

Lecture – 4

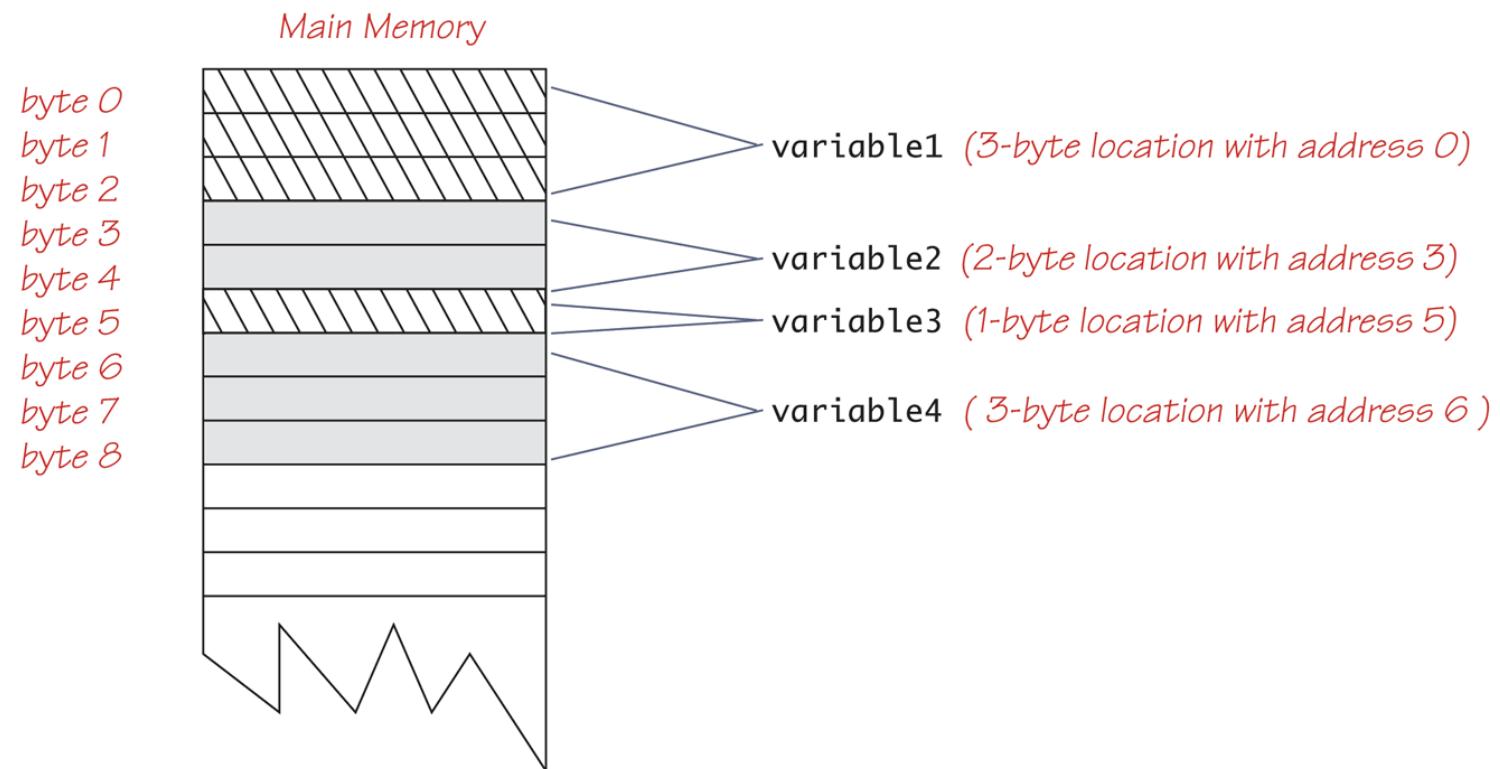
Objects and Memory

Variables and Memory

- Values of most data types require more than one byte of storage
 - Several adjacent bytes are then used to hold the data item
 - The entire chunk of memory that holds the data is called its *memory location*
 - The address of the first byte of this memory location is used as the address for the data item
- A computer's main memory can be thought of as a long list of memory locations of *varying sizes*

Variables in Memory

Display 5.10 **Variables in Memory**

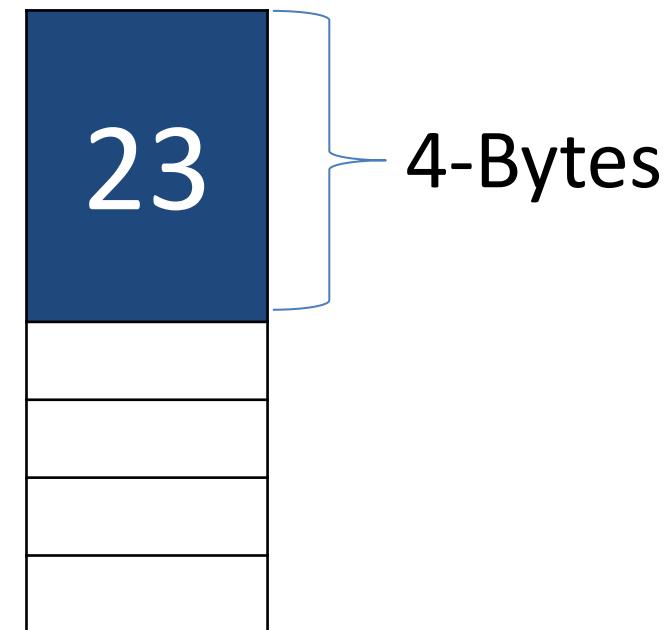


References

- Every variable is implemented as a location in computer memory
- When the variable is a primitive type, the value of the variable is stored in the memory location assigned to the variable
 - Each primitive type always require the same amount of memory to store its values

```
int age = 23
```

age



References

- When the variable is a class type, only the memory address (or *reference*) where its object is located is stored in the memory location assigned to the variable
 - The object named by the variable is stored in some other location in memory
 - Like primitives, the value of a class variable is a fixed size
 - Unlike primitives, the value of a class variable is a memory address or reference
 - The object, whose address is stored in the variable, can be of any size

Class Type Variables Store a Reference (Part 1 of 2)

Display 5.12 Class Type Variables Store a Reference

```
public class ToyClass
{
    private String name;
    private int number;
```

The complete definition of the class ToyClass is given in Display 5.11.

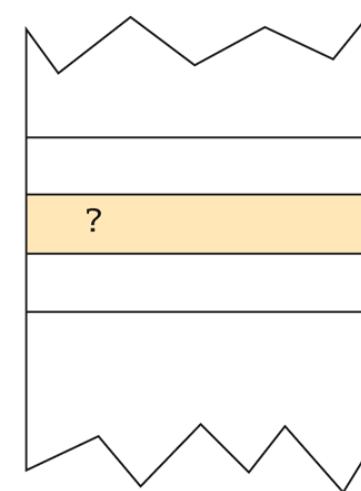
```
ToyClass sampleVariable;
```

Creates the variable sampleVariable in memory but assigns it no value.

sampleVariable

```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

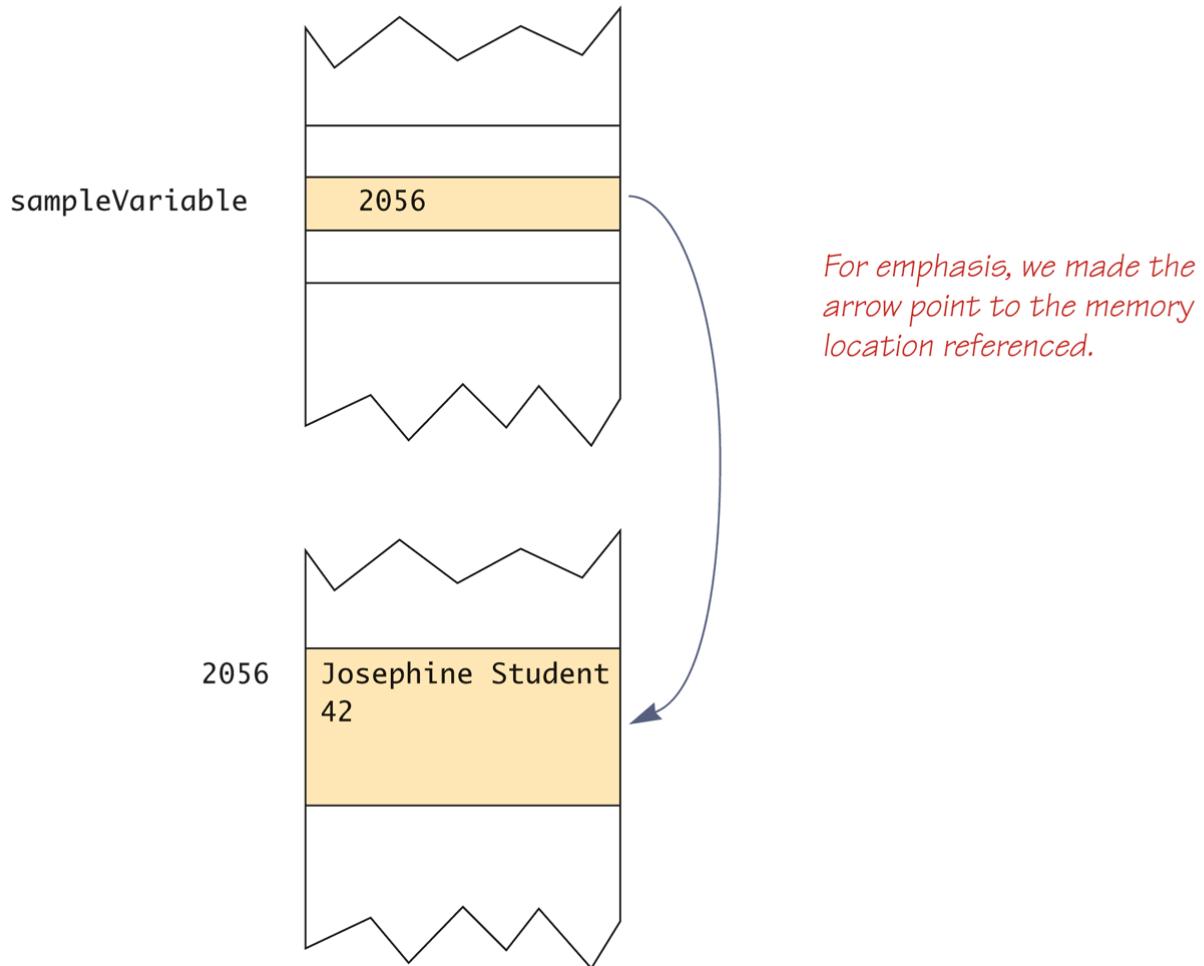
Creates an object, places the object someplace in memory, and then places the address of the object in the variable sampleVariable. We do not know what the address of the object is, but let's assume it is 2056. The exact number does not matter.



(continued)

Class Type Variables Store a Reference (Part 2 of 2)

Display 5.12 Class Type Variables Store a Reference



Assignment Operator with Class Type Variables

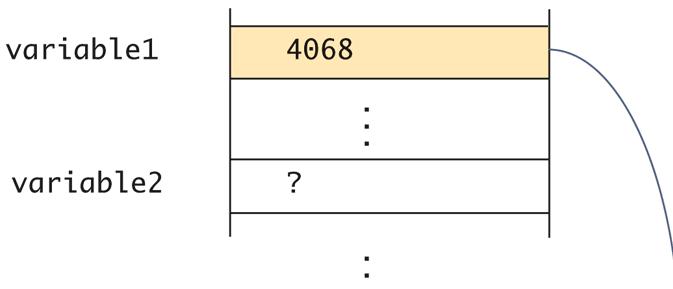
- Two reference variables can contain the same reference, and therefore name the same object
 - The assignment operator sets the reference (memory address) of one class type variable equal to that of another
 - Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they are the same object

```
variable2 = variable1;
```

Assignment Operator with Class Type Variables (Part 1 of 3)

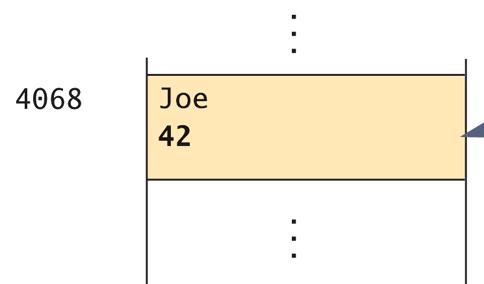
Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);  
ToyClass variable2;
```



We do not know what memory address (reference) is stored in the variable `variable1`. Let's say it is `4068`. The exact number does not matter.

Someplace else in memory:



Note that you can think of

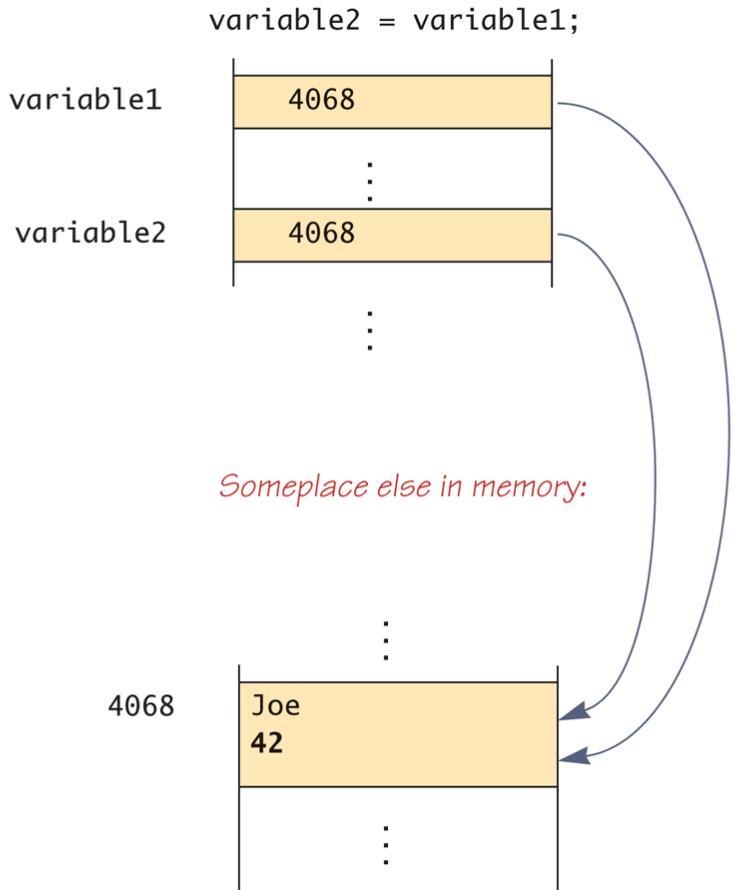
`new ToyClass("Joe", 42)`

as returning a reference.

(continued)

Assignment Operator with Class Type Variables (Part 2 of 3)

Display 5.13 Assignment Operator with Class Type Variables

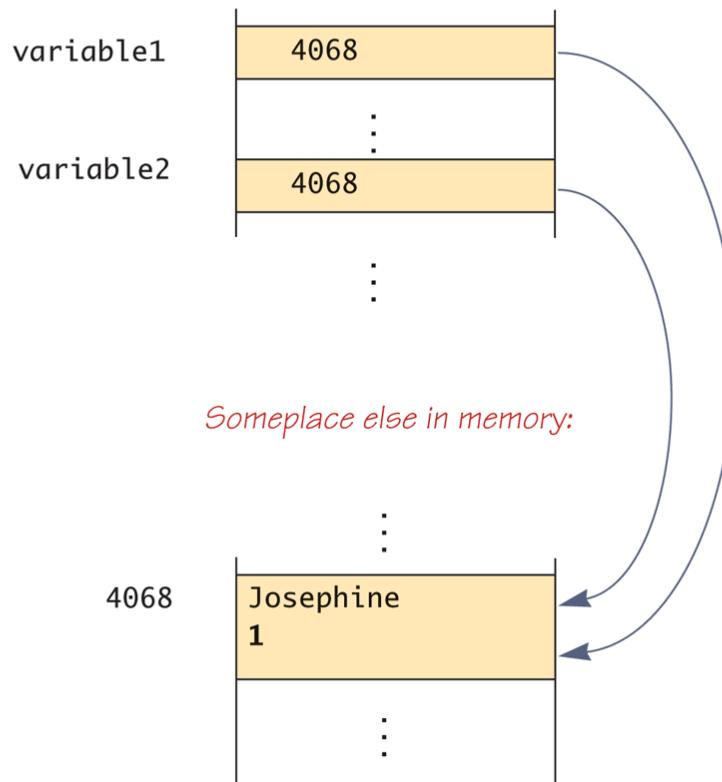


(continued)

Assignment Operator with Class Type Variables (Part 3 of 3)

Display 5.13 Assignment Operator with Class Type Variables

```
variable2.set("Josephine", 1);
```



The equals Method

- When the == operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
    System.out.println("The objects are the same.");
else
    System.out.println("The objects are not the same.");
```

- In above code segment only the addresses of the objects are compared.

The equals Method

- Java expects certain methods, such as **equals** to be in all, or almost all, classes
- The purpose of **equals**, a **boolean** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
 - Note: You cannot use **==** to compare objects

```
public boolean equals(ClassName objectName)
```

The equals Method

- Instead of using the == operator to compare two Stock objects, we should use the equals method.

```
public boolean equals(Stock object2)
{
    boolean status;

    if(symbol.equals(Object2.symbol) && sharePrice == Object2.sharePrice)
        status = true;
    else
        status = false;
    return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.

Methods That Copy Objects

- There are two ways to copy an object.
 - You cannot use the assignment operator to copy reference types
 - Reference only copy (shallow Copy)
 - This is simply copying the address of an object into another reference variable.

```
Stock stock1 = new Stock("GMX", 55.3);  
Stock stock2 = stock1;
```

Deep copy (correct)

- This involves creating a new instance of the class and copying the values from one object into the new object.

Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object1)
{
    if (object1 == null) //Not a real stock.
    {
        System.out.println("Fatal Error.");
        System.exit(0);
    }
    symbol = object1.symbol;
    sharePrice = object1.sharePrice;
}

// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);

//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```