Java language.



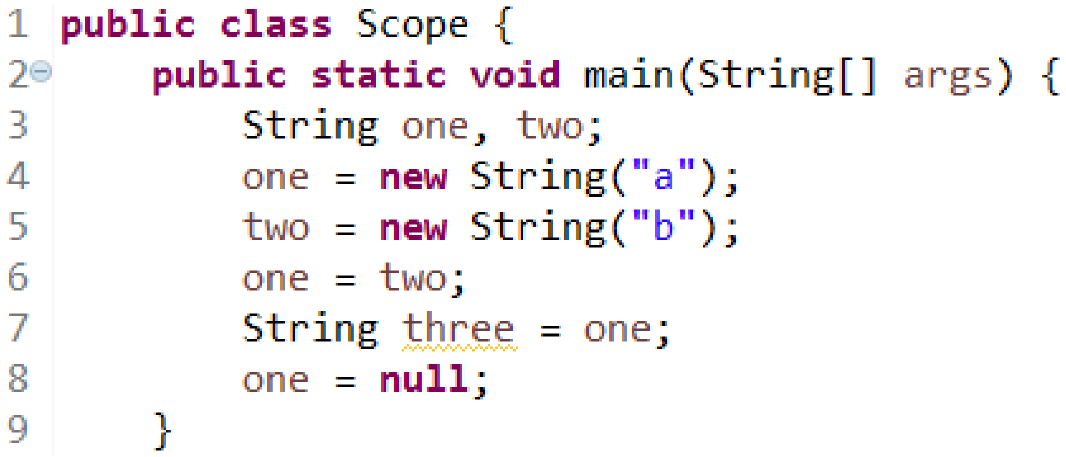
# 3) DESTROYING OBJECTS

## 3.1) GARBAGE COLLECTION

- Let’s look at this code and see if we can figure out when each object first becomes eligible for garbage collection.
- After line 5:

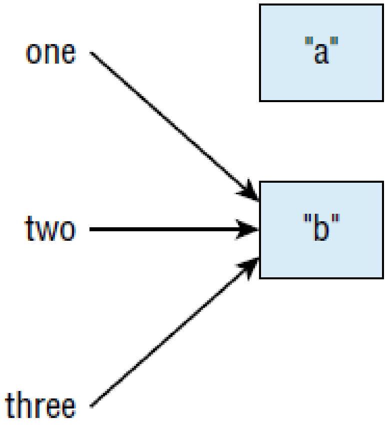


- On line 6, the variable one changes to point to "b".
- On line 7, we have a new variable, three, pointing to "b".



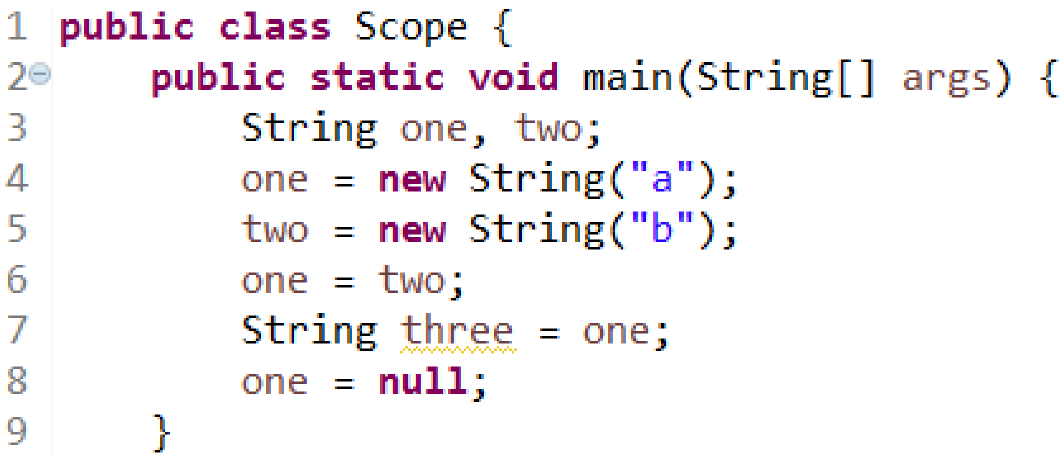
```
1 public class Scope {
2     public static void main(String[] args) {
3         String one, two;
4         one = new String("a");
5         two = new String("b");
6         one = two;
7         String three = one;
8         one = null;
9     }
```

- After line 7:



# 3) DESTROYING OBJECTS
## 3.1) GARBAGE COLLECTION

- Finally, we can cross out the line between one and "b" since line 8 sets this variable to null.
- Now, we were trying to find out when the objects were first eligible for garbage collection.
- On line 6, we got rid of the only arrow pointing to "a", making that object eligible for garbage collection.
- "b" has arrows pointing to it until it goes out of scope.
- This means "b“ doesn’t go out of scope until the end of the method on line 9.



```
1 public class Scope {
2     public static void main(String[] args) {
3         String one, two;
4         one = new String("a");
5         two = new String("b");
6         one = two;
7         String three = one;
8         one = null;
9     }
```

# 3) DESTROYING OBJECTS
## 3.2) FINALIZE

- Java allows objects to implement a method called finalize() that might get called.
- This method gets called if the garbage collector tries to collect the object.
- If the garbage collector doesn’t run, the method doesn’t get called.
- If the garbage collector fails to collect the object and tries to run it again later, the method doesn’t get called a second time.
- In practice, this means you are highly unlikely to use it in real projects.
- **Just keep in mind that it might not get called and that it definitely won’t be called twice.**