# Java Programming

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

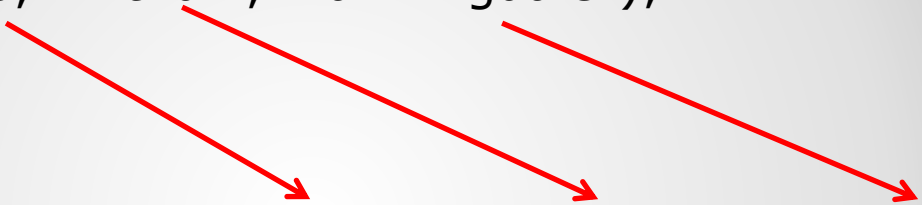
# Today's Lecture

- Chapter 6 (continued)
- Review methods
- Review stack and heap memory
- Call stack and activation records
- Method signatures
- Overloading

- Method Review

# Next Section

```
public class Test
{
   public static void main(String[] args)
   {
         ShowData(10, "Arthur", "Farmingdale");
   }


   public static void ShowData(int id, String name, String school)
   {
         System.out.println(id);
         System.out.println(name);
         System.out.println(school);
         return;
   }
}
```



# Method With Multiple Parameters

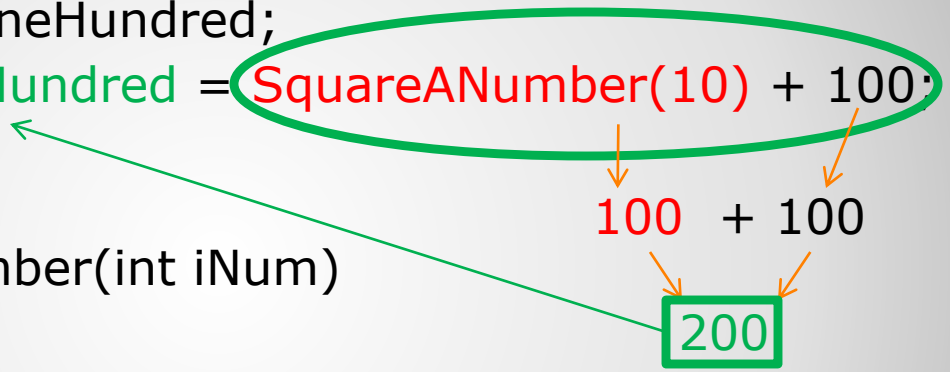
```
public class Test
{
   public void SomeMethod()
   {
         int iSquaredNum;
         iSquaredNum = SquareANumber(10);
   }

   public int SquareANumber(int iNum)
   {
         int iResult;
         iResult = iNum * iNum;

         return iResult;
   }
}
```

Returns an int

Takes an int as a parameter

# Methods and Assignment REVIEW

```
public class Test
{
    public void SomeMethod()
    {
        int iSquarePlusOneHundred;
        iSquarePlusOneHundred = SquareANumber(10) + 100;
    }

    public int SquareANumber(int iNum)
    {
        int iResult;
        iResult = iNum * iNum;

        return iResult;
    }
}
```




**SquareANumber() evaluates to 100 which is then added to the constant 100 creating the value 200.**

# Methods and Assignment REVIEW

- Stack and heap memory

# Next Section

# Two types of Memory

## Stack

All local variables and parameters

## Heap

Member variables of reference types

# Memory

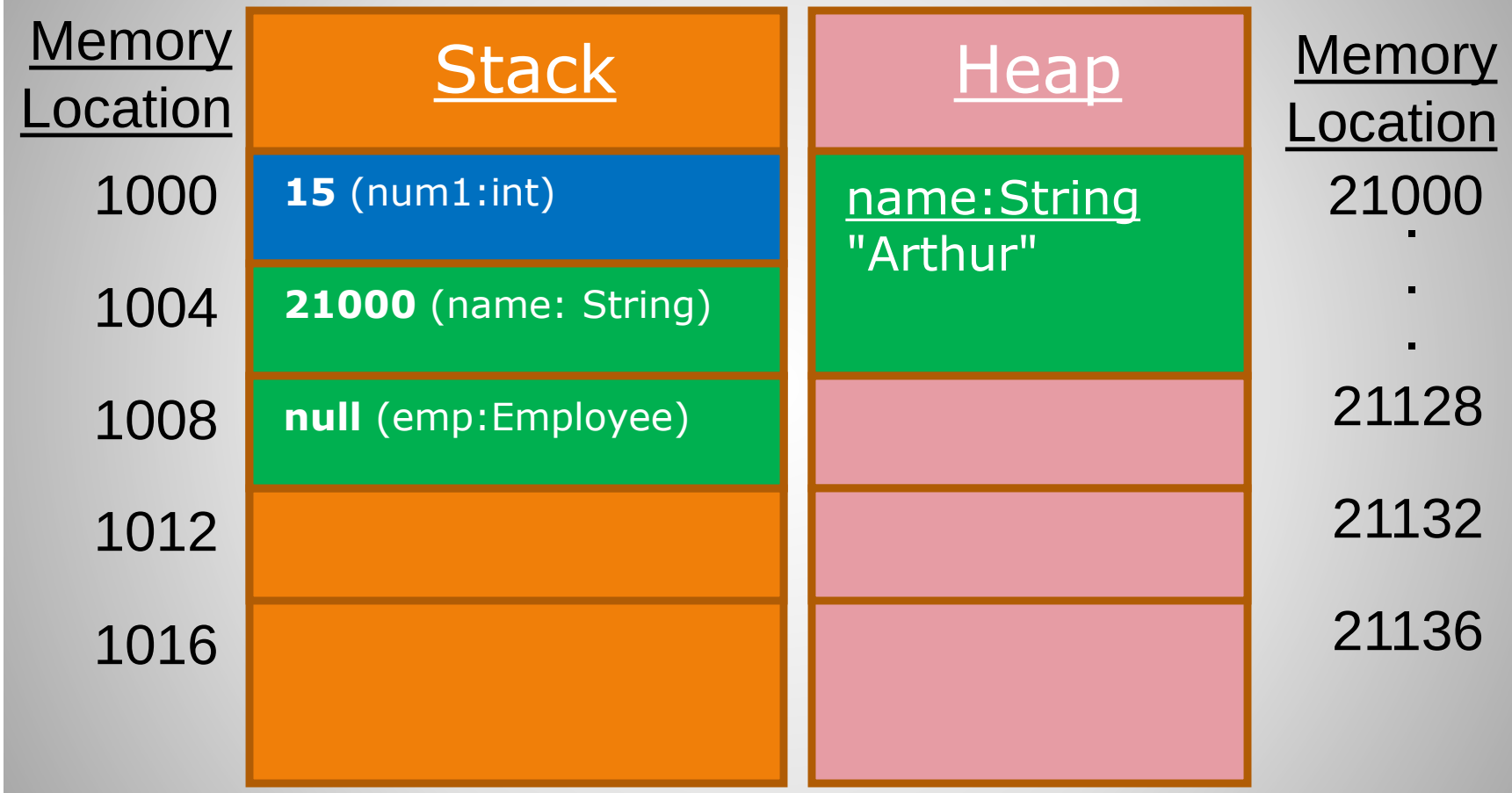
- Memory layout example…
- Both primitive and reference types are included.

```
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;
         String name = new String("Arthur");
         Employee emp;
   }
};
```

**What does memory look like?**

# Memory

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



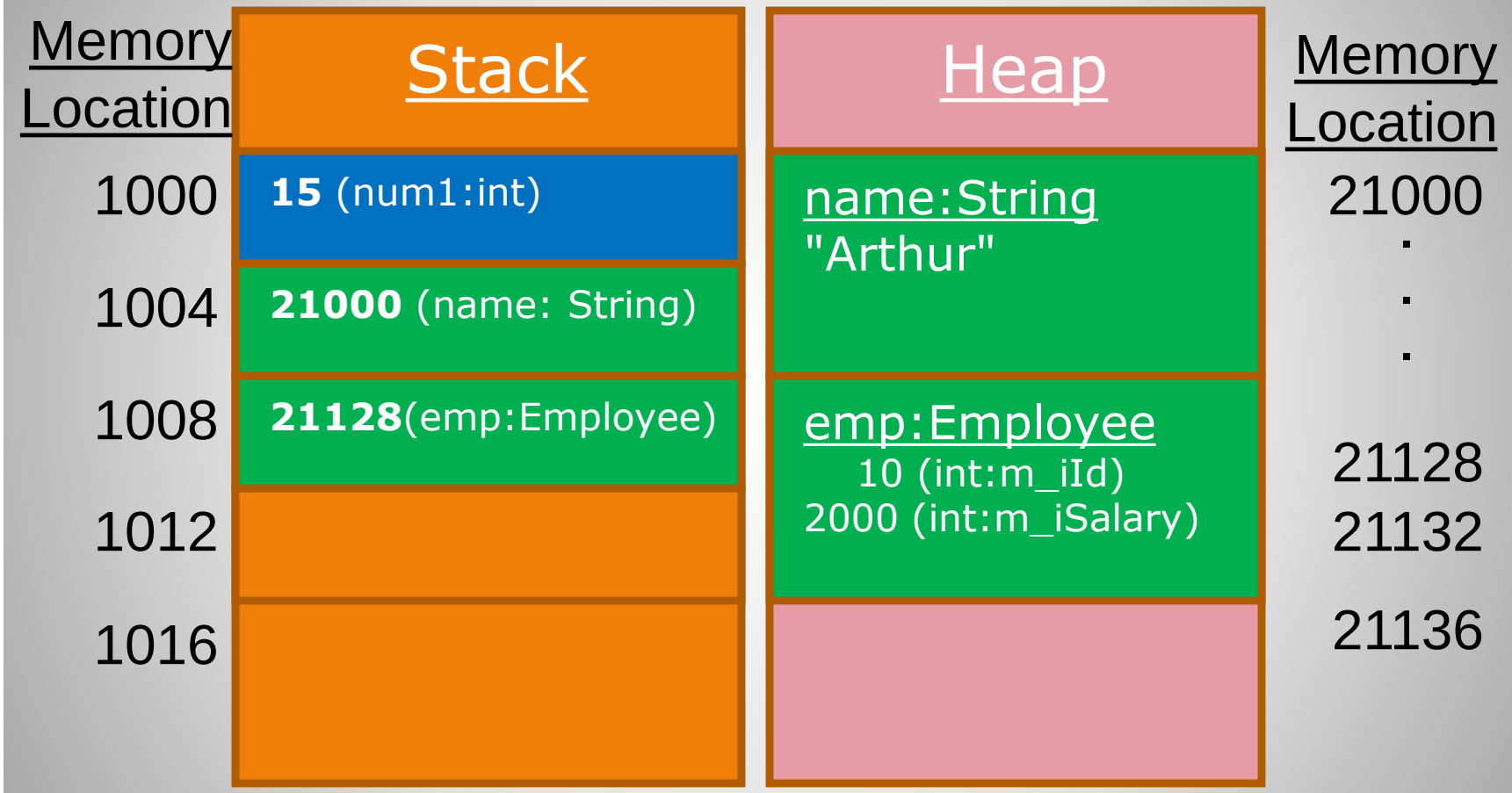

# Memory

```
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;
         String name = new String("Arthur");
         Employee emp = new Employee(10, 2000);
   }
};
```

**What does memory look like?**

# Memory

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




# Memory

*Show the memory layout of the following:*

```
public class Student {
   private int id = 1;
   private int credits = 12;
   public static void main(String args[]) {
         Student s = new Student();
         int num = 10;
         Student s2 = new Student();
}
```

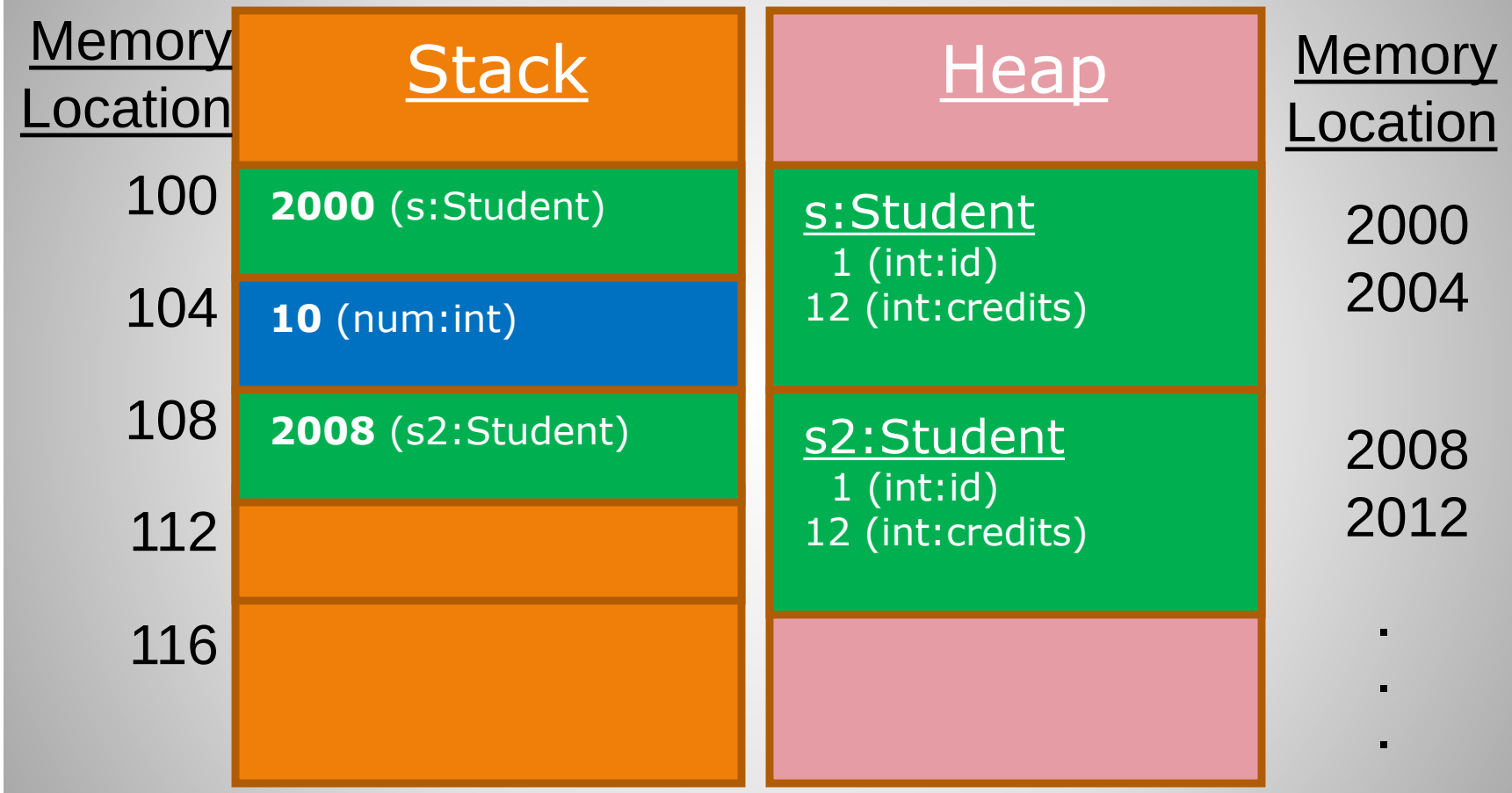
***Hints:***

1. ***int variable takes up 4 bytes***
2. ***Reference pointer takes up 4 bytes***

| Stack | | Heap | |
|---|---|---|---|
| 100 | value (name, type) | 2000 | value (name, type) |
| 104 | | 2004 | |
| 108 | | 2008 | |
| 112 | | 2012 | |

# Problem #1

- ***SOLUTION***




# Memory

- Call stack and activation records

# Next Section

# Stacks

- A stack is a data structure (a collection of related items).
- Similar to a "stack of dishes".

- If you add a dish to the pile it will always be placed on top.

# Stacks

Assume the following:

1. **Only add to the top of the stack.**
2. **Only remove from the top of the stack.**

- So, if you add a dish on top of a stack then that dish will be the first one removed (because it is on top).
- Last In First Out (LIFO). The last one in is the first one out.

# Stacks

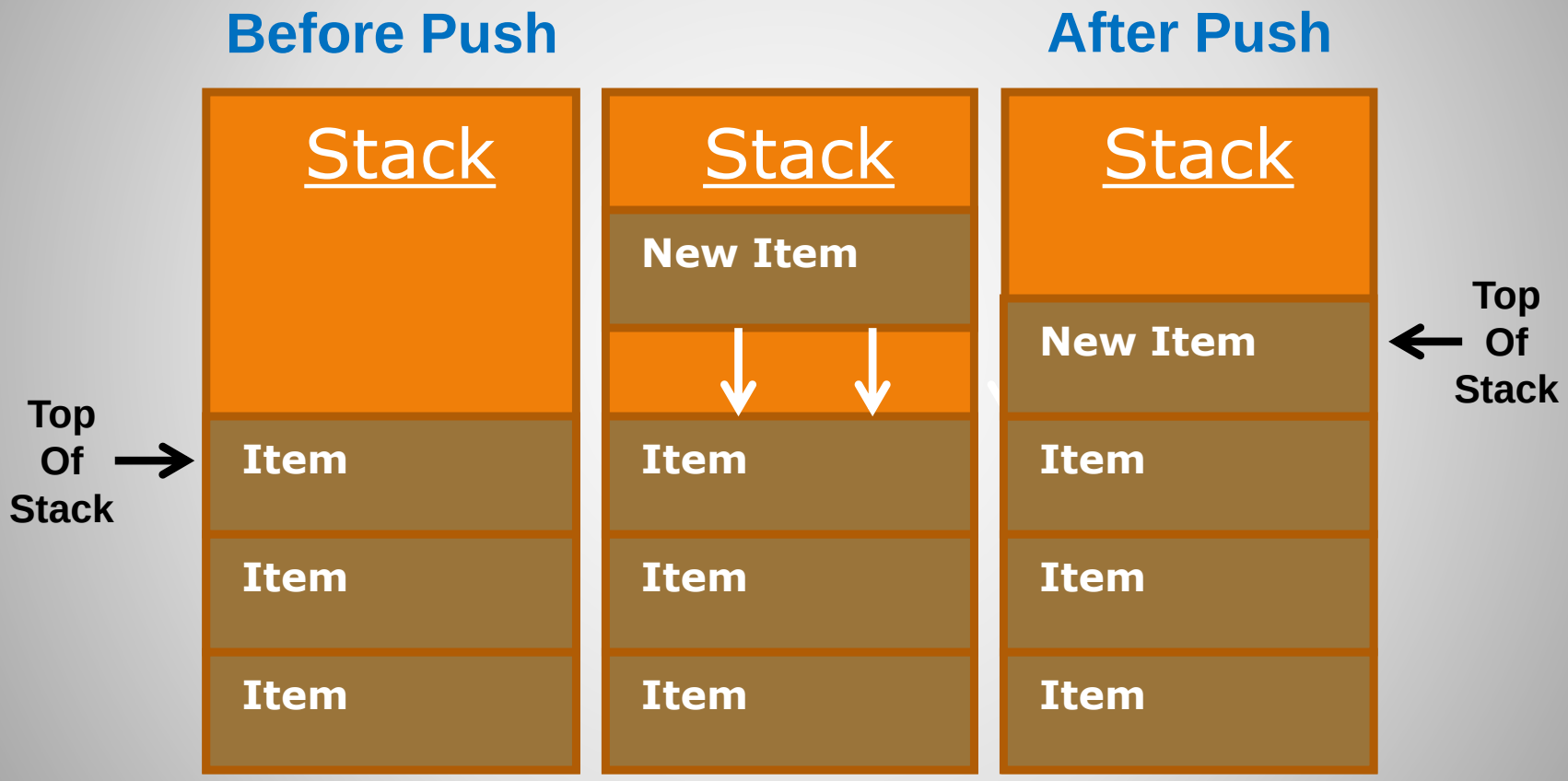
- Terminology:
  - **Push**: Put something on the stack.
  - **Pop**: Take something off the stack.

- You ***push*** items on to a stack.
- You ***pop*** items off of a stack.

- Pushing and popping only occur from the top of the stack.

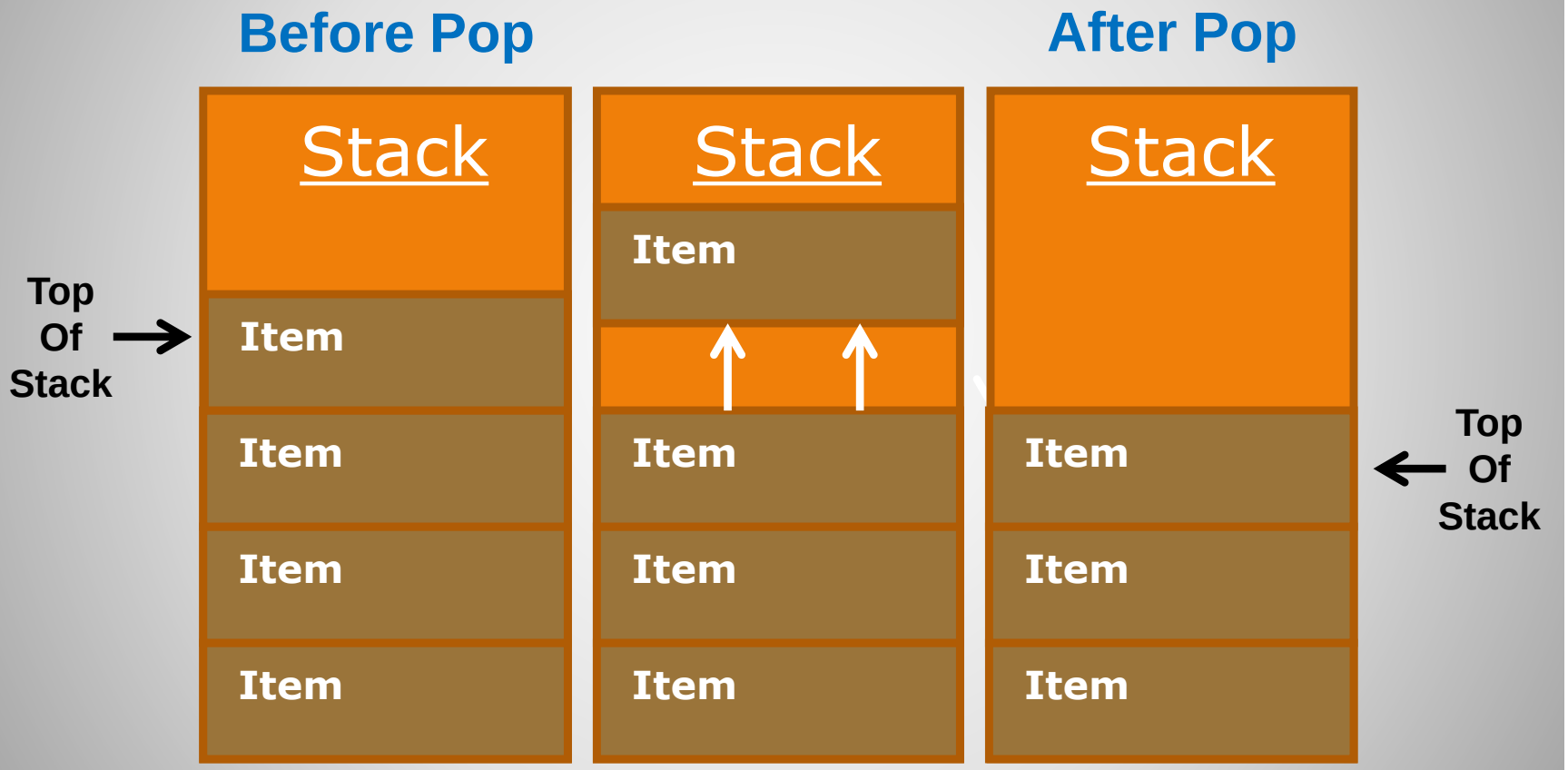
- For example…

- Add items – "Push" on to top of stack




# Stack - Push

- Remove items – "Pop" from top of stack




# Stack - Pop

## More details about the JVM stack.

- Proper name: ***Method call stack*** or ***program execution stack***.
- Variables are not just stored anywhere on the stack.
- Variables from the ***same method*** are grouped together on the stack.

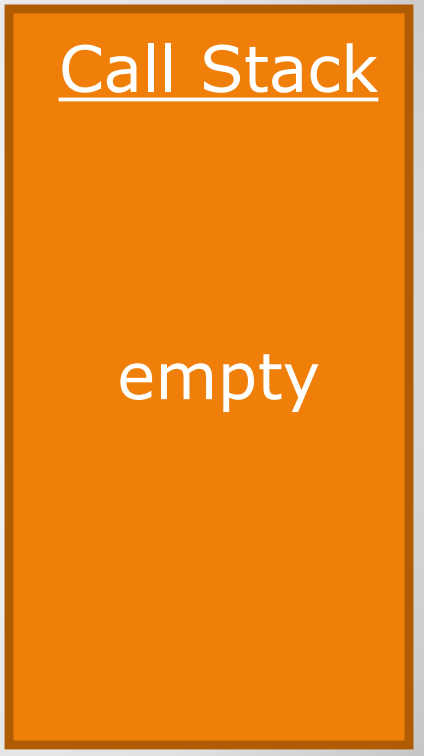
# Method Call Stack

# Method Call Stack

- All variables declared in a method are stored in an ***activation record (or stack frame)***.
- The activation record for a method call stores all the variables declared in that method.
- Call Stack Actions
  - **Call Method: Push activation record on stack.**
  - **End Method: Pop activation record off stack.**
- For example…

Program has not started yet. No activation records on stack.

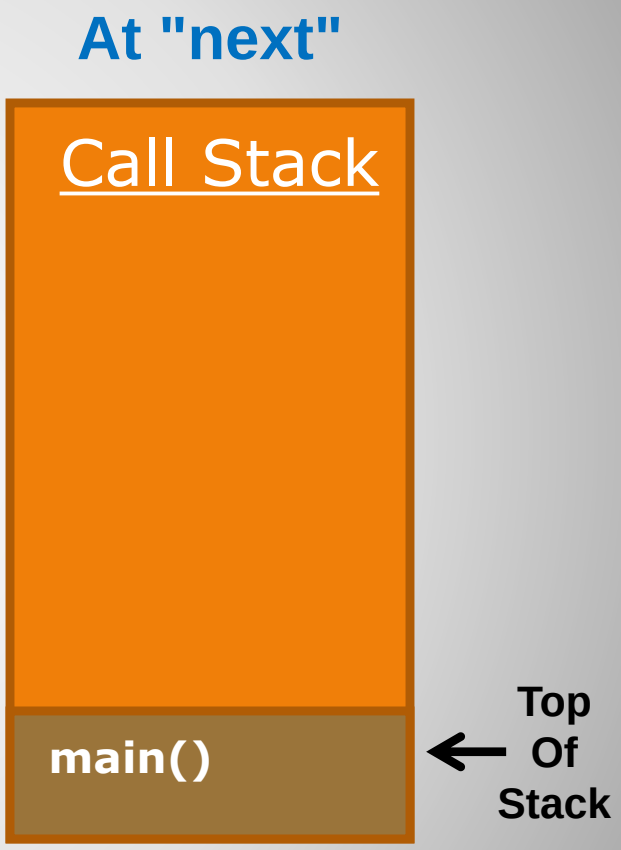
```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B();
   B();
}

void main(…) {
 System.out.println("In main");
 A();
}
```




# Method Calls and Call Stack

Program started. In main and about to execute the "next" line (in bold).

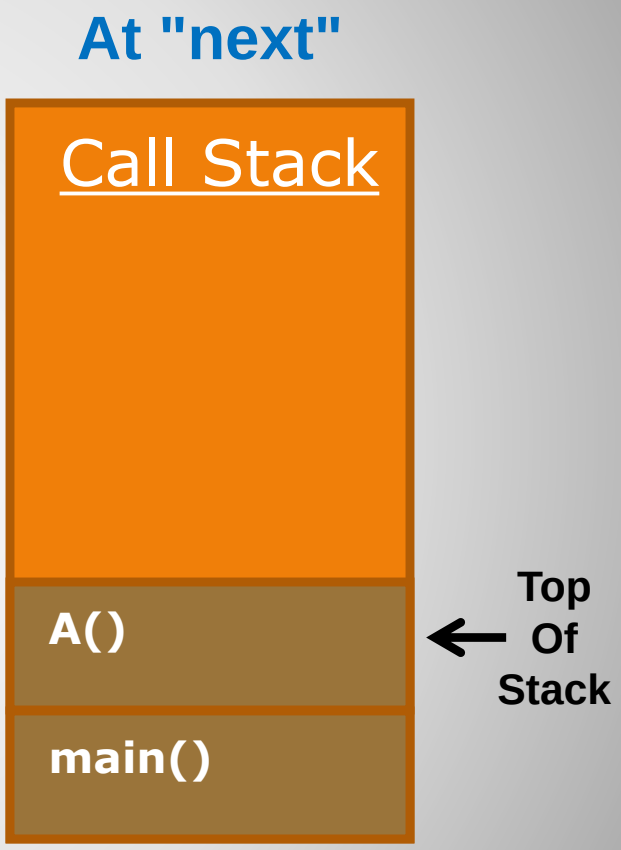
```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B();
   B();
}

void main(…) {
 System.out.println("In main"); // next
 A();
}
```




# Method Calls and Call Stack

Main called A. This causes an activation record for A to be pushed on stack.

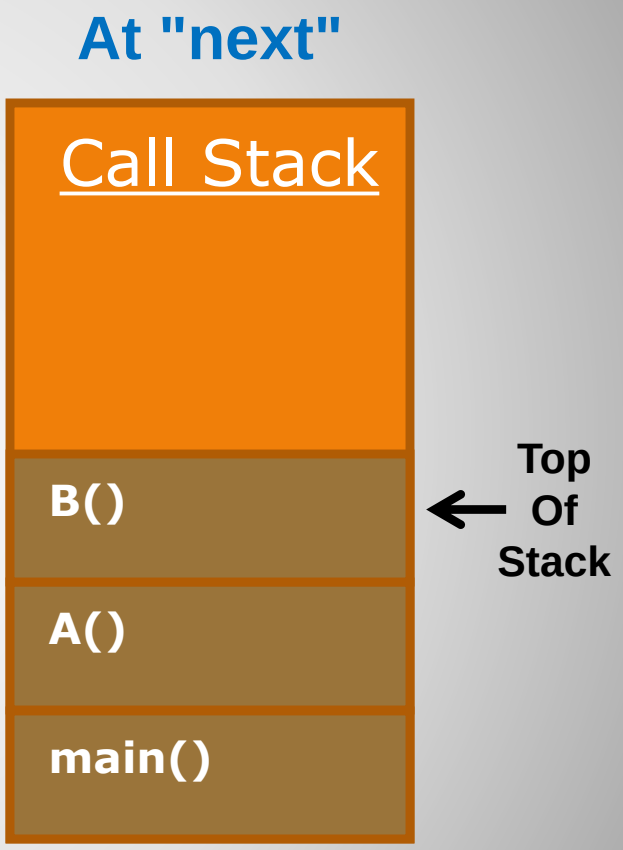
```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B(); // next
   B();
}

void main(…) {
 System.out.println("In main");
 A(); // called from here…
}
```




# Method Calls and Call Stack

A called B. This causes an activation record for B to be pushed on stack.

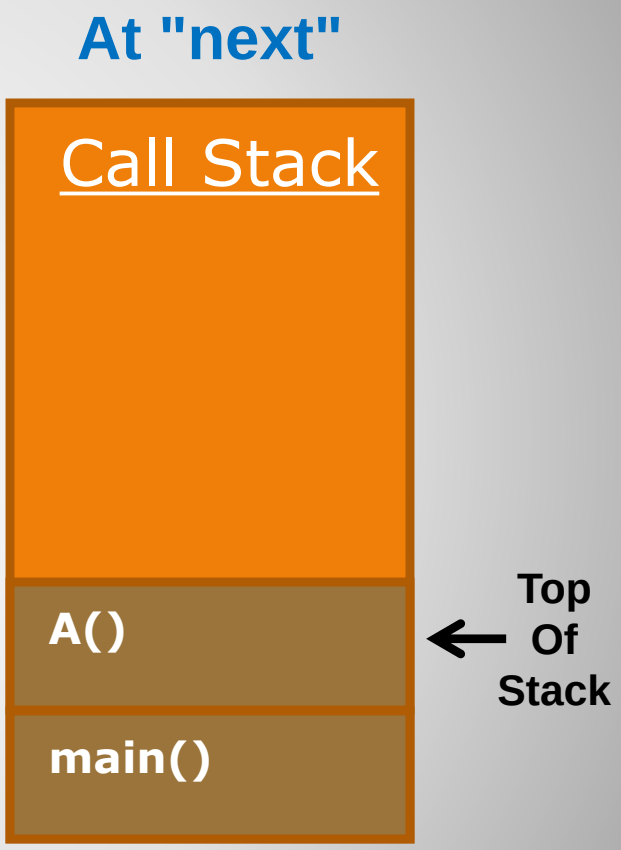
```
void B() {
   System.out.println("In B"); // next
}

void A() {
   System.out.println("In A");
   B(); // called from here...
   B();
}

void main(…) {
 System.out.println("In main");
 A();
}
```




# Method Calls and Call Stack

B ended. This causes B activation record to be popped. A will call B again.

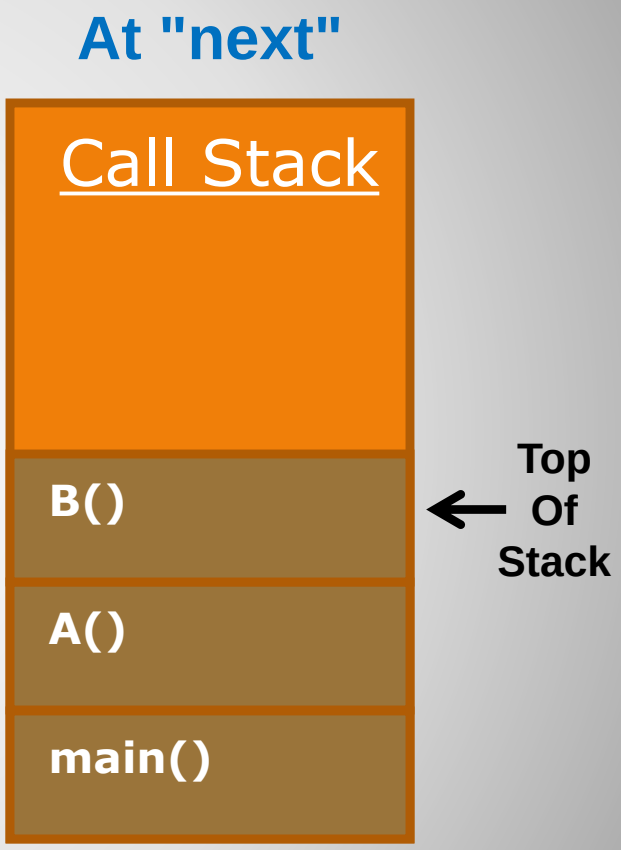
```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B();
   B(); // next
}

void main(…) {
 System.out.println("In main");
 A();
}
```




# Method Calls and Call Stack

A called B again. An activation record for B is pushed on the stack again.

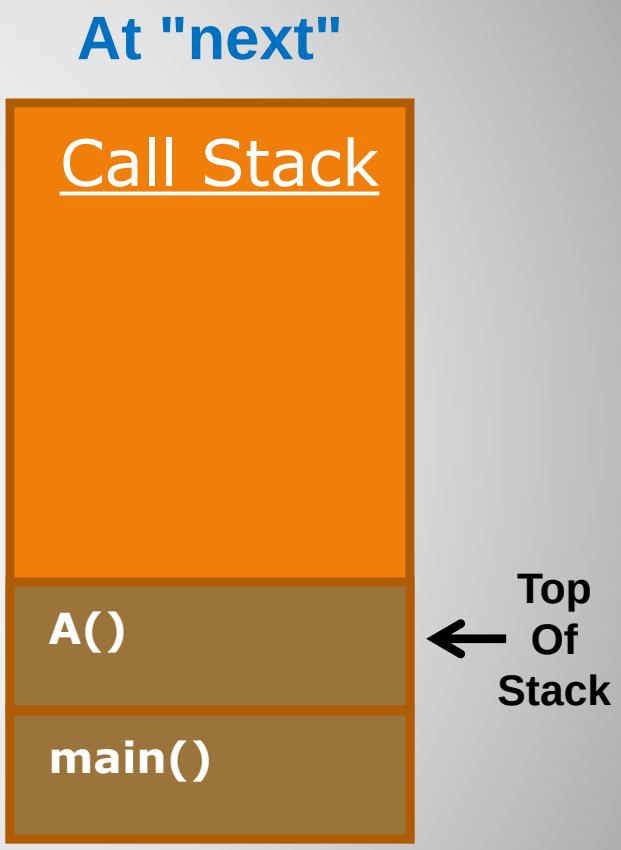
```
void B() {
   System.out.println("In B"); // next
}

void A() {
   System.out.println("In A");
   B();
   B(); // called from here...
}

void main(…) {
 System.out.println("In main");
 A();
}
```




# Method Calls and Call Stack

B ended. This causes B activation record to be popped. A about to end.

```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B();
   B();
} // next

void main(…) {
 System.out.println("In main");
 A();
}
```




# Method Calls and Call Stack

A ended. This causes A activation record to be popped. main about to end.

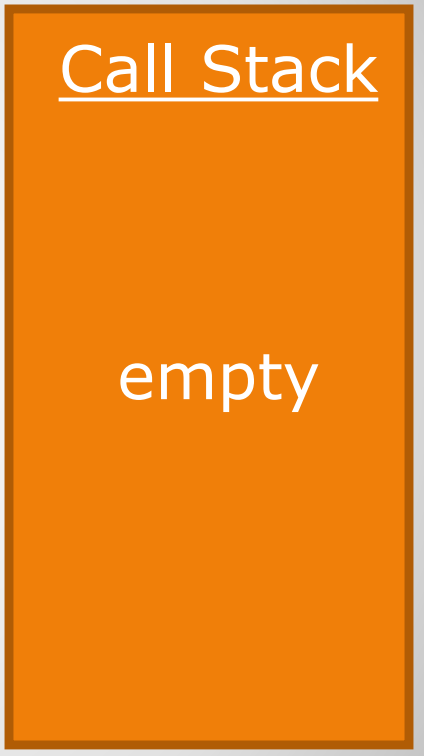
```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B();
}

void main(…) {
 System.out.println("In main");
 A();
} // next
```


# Method Calls and Call Stack

main ended. Program Done. No more activation records on stack.

```
void B() {
   System.out.println("In B");
}

void A() {
   System.out.println("In A");
   B();
}

void main(…) {
 System.out.println("In main");
 A();
}
```

**At "next"**



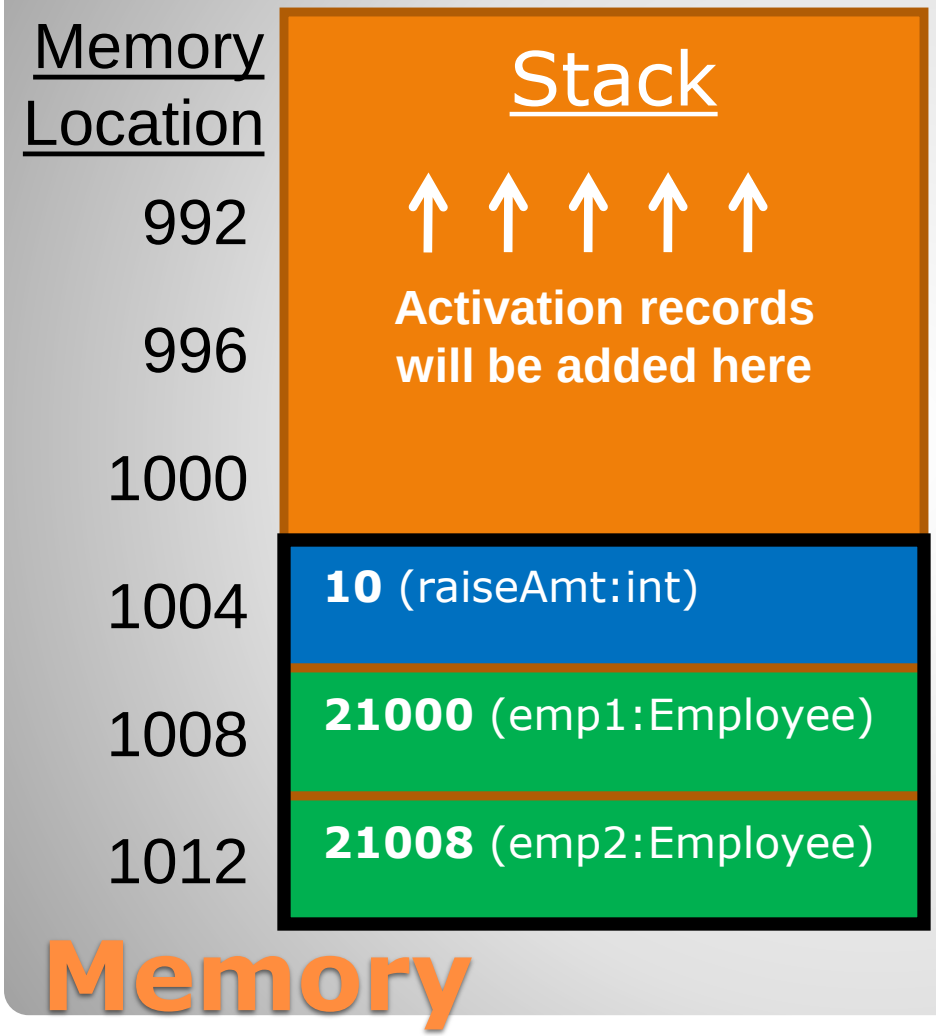

# Method Calls and Call Stack

```
public class Employee {
    int m_Id;
    int m_Salary;

    public Employee(int id, int salary) {
            m_Id = id;
            m_Salary = salary;
    }

    public void Raise(int amount) {
            m_Salary = m_Salary + amount;
    }

    public static void main(…) {
            Employee emp1 = new Employee(111, 20);
            Employee emp2 = new Employee(222, 50);
            int raiseAmt = 10;
            emp1.Raise(raiseAmt); // next
            emp2.Raise(raiseAmt);
    }
};
```

**Assume the program has executed to the "next" line.**

**What does memory look like in more detail using activation records?**

- **main local variables grouped together**




main()

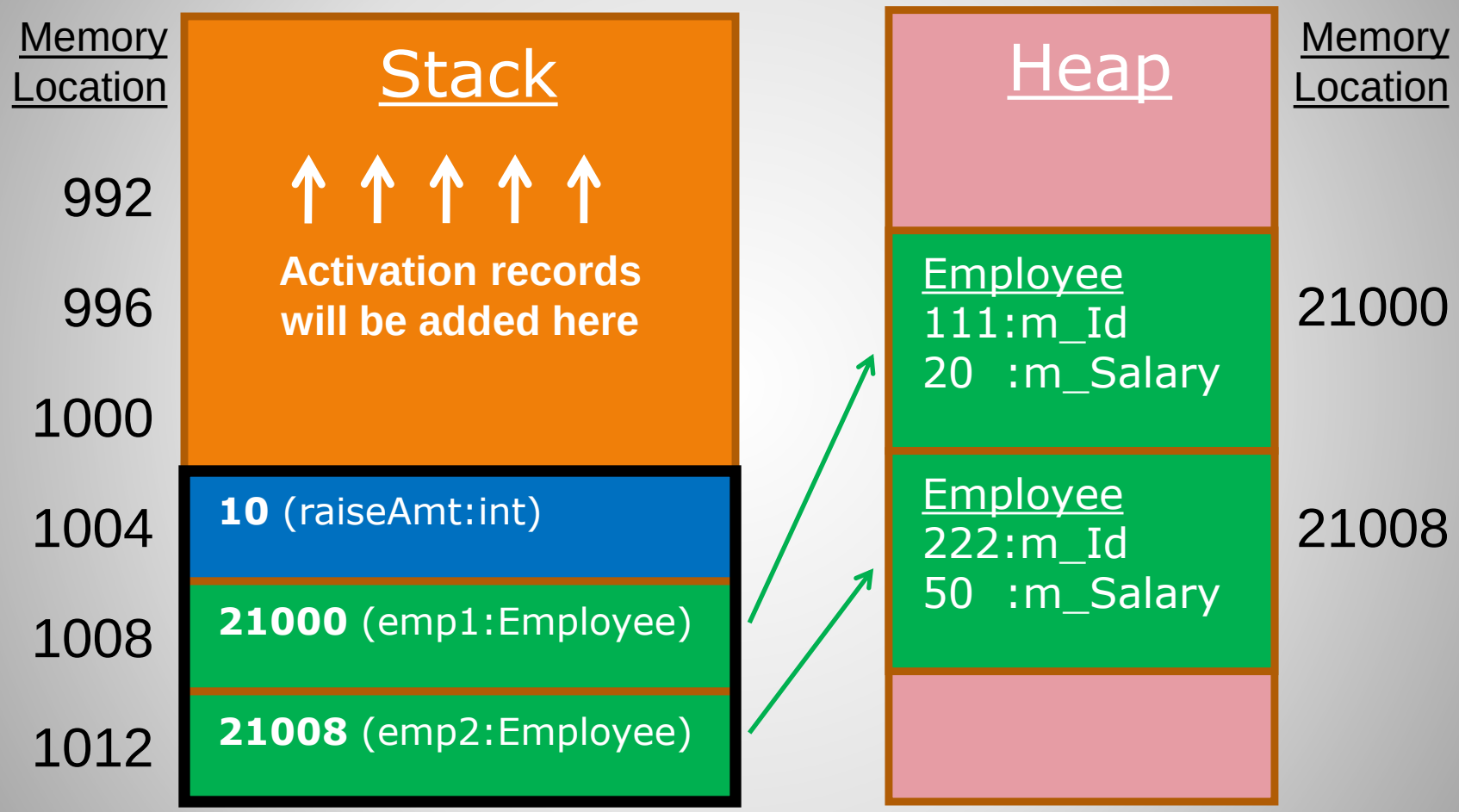
**main() method's activation record**

**The activation record is colored black**

**Important**

**Activation record holds all local variables and parameters**

- **emp1 and emp2 refer to heap locations**



```
public class Employee {
    int m_Id;
    int m_Salary;

    public Employee(int id, int salary) {
            m_Id = id;
            m_Salary = salary;
    }

    public void Raise(int amount) {
            m_Salary = m_Salary + amount;
    }

    public static void main(…) {
            Employee emp1 = new Employee(111, 20);
            Employee emp2 = new Employee(222, 50);
            int raiseAmt = 10;
            emp1.Raise(raiseAmt);
            emp2.Raise(raiseAmt);
    }
};
```

**When inside the Raise method how does it know which m_Salary to use?**

**Is the value 20 or 50?**

# this Reference

- How does it know which m_Salary to use?

**Answer: It passes in the base address of the instance to work with when Raise is called.**

- In general, when an instance method is called the instances reference is passed inside the **this** reference.
- It is a hidden parameter that gets passed into the method.

# this Reference

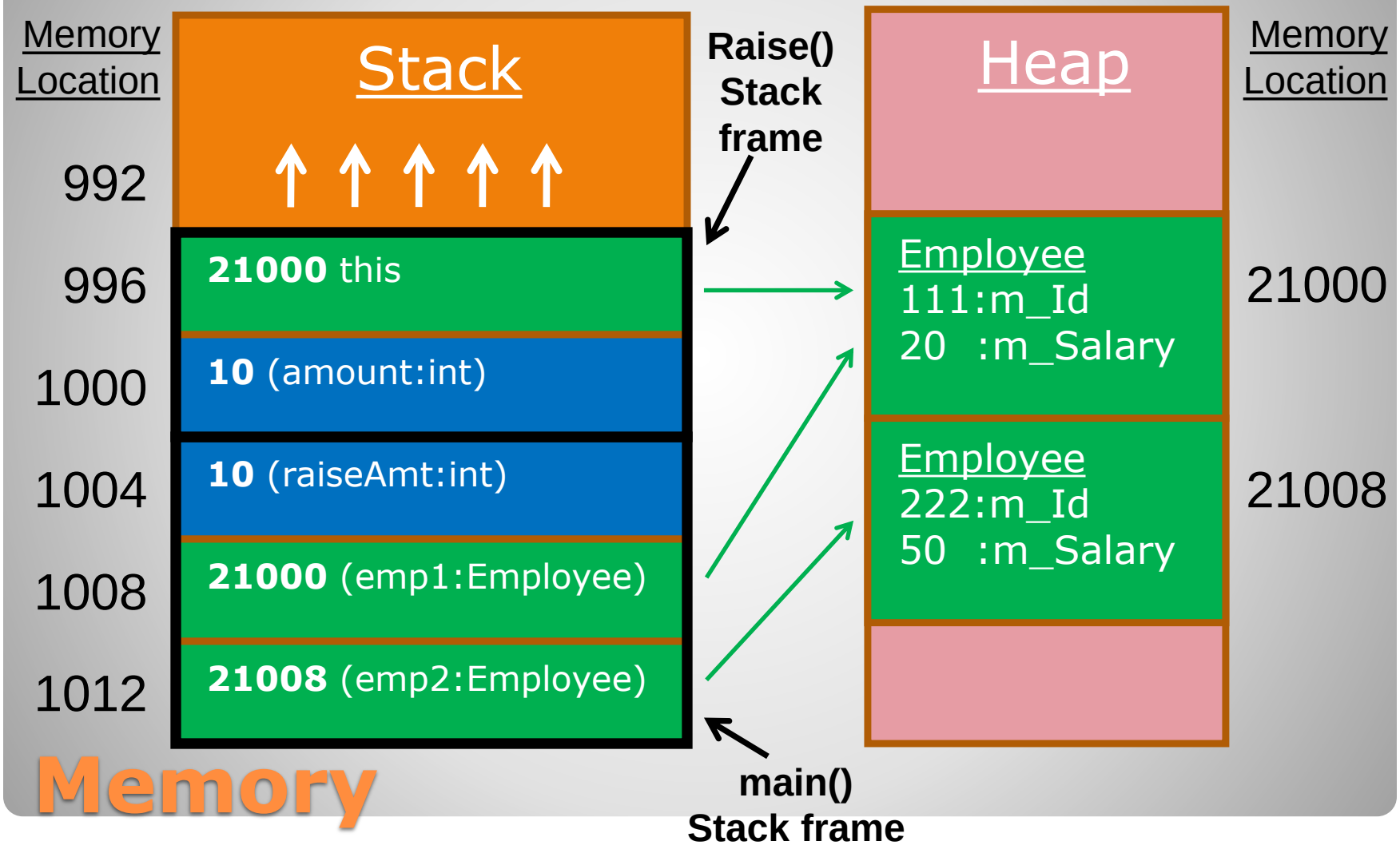
- The **this** reference is used to get access to the current instances member variables.
- **this** is automatically populated with the address of the current instance when an instance method is called.
- The value of **this** will change depending on which instance it was called from.

```
public class Employee {
    int m_Id;
    int m_Salary;

    public Employee(int id, int salary) {
            m_Id = id;
            m_Salary = salary;
    }

    public void Raise(int amount) {
            m_Salary = m_Salary + amount;
    }

    public static void main(…) {
            Employee emp1 = new Employee(111, 20);
            Employee emp2 = new Employee(222, 50);
            int raiseAmt = 10;
            emp1.Raise(raiseAmt); // called from here
            emp2.Raise(raiseAmt);
    }
};
```

**When inside the Raise method how does it know which m_Salary to use?**

**Is the value 20 or 50?**

- **this reference has 21000 (m_Salary is 20)**

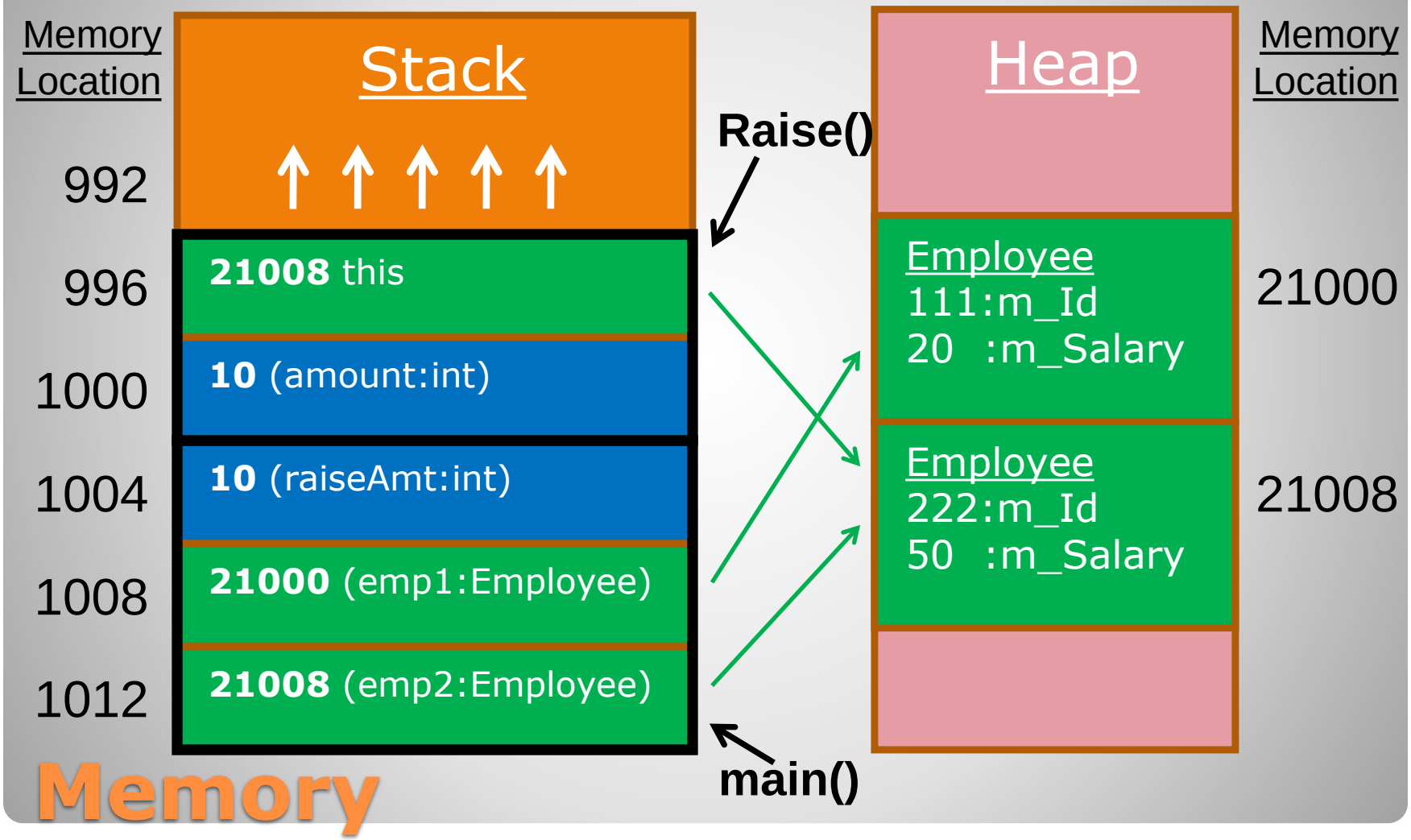


```
public class Employee {
    int m_Id;
    int m_Salary;

    public Employee(int id, int salary) {
            m_Id = id;
            m_Salary = salary;
    }

    public void Raise(int amount) {
            m_Salary = m_Salary + amount;
    }

    public static void main(…) {
            Employee emp1 = new Employee(111, 20);
            Employee emp2 = new Employee(222, 50);
            int raiseAmt = 10;
            emp1.Raise(raiseAmt);
            emp2.Raise(raiseAmt); // called from here
    }
};
```

**When inside the Raise method how does it know which m_Salary to use?**

**Is the value 20 or 50?**

- **this reference has 21008 (m_Salary is 50)**

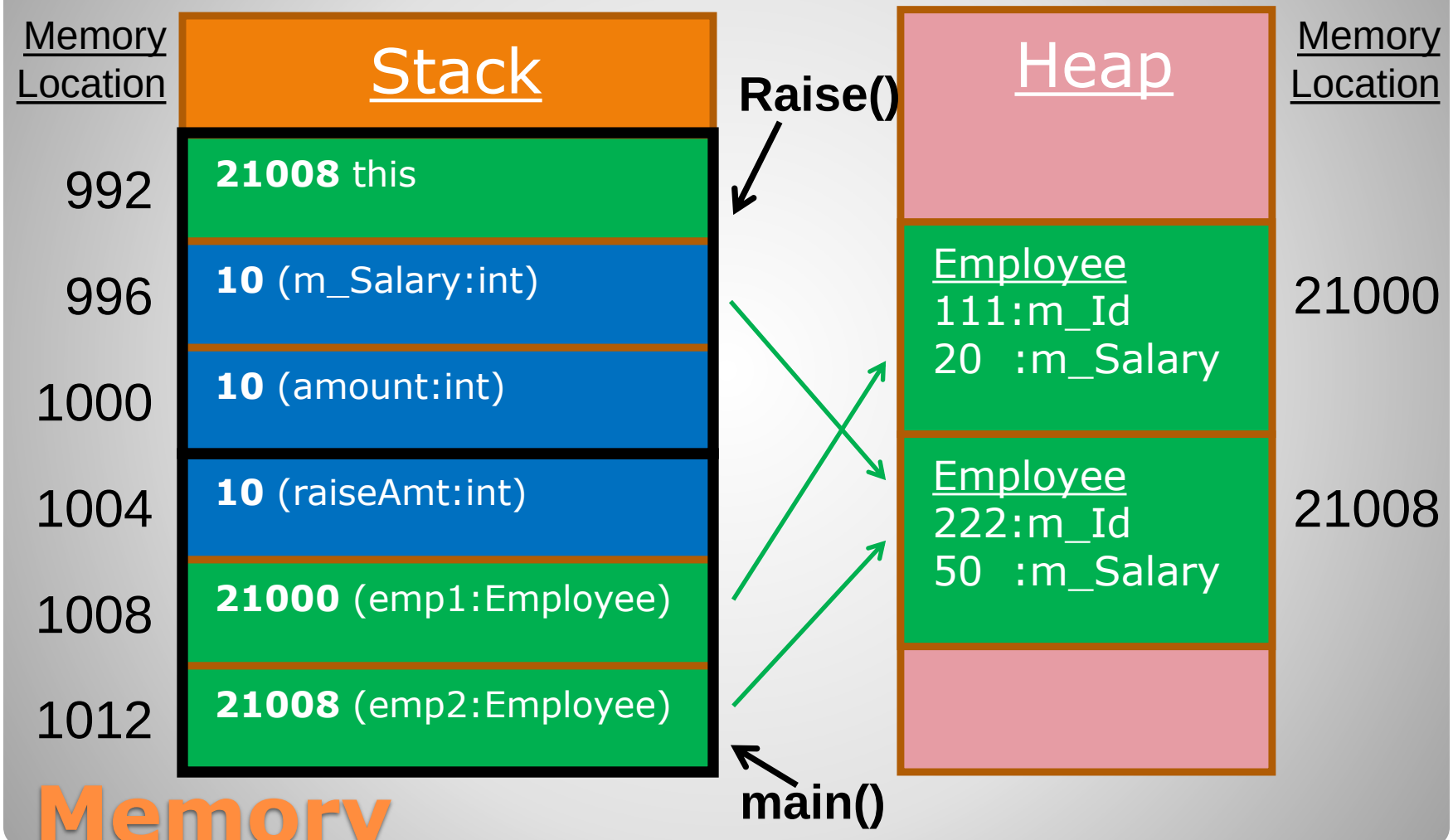


```
public class Employee {
    int m_Id;
    int m_Salary;

    public Employee(int id, int salary) {
            m_Id = id;
            m_Salary = salary;
    }

    public void Raise(int amount) {
            int m_Salary;
            m_Salary = m_Salary + amount; // next
    }

    public static void main(…) {
            Employee emp1 = new Employee(111, 20);
            Employee emp2 = new Employee(222, 50);
            int raiseAmt = 10;
            emp1.Raise(raiseAmt);
            emp2.Raise(raiseAmt); // called from here
    }
};
```

**Which m_Salary gets used?**

**What is the value of m_Salary before running "next" line?**

- **Two m_Salary (local and member)**



**Find the Correct Variable Inside a Method**

1. Look for it as a local variable first (stored in activation record).
2. If not found then use **this** reference to find it as a member variable.

- If a variable is being used that is **not** declared in the current activation record it will follow the **this** reference and look for it as a member of the class.

# Finding Correct Variable

# Shadowing

- BE CAREFUL !!!
- The local variable m_Salary hides or "shadows" the member variable m_Salary.

```
public class Employee {
    int m_Id;
    int m_Salary;

    // other code here…

    public void Raise(int amount) {
            int m_Salary;   // Shadows member variable
            m_Salary = m_Salary + amount;
    }

    // other code here…
}
```

**This will change the local m_Salary. The member variable m_Salary will remain unchanged.**

# Shadowing

- You are allowed to explicitly use "this" in your code.
- Allows you to get around shadowing.

```
public class Employee {
    int m_Id;
    int m_Salary;

    // other code here…

    public void Raise(int amount) {
            int m_Salary;   // Shadows member variable
            this.m_Salary = this.m_Salary + amount;
    }

    // other code here…
}
```

**You can explicitly use the "this" reference to avoid the shadowing**

- End of Slides

# End of Slides