# Lecture: Reference and Class parameters

## Acknowledgement

Most of this content is (c) Pearson Addison-Wesley.

It has been transferred from static slides to Ed Stem by Lachlan Andrew, with minor changes.

## Storing data

It is important to understand how computers store data, such as Java variables.

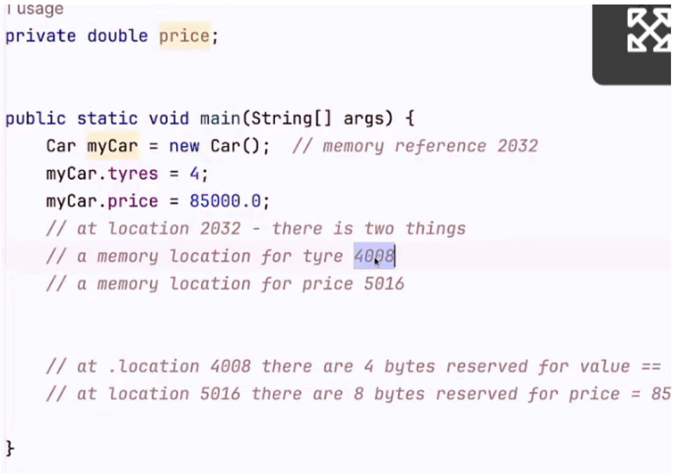
A computer has two forms of memory:

- Secondary memory (typically a hard disk, or flash drive like SSD) is used to hold files for "permanent" storage
- Main memory (a.k.a. Primary memory, typically RAM) is used by a computer when it is running a program.

Values stored in a program's variables are kept in main memory.

## Advanced: Virtual memory

"Virtual memory" allows the operating system allows some data from "main memory" to be stored on the hard drive instead of in RAM.  That is one reason we talk about main and secondary memory, instead of RAM and hard drive.  Main memory includes RAM plus virtual memory "swap space".



```
1 usage
private double price;


public static void main(String[] args) {
    Car myCar = new Car();  // memory reference 2032
    myCar.tyres = 4;
    myCar.price = 85000.0;
    // at location 2032 - there is two things
    // a memory location for tyre 4008
    // a memory location for price 5016


    // at .location 4008 there are 4 bytes reserved for value ==
    // at location 5016 there are 8 bytes reserved for price = 85

}
```

# Variables and memory

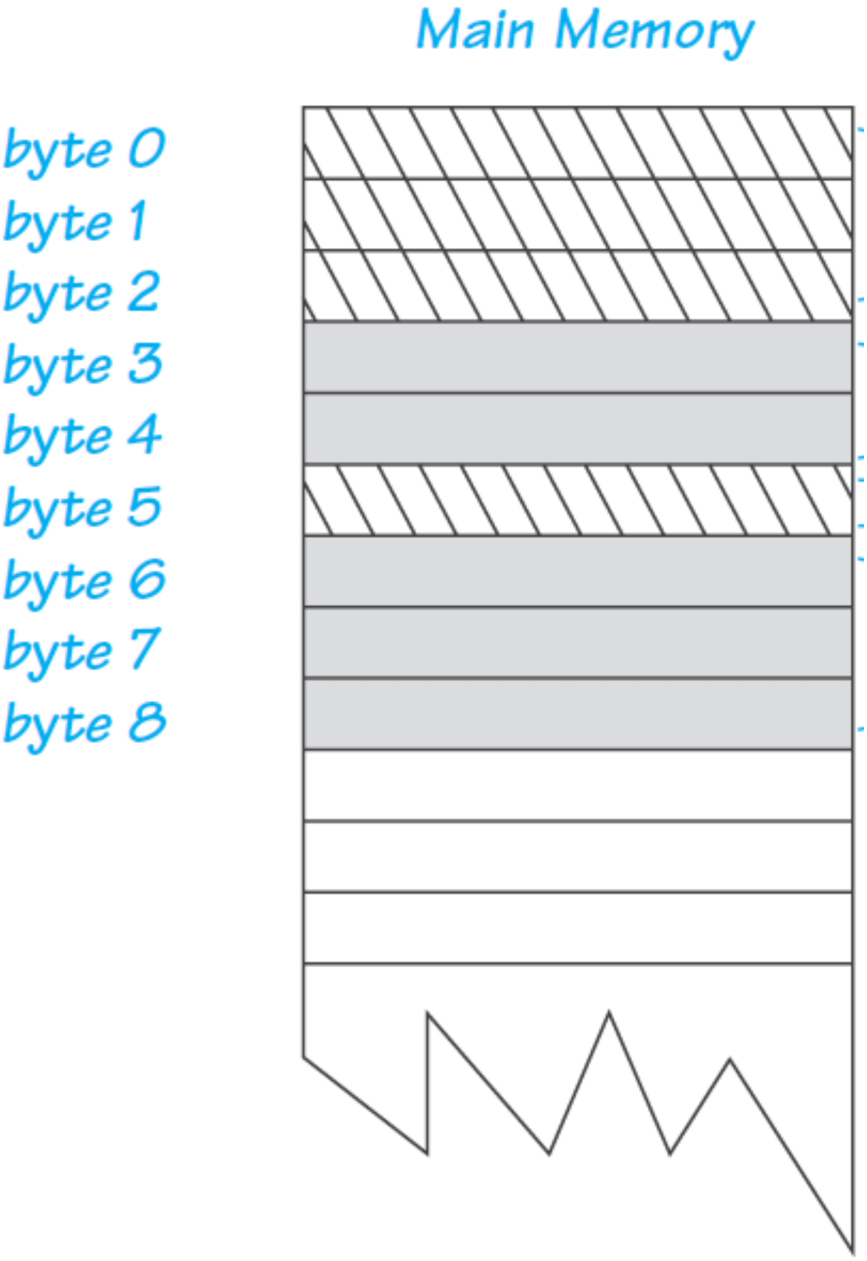
Main memory consists of a long list of numbered locations, each of which stores a *byte* (an integer from 0 to 255)

A byte contains eight ***b***inary dig***its***, called *bits*, that can each be either 0 or 1.

Each byte location in memory has an *address*, which is an integer that identifies the location.  On a 64-bit machine, it is typically a 64-bit number.

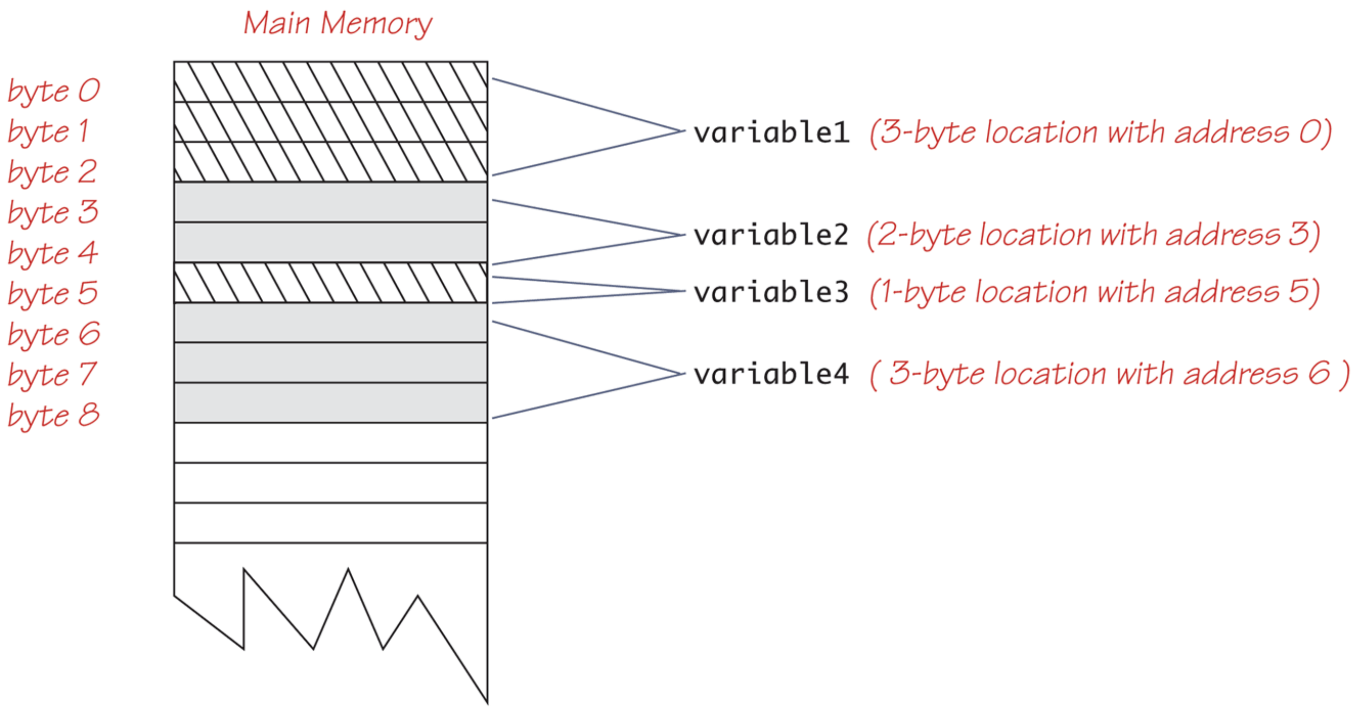
A data item is stored in one or (typically) more of these bytes.

The address of the (first) byte of the data item is used to find the data item when it is needed, either to check its value or to write a new one.


Main Memory
byte 0
byte 1
byte 2
byte 3
byte 4
byte 5
byte 6
byte 7
byte 8

# Variables and Memory

**Display 5.10 Variables in Memory**


Main Memory
byte 0
byte 1
byte 2
byte 3
byte 4
byte 5
byte 6
byte 7
byte 8
variable1 (3-byte location with address 0)
variable2 (2-byte location with address 3)
variable3 (1-byte location with address 5)
variable4 ( 3-byte location with address 6 )

# References

Every variable is stored in a *location* in computer memory.

- When the variable is a *primitive* type (int, double etc.), the value of the variable is stored **directly** in the memory location assigned to the variable. This is possible because each value of a primitive type always requires the same amount of memory to store its values.
- When the variable is a *class* type, only the memory address (or ***reference***) of where the object is located is stored at the memory location assigned to the variable. (Note: two different locations are involved.)

For classes:

- The object named by the variable is stored in some other location in memory, not the location assigned to the variable.
- Like primitives, the value of a class variable (i.e., the reference) 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 variable size.*

Because a class variable only holds a reference to the object, two class 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;
```

# Example

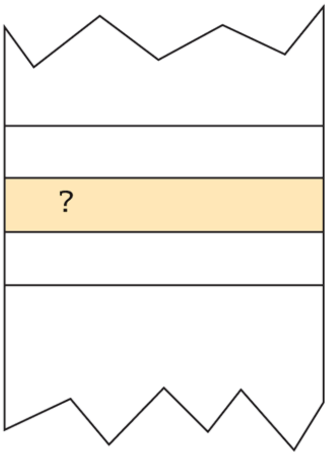
**Display 5.12 Class Type Variables Store a Reference**

```
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.*

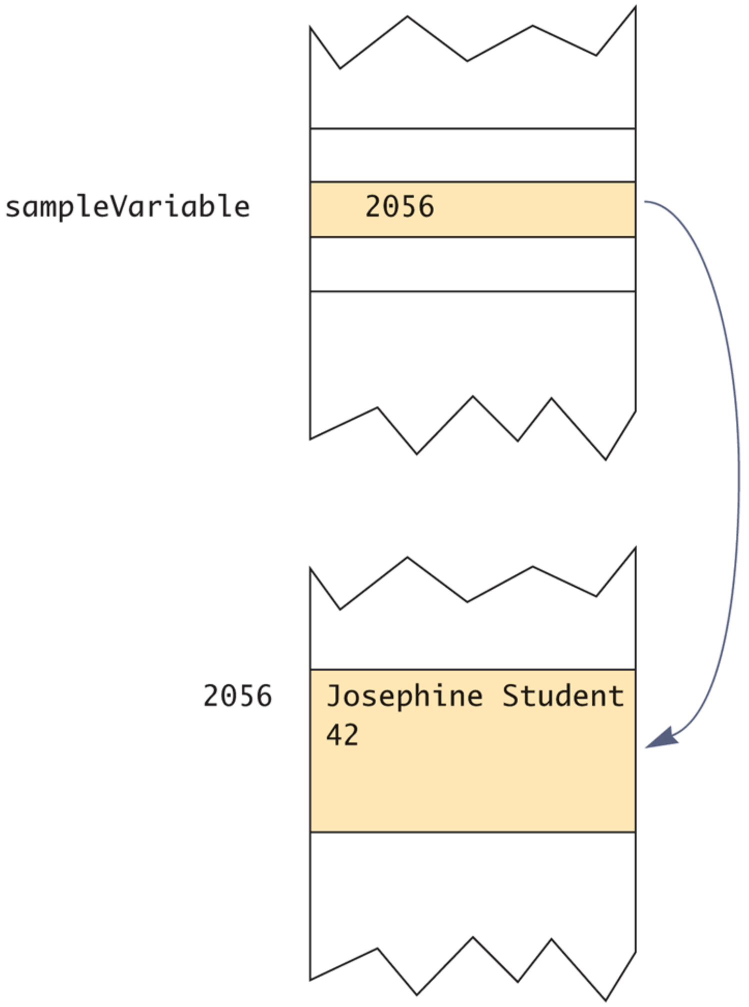


```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

*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)

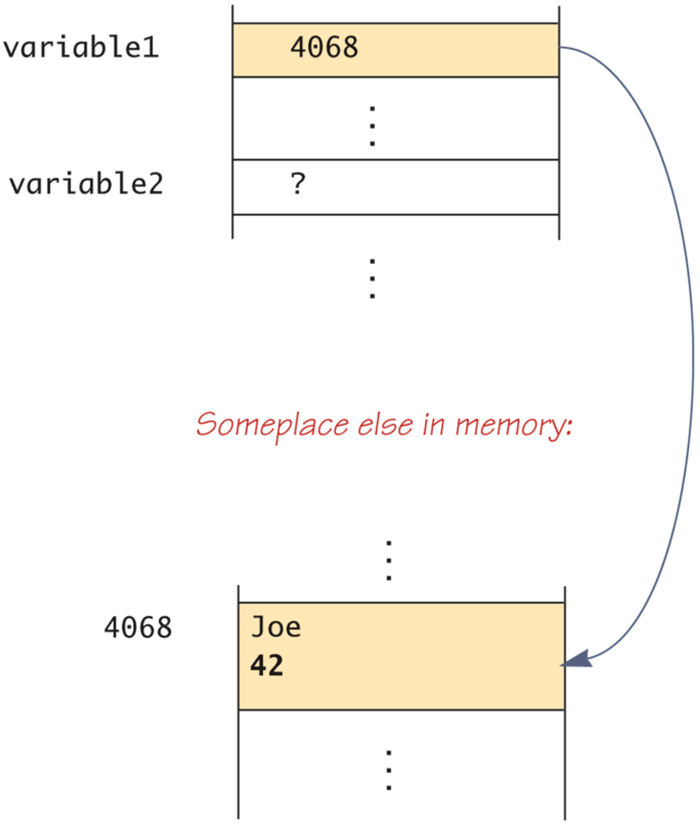
Display 5.12 **Class Type Variables Store a Reference**



*For emphasis, we made the arrow point to the memory location referenced.*

### 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.*

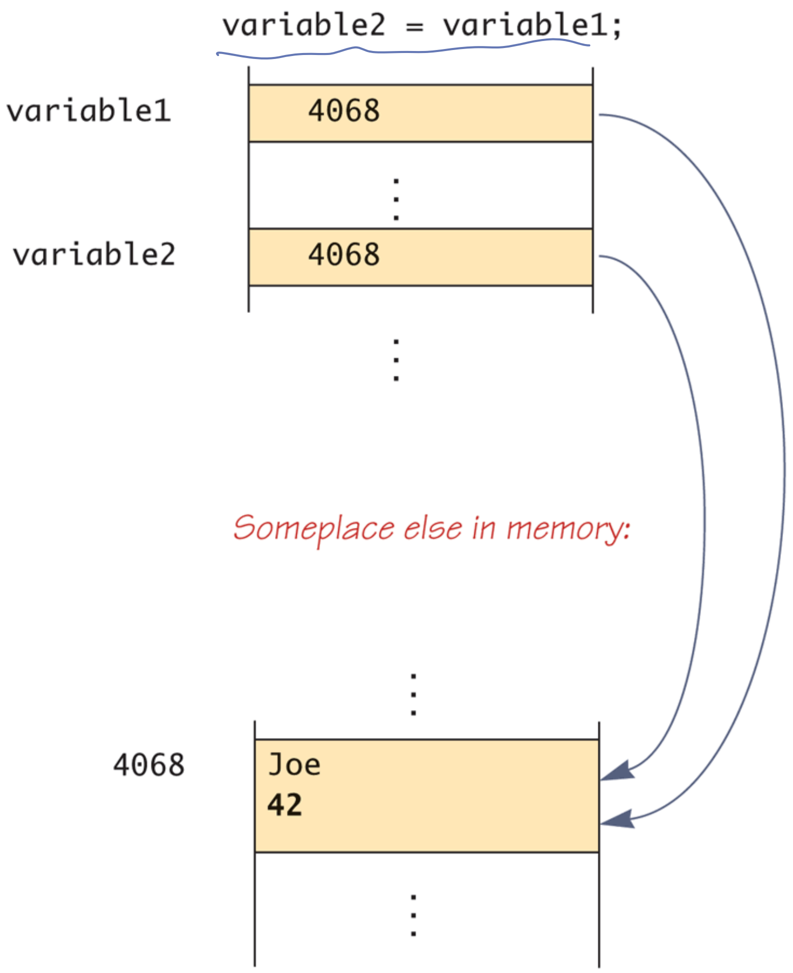
*Note that you can think of*

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

*as returning a reference.*

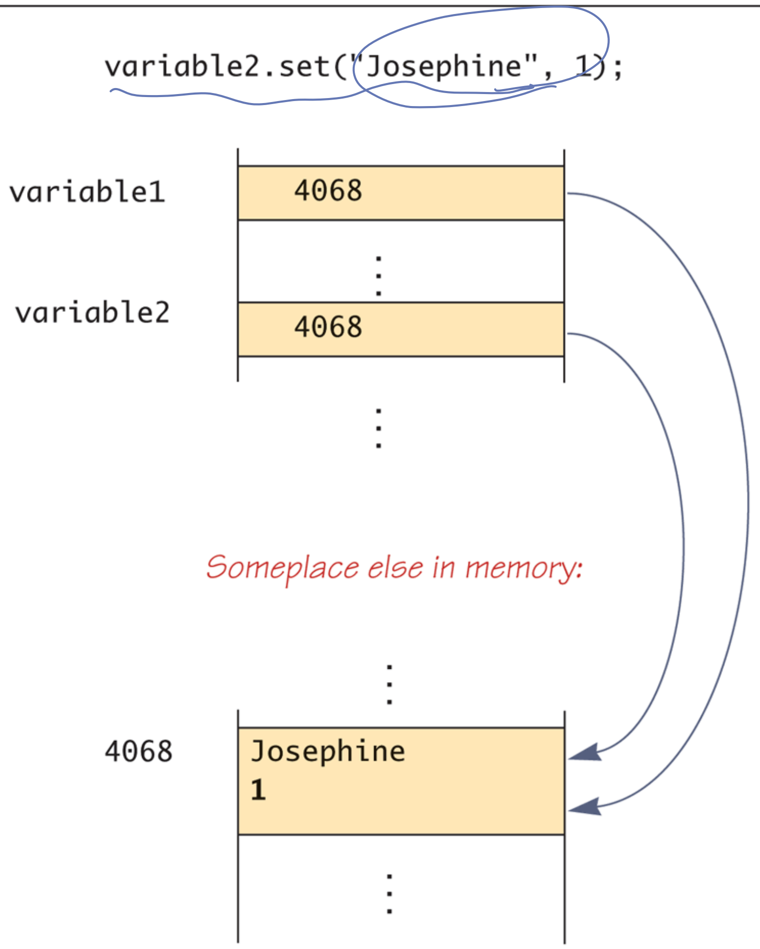
(continued)

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



(continued)

## Display 5.13 Assignment Operator with Class Type Variables



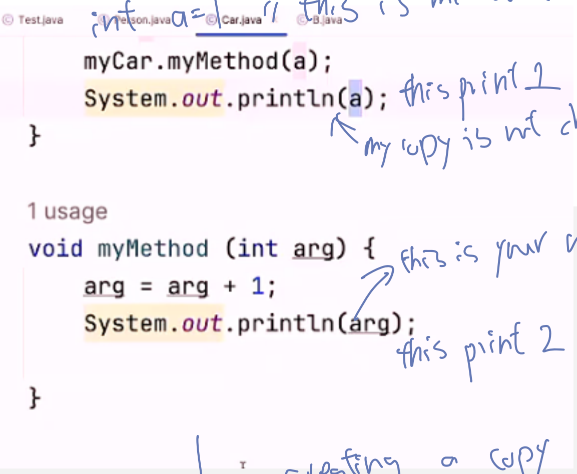
```
variable2.set("Josephine", 1);
```

# Call-by-value and call-by-reference

Among many programming languages, there are two ways to pass arguments into a method

- call-by-value
- call-by-reference

To see how these two types relate, consider the code



```
    myCar.myMethod(a);
    System.out.println(a);
}

1 usage
void myMethod (int arg) {
    arg = arg + 1;
    System.out.println(arg);

}
```

int a=1 // this is me and my refe

this print 1

my copy is not change

this is your note.

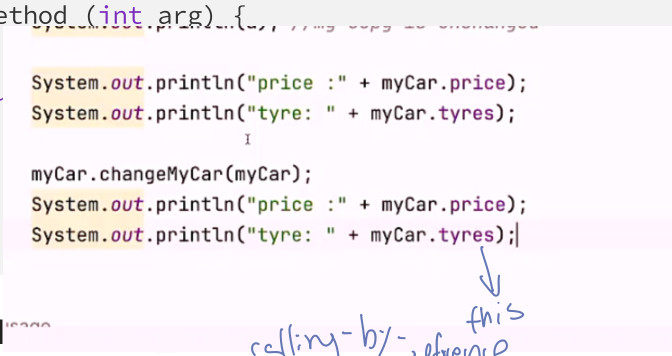
this print 2

creating a copy

call-by- value

```
int myMethod (int arg) {
    arg
    retu
}

int a =
int b =
```



```
System.out.println("price :" + myCar.price);
System.out.println("tyre: " + myCar.tyres);

myCar.changeMyCar(myCar);
System.out.println("price :" + myCar.price);
System.out.println("tyre: " + myCar.tyres);
```

this

calling-by-refrence.

What val

In a call-by-value function call, a copy of `a` would be passed to `myFunction` and so `a` would not be affected by the line `arg = arg + 1;`

In a call-by-reference function call, `arg` would actually stand for `a` during the function call -- `arg` would be a reference to the location where `a` is stored, and so after the function call, `a == 2`.

So which does Java use?

Technically, all Java methods are call-by-value.  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.

However, since class variables are actually references to objects, they behave very similarly to call-by-reference parameters.  In particular, the function call can change the value of the **object** that the parameter refers to, even though it cannot change the parameter itself (i.e., can't make it refer to a different object).

In the following code, try to predict what will be printed when you uncomment one of the lines starting "//", and then run the code to check.  You can also uncomment more than one, or reorder them if you like.

Which of the functions do you think better reflects the name "increment"?

```
class Main {
    int value;
    Main () {
        value = 0;
    }

    static Main tryIncrement1 (Main m) {
        m.value++;
        return m;
    }

    static Main tryIncrement2 (Main m) {
        Main n = new Main ();
        n.value = m.value + 1;
        m = n;
        return m;
    }

    public static void main (String[] arg) {
        Main m = new Main ();
        Main n = new Main ();
        /* Uncomment one of the following lines */

        // n = tryIncrement1 (m);
        // n = tryIncrement2 (m);
        // m = tryIncrement1 (m);
        // m = tryIncrement2 (m);

        System.out.println ("Values are " + m.value + " " + n.value);
    }
}
```

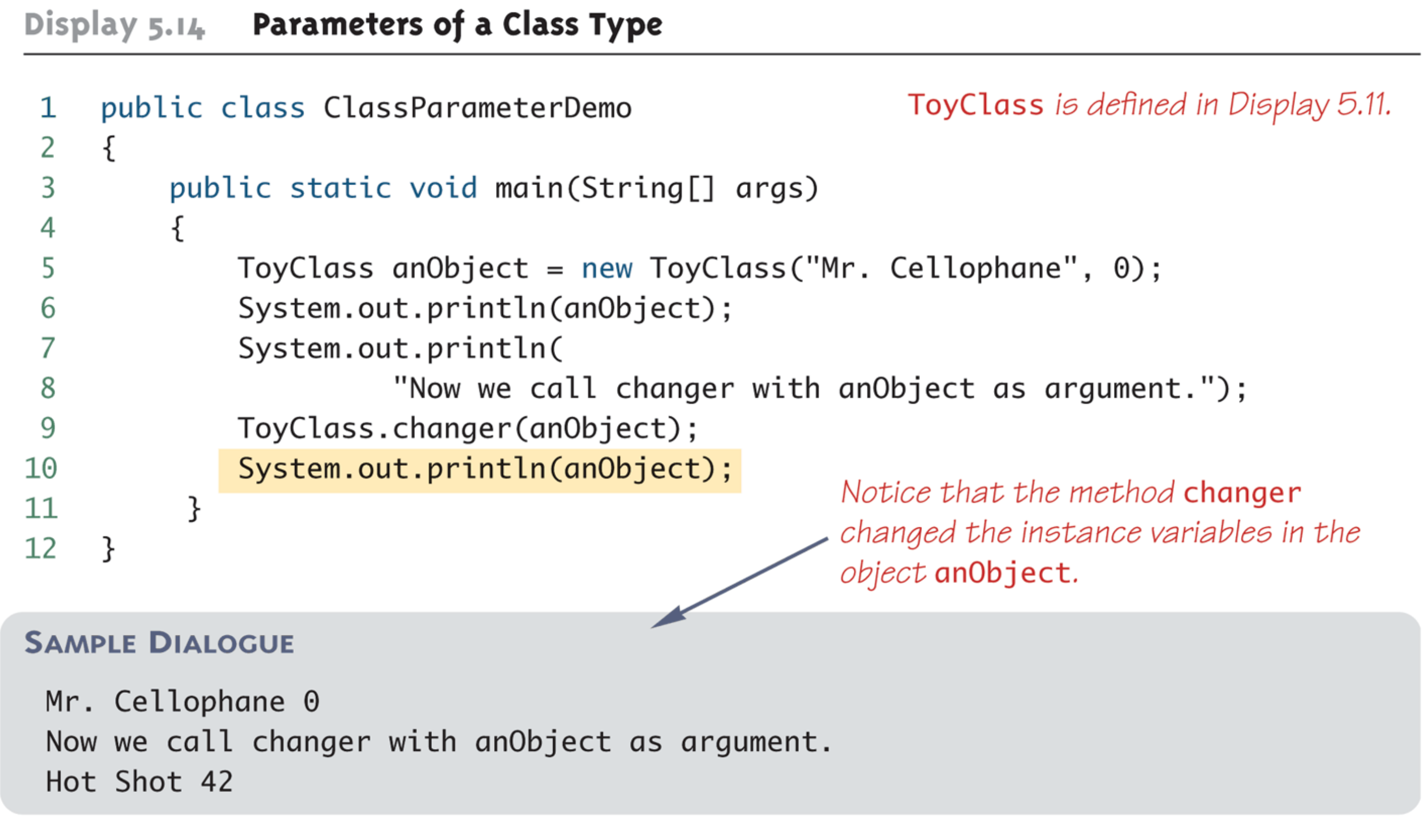
The value plugged into a class type parameter is a reference (memory address)

- 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)

The *value of the object* named by the argument can be updated but the *argument itself* will not be changed.

# Parameters of a class type

The following is based on Display 5.14 of the text book.



**Display 5.14 Parameters of a Class Type**

```
public class ClassParameterDemo
{
    public static void main(String[] args)
    {
        ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
        System.out.println(anObject);
        System.out.println(
                  "Now we call changer with anObject as argument.");
        ToyClass.changer(anObject);
        System.out.println(anObject);
     }
}
```

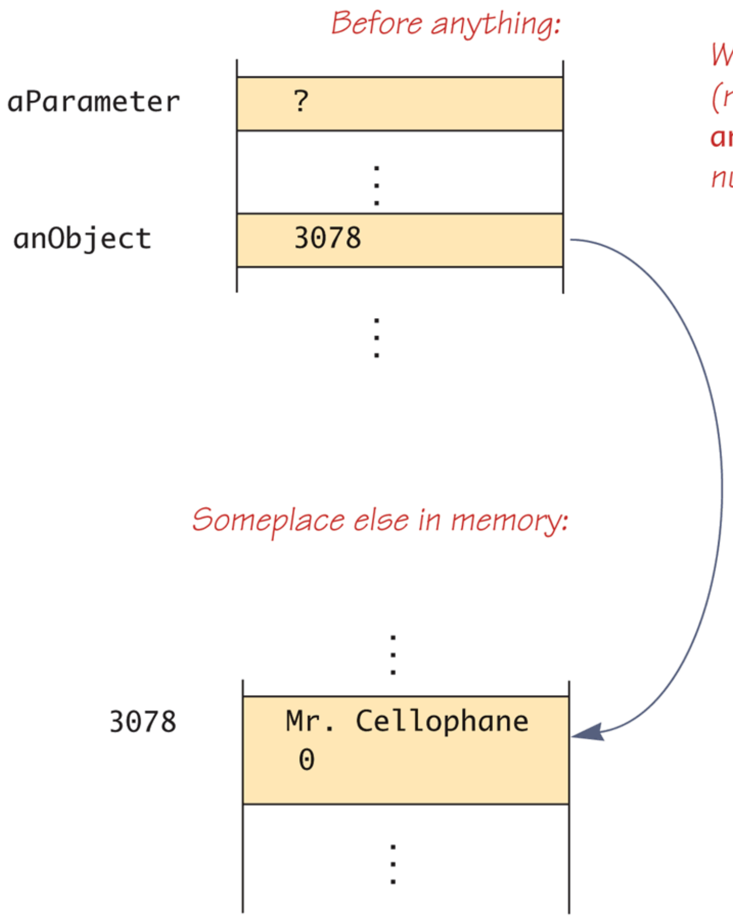
*`ToyClass` is defined in Display 5.11.*

*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
```

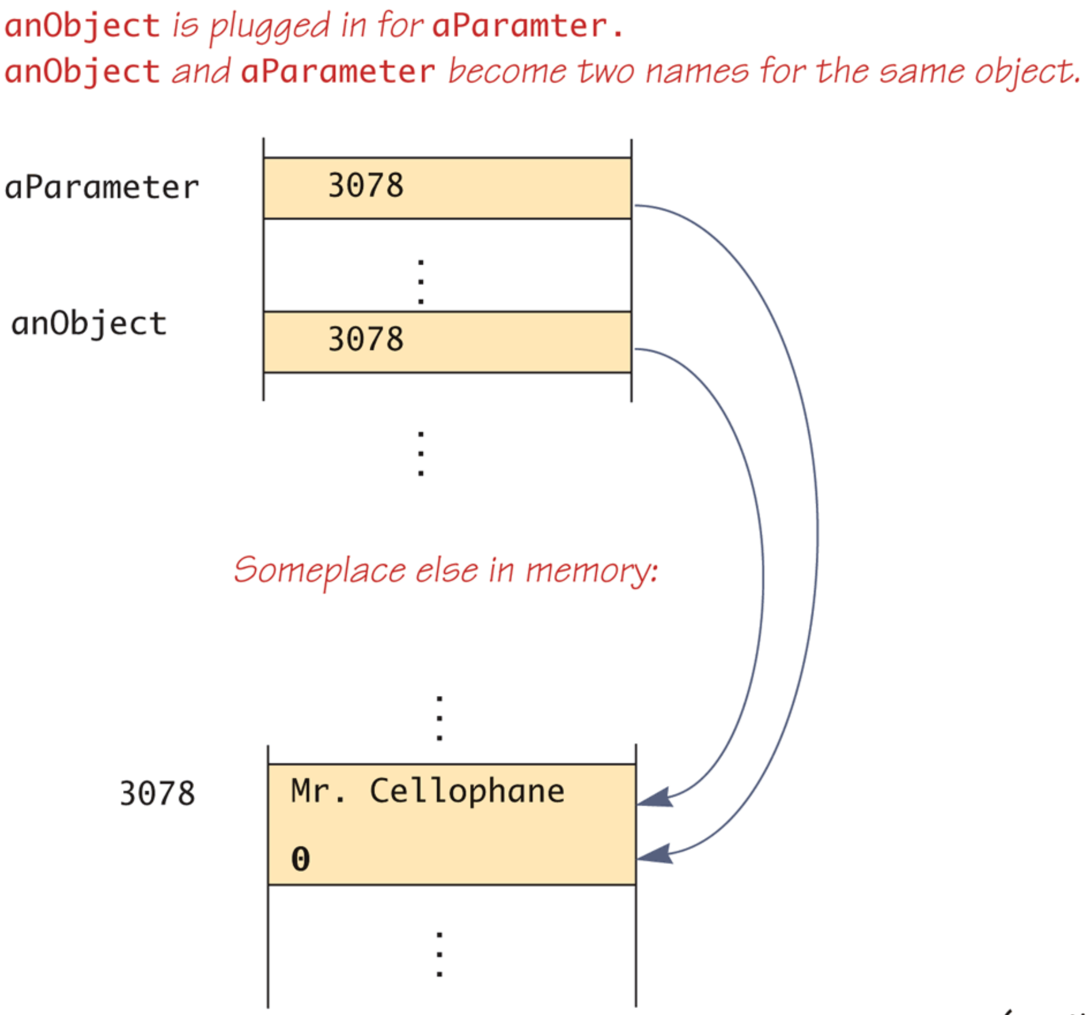
**Display 5.15** **Memory Picture for Display 5.14**



*We do not know what memory address (reference) is stored in the variable* `anObject`*. Let's say it is 3078. The exact number does not matter.*

(continued)

## Display 5.15 Memory Picture for Display 5.14


anObject is plugged in for aParamter.
anObject and aParameter become two names for the same object.
aParameter
3078
anObject
3078
Someplace else in memory:
3078
Mr. Cellophane
0

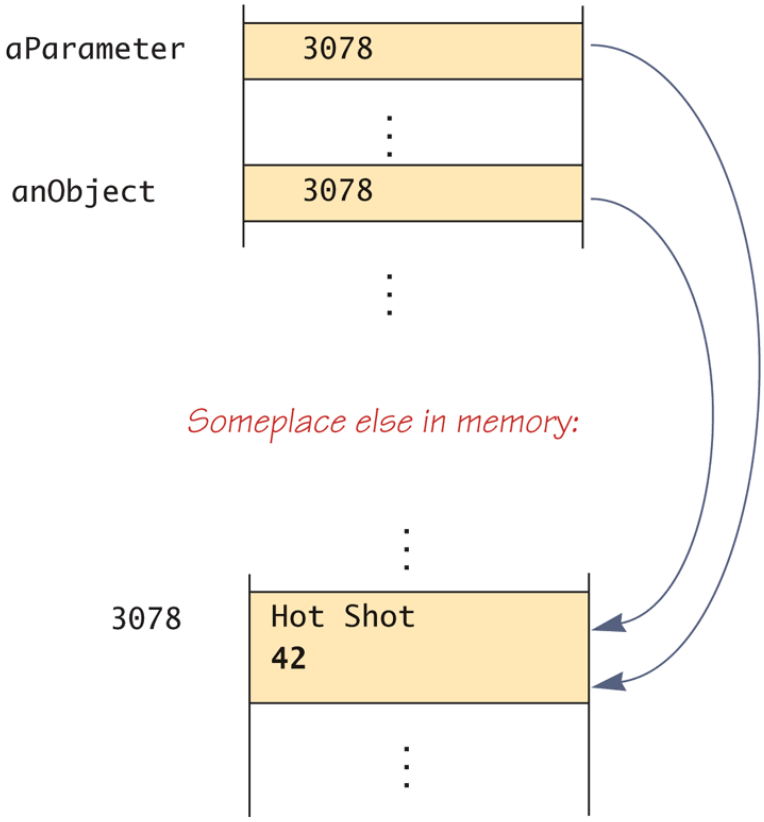
(continued)

### 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.*



*Someplace else in memory:*

## Pitfall: = and ==

The operators `=` and `==` don't do what you might expect when used on class variables.

- Assignment (=) causes two variables to be names for the same object; it does not create a copy. Changing the instance variables of the object referred to by one variable will cause cause changes in the object referred to by the other  variable, because it is the *same object*.
- Testing for equality (==) only checks that two variables of a class type *refer to the same object*. If they refer to two objects with the same instance variables, it will return `false`.

  **To test for equality, use the member method `equals()`.**

## The constant null

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

- e.g., `YourClass  yourObject = null;`

It indicates that your variable does not currently refer to *any* object at all.

It is often used in constructors to initialize class type instance variables when there is no obvious object to use yet.  If a variable of a class type is not initialized, it will default to being `null`.

`null` is not an object.  It is a placeholder that doesn't refer to anywhere in memory

To test if a variable is `null`, use `==` or `!=` (instead of `equals()` ).  We are testing whether or not the variable refers to something, not testing its value.

- e.g.  `if (myObject == null)`

## Pitfall

Because `null` doesn't refer to an object, it is meaningless to refer to its instance variables.  Because non-static member functions are allowed to refer to their instance variables, we are also not allowed to call them on a `null` variable.

Accessing a non-static member through a null variable will result in a "Null Pointer Exception" error message.

# 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*

```
    ToyClass variable1 =  new ToyClass(“Joe”, 42);
    if (variable1.equals (new ToyClass(“JOE”, 42)))
```

# Keyboard input using Double.parseDouble

We have seen one way to convert keyboard input from strings to numeric values. Another is to use the static method Double.parseDouble . (Of course, this applies to any strings, not only those read from the keyboard.)

Run this code. Try numbers like 10000000 or 0.0000001 . What are the biggest and smallest numbers it will recognize?

**Exercise:** Modify this code to print only one number if only one is entered.

```
import java.util.Scanner;
import java.util.StringTokenizer;

public class Main {
    public static void main (String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter two numbers on a line.");
        System.out.println("Place a comma between the numbers.");
        System.out.println("Extra blank space is OK.");
        String inputLine = keyboard.nextLine();

        String delimiters = ", "; // comma and blank space
        StringTokenizer numberFactory =
            new StringTokenizer(inputLine, delimiters);

        String string1 = null;
        String string2 = null;

        if (numberFactory.countTokens() < 2) {
            System.err.println("Fatal Error.  Insufficient numbers");
            System.exit(1);
        } else {
            string1 = numberFactory.nextToken();
            string2 = numberFactory.nextToken();
        }

        // Now, finally, the static class gets used.
        double number1 = Double.parseDouble(string1);
        double number2 = Double.parseDouble(string2);

        System.out.print("You input ");
        System.out.println(number1 + " and " + number2);
    }
}
```

# Using and misusing references

When writing a program, it is very important to ensure that private instance variables remain truly private

For a primitive type instance variable, just adding the `private` modifier to its declaration should ensure that there will be no privacy leaks

For a class type instance variable, however, adding the `private` modifier alone is not sufficient

We'll see that in the following example of a class recording data for a person not finish yet

# (long) class Person

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.

- E.g. Person.java

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 if still alive
```

We will build this class up in stages.  If you want to experiment with it, please use the runnable version at the end of this slide set.

## Constructor

In order to exist, a person must have (at least) a name and a birth date.  Therefore, makes sense not to have a no-argument Person class constructor.

A person who is still alive does not yet have a date of death. Therefore, the `Person` class constructor will need to be able to deal with a `null` value for date of death

A person who has died must have had a birth date that preceded his or her date of death. Therefore, when both dates are provided, they will need to be checked for consistency

```
public Person(String initialName, Date birthDate, Date deathDate)
{
    if (consistent(birthDate, deathDate))
    {
        name = initialName;
        born = new Date(birthDate);
        if (deathDate == null)
            died = null;
        else
            died = new Date(deathDate);
        }
    else
    {
        System.out.println("Inconsistent dates.");
        System.exit(1);
    }
}
```

## The class invariant

A statement that is always true for every object of the class is called a *class invariant*. A class invariant can help to define a class in a consistent and organized way.

For the `Person` class, the following should always be true: An object of the class `Person` has a date of birth (which is not `null`), and if the object has a date of death, then the date of death is equal to or later than the date of birth

Checking the `Person` class confirms that this is true of every object created by a constructor, and all the other methods (e.g., the private method `consistent`) preserve the truth of this statement.

```
/**
Class invariant: A Person always has a date of birth,
and if the Person has a date of death, then the date of
death is equal to or later than the date of birth.
To be consistent, birthDate must not be null. If there
is no date of death (deathDate == null), that is
consistent with any birthDate. Otherwise, the birthDate
must come before or be equal to the deathDate.
*/
private static boolean consistent(Date birthDate, Date deathDate)
{
    if (birthDate == null) return false;
    else if (deathDate == null) return true;
    else return (birthDate.precedes(deathDate) ||
                 birthDate.equals(deathDate));
}
```

## Methods equals and datesMatch

The definition of `equals` for the class `Person` includes an invocation of `equals` for the class `String`, and an invocation of the method `equals` for the class `Date`

Java determines which `equals` method is being invoked from the type of its calling object

Also note that the `died` instance variables are compared using the `datesMatch` method instead of the `equals` method, since their values may be `null`

```
public boolean equals(Person otherPerson)
{
    if (otherPerson == null)
        return false;
    else
        return (name.equals(otherPerson.name) &&
                born.equals(otherPerson.born) &&
                datesMatch(died, otherPerson.died));
}
```

## Method datesMatch

```
/**
To match date1 and date2 must either be the
same date or both be null.
*/
private static boolean datesMatch(Date date1, Date date2)
{
    if (date1 == null)
        return (date2 == null);
    else if (date2 == null) //&& date1 != null
        return false;
    else // both dates are not null.
        return(date1.equals(date2));
}
```

## Method toString

The way Java prints an object is to convert it to a string and then print the string.  That means that most classes will have a `toString` method.

Like the equals method, the `Person` class `toString` method includes invocations of the `Date` class `toString` method.

```
public String toString( )
{
    String diedString;
    if (died == null)
        diedString = ""; //Empty string
    else
        diedString = died.toString( );
    return (name + ", " + born + "-" + diedString);
}
```

## 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.

```
public Date(Date aDate) //constructor - chapter 4
{
    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, referring to the same `Date` objects.

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

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

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*

```
public Person(Person original)
{
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(1);
    }
    name = original.name;
    born = new Date(original.born);
    if (original.died == null)
        died = null;
    else
        died = new Date(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
}
```

```
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.

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.

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 instance variable.** Instead, use a copy constructor to return a reference to a completely independent copy of the mutable object

## Deep copy vs Shallow copy

A *deep copy* of an object is a copy that, with one exception, has *no references* in common with the original. The exception is that 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.

*Advanced*: If they are used carefully then shallow copies can be more efficient than performing deep copies of large objects. Best practice is to start making a deep copy, and then once your code is working, you have a thorough test suite and you are optimizing the code, carefully consider using shallow copies in a few limited places for efficiency.

## The complete Person class

The complete classes `Person` and `Date` are below.

You can run it, but it will currently give meaningless answers.

**Exercise:** Implement `Date.precedes()` near the end of the code, and fill in your name and birth date, and those of a friend or other person in the method `Main.main` at the start of the code.

You can also test the unused methods such as `Person.consistent`, and/or check which of two people with death dates lived longer.

```
class Main {
    public static void main (String[] args) {
        Person me    =new Person("",new Date(0/*yy*/,0/*mm*/,0/*dd*/), null);
        Person friend=new Person("",new Date(0/*yy*/,0/*mm*/,0/*dd*/), null);

        System.out.println(me.oldest(friend) + " is older");
    }
}

class Person
{
    private String name;
    private Date born;
    private Date died; //null if still alive

    public Person(String initialName, Date birthDate, Date deathDate)
    {
        if (consistent(birthDate, deathDate))
        {
            name = initialName;
            born = new Date(birthDate);
            if (deathDate == null)
                died = null;
            else
                died = new Date(deathDate);
            }
        else
        {
            System.out.println("Inconsistent dates.");
            System.exit(1);
        }
    }

    public Person(Person original)
    {
        if (original == null)
        {
            System.out.println("Fatal error.");
            System.exit(1);
        }
        name = original.name;
        born = new Date(original.born);
        if (original.died == null)
            died = null;
        else
            died = new Date(original.died);
    }

    /**
    Class invariant: A Person always has a date of birth,
    and if the Person has a date of death, then the date of
    death is equal to or later than the date of birth.
    To be consistent, birthDate must not be null. If there
    is no date of death (deathDate == null), that is
    consistent with any birthDate. Otherwise, the birthDate
    must come before or be equal to the deathDate.
    */
    private static boolean consistent(Date birthDate, Date deathDate)
    {
        if (birthDate == null) return false;
        else if (deathDate == null) return true;
        else return (birthDate.precedes(deathDate) ||
                    birthDate.equals(deathDate));
    }

    public boolean equals(Person otherPerson)
    {
        if (otherPerson == null)
            return false;
        else
            return (name.equals(otherPerson.name) &&
                    born.equals(otherPerson.born) &&
                    datesMatch(died, otherPerson.died));
    }

    /**
    To match date1 and date2 must either be the
    same date or both be null.
    */
    private static boolean datesMatch(Date date1, Date date2)
    {
        if (date1 == null)
            return (date2 == null);
        else if (date2 == null) //&& date1 != null
            return false;
        else // both dates are not null.
            return(date1.equals(date2));
    }

    String oldest (Person p) {
        if (born.precedes(p.born))
            return name;
        else
            return p.name;
    }

    @Override
    public String toString( )
    {
        String diedString;
        if (died == null)
            diedString = ""; //Empty string
        else
            diedString = died.toString( );
        return (name + ", " + born + "-" + diedString);
    }

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

    public Date getDeathDate()
    {
        return (died == null) ? null : new Date(died);
    }

    public String getName()
    {
        return name;
    }
}

class Date {
    int day, month, year;

    public Date(int yy, int mm, int dd) {
        day = dd;
        month = mm;
        year = yy;
    }

    public Date(Date aDate) //constructor - chapter 4
    {
        if (aDate == null) //Not a real date.
        {
            System.out.println("Fatal Error.");
            System.exit(0);
        }
        month = aDate.month;
        day = aDate.day;
        year = aDate.year;
    }

    public boolean equals (Date other) {
        return year == other.year
            && month == other.month
            && day == other.day;
    }

    /**
     * Return true if  this  is earlier than  other
     * Return false otherwise
     */
    public boolean precedes (Date other) {
    /**************************************/
    /*****  Complete me *******************/
    /**************************************/
        return false;
    }
}
```