Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

- Chapter 6
- Static variables and methods
- Primitive Vs Reference Variables
- Stack Vs Heap Memory
- Memory Allocation: Primitive Types
- Memory Allocation: Reference Types

Today's Lecture

Static

- Both methods and variables can be declared ***static***.
- A static variable is shared by every instance of the class.
- If you make a change from one instance you will "see" that change in another instance.
- Use a static variable if you *don't* need a different version of that variable for EVERY instance of the class.

- INSTANCE VARIABLES:
- Suppose a student class:

```
public class Student {
   public int id;
   public int rank;

   public Student(int newId, int newRank) {
         id = newId;
         rank = newRank;
   }
}
```

Static

- INSTANCE VARIABLES:
- 3 Instances of student. Each has its own set of variables.

```
Student s1, s2, s3;

s1 = new Student(12, 3);
s2 = new Student(10, 100);
s3 = new Student(7, 70);
```

s1:Student	s2:Student	s3:Student
12(id)	10 (id)	7 (id)
3 (rank)	100 (rank)	70 (rank)

Static

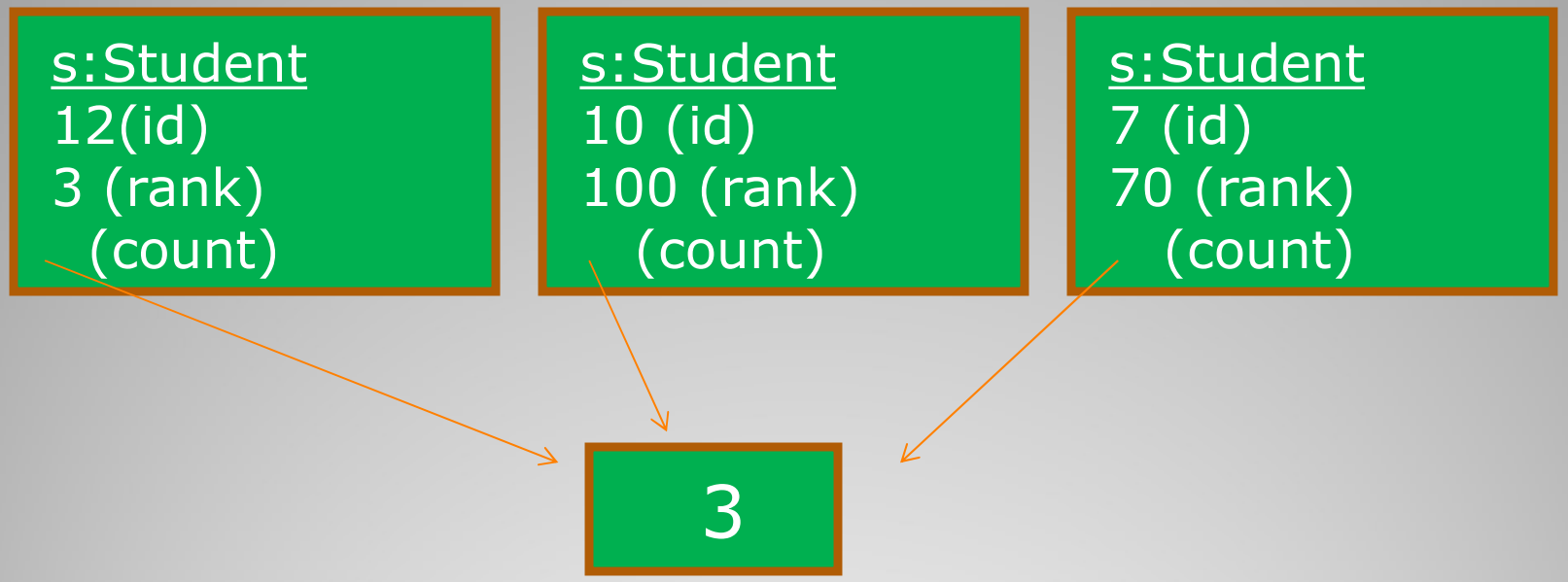
- Now suppose we want to store a count of the total number of students.
- The number of students is not specific to any instance so it should be defined as static.
- For example…

Static

```
public class Student {
   int id;
   int rank;
   static int count;

   public Student(int newId, int newRank) {
         id = newId;
         rank = newRank;
   }
}
```

Use the static keyword to declare a variable as static.

Static



- A static variable is shared by all instances.

Static

Static

- You can access a static variable even if you do not declare an instance of the class.
- Use the class name followed by a dot and then the variable name.

For example:

```
Student.count = 3;
```

This sets the count variable to 3.

- Methods can also be declared as static.
- Static methods can only access static variables.
- Static methods CANNOT access instance variables.

Static

Static

- Why is main() declared static?
- It is static because when the program starts there are no objects of the class.
- If main() was not static then we would not be able to use it at the start of the program.
- Regular instance methods can only be used from an instance of a class.

- Now we will examine what is going on behind the scenes when a variable is declared.
- How does memory get allocated?
- Where do variables get stored?
- **Primitive** Vs. **Reference** Types *revisited*.

Memory

Two types of Memory

Stack

All local variables and parameters

Heap

Member variables of reference types

Memory

- When you declare a **primitive** type variable the data gets stored in the variable itself (does not store an address).

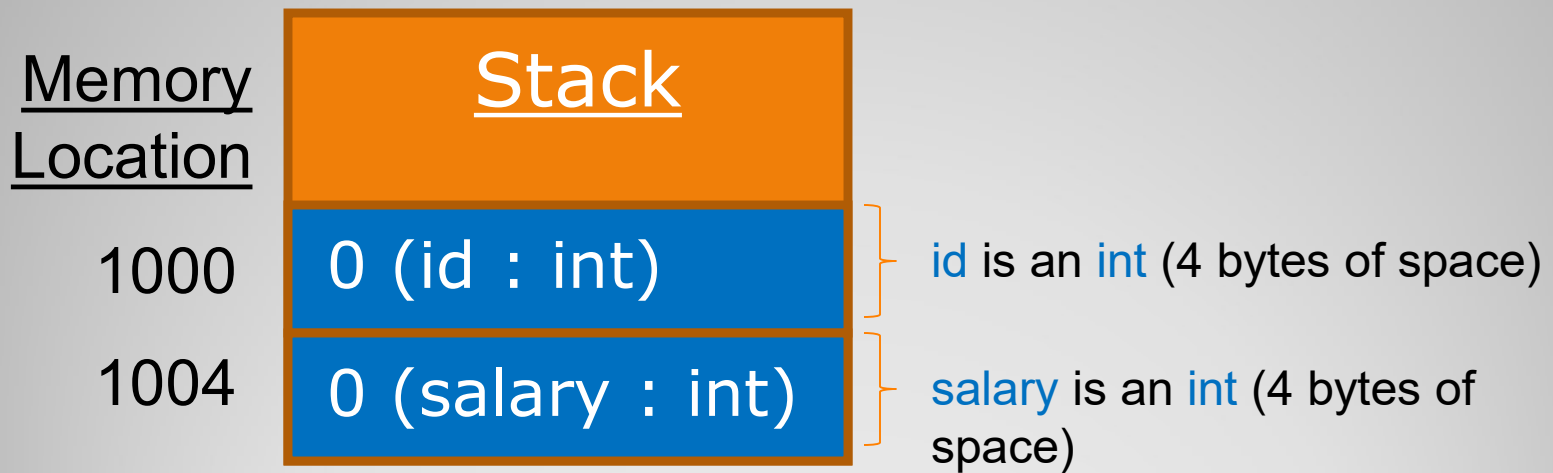
Primitive Types:

1. int	2. short	3. long
4. float	5. double	6. byte
7. char	8. boolean	

- int, short, long, float, double, byte, char are initialized to 0.
- boolean is initialized to false.

Memory

- What happens when primitive variables are declared in a method?
- For example: int id, salary;



Memory

- What is a **reference** type?
- Something that is not a **primitive** type.
- Types defined using the keyword "class" are **reference** types.
- Predefined classes stored in the Java Standard Library. For example: String, Scanner etc.

Memory

- Now assume that we declared the following class:

```
public class Student
{
      private int id;      // Primitive Instance Within Class
      private int rank;  // Primitive Instance Within Class

    // Assume the proper Get, Set, and Constructors
    // are declared
}
```

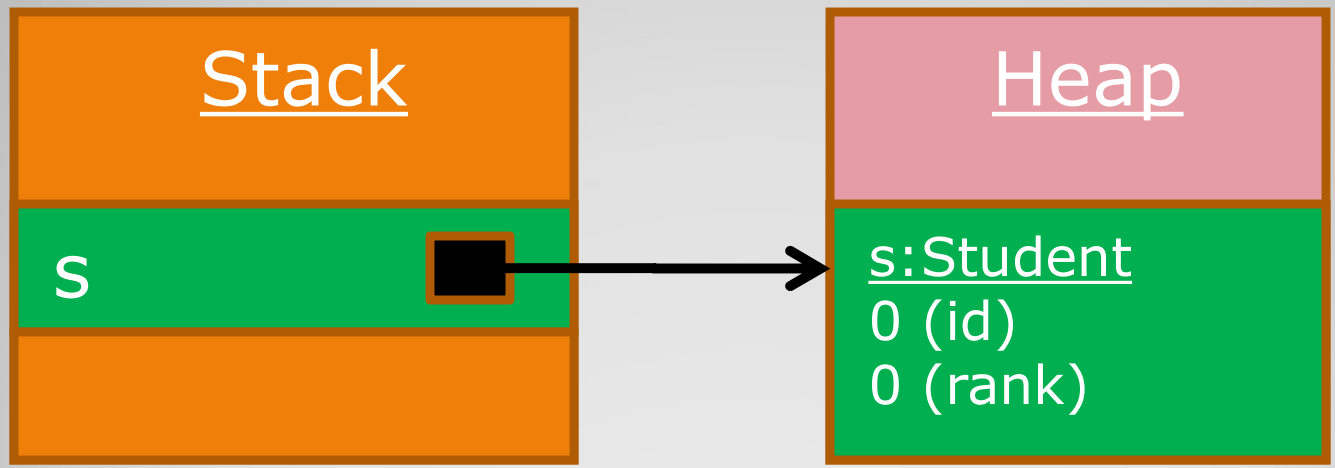
- *There are two member variables in Student.*

Memory

- How do **reference type** variables get stored?
- **Reference type** variables "refer" to a location.
- **The variable stores the address where the member variables are located on the heap**
- ***How is Student stored in memory?***

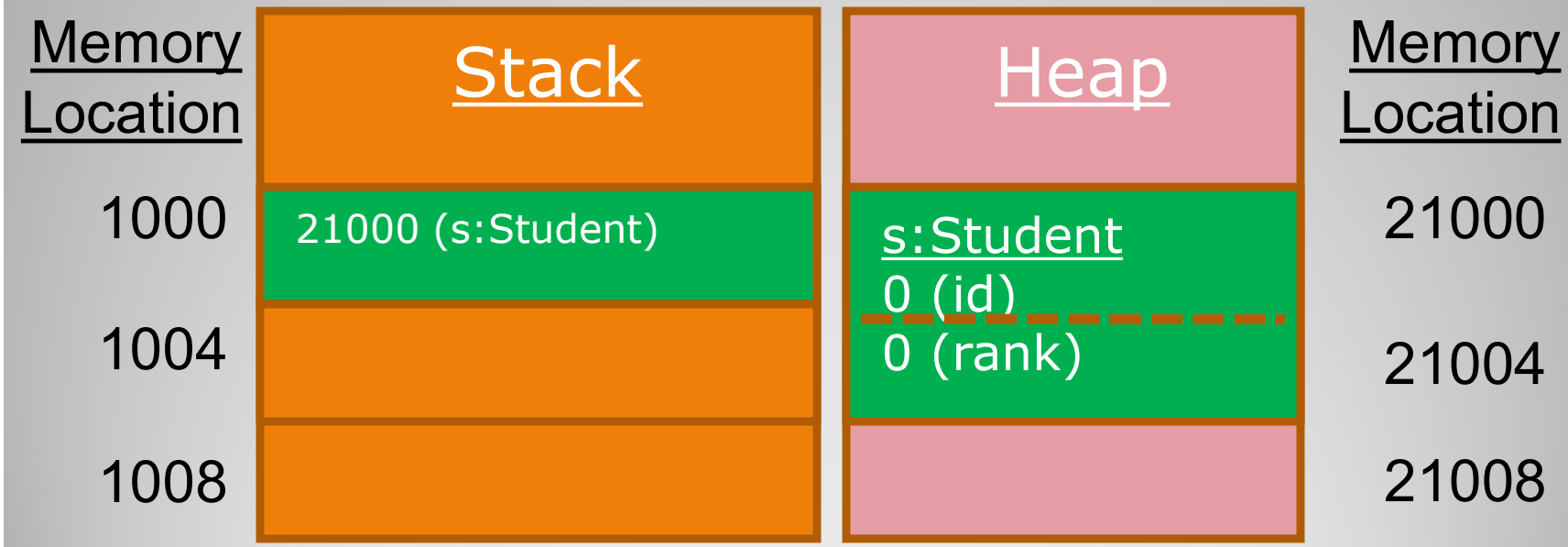
Memory

- Any class is a reference type. Now declare a variable of type Student.
- Declare an instance of Student in a method:
  Student s = new Student();



Memory

- The number 21000 is a location in memory (an address).
- 21000 "refers" to the location in memory where the s variable data is located.

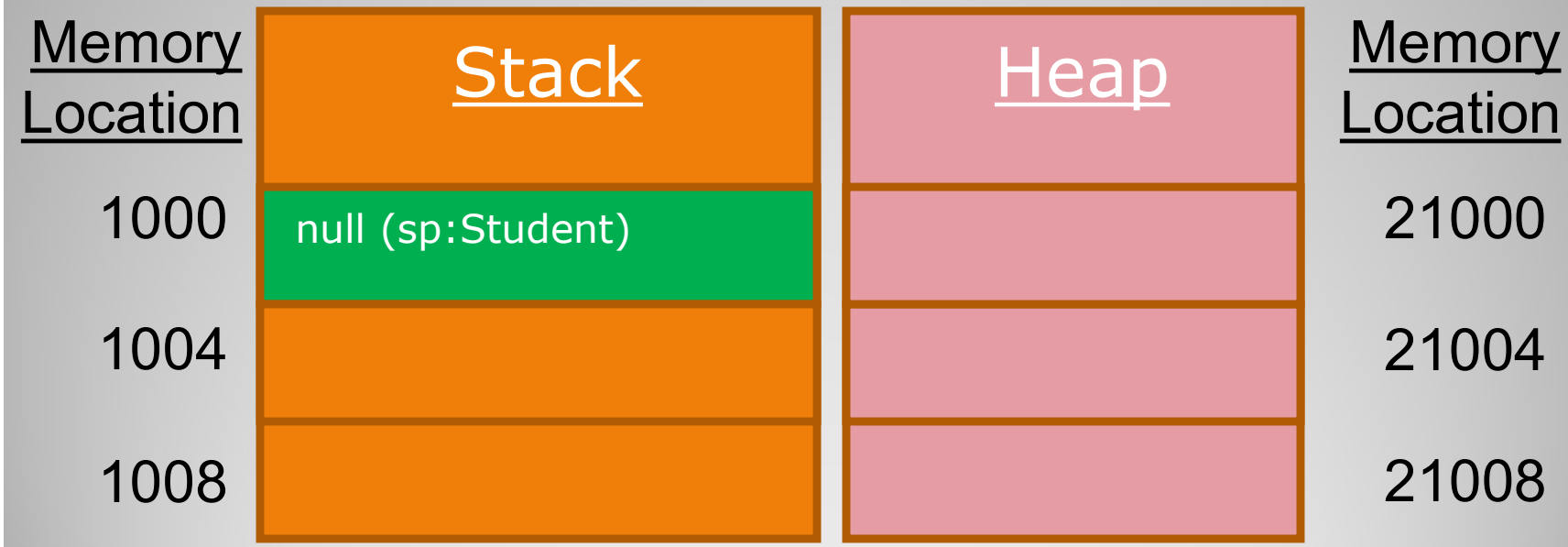


Memory

- Declare a variable in a method but do not call new.

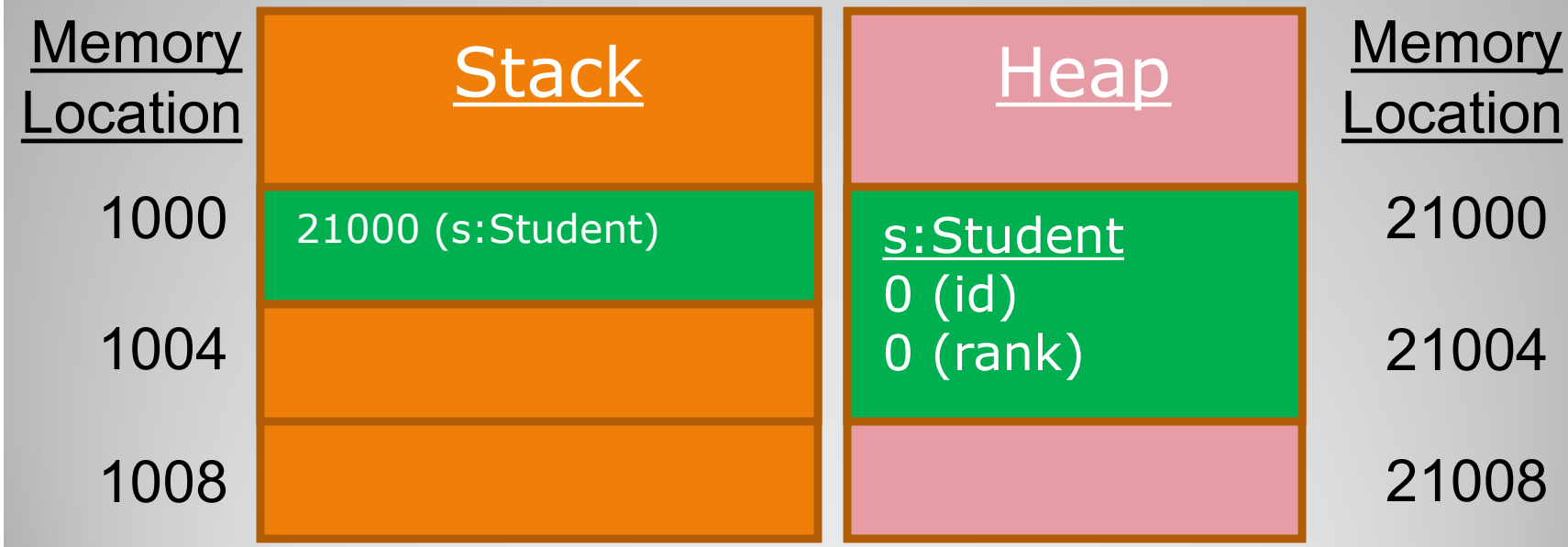
Student s; // New is NOT called!

- The heap piece is **NOT** allocated until new is called!!!



Memory

- The heap piece is allocated the moment that new is called!
- Student s;
  s = new Student(); // Call new



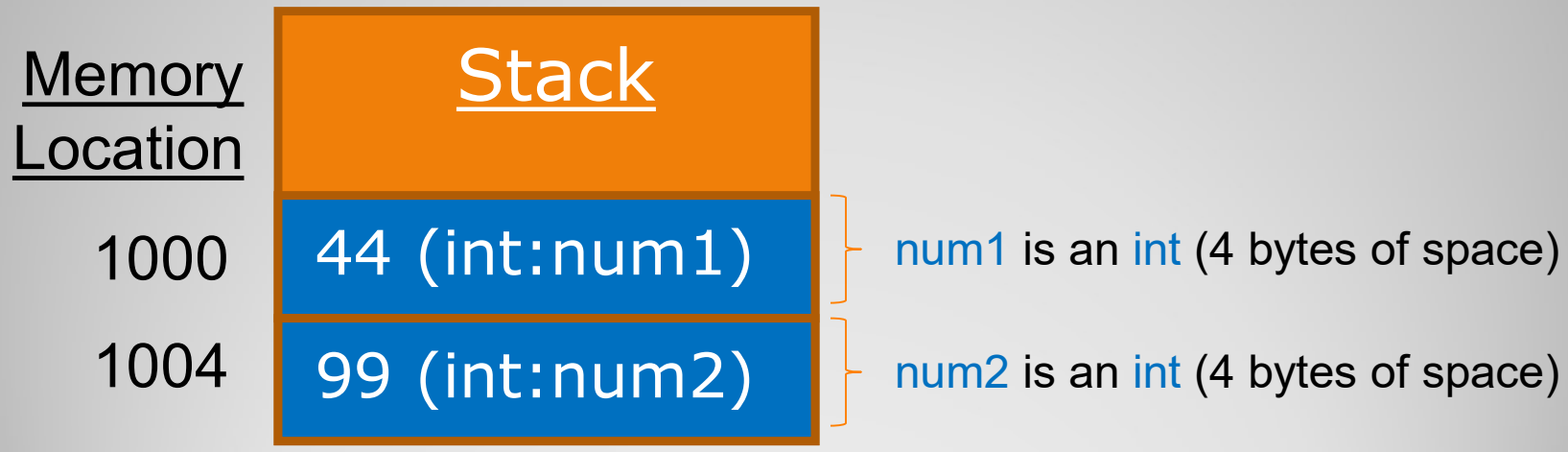
Memory

- new allocates memory on the heap.
- If new is not called, then the value of the variable is null.
- **If a reference variable has the value null it cannot be used!**

Memory

- Now look at primitive values again.
- ***What does memory look like after declaring primitive variables in a method?***

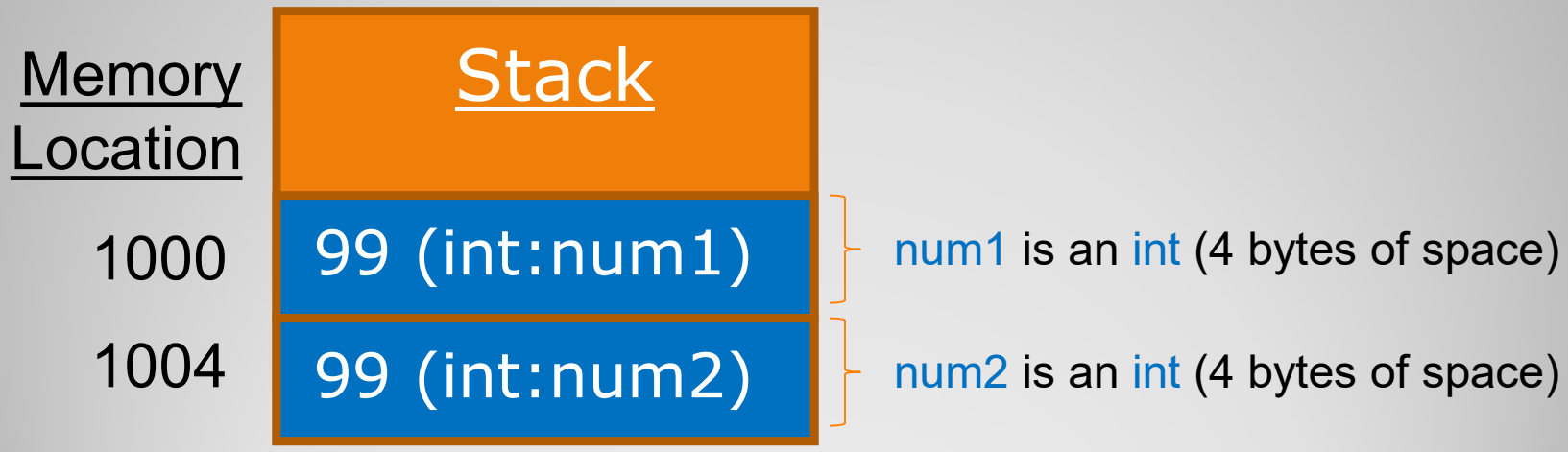
```
int num1=44, num2=99;
num1 = num2;   // Assignment
```



Memory

- The variable num1 is assigned the value stored in num2.
- 99 is copied into num1.

```
int num1=44, num2=99;
num1 = num2;   // Assignment
```



Memory

- When you assign one variable to another you copy whatever value is inside it and put it into the other variable.

- 99 is copied into num1 from num2.

```
int num1=44, num2=99;
num1 = num2;   // Assignment
```

99

Memory

- Now declare two Student type variables in a method.

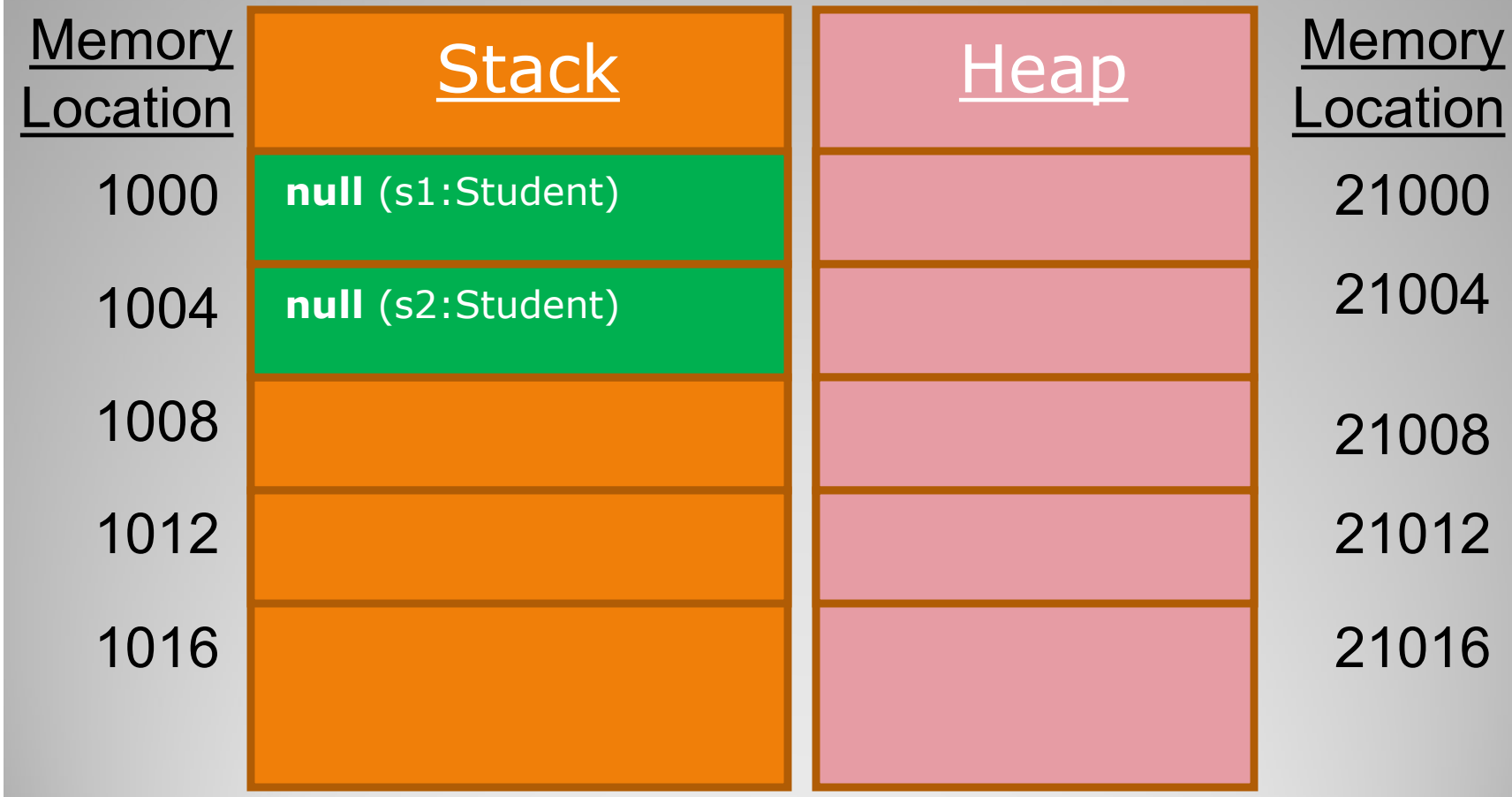
- For example:

```
Student s1;
Student s2;
```

- **What does memory look like?**

Memory

- **new was NOT called so no memory on heap.**



Memory

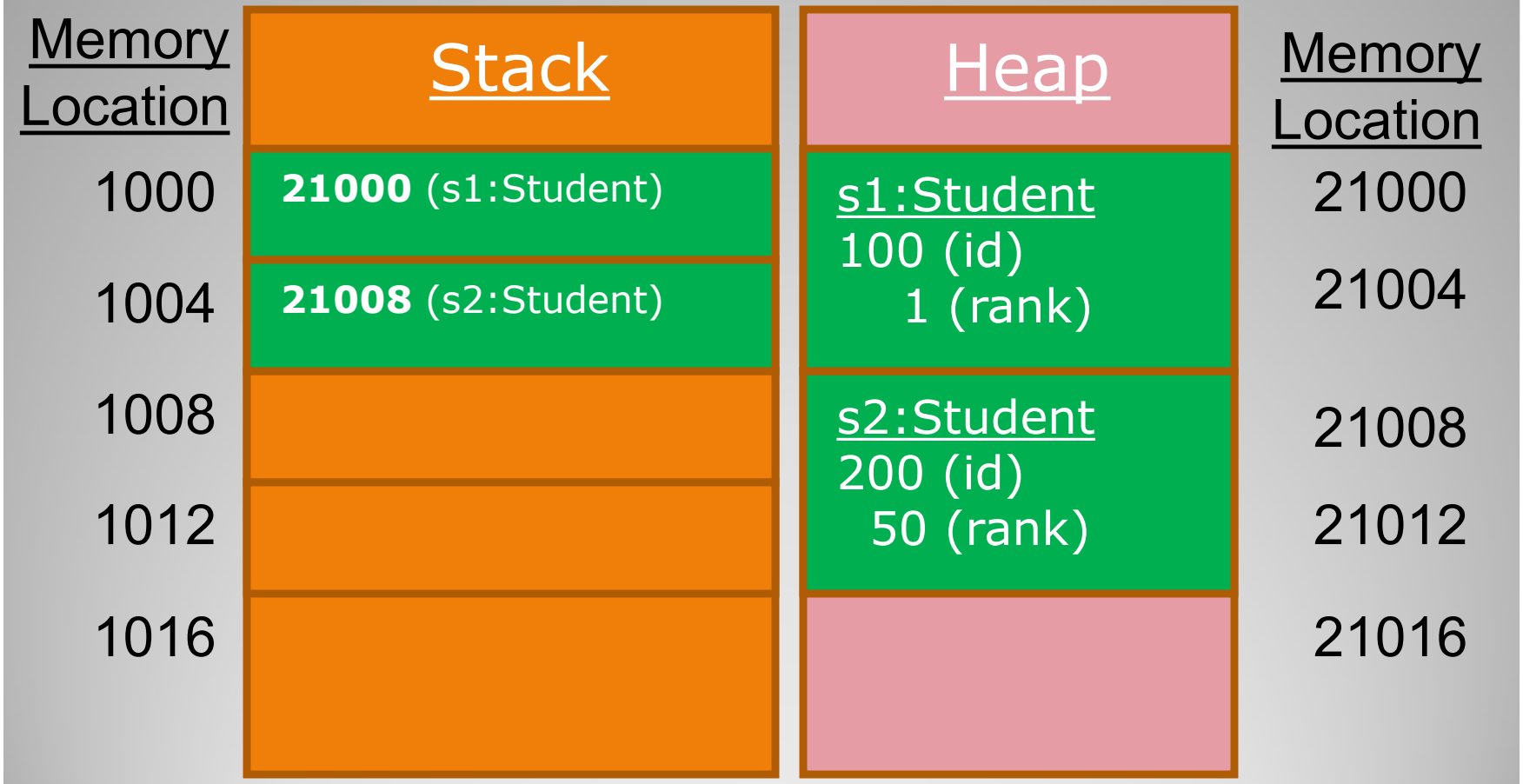
- Now declare two Student type variables.
- ***This time new is called for each.***
- For example:

```
Student s1 = new Student(100, 1);
Student s2 = new Student(200, 50);
```

- **What does memory look like?**

Memory

- **s1 and s2 have different addresses.**



Memory

- What happens when you assign one reference to another?

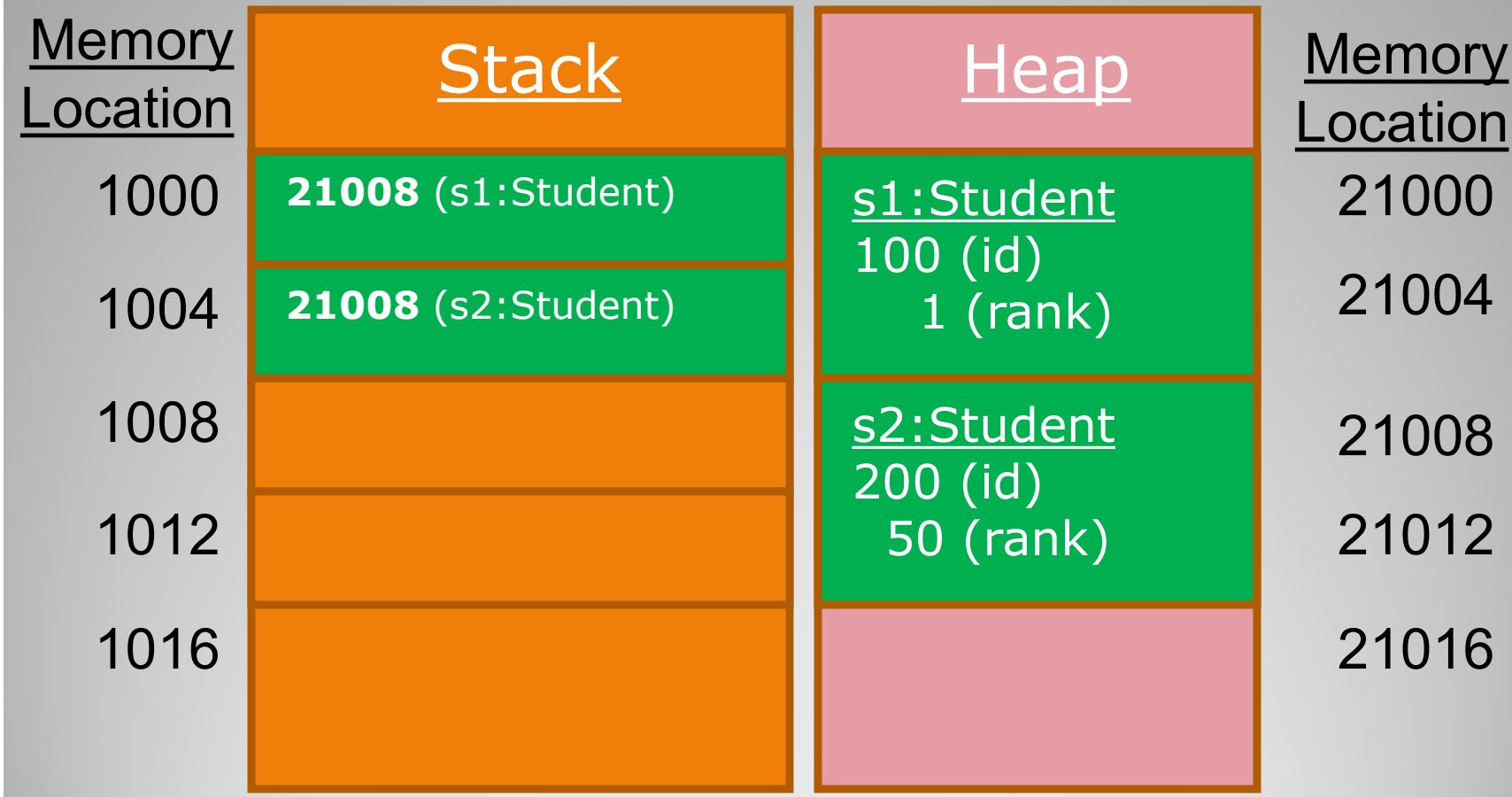
- For example:

```
Student s1 = new Student(100, 1);
Student s2 = new Student(200, 50);

s1 = s2;  // Assignment
```

- ***What does memory look like?***

Memory

- **s1 has the same ADDRESS as s2**



Memory

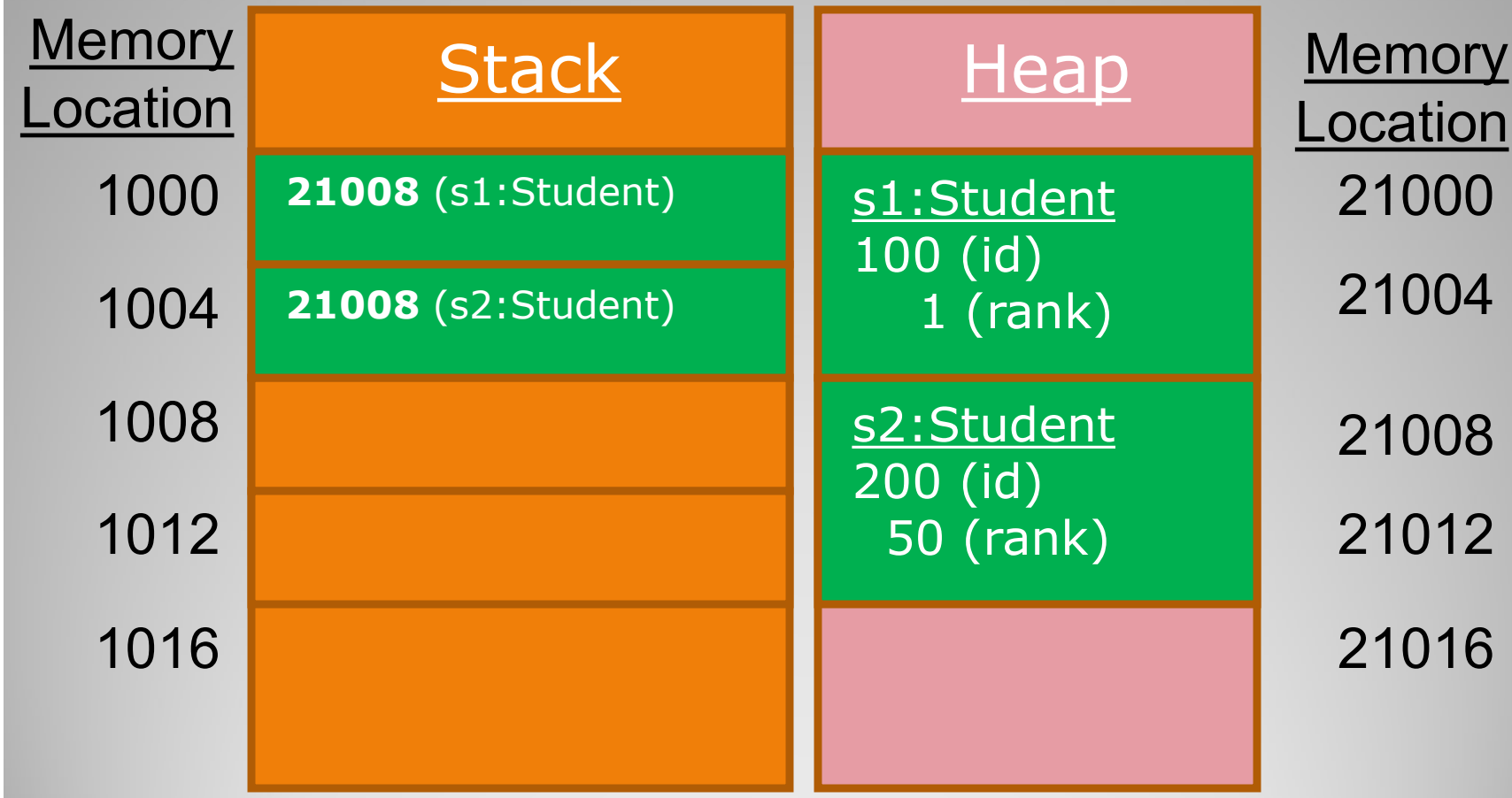
- Code:

```
Student s1 = new Student(100, 1);
Student s2 = new Student(200, 50);
s1 = s2;  // Assignment
```

- s1 and s2 now point to the SAME memory location in the heap.
- Any change you make to either one will effect the other.

Memory

- *Memory location 21000 is now unreachable!!!*



Memory

- Unreachable memory locations are a waste of space and must be given back to the system.
- Any memory locations on the heap that are not “referenced” will be given back to the system.
- This is called **"garbage collection"**.

Memory

- Do in-class problem for ch 6 p1.

In-Class Problem

- Define a class that contains another class.

```
public class School {
        int dist;
        Student s1; // Previously defined
        Student s2; // Previously defined

        public School(int newDist, int id1, rank1, id2, rank2)
        {
                dist = newDist;
                s1 = new Student(id1, rank1);
                s2 = new Student(id2, rank2);
        }
        // Assume Get/Set and main defined
};
```

Memory

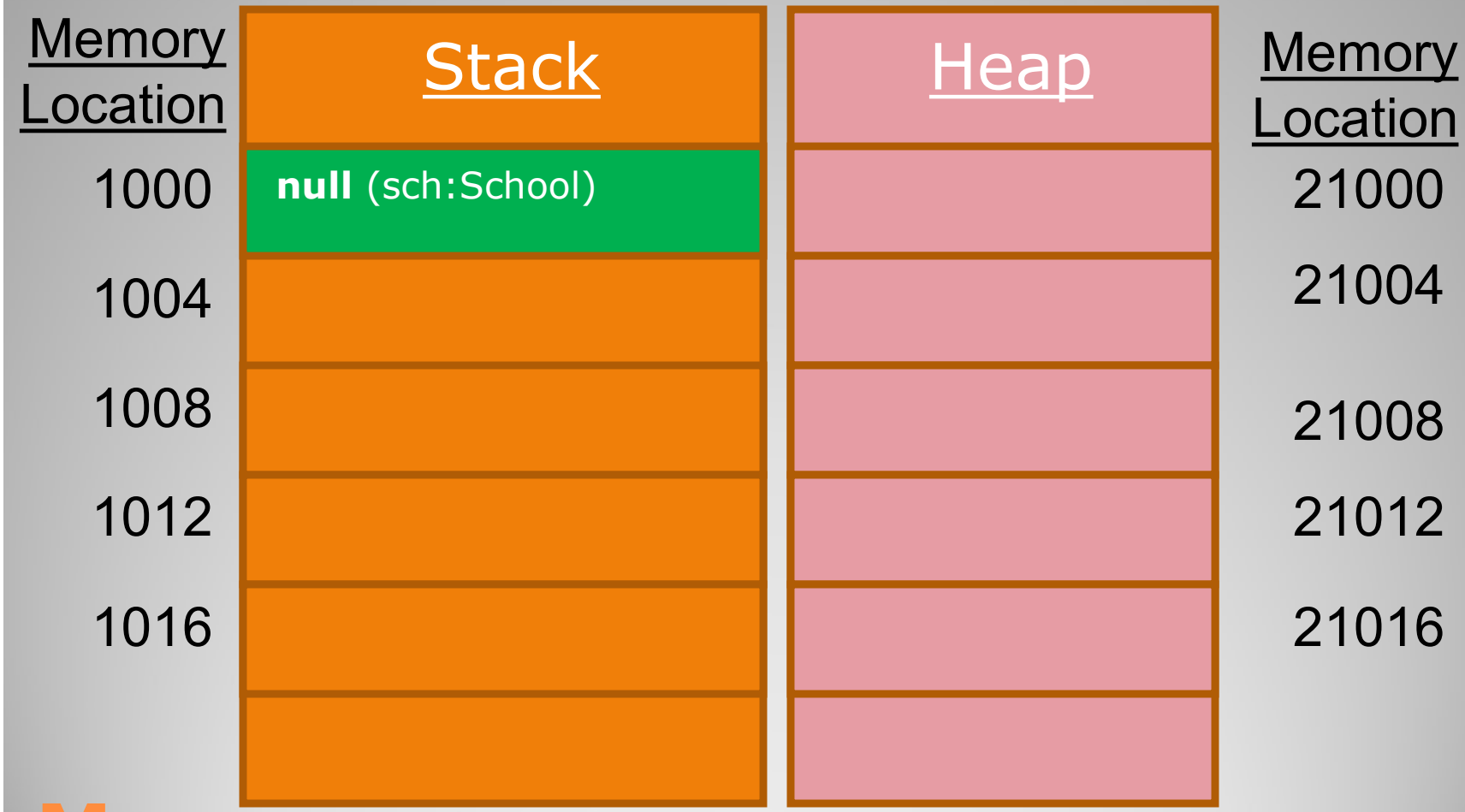
- Create an instance of the School type but do not call new:

```
public static void main(String[] args)
{
    School sch;
}
```

- ***What does the variable sch look like in memory?***

Memory

- *No memory allocated on the heap!*



Memory

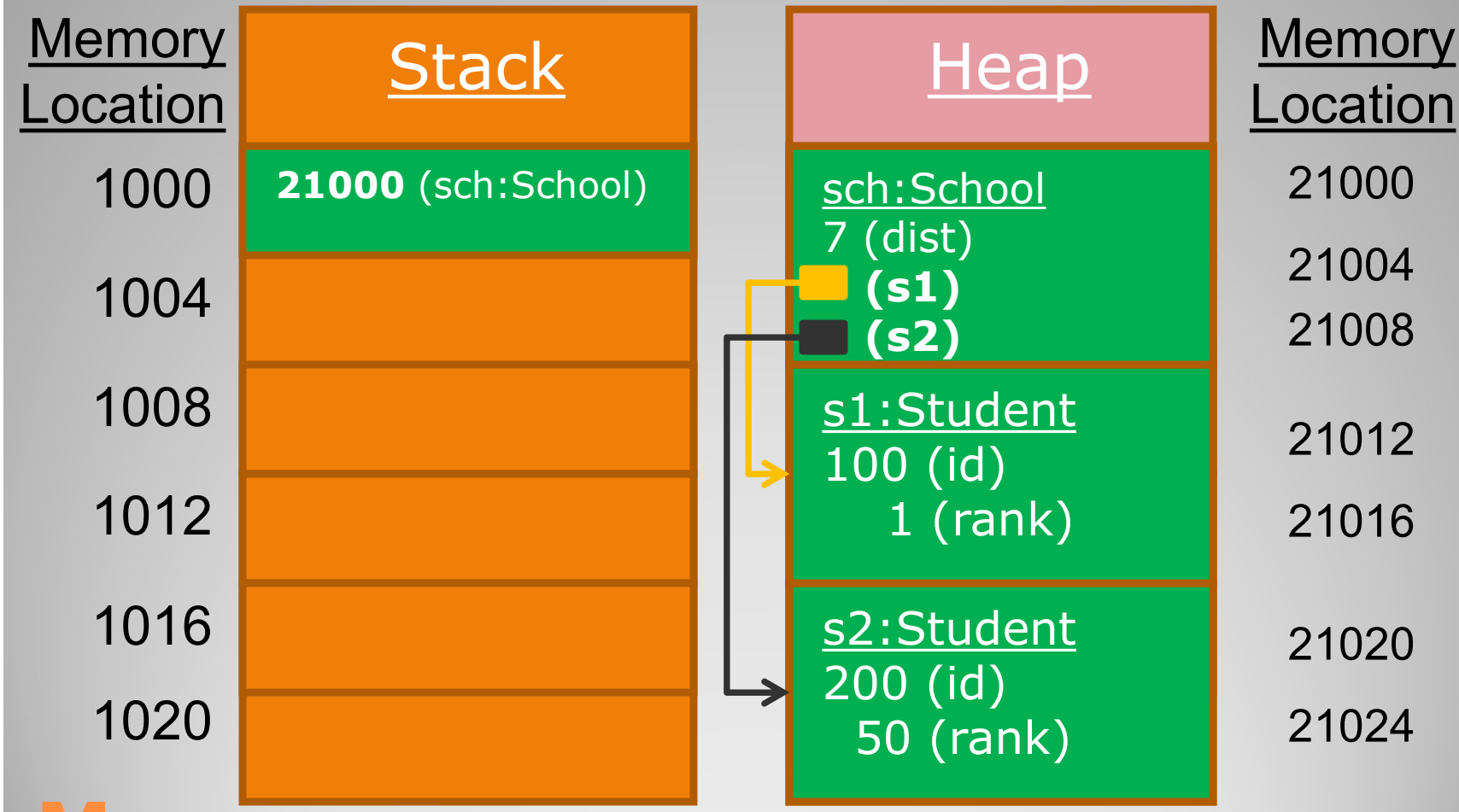
- Create an instance of the school class and call new on it:

```
public static void main(String[] args)
{
    School sch;
    sch = new School(7, 100, 1, 200, 50);

}
```

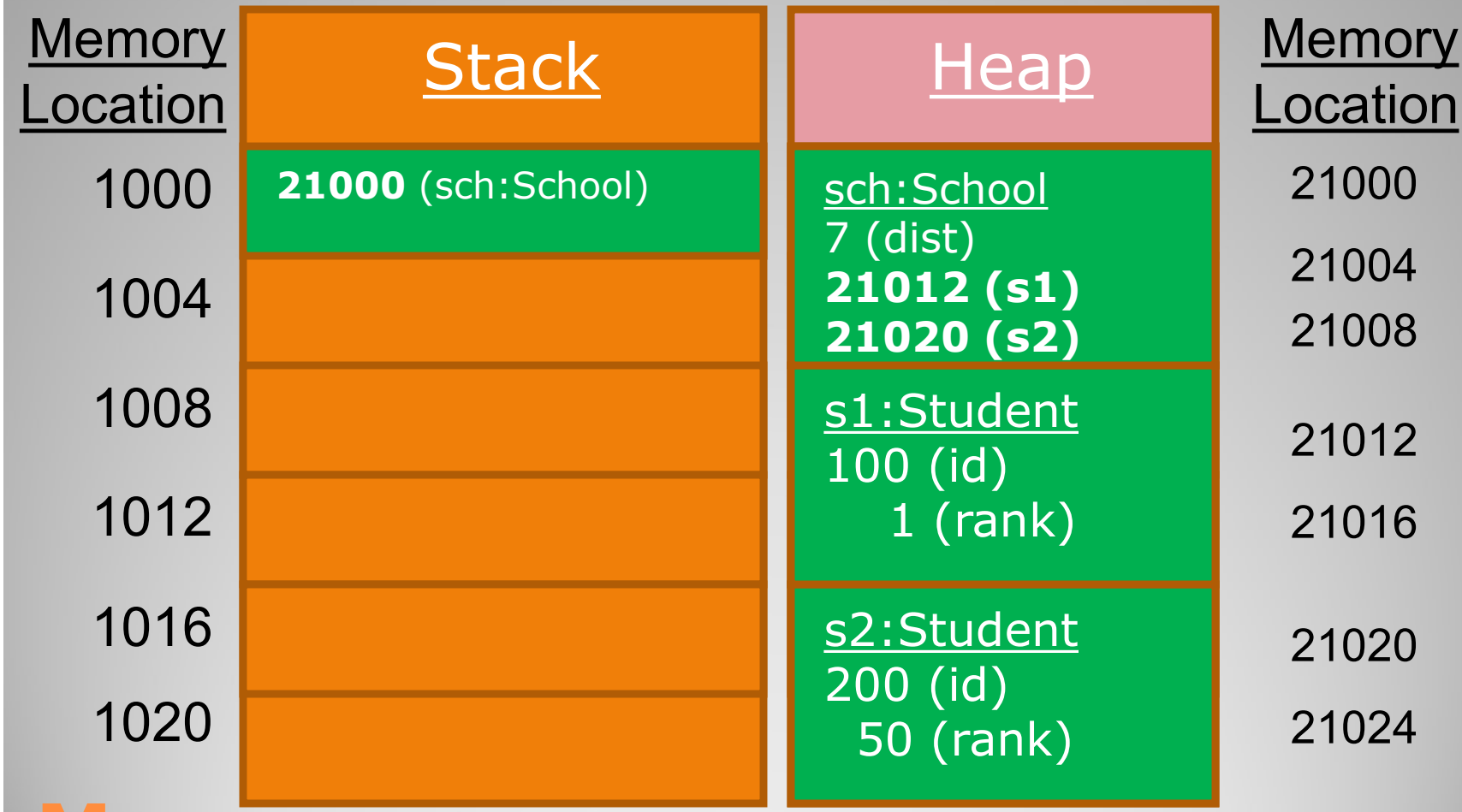
- ***What does the variable sch look like in memory?***

Memory

- *School AND two Student instances on heap.*



- *s1 and s2 members of School are references!*



Memory

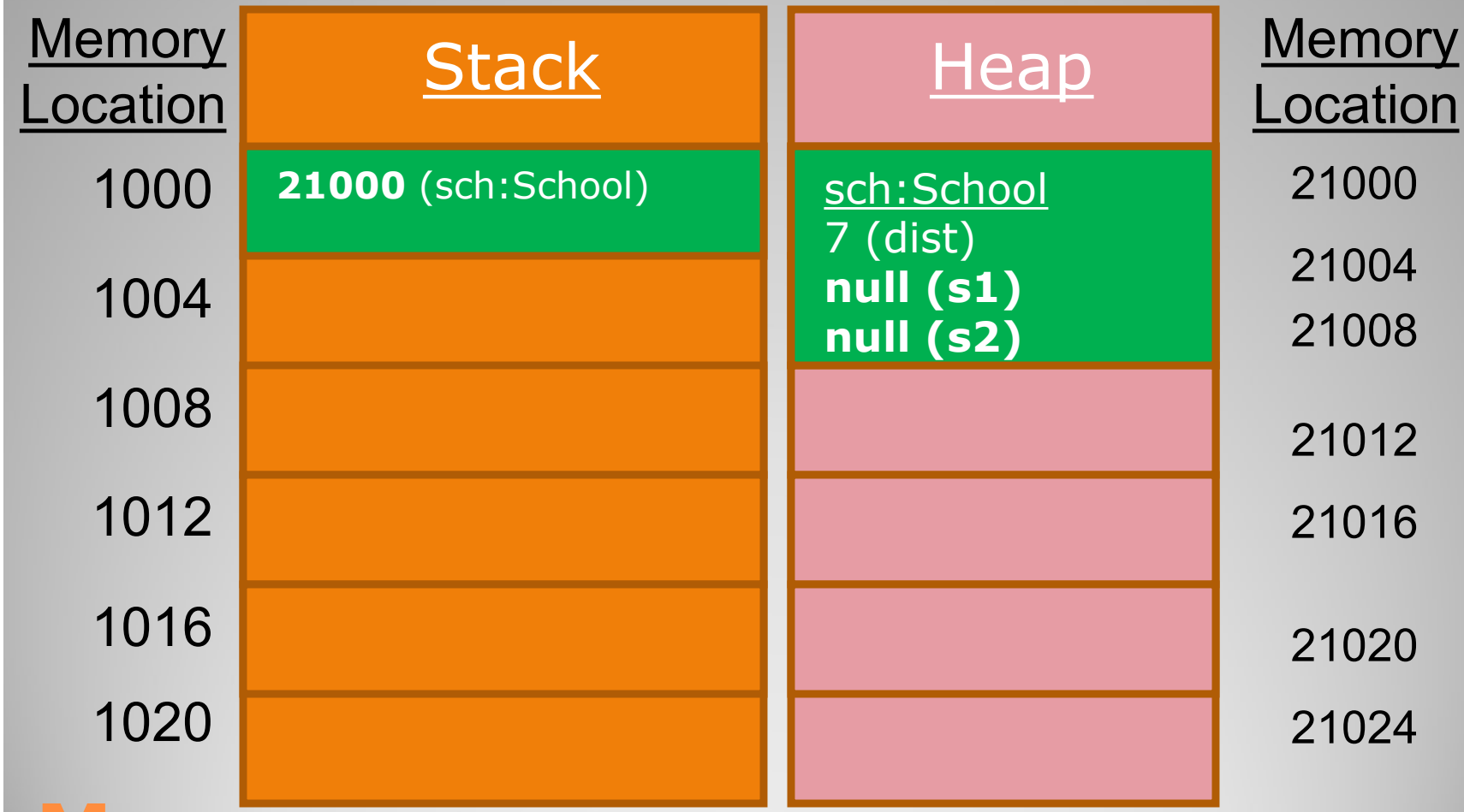
- What if we did **NOT** call new for each Student inside the School constructor?

- For example:

```
public School(int newDist, int id1, rank1, id2, rank2)
{
        dist = newDist;

        //s1 = new Student(id1, rank1);  Don't run this line
        //s2 = new Student(id2, rank2);  Don't run this line
}
```

Memory

- ***No place to store Student data!***



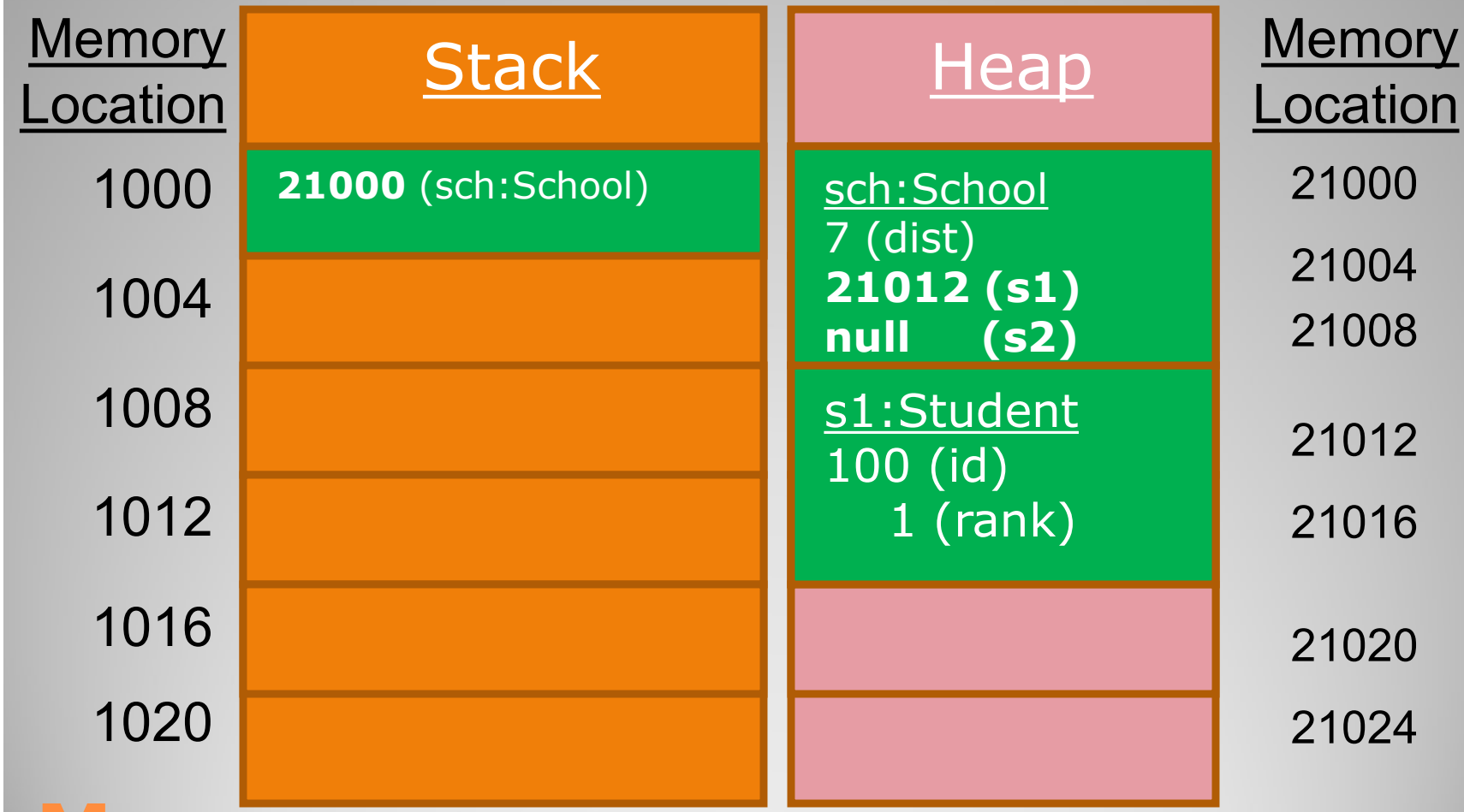
Memory

- What if we called new on s1 but NOT s2?
- For example:

```
public School(int newDist, int id1, rank1, id2, rank2)
{
        dist = newDist;

        s1 = new Student(id1, rank1);

        //s2 = new Student(id2, rank2);  Don't run this line
}
```

Memory

- *s1 is usable but s2 is not.*



Memory

- **Primitive** types:
- int, short, long, float, double, byte, char, and boolean.

- Is **String** a primitive or a reference type?

- *Are the following declarations legal?*

```
int num = 44;
String name = "Arthur";
```

Strings

Strings

- String is a reference type!
- If String is a reference type, then ***why don't you have to call new to use it?***
- *For example (this is legal Java code):*

```
String name = "Arthur";
```

Strings

- String is a **special** reference type!
- Call to new is NOT required.

- Strings can be stored in two different areas of the heap:
  - **String Constant Pool**
  - **Normal Heap Memory**

- Where the string is stored depends on how it is initialized.

- **Store in String Constant Pool**:

```
String name = "Arthur";
```

- **Store in Normal Heap Memory:**

```
String name = new String("Arthur");
```

String Constant Pool

- **String Constant Pool**:

```
String s1 = "Arthur";
```

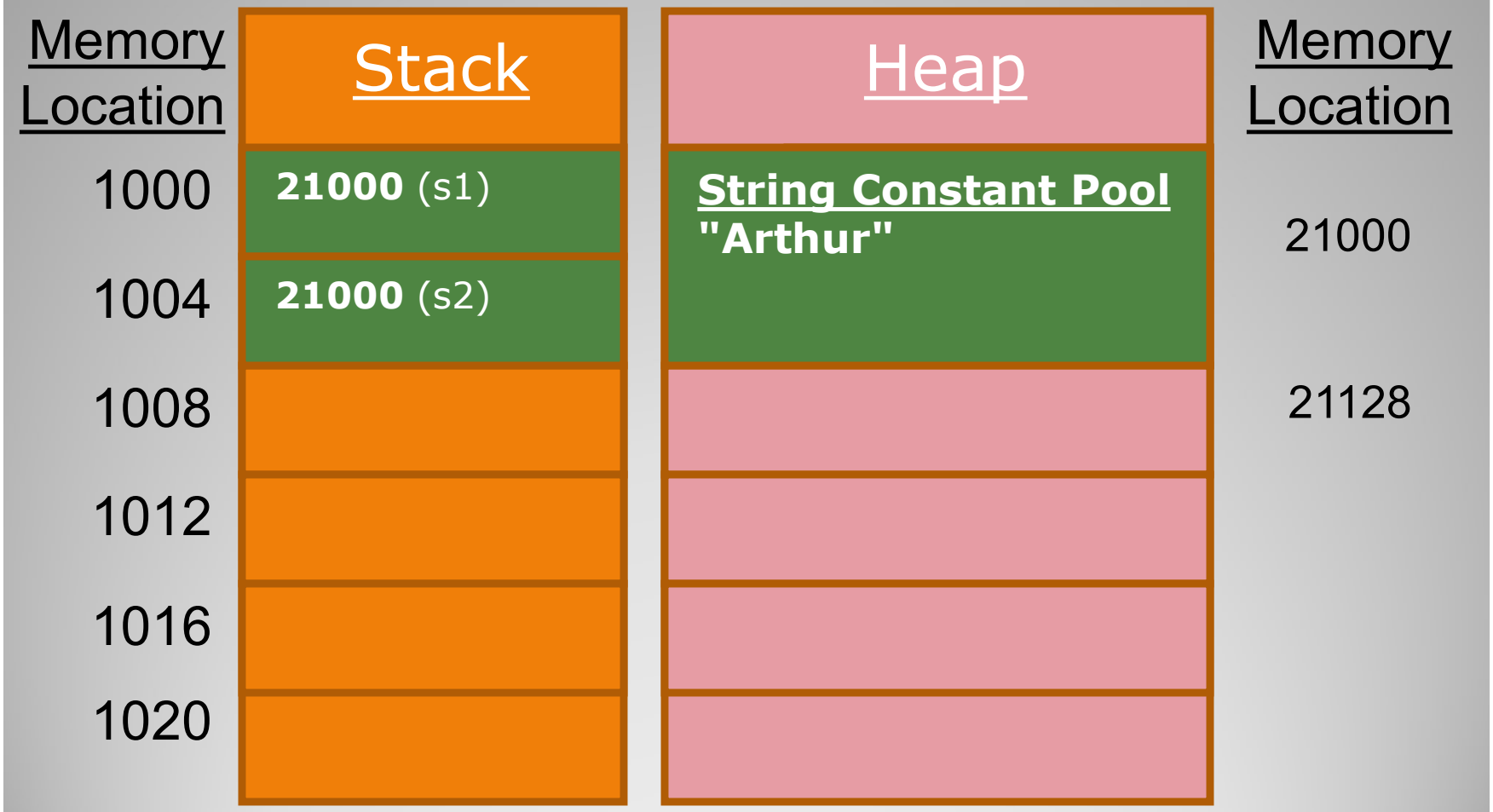
- The String Constant Pool stores all string constants.
- String constants in the pool are shared by all instances that use it (no duplicates).

```
String s1 = "Arthur";
String s2 = "Arthur";
```

s1.equals(s2) – Returns true

s1 == s2 – Returns true (refer to same exact location)

- *s1 and s2 share same string constant*



String Constant Pool

- **String Normal Heap Memory**:

String s3 = new String("Aidan");

- Behaves like normal references type.
- String is NOT in the string constant pool.
- Actual strings are NOT shared by all instances.

String s3 = new String("Aidan");
String s4 = new String("Aidan");

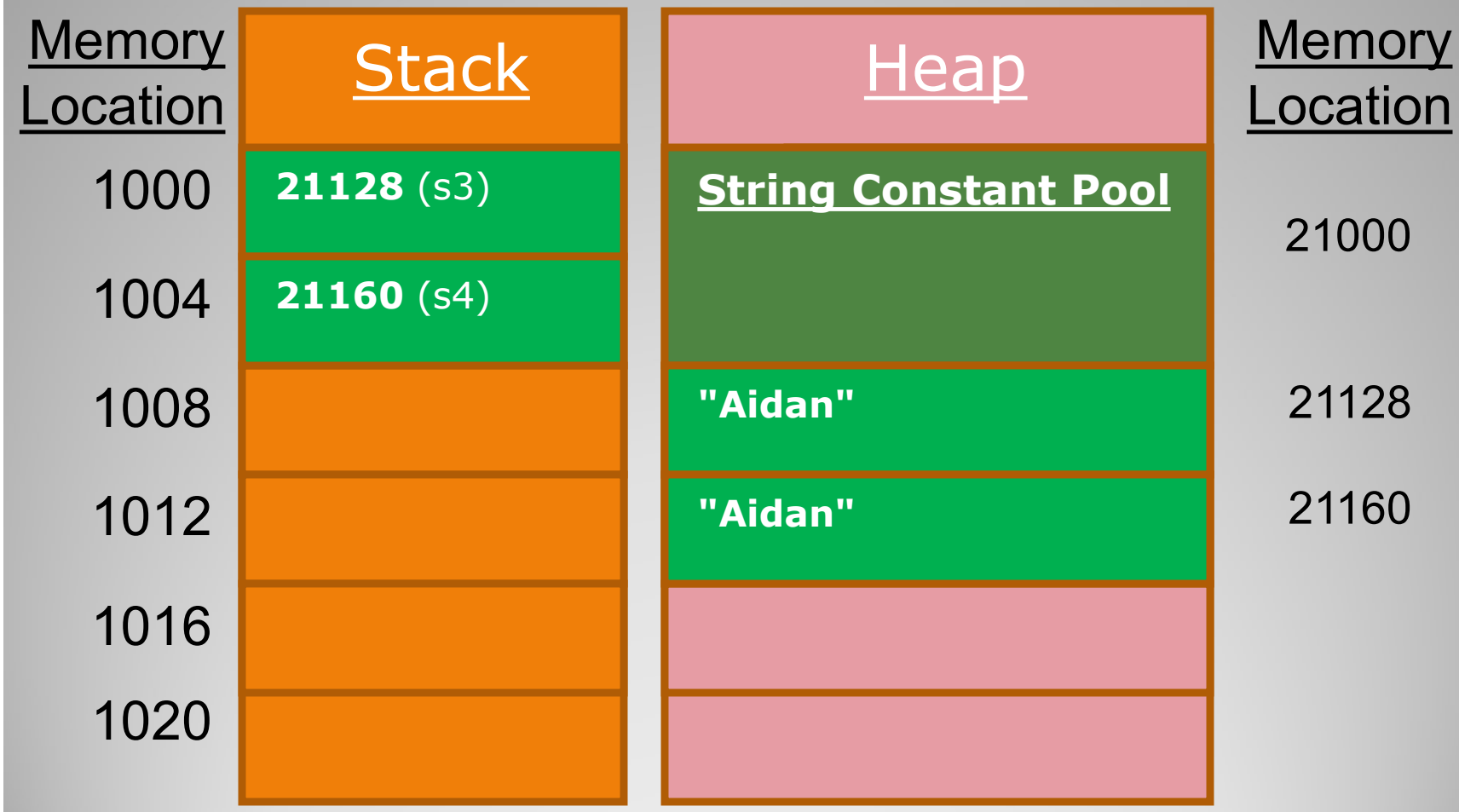
s3.equals(s4) – Returns true

s3 == s4 – Returns false

There will be two copies of "Aidan"

String Normal Heap Memory

- *s3 and s4 are NOT in the string constant pool*



String Normal Heap Memory

```
String s1 = "Arthur"
String s2 = "Arthur"
String s3 = new String("Aidan");
String s4 = new String("Aidan");
```

s1.equals(s2)	true
s3.equals(s4)	true
s1==s2	true
s3==s4	false

Strings Comparisons

- Now another example…
- Both primitive and reference types are included.

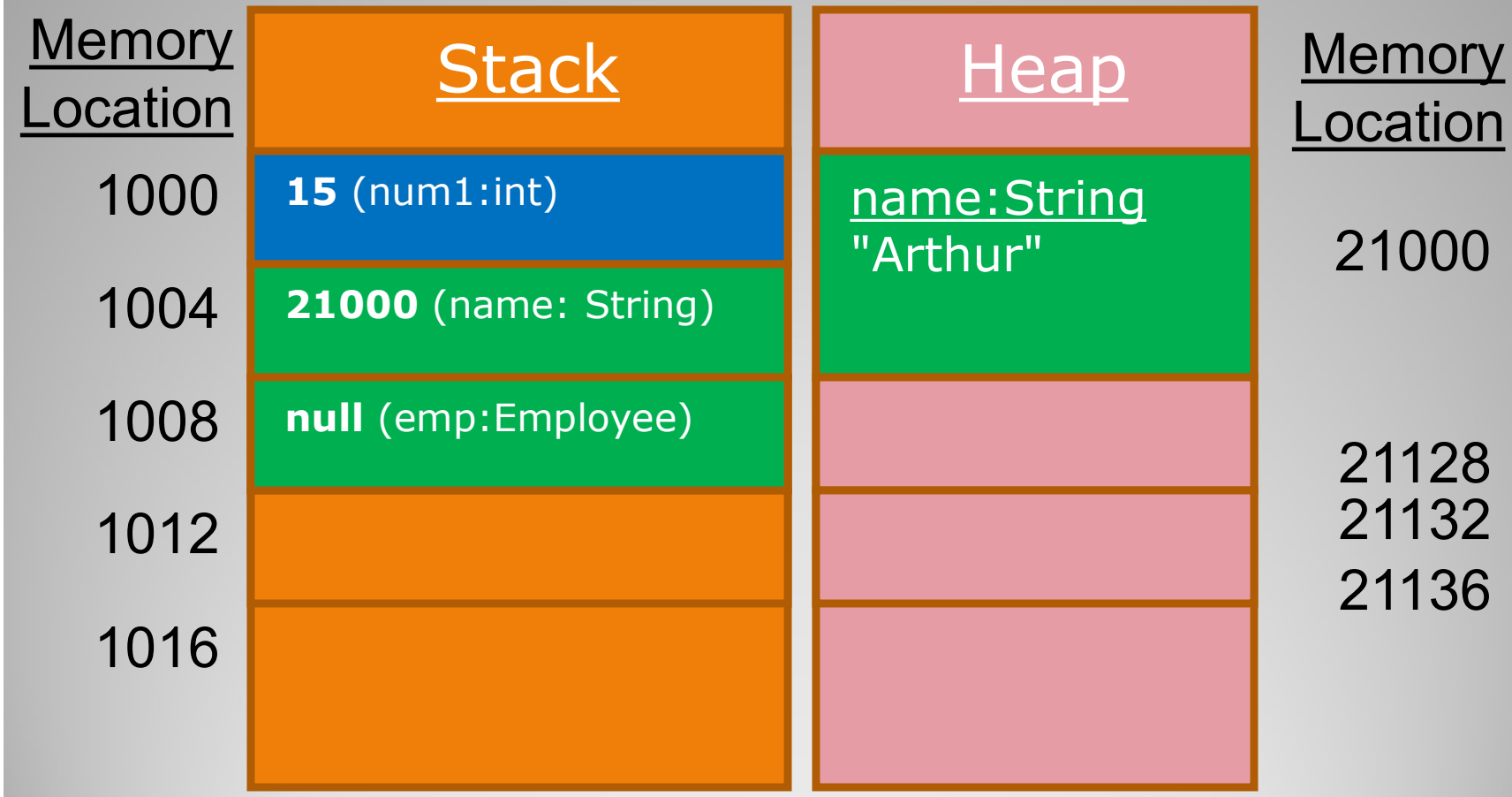
Memory

```
public class Employee {
   int    m_iId;
   int    m_iSalary;

   public Employee(int id, int salary) {
          m_iId = id;
          m_iSalary = salary;
   }

   public static void main(String[] args) {
          int num1 = 15;                    // Declare 3 variables
          String name = new String("Arthur");
          Employee emp;
   }
};
```

What does memory look like?

Memory

- ***Did not call new on Employee.***



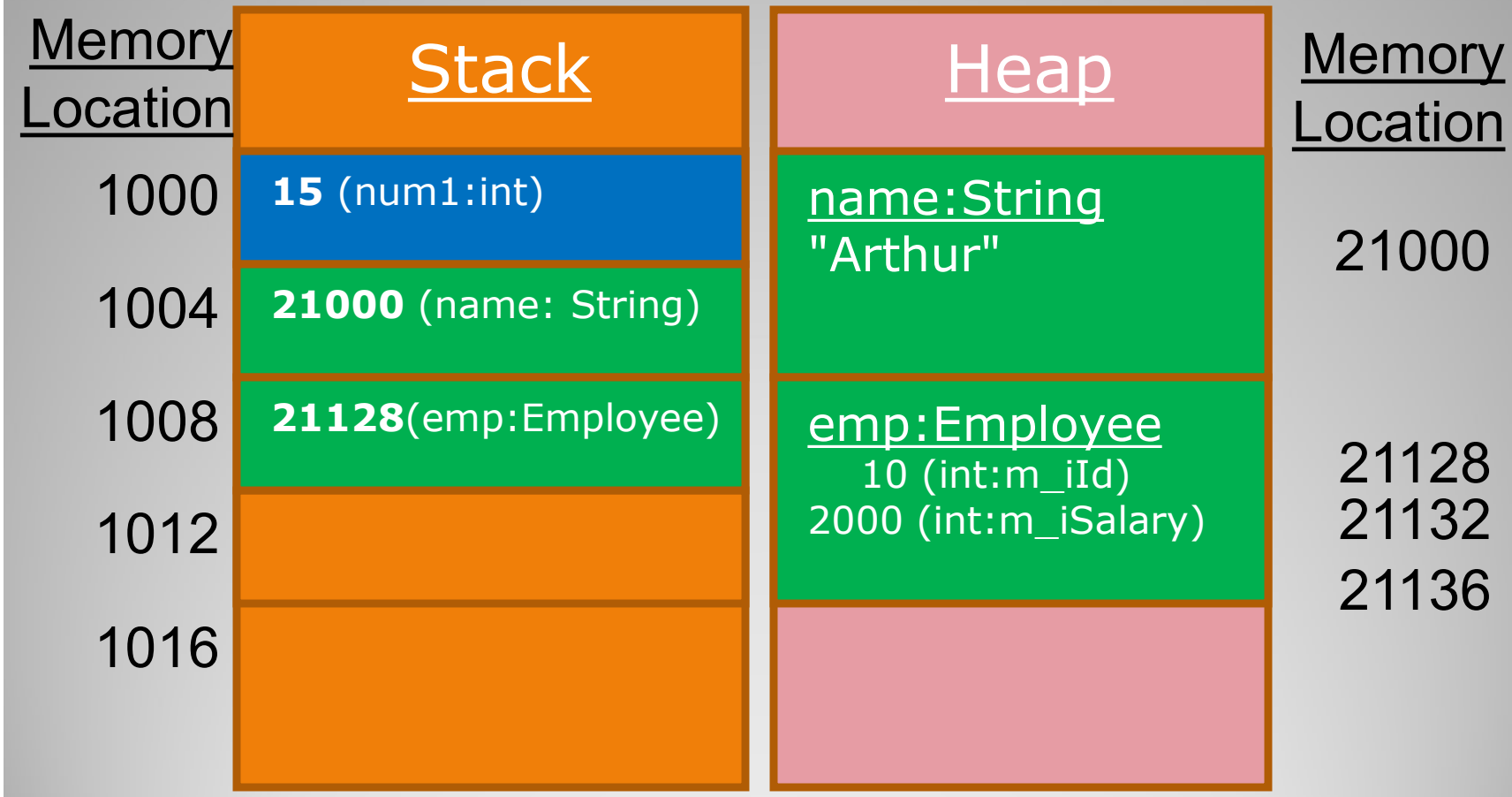
Memory

```
public class Employee {
   int m_iId;
   int    m_iSalary;

   public Employee(int id, int salary) {
         m_iId = id;
         m_iSalary = salary;
   }

   public static void main(String[] args) {
         int num1 = 15;                    // Declare 3 variables
         String name = new String("Arthur");// AND call new
         Employee emp = new Employee(10, 2000);
   }
};
```

What does memory look like?

Memory

- ***new is called for Employee.***



Memory

- **Take attendance now!!!**

Attendance

- **End of Slides**

End of Slides