

# Chapter 5

## Defining Classes II

### (Part 2)

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering  
Hanyang University

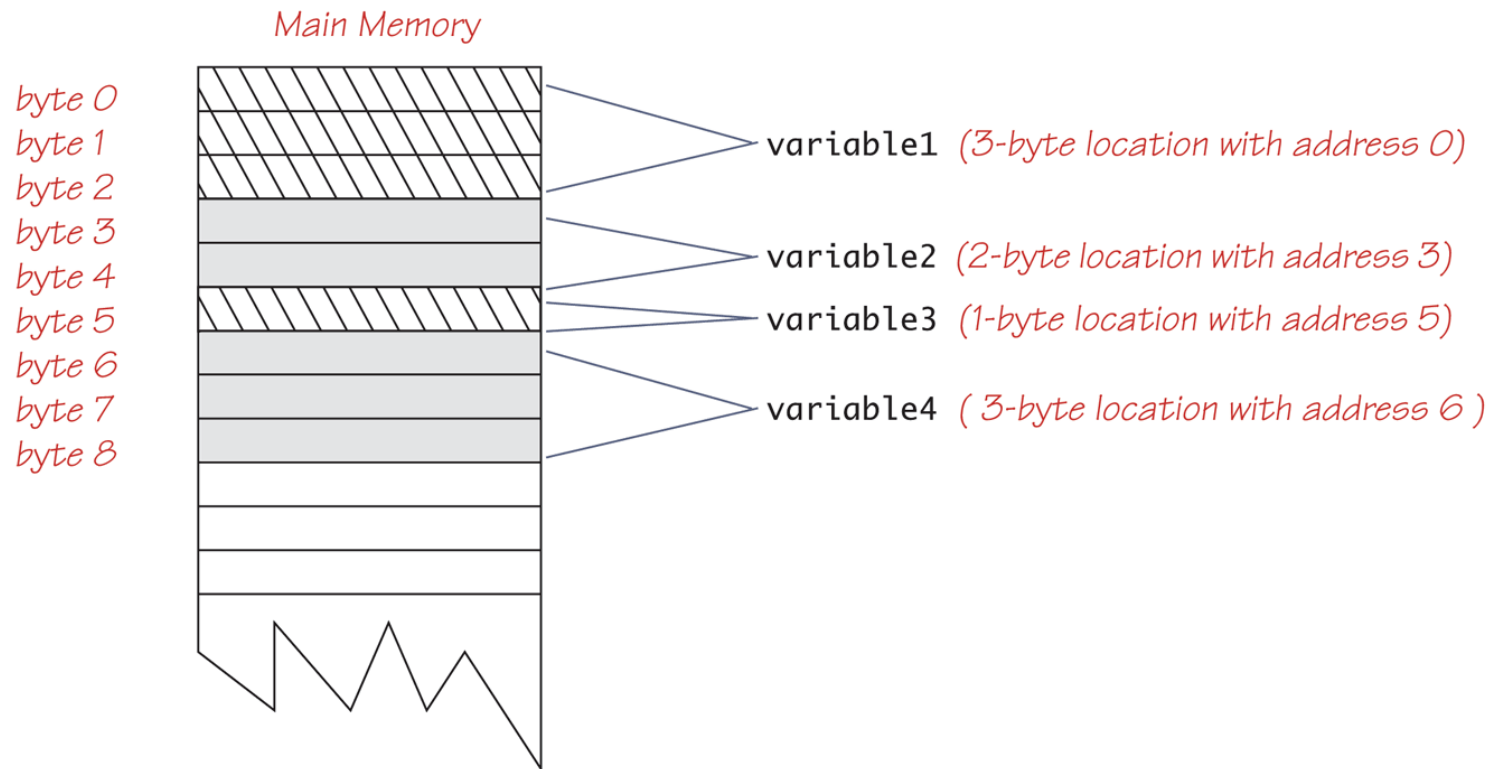
# Variables and Memory

- 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;
```

# Display 5.11 A Simple Class

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

    public ToyClass(String initialName, int initialNumber)
    {
        name = initialName;
        number = initialNumber;
    }

    public ToyClass()
    {
        name = "No name yet.";
        number = 0;
    }

    public void set(String newName, int newNumber)
    {
        name = newName;
        number = newNumber;
    }
}
```

```
    public String toString()
    {
        return (name + " " + number);
    }

    public static void changer(ToyClass aParameter)
    {
        aParameter.name = "Hot Shot";
        aParameter.number = 42;
    }

    public boolean equals(ToyClass otherObject)
    {
        return ( (name.equals(otherObject.name))
                && (number == otherObject.number) );
    }
}
```

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

## Display 5.12 Class Type Variables Store a Reference

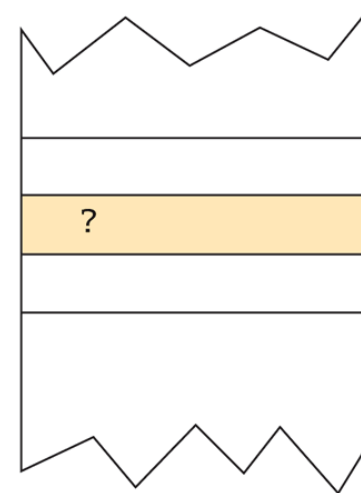
```
public class ToyClass
{
    private String name;
    private int number;
```

*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.*

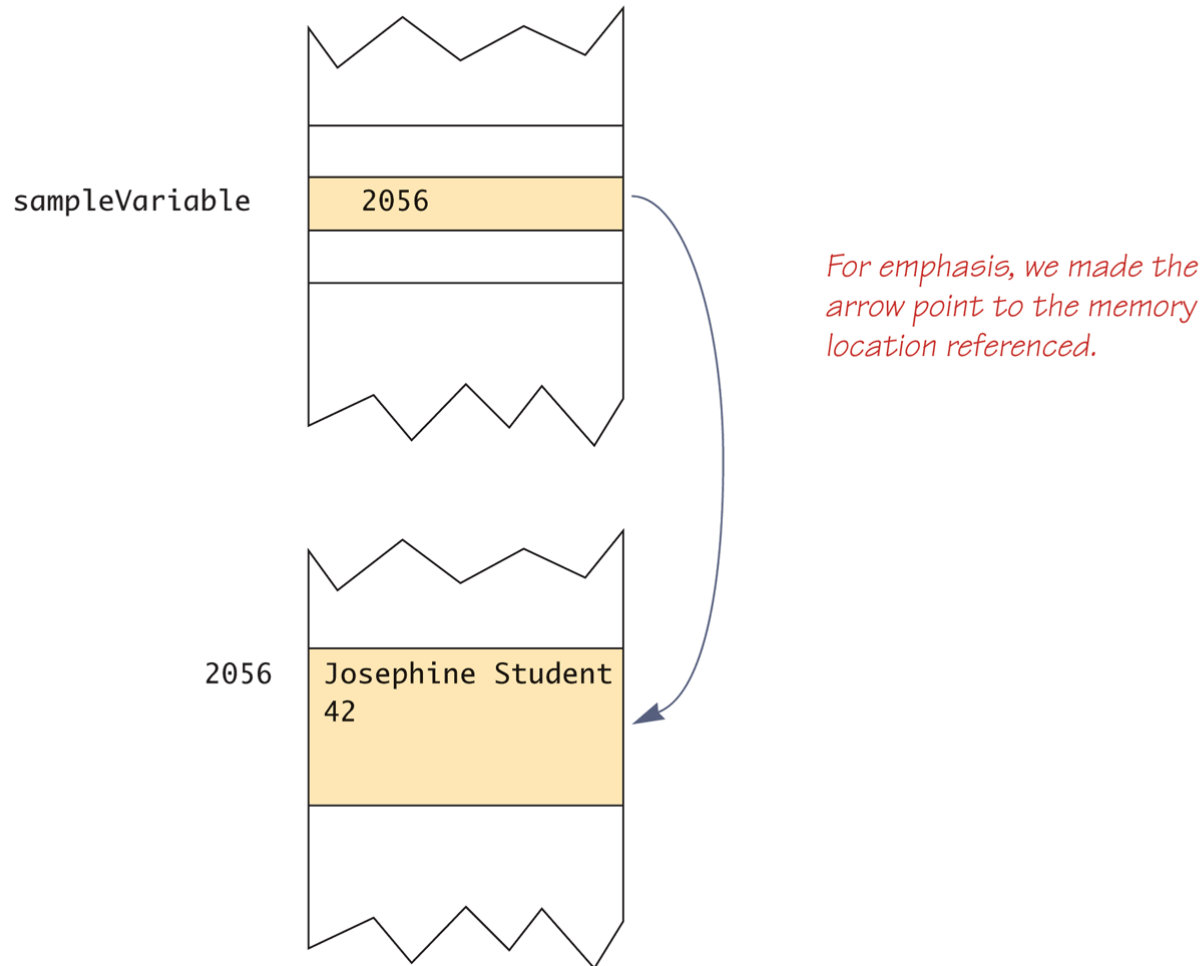
(continued)



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

**Display 5.12** Class Type Variables Store a Reference

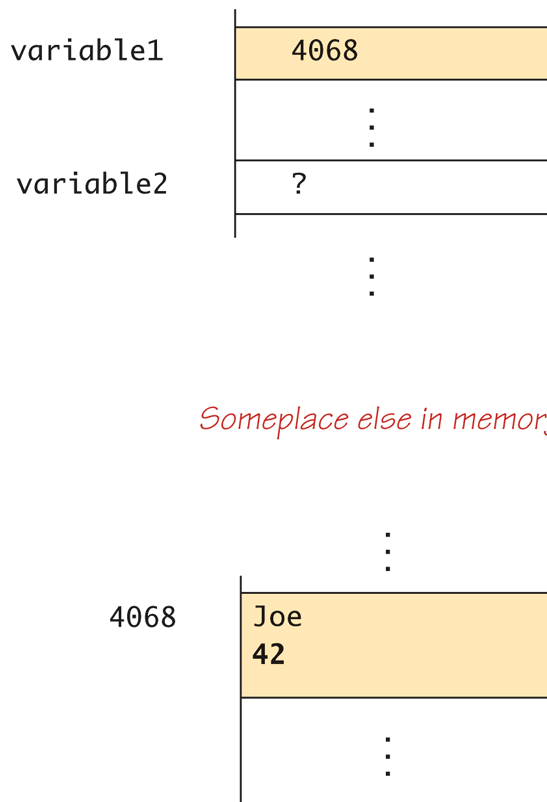
---



# Assignment Operator with Class Type Variables (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.*

*Note that you can think of*

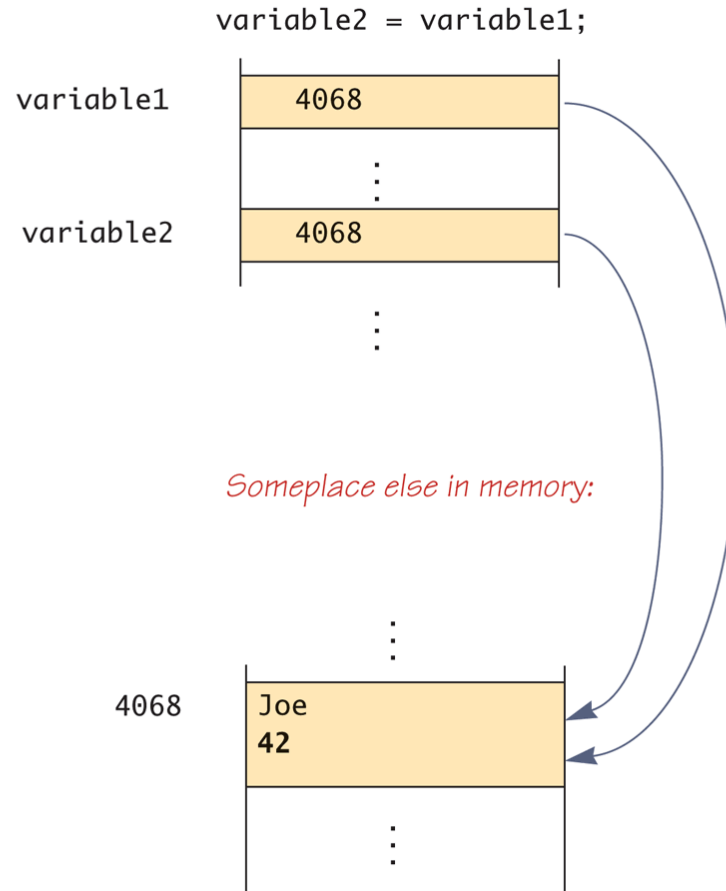
```
new ToyClass("Joe", 42)
```

*as returning a reference.*

(continued)

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

**Display 5.13** Assignment Operator with Class Type Variables

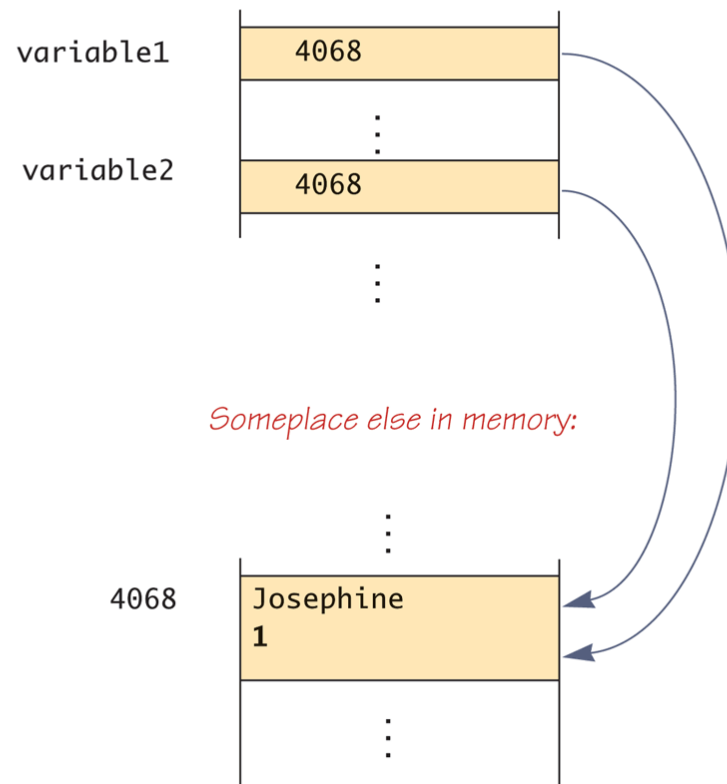


(continued)

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

**Display 5.13** Assignment Operator with Class Type Variables

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



# 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**.*



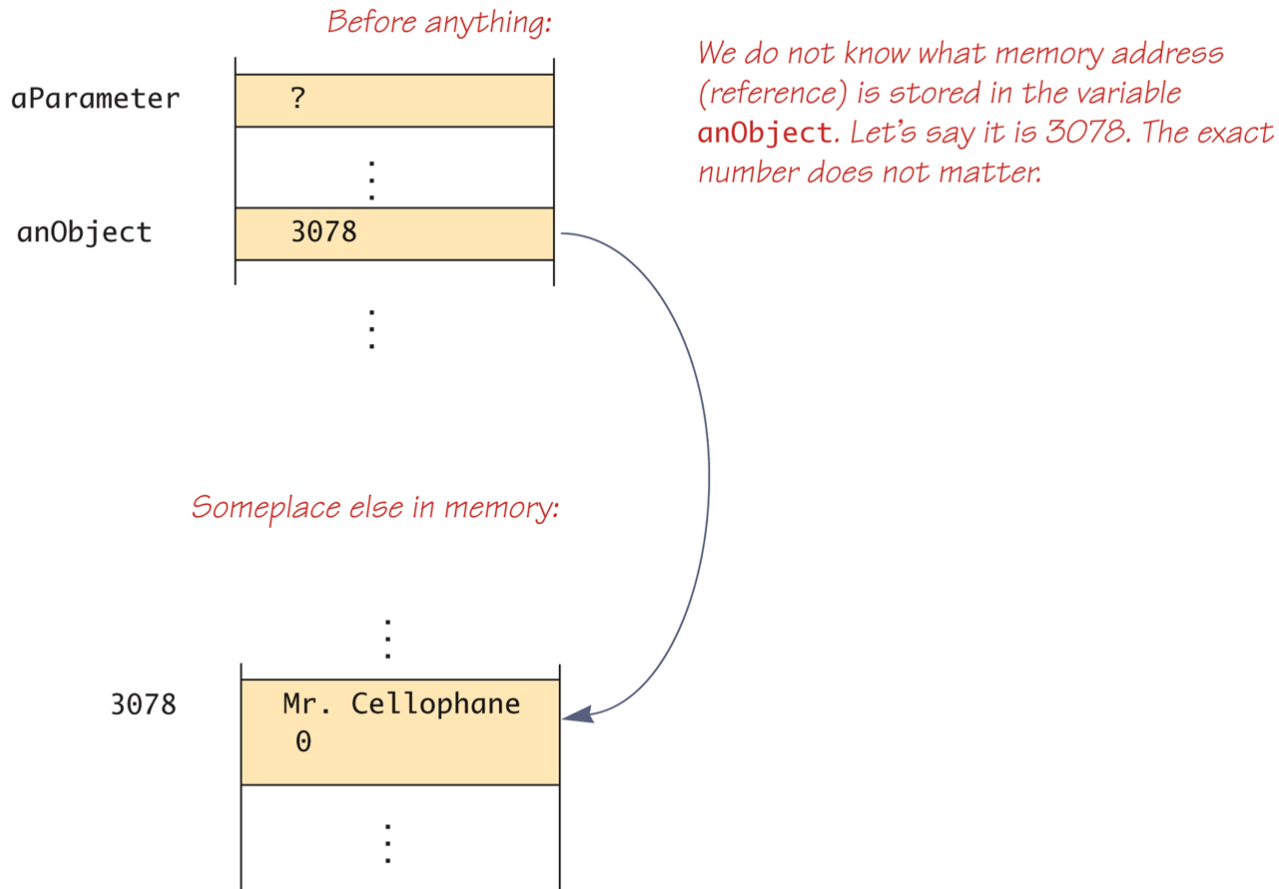
### SAMPLE DIALOGUE

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

# Memory Picture for Display 5.14

## (Part 1 of 3)

**Display 5.15** Memory Picture for Display 5.14



(continued)



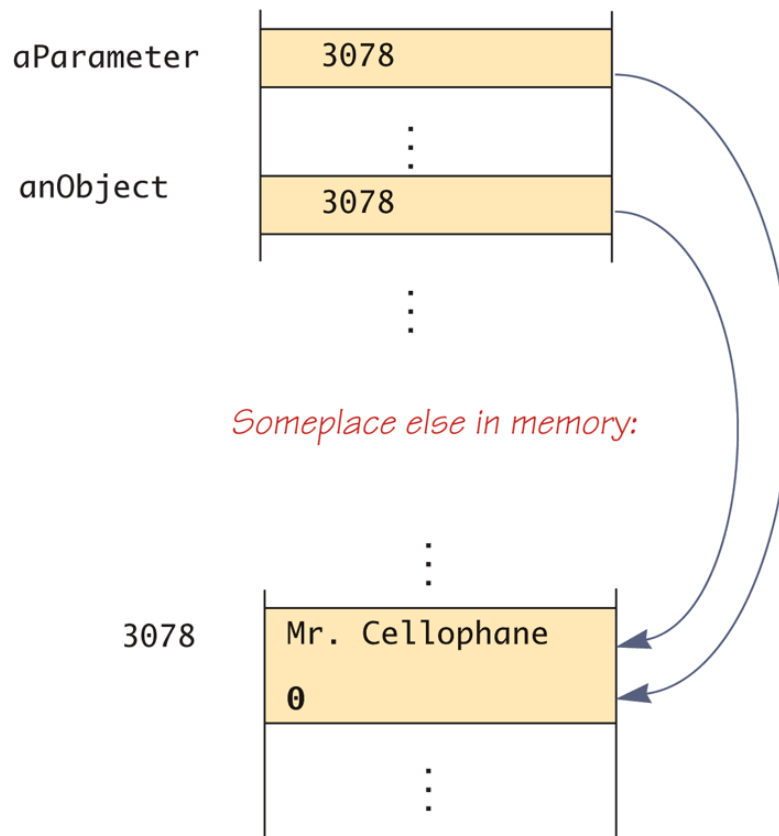
# Memory Picture for Display 5.14

## (Part 2 of 3)

**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.*



*Someplace else in memory:*

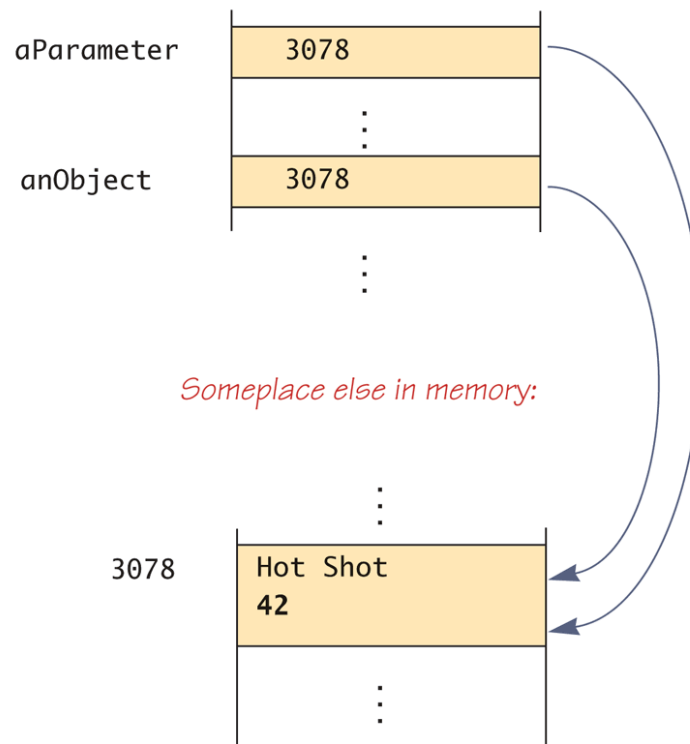
(continued)

# Memory Picture for Display 5.14

## (Part 3 of 3)

**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)

# A Toy Class to Use in Display 5.16

## (Part 1 of 2)

### 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

## (Part 2 of 2)

### 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
```

# Comparing Parameters of a Class Type and a Primitive Type (Part 2 of 2)

## 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.*

# 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)    . . .
```

# 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

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

# The **new** Operator and Anonymous Objects

- 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

```
if (variable1.equals(new ToyClass("Joe", 42)))  
    System.out.println("Equal");  
else  
    System.out.println("Not equal");
```

- An object whose reference is not assigned to a variable is called an **anonymous object**

# Using and Misusing References

- When writing a program, it is very important to insure that private instance variables remain truly private
- For a primitive type instance variable, just adding the **private** modifier to its declaration should insure that there will be no *privacy leaks*
- For a class type instance variable, however, adding the **private** modifier alone is not sufficient

# Designing A **Person** Class: Instance Variables

- A simple **Person** class could contain instance variables representing a person's name, the date on which they were born, and the date on which they died
- These instance variables would all be class types: name of type **String**, and two dates of type **Date**
- As a first line of defense for privacy, each of the instance variables would be declared **private**

```
public class Person
{
    private String name;
    private Date born;
    private Date died;    //null is still alive
    . . .
}
```

# Copy Constructors

- A *copy constructor* is a constructor with a single argument of the same type as the class
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object
- Note how, in the **Date** copy constructor, the values of all of the primitive type private instance variables are merely copied

# Copy Constructor for a Class with Primitive Type Instance Variables

```
public Date(Date aDate)
{
    if (aDate == null) //Not a real date.
    {
        System.out.println("Fatal Error.");
        System.exit(0);
    }

    month = aDate.month;
    day = aDate.day;
    year = aDate.year;
}
```

# Copy Constructor for a Class with Class Type Instance Variables

- Unlike the **Date** class, the **Person** class contains three class type instance variables
- If the **born** and **died** class type instance variables for the new **Person** object were merely copied, then they would simply rename the **born** and **died** variables from the original **Person** object

```
born = original.born //dangerous
```

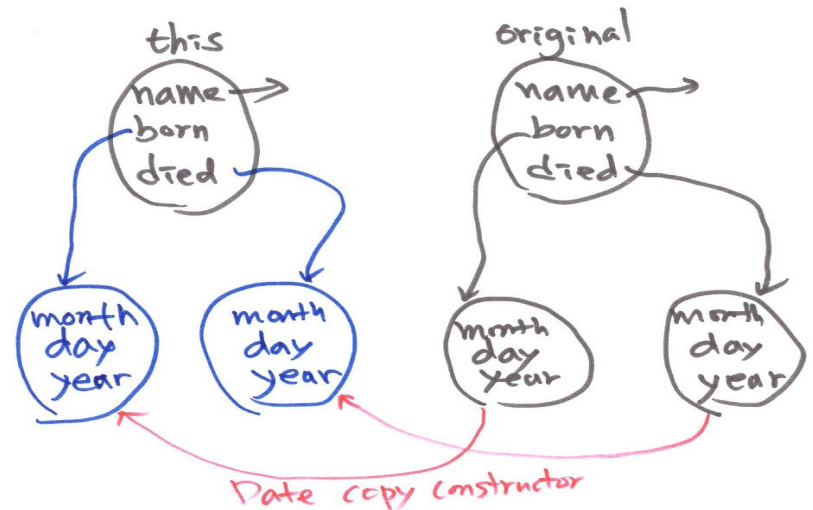
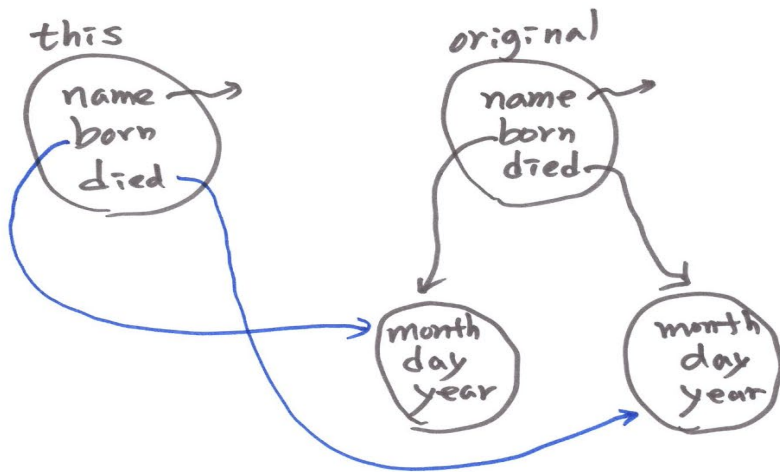
```
died = original.died //dangerous
```

- This would not create an independent copy of the original object



# Copy Constructor for a Class with Class Type Instance Variables

```
public Person(Person original)
{
    ...
    born = original.born;
    died = original.died;
    ...
}
```



# Copy Constructor for a Class with Class Type Instance Variables

- The actual copy constructor for the **Person** class is a "safe" version that creates completely new and independent copies of **born** and **died**, and therefore, a completely new and independent copy of the original **Person** object
  - For example:  
`born = new Date(original.born) ;`
- Note that in order to define a correct copy constructor for a class that has class type instance variables, copy constructors must already be defined for the instance variables' classes

# Copy Constructor for a Class with Class Type Instance Variables

```
public Person(Person original)
{
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = new Date(original.born);    // ? born = original.born;
    if (original.died == null)
        died = null;
    else
        died = new Date(original.died); // ? Died = original.died;
}
```

# Pitfall: Privacy Leaks

- The previously illustrated examples from the **Person** class show how an incorrect definition of a constructor can result in a *privacy leak*
- A similar problem can occur with incorrectly defined mutator or accessor methods

- For example:

```
public Date getBirthDate()  
{  
    return born;    //dangerous  
}
```

- Instead of:

```
public Date getBirthDate()  
{  
    return new Date(born);    //correct  
}
```

# Mutable and Immutable Classes

- The accessor method `getName` from the `Person` class appears to contradict the rules for avoiding privacy leaks:

```
public String getName()  
{  
    return name; //Isn't this dangerous?  
}
```
- Although it appears the same as some of the previous examples, it is not: The class `String` contains no mutator methods that can change any of the data in a `String` object

# Mutable and Immutable Classes

- A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an *immutable class*
  - Objects of such a class are called *immutable objects*
  - It is perfectly safe to return a reference to an immutable object because the object cannot be changed in any way
  - The **String** class is an immutable class

# Mutable and Immutable Classes

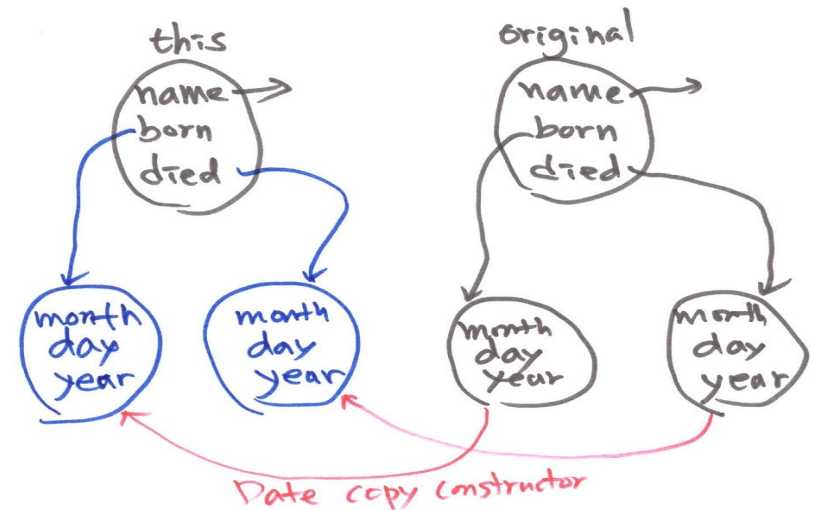
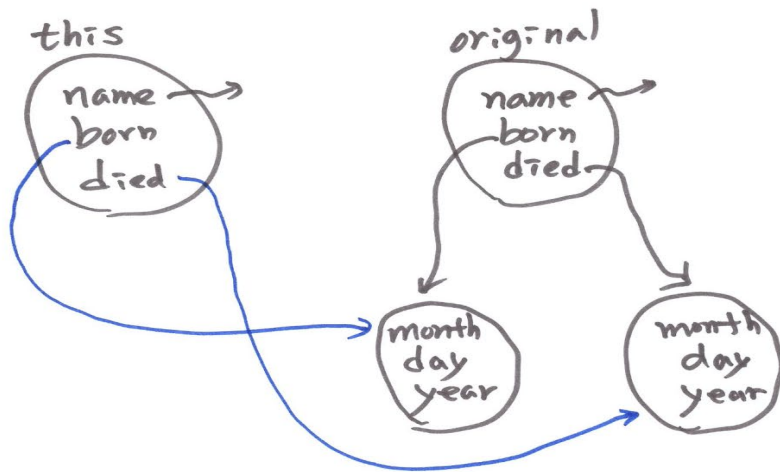
- A class that contains public mutator methods or other public methods that can change the data in its objects is called a *mutable class*, and its objects are called *mutable objects*
  - Never write a method that returns a mutable object
  - Instead, use a copy constructor to return a reference to a completely independent copy of the mutable object

# Deep Copy Versus Shallow Copy

- A *deep copy* of an object is a copy that, with one exception, has no references in common with the original
  - Exception: References to immutable objects are allowed to be shared
- Any copy that is not a deep copy is called a *shallow copy*
  - This type of copy can cause dangerous privacy leaks in a program



# Deep Copy Versus Shallow Copy



# Deep Copy Versus Shallow Copy

```
public Date getBirthDate()  
{  
    return new Date(born);  
}
```

vs.

```
public Date getBirthDate()  
{  
    return born;  
}
```

# Packages and Import Statements

- Java uses *packages* to form libraries of classes
- A package is a group of classes that have been placed in a directory or folder, and that can be used in any program that includes an *import statement* that names the package
  - The import statement must be located at the beginning of the program file: Only blank lines, comments, and package statements may precede it
  - The program can be in a different directory from the package

# Import Statements

- We have already used import statements to include some predefined packages in Java, such as **Scanner** from the **java.util** package  
`import java.util.Scanner;`
- It is possible to make all the classes in a package available instead of just one class:  
`import java.util.*;`
  - Note that there is no additional overhead for importing the entire package

# The Package `java.lang`

- The package `java.lang` contains the classes that are fundamental to Java programming
  - It is imported automatically, so no import statement is needed
  - Classes made available by `java.lang` include `Math`, `String`, and the wrapper classes

# The **package** Statement

- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each class file:

**package package\_name;**

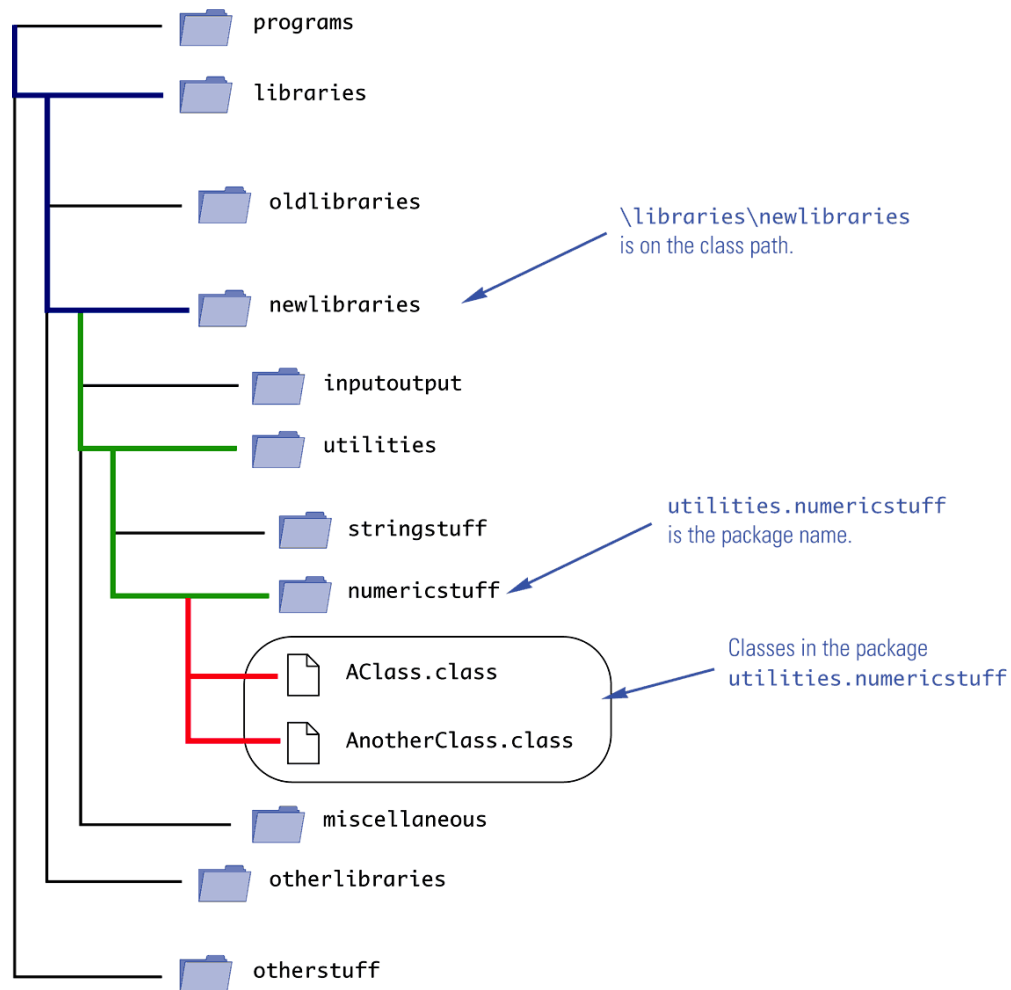
- Only the **.class** files must be in the directory or folder, the **.java** files are optional
- Only blank lines and comments may precede the package statement
- If there are both import and package statements, the package statement must precede any import statements

# Package Names and Directories

- A package name is the path name for the directory or subdirectories that contain the package classes
- Java needs two things to find the directory for a package: the name of the package and the value of the **CLASSPATH** variable
  - The **CLASSPATH** environment variable is similar to the **PATH** variable, and is set in the same way for a given operating system
  - The **CLASSPATH** variable is set equal to the list of directories (including the current directory, ".") in which Java will look for packages on a particular computer
  - Java searches this list of directories in order, and uses the first directory on the list in which the package is found

# A Package Name

Display 5.14 A Package Name





# The Default Package

- All the classes in the current directory belong to an unnamed package called the *default package*
- As long as the current directory ( `.` ) is part of the **CLASSPATH** variable, all the classes in the default package are automatically available to a program

# Name Clashes

- In addition to keeping class libraries organized, packages provide a way to deal with *name clashes*: a situation in which two classes have the same name
  - Different programmers writing different packages may use the same name for one or more of their classes
  - This ambiguity can be resolved by using the *fully qualified name* (i.e., precede the class name by its package name) to distinguish between each class

**package\_name.ClassName**

- A package named **sallyspack** contains a class called **HighClass**, and another package named **joespack** contains a class named **HighClass**.

```
sallyspack.HighClass object1 = new sallyspack.HighClass();  
joespack.HighClass object2 = new joespack.HighClass();
```

- If the fully qualified name is used, it is no longer necessary to import the class (because it includes the package name already)