



Classes 3/5



Outline

- Variables and Memory
- References
- Class Type Variables Store a Reference
- Assignment Operator with Class Type Variables
- Class Parameters
- Parameters of a Class Type
- Differences Between Primitive and Class-Type Parameters
- Comparing Parameters of a Class Type and a Primitive Type
- Example: A Toy Class
- The Constant `null`
- The `new` Operator and Anonymous Objects



- Variables and Memory ...

- A computer has two forms of memory
- *Secondary memory* is used to hold files for "permanent" storage
- *Main memory* is used by a computer when it is running a program
 - Values stored in a program's variables are kept in main memory



... - Variables and Memory ...

- Main memory consists of a long list of numbered locations called *bytes*
 - Each byte contains eight *bits*: eight 0 or 1 digits
- The number that identifies a byte is called its *address*
 - A data item can be stored in one (or more) of these bytes
 - The address of the byte is used to find the data item when needed

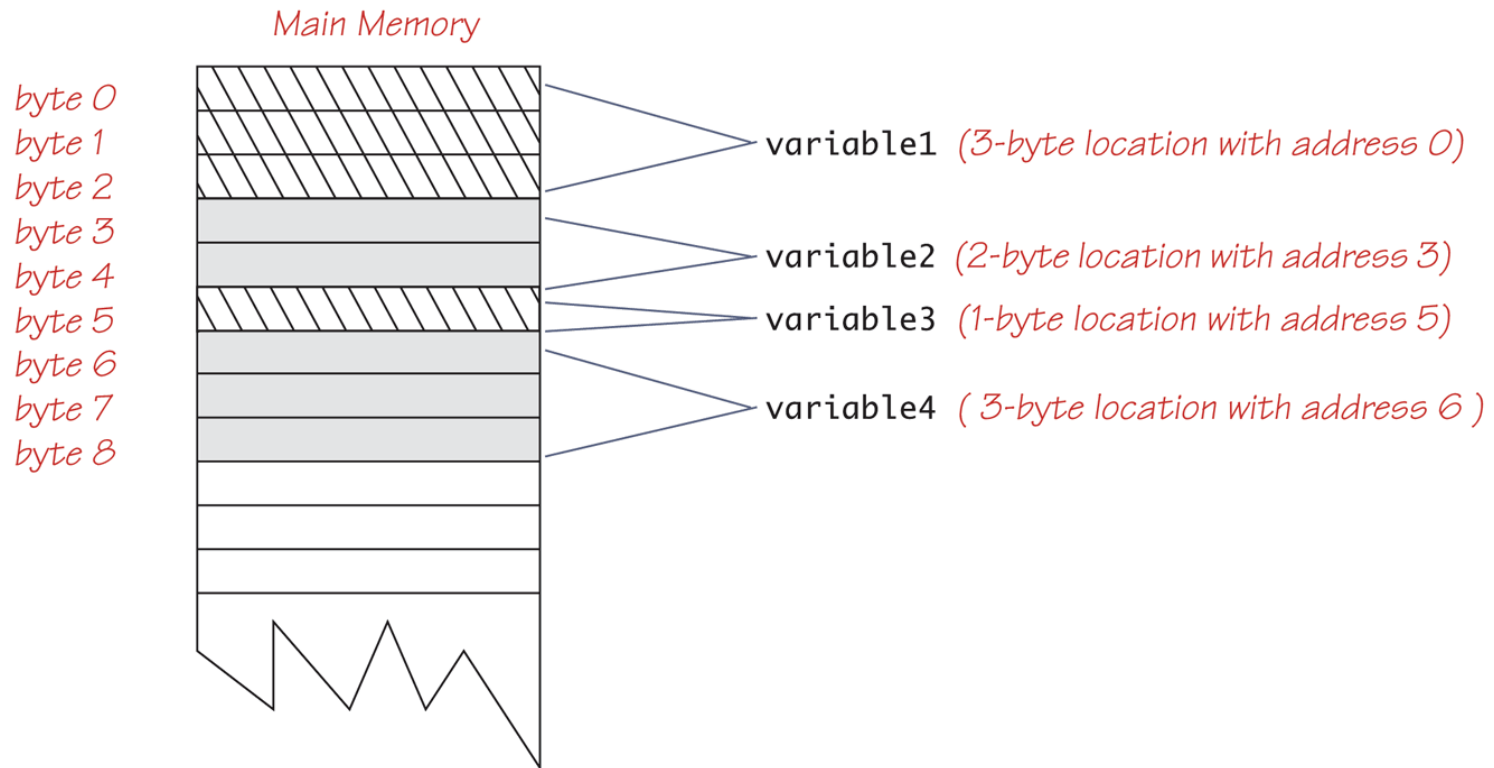


... - Variables and Memory ...

- 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



... - 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



... - References

- 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;

- Class Type Variables Store a Reference ...

Display 5.12 Class Type Variables Store a Reference

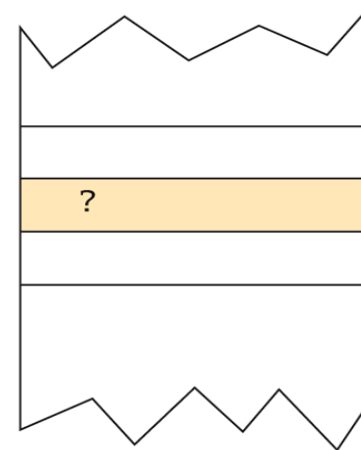
```
public class ToyClass
{
    private String name;
    private int number;
```

*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



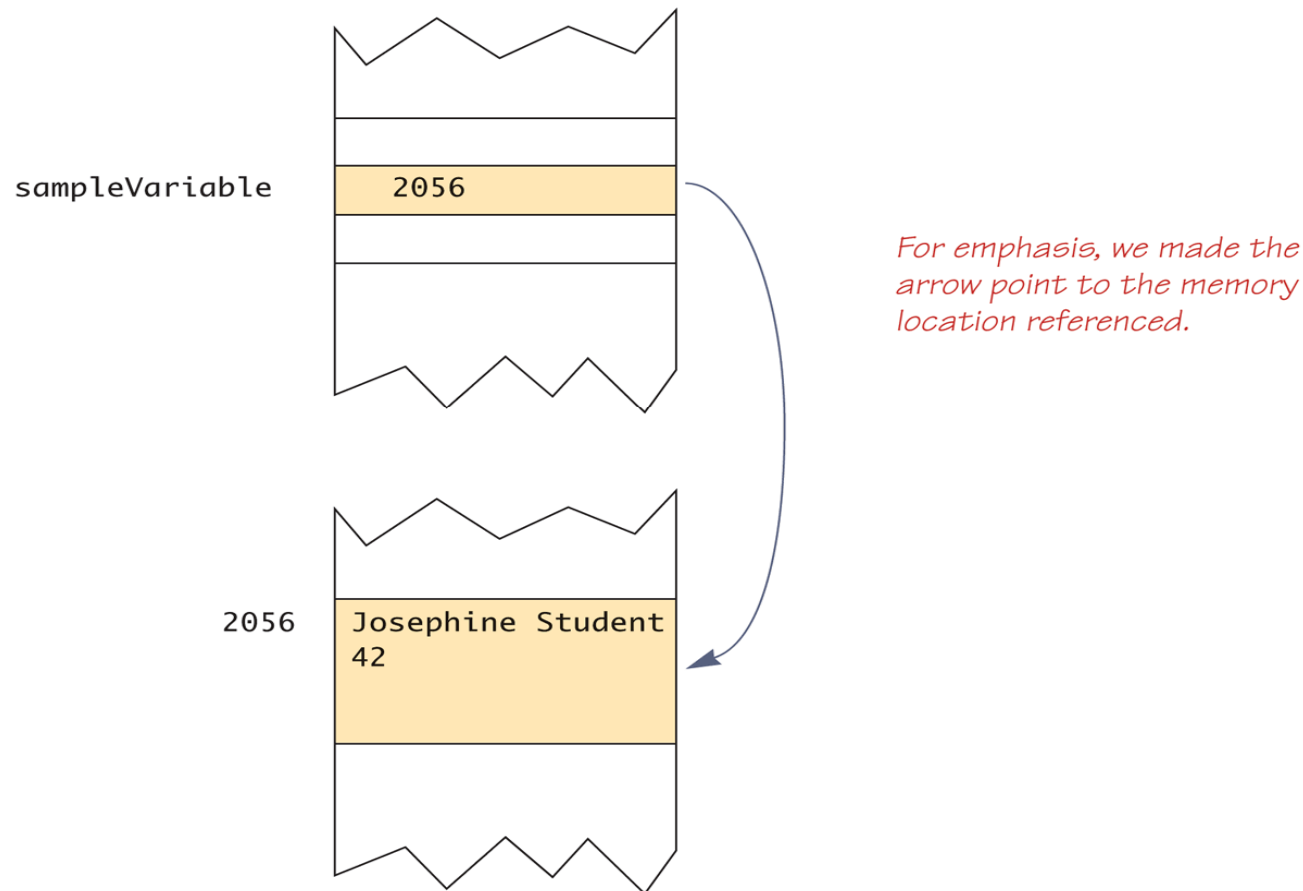
```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

*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

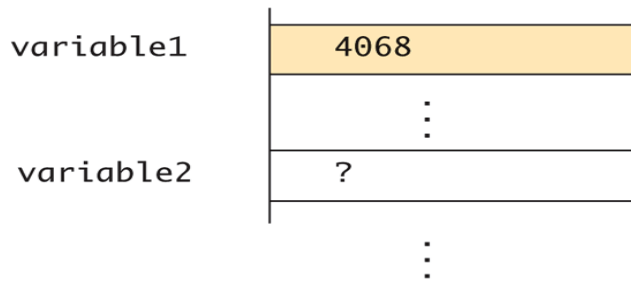
Display 5.12 Class Type Variables Store a Reference



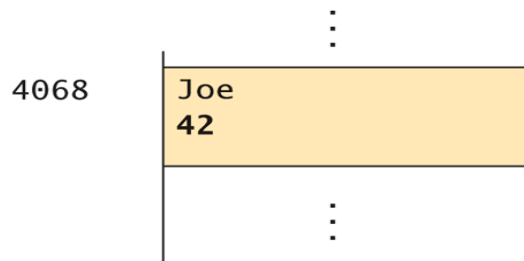
- Assignment Operator with Class Type Variables ...

Display 5.13 Assignment Operator with Class Type Variables

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



Someplace else in memory:



*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.*

Note that you can think of

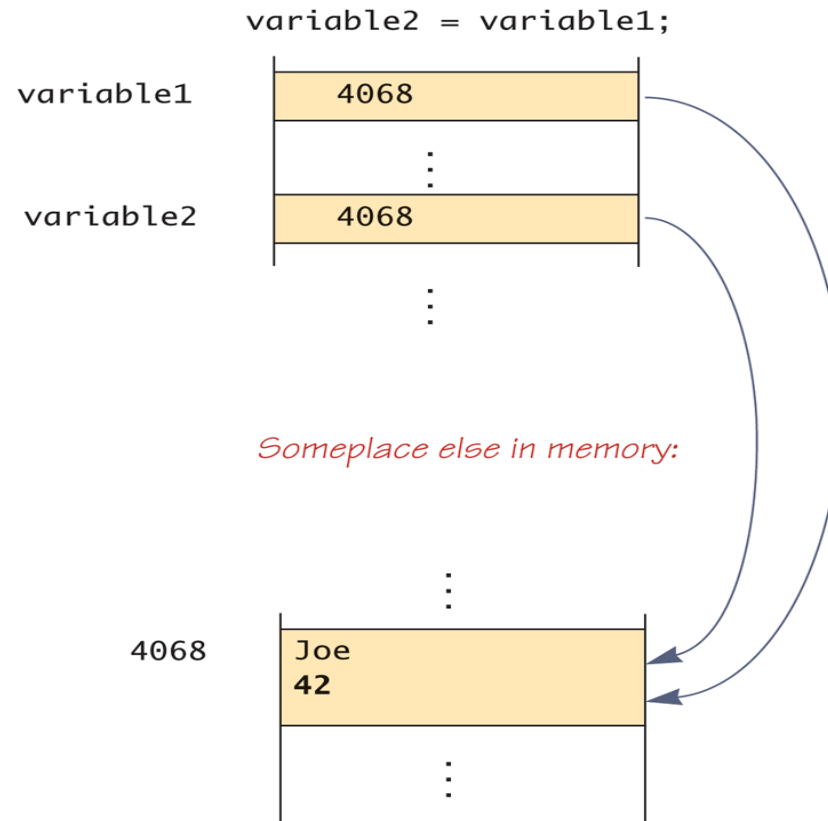
`new ToyClass("Joe", 42)`

as returning a reference.

(continued)

... - Assignment Operator with Class Type Variables ...

Display 5.13 Assignment Operator with Class Type Variables

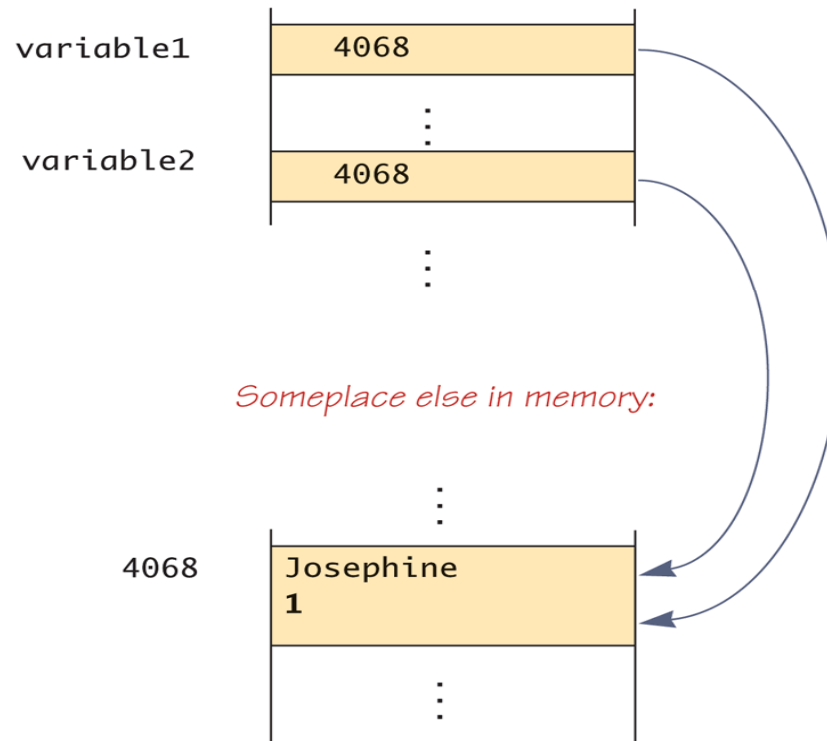


(continued)

... - Assignment Operator with Class Type Variables

Display 5.13 Assignment Operator with Class Type Variables

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





- Class Parameters ...

- All parameters in Java are *call-by-value* parameters
 - A parameter is a *local variable* that is set equal to the value of its argument
 - Therefore, any change to the value of the parameter cannot change the value of its argument
- Class type parameters appear to behave differently from primitive type parameters
 - They appear to behave in a way similar to parameters in languages that have the *call-by-reference* parameter passing mechanism



... - Class Parameters

- The value plugged into a class type parameter is a reference (memory address)
 - Therefore, the parameter becomes another name for the argument
 - Any change made to the object named by the parameter (i.e., changes made to the values of its instance variables) will be made to the object named by the argument, because they are the same object
 - Note that, because it still is a call-by-value parameter, any change made to the class type parameter itself (i.e., its address) will not change its argument (the reference or memory address)



- Parameters of a Class Type

Display 5.14 Parameters of a Class Type

```
1  public class ClassParameterDemo
2  {
3      public static void main(String[] args)
4      {
5          ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
6          System.out.println(anObject);
7          System.out.println(
8              "Now we call changer with anObject as argument.");
9          ToyClass.changer(anObject);
10         System.out.println(anObject);
11     }
12 }
```

ToyClass is defined in Display 5.11.

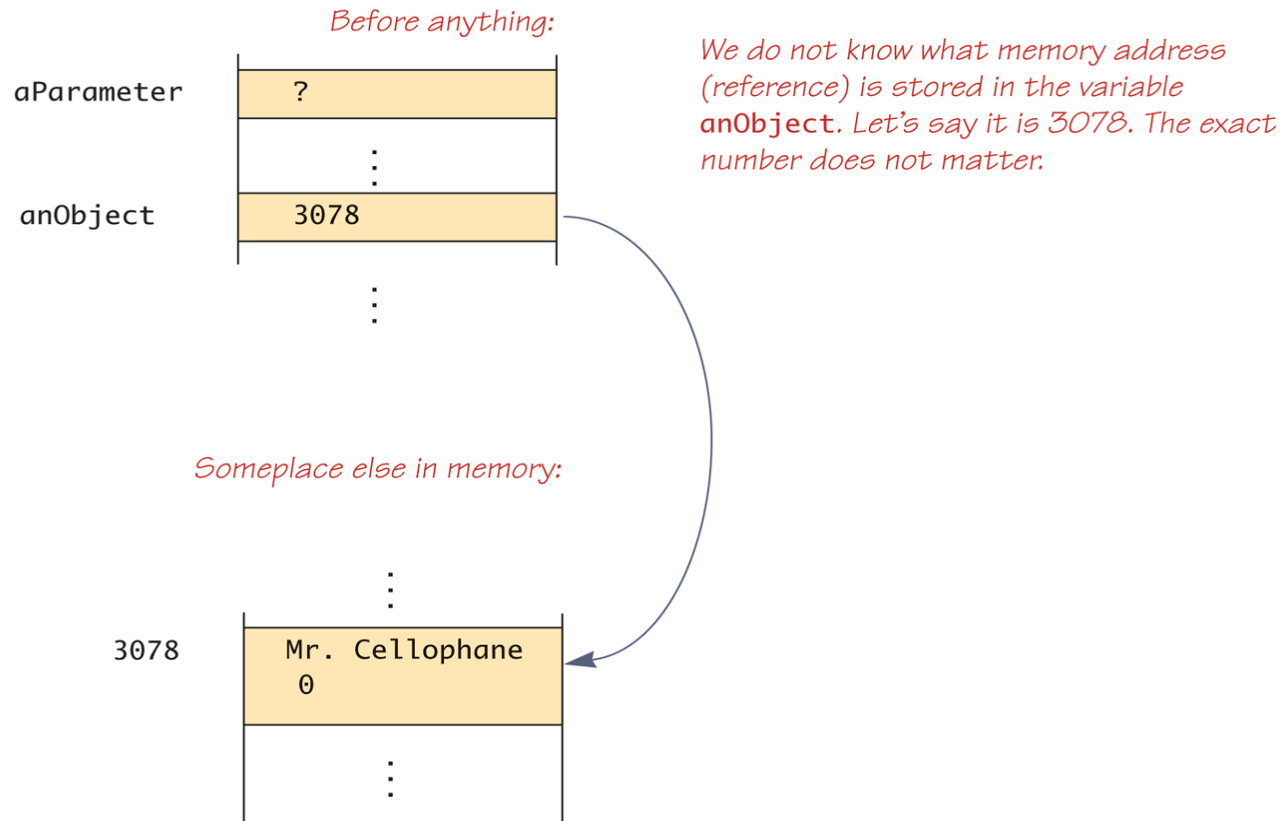
Notice that the method changer changed the instance variables in the object anObject.

SAMPLE DIALOGUE

```
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
```

- Memory Picture for Display 5.14 ...

Display 5.15 Memory Picture for Display 5.14

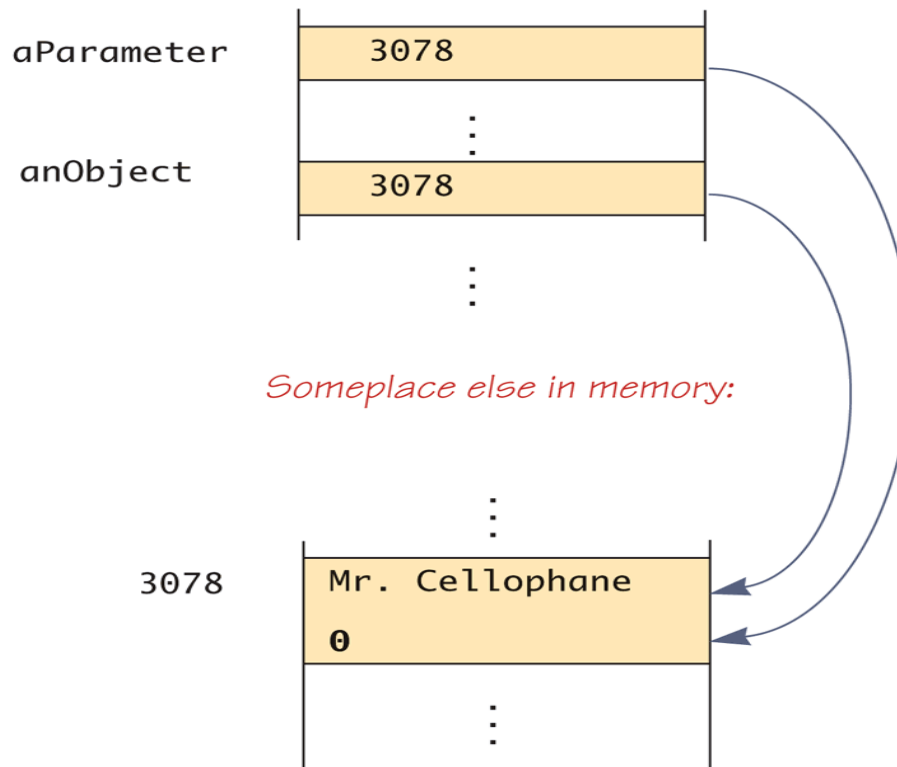


(continued)

... - Memory Picture for Display 5.14 ...

Display 5.15 Memory Picture for Display 5.14

*anObject is plugged in for aParameter .
anObject and aParameter become two names for the same object.*

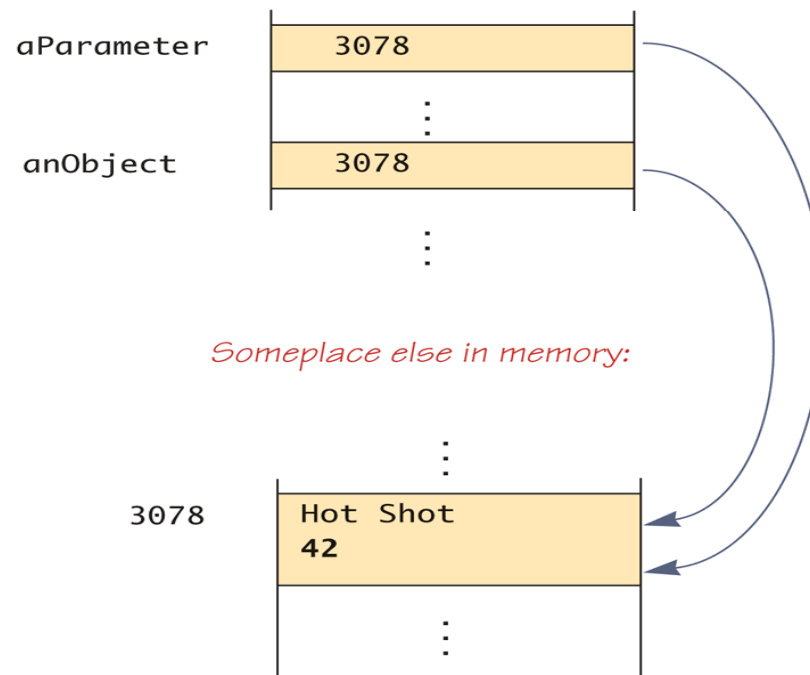


(continued)

... - Memory Picture for Display 5.14

Display 5.15 Memory Picture for Display 5.14

*ToyClass.changer(anObject); is executed
and so the following are executed:
 aParameter.name = "Hot Shot";
 aParameter.number = 42;
As a result, anObject is changed.*





- Differences Between Primitive and Class-Type Parameters

- A method cannot change the value of a variable of a primitive type that is an argument to the method
- In contrast, a method can change the values of the instance variables of a class type that is an argument to the method



- Comparing Parameters of a Class Type and a Primitive Type ...

Display 5.16 Comparing Parameters of a Class Type and a Primitive Type

```
1  public class ParametersDemo
2  {
3      public static void main(String[] args)
4      {
5          ToyClass2 object1 = new ToyClass2(),
6              object2 = new ToyClass2();
7          object1.set("Scorpius", 1);
8          object2.set("John Crichton", 2);
9          System.out.println("Value of object2 before call to method:");
10         System.out.println(object2);
11         object1.makeEqual(object2);
12         System.out.println("Value of object2 after call to method:");
13         System.out.println(object2);
14
15         int aNumber = 42;
16         System.out.println("Value of aNumber before call to method: "
17             + aNumber);
18         object1.tryToMakeEqual(aNumber);
19         System.out.println("Value of aNumber after call to method: "
20             + aNumber);
21     }
22 }
```

*ToyClass2 is defined in
Display 5.17.*

(continued)



... - Comparing Parameters of a Class Type and a Primitive Type

Display 5.16 Comparing Parameters of a Class Type and a Primitive Type

SAMPLE DIALOGUE

Value of object2 before call to method:

John Crichton 2

Value of object2 after call to method:

Scorpius 1

Value of aNumber before call to method: 42

Value of aNumber after call to method: 42

*An argument of a class type
can change.*

*An argument of a primitive
type cannot change.*



- A Toy Class to Use in Display 5.16 ...

Display 5.17 A Toy Class to Use in Display 5.16

```
1  public class ToyClass2
2  {
3      private String name;
4      private int number;

5      public void set(String newName, int newNumber)
6      {
7          name = newName;
8          number = newNumber;
9      }

10     public String toString()
11     {
12         return (name + " " + number);
13     }
```

(continued)



... - A Toy Class to Use in Display 5.16


Display 5.17 A Toy Class to Use in Display 5.16

```
14     public void makeEqual(ToyClass2 anObject)
15     {
16         anObject.name = this.name;
17         anObject.number = this.number;
18     }

19     public void tryToMakeEqual(int aNumber)
20     {
21         aNumber = this.number;
22     }

23     public boolean equals(ToyClass2 otherObject)
24     {
25         return ( (name.equals(otherObject.name))
26                 && (number == otherObject.number) );
27     }
```

Read the text for a discussion of the problem with this method.



<Other methods can be the same as in Display 5.11, although no other methods are needed or used in the current discussion.>

```
28 }
29
```



Pitfall: Use of = and == with Variables of a Class Type

- Used with variables of a class type, the assignment operator (=) produces two variables that name the same object
 - This is very different from how it behaves with primitive type variables
- The test for equality (==) also behaves differently for class type variables
 - The == operator only checks that two class type variables have the same memory address
 - Unlike the **equals** method, it does not check that their instance variables have the same values
 - Two objects in two different locations whose instance variables have exactly the same values would still test as being "not equal"



- The Constant `null`

- `null` is a special constant that may be assigned to a variable of any class type

`YourClass yourObject = null;`

- It is used to indicate that the variable has no "real value"
 - It is often used in constructors to initialize class type instance variables when there is no obvious object to use
- `null` is not an object: It is, rather, a kind of "placeholder" for a reference that does not name any memory location
 - Because it is like a memory address, use `==` or `!=` (instead of `equals`) to test if a class variable contains null

`if (yourObject == null) . . .`



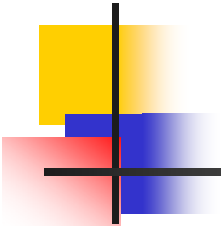
Pitfall: Null Pointer Exception

- Even though a class variable can be initialized to `null`, this does not mean that `null` is an object
 - `null` is only a placeholder for an object
- A method cannot be invoked using a variable that is initialized to `null`
 - The calling object that must invoke a method does not exist
- Any attempt to do this will result in a "Null Pointer Exception" error message
 - For example, if the class variable has not been initialized at all (and is not assigned to `null`), the results will be the same



- The **new** Operator and Anonymous Objects

- The **new** operator invokes a constructor which initializes an object, and returns a reference to the location in memory of the object created
 - This reference can be assigned to a variable of the object's class type
- Sometimes the object created is used as an argument to a method, and never used again
 - In this case, the object need not be assigned to a variable, i.e., given a name
- An object whose reference is not assigned to a variable is called an **anonymous object**



THE END