

MET CS 520 – Information Structures with Java

Classes

Static Methods

- A static method is one that can be used without a calling object
- A static method still belongs to a class, and its definition is given inside the class definition
- When a static method is defined, the keyword `static` is placed in the method header

```
public static returnType myMethod(parameters)
{ ... }
```

- Static methods are invoked using the class name in place of a calling object

```
returnedValue = MyClass.myMethod(arguments);
```

Pitfall: Invoking a Nonstatic Method Within a Static Method

- A static method cannot refer to an instance variable of the class, and it cannot invoke a nonstatic method of the class
 - A static method has no this, so it cannot use an instance variable or method that has an implicit or explicit this for a calling object
 - A static method can invoke another static method, however

Tip: You Can Put a main in any Class

- Although the main method is often by itself in a class separate from the other classes of a program, it can also be contained within a regular class definition
 - In this way the class in which it is contained can be used to create objects in other classes, or it can be run as a program
 - A main method so included in a regular class definition is especially useful when it contains diagnostic code for the class

Demo Temperature.java

Static Variables

- A static variable is a variable that belongs to the class as a whole, and not just to one object
 - There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
- All objects of the class can read and change a static variable
- Although a static method cannot access an instance variable, a static method can access a static variable
- A static variable is declared like an instance variable, with the addition of the modifier static

```
private static int myStaticVariable;
```

Static Variables

- Static variables can be declared and initialized at the same time

```
private static int myStaticVariable = 0;
```

- If not explicitly initialized, a static variable will be automatically initialized to a default value
 - boolean static variables are initialized to false
 - Other primitive types static variables are initialized to the zero of their type
 - Class type static variables are initialized to null
- It is always preferable to explicitly initialize static variables rather than rely on the default initialization

Static Variables

- A static variable should always be defined private, unless it is also a defined constant
 - The value of a static defined constant cannot be altered, therefore it is safe to make it public
 - In addition to static, the declaration for a static defined constant must include the modifier final, which indicates that its value cannot be changed

```
public static final int BIRTH_YEAR = 1954;
```

- When referring to such a defined constant outside its class, use the name of its class in place of a calling object

```
int year = MyClass.BIRTH_YEAR;
```


The Math Class

- The Math class provides a number of standard mathematical methods
 - It is found in the java.lang package, so it does not require an import statement
 - All of its methods and data are static, therefore they are invoked with the class name Math instead of a calling object
 - The Math class has two predefined constants, E (e, the base of the natural logarithm system) and PI (π , 3.1415 . . .)
`area = Math.PI * radius * radius;`

Review Math Class

Random Numbers

- The Math class also provides a facility to generate pseudo-random numbers

```
public static double random()
```

- A pseudo-random number appears random but is really generated by a deterministic function
 - There is also a more flexible class named Random
- Sample use:
- Returns a pseudo-random number greater than or equal to 0.0 and less than 1.0

```
double num = Math.random() ;
```

Wrapper Classes

- Wrapper classes provide a class type corresponding to each of the primitive types
 - This makes it possible to have class types that behave somewhat like primitive types
 - The wrapper classes for the primitive types `byte`, `short`, `long`, `float`, `double`, and `char` are (in order) `Byte`, `Short`, `Long`, `Float`, `Double`, and `Character`
- Wrapper classes also contain a number of useful predefined constants and static methods

Wrapper Classes

- **Boxing:** the process of going from a value of a primitive type to an object of its wrapper class
 - To convert a primitive value to an "equivalent" class type value, create an object of the corresponding wrapper class using the primitive value as an argument
 - The new object will contain an instance variable that stores a copy of the primitive value
 - Unlike most other classes, a wrapper class does not have a no-argument constructor

```
Integer integerObject = new Integer(42);
```

Wrapper Classes

- **Unboxing:** the process of going from an object of a wrapper class to the corresponding value of a primitive type
 - The methods for converting an object from the wrapper classes `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Character` to their corresponding primitive type are (in order) `byteValue`, `shortValue`, `intValue`, `longValue`, `floatValue`, `doubleValue`, and `charValue`
 - None of these methods take an argument
`int i = integerObject.intValue();`

Automatic Boxing and Unboxing

- Starting with version 5.0, Java can automatically do boxing and unboxing
- Instead of creating a wrapper class object using the new operation (as shown before), it can be done as an automatic type cast:

```
Integer integerObject = 42;
```

- Instead of having to invoke the appropriate method (such as intValue, doubleValue, charValue, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically

```
int i = integerObject;
```

Constants and Static Methods in Wrapper Classes

- Wrapper classes include useful constants that provide the largest and smallest values for any of the primitive number types
 - For example, `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Double.MAX_VALUE`, `Double.MIN_VALUE`, etc.
- The `Boolean` class has names for two constants of type `Boolean`
 - `Boolean.TRUE` and `Boolean.FALSE` are the Boolean objects that correspond to the values `true` and `false` of the primitive type `boolean`

Constants and Static Methods in Wrapper Classes

- Wrapper classes have static methods that convert a correctly formed string representation of a number to the number of a given type
 - The methods `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat`, and `Double.parseDouble` do this for the primitive types (in order) int, long, float, and double
- Wrapper classes also have static methods that convert from a numeric value to a string representation of the value
 - For example, the expression
`Double.toString(123.99);`
returns the string value "123.99"
- The `Character` class contains a number of static methods that are useful for string processing

Review Integer and Character Classe

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

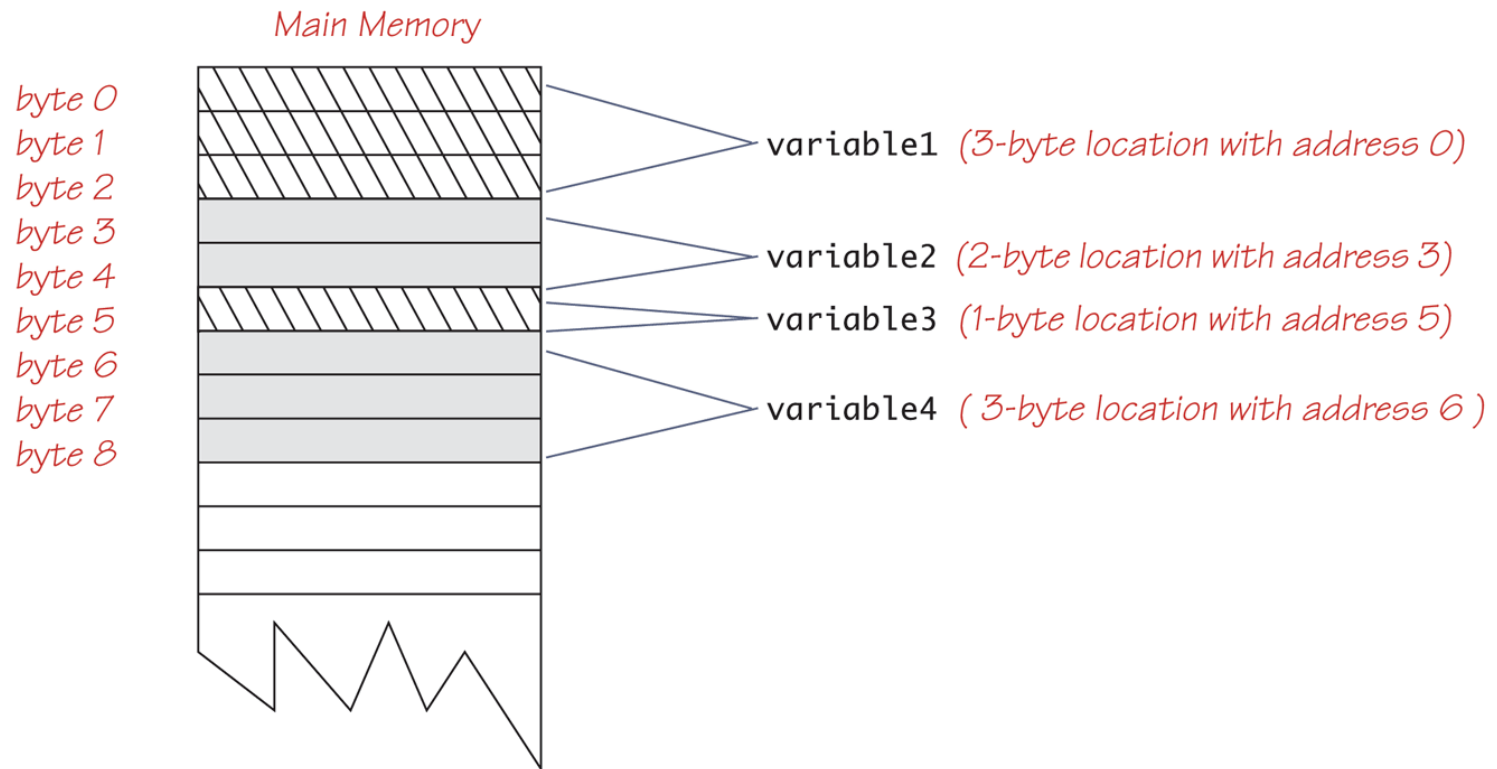
- Main memory consists of a long list of numbered locations called bytes
 - Each byte contains eight bits: eight 0 or 1 digits
- The number that identifies a byte is called its address
 - A data item can be stored in one (or more) of these bytes
 - The address of the byte is used to find the data item 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 (Part 1 of 2)

Display 5.12 Class Type Variables Store a Reference

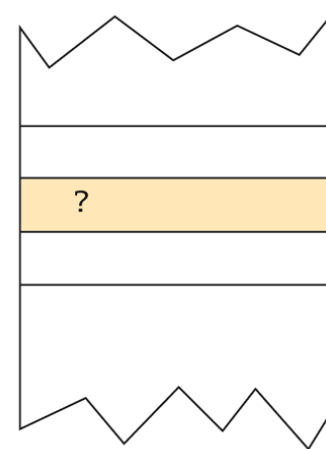
```
public class ToyClass
{
    private String name;
    private int number;
```

*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



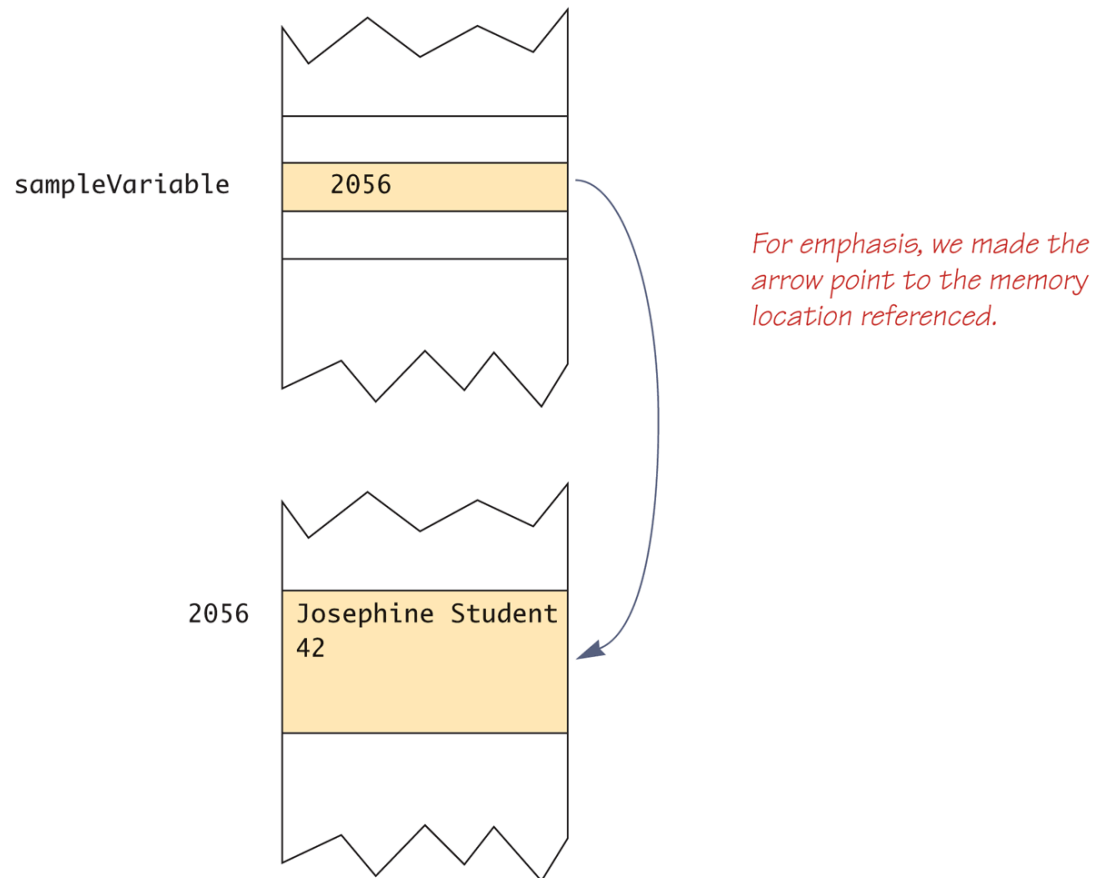
```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

*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)

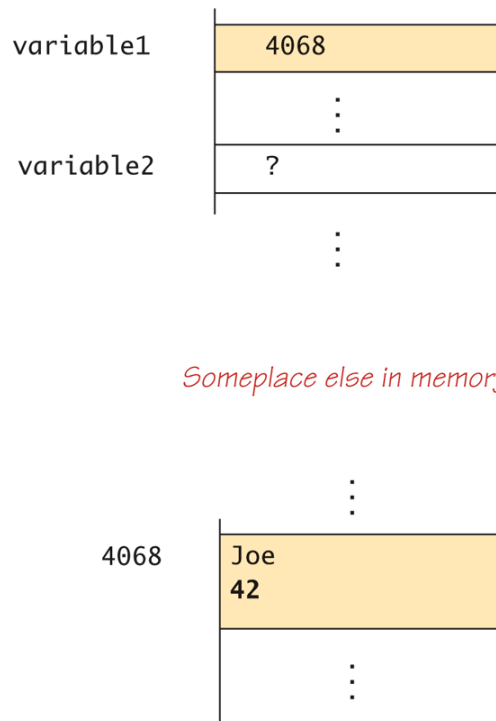
Display 5.12 Class Type Variables Store a Reference



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

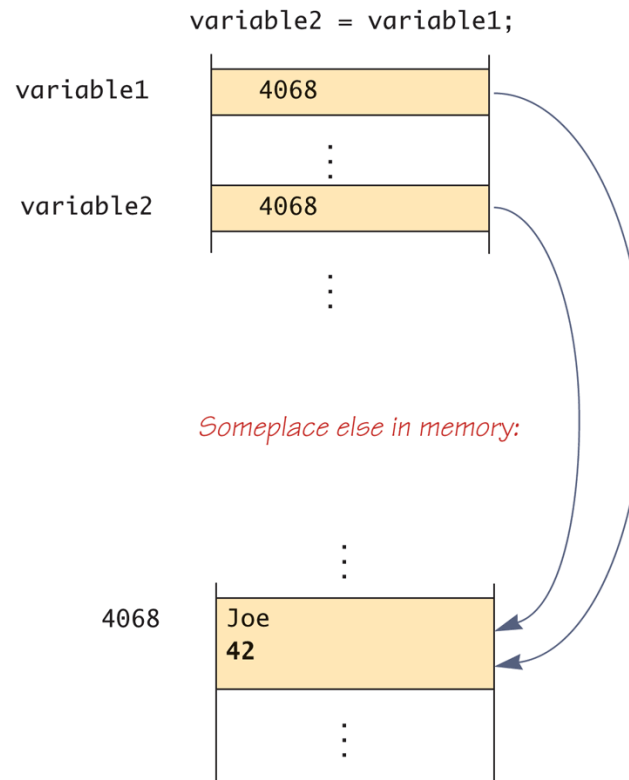
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

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

Display 5.13 Assignment Operator with Class Type Variables

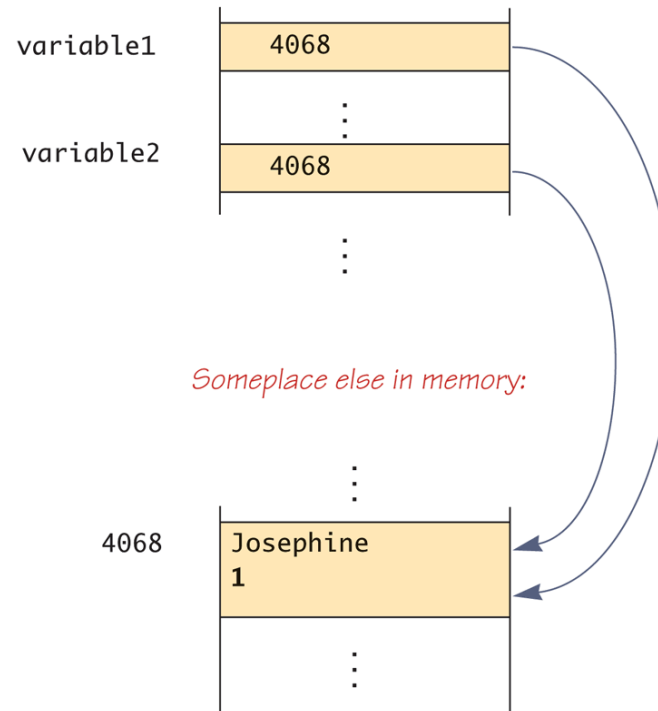


(continued)

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

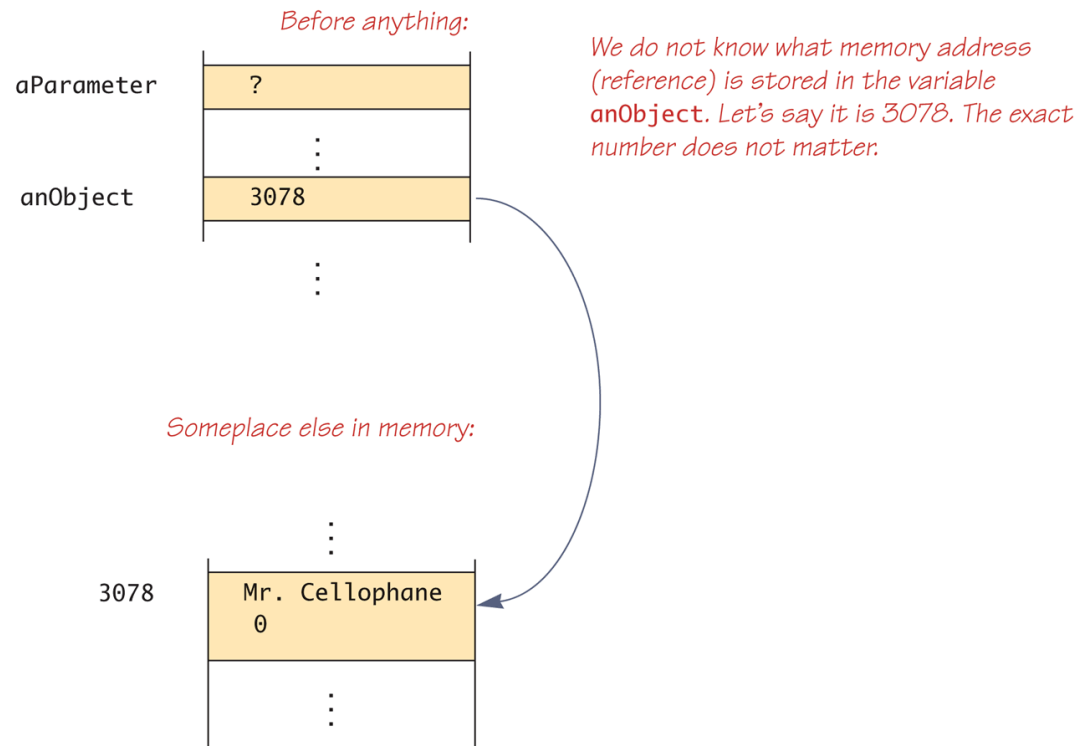
Class Parameters

- 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)

Demo ClassParameterDemo and ToyClass

Memory Picture for Display 5.14 (Part 1 of 3)

Display 5.15 Memory Picture for Display 5.14



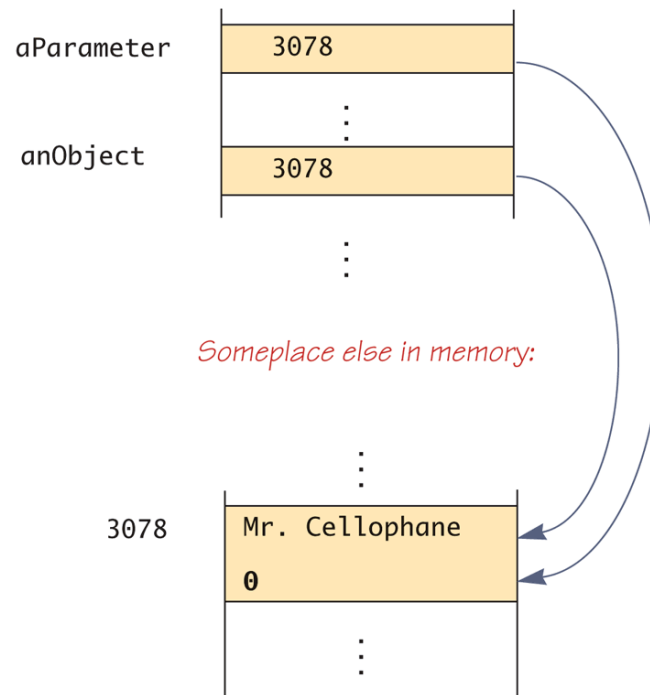
(continued)

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.

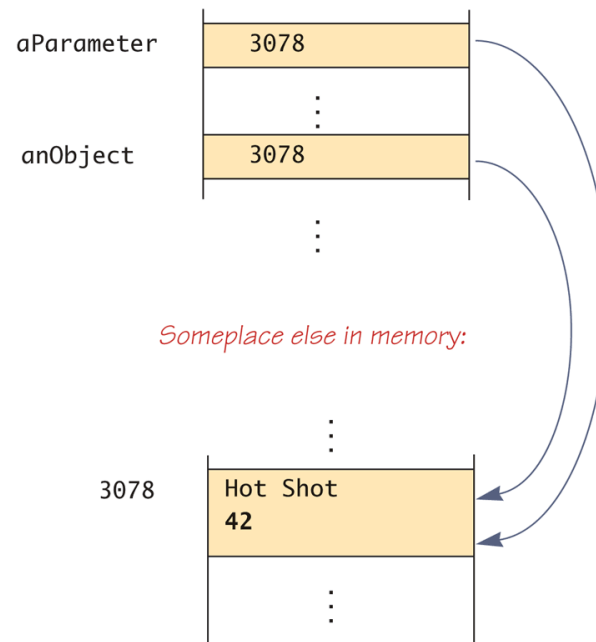


(continued)

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

Demo ParametersClassDemo and ToyClass2

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

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
```

...

Designing a Person Class: Constructor

- In order to exist, a person must have (at least) a name and a birth date
 - Therefore, it would make no sense 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

A Person Class Constructor

```
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(0);
    }
}
```

Designing a Person Class: 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

Designing a Person Class: the Class Invariant

```
/** 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));  
}
```


Designing a Person Class: the equals and datesMatch Methods

- 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

Designing a Person Class: the equals Method

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

Designing a Person Class: the matchDate Method

```
/** 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));
}
```

Designing a Person Class: the toString Method

- Like the equals method, note that 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

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

- 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);
    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  
}
```

- 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

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 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

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

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

Pitfall: Subdirectories Are Not Automatically Imported

- When a package is stored in a subdirectory of the directory containing another package, importing the enclosing package does not import the subdirectory package
- The import statement:
`import utilities.numericstuff.*;`
 - imports the utilities.numericstuff package only
- The import statements:
`import utilities.numericstuff.*;`
`import utilities.numericstuff.statistical.*;`
 - import both the utilities.numericstuff and
 - utilities.numericstuff.statistical packages

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

Pitfall: Not Including the Current Directory in Your Class Path

- If the **CLASSPATH** variable is set, the current directory must be included as one of the alternatives
 - Otherwise, Java may not even be able to find the .class files for the program itself
- If the **CLASSPATH** variable is not set, then all the class files for a program must be put in the current directory

Specifying a Class Path When You Compile

- The class path can be manually specified when a class is compiled
 - Just add `-classpath` followed by the desired class path
 - This will compile the class, overriding any previous `CLASSPATH` setting
- You should use the `-classpath` option again when the class is run

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

Introduction to javadoc

- Unlike a language such as C++, Java places both the interface and the implementation of a class in the same file
- However, Java has a program called **javadoc** that automatically extracts the interface from a class definition and produces documentation
 - This information is presented in HTML format, and can be viewed with a Web browser
 - If a class is correctly commented, a programmer need only refer to this API (Application Programming Interface) documentation in order to use the class
 - javadoc can obtain documentation for anything from a single class to an entire package

Commenting Classes for javadoc

- The javadoc program extracts class headings, the headings for some comments, and headings for all public methods, instance variables, and static variables
 - In the normal default mode, no method bodies or private items are extracted
- To extract a comment, the following must be true:
 - The comment must immediately precede a public class or method definition, or some other public item
 - The comment must be a block comment, and the opening `/*` must contain an extra `*` (`/** . . . */`)
 - Note: Extra options would have to be set in order to extract line comments (`//`) and private items

Commenting Classes for javadoc

- In addition to any general information, the comment preceding a public method definition should include descriptions of parameters, any value returned, and any exceptions that might be thrown
 - This type of information is preceded by the @ symbol and is called an *@ tag*
 - @ tags come after any general comment, and each one is on a line by itself

```
/**  
General Comments about the method . . .  
@param aParameter Description of aParameter  
@return What is returned  
...  
*/
```

@ Tags

- @ tags should be placed in the order found below
- If there are multiple parameters, each should have its own `@param` on a separate line, and each should be listed according to its left-to-right order on the parameter list
- If there are multiple authors, each should have its own `@author` on a separate line

`@param Parameter_Name Parameter_Description`

`@return Description_Of_Value_Returned`

`@throws Exception_Type Explanation`

`@deprecated`

`@see Package_Name.Class_Name`

`@author Author`

`@version Version_Information`

Running javadoc

- To run `javadoc` on a package, give the following command:

`javadoc -d Documentation_Directory Package_Name`

- The HTML documents produced will be placed in the `Documentation_Directory`
 - If the `-d` and `Documentation_Directory` are omitted, `javadoc` will create suitable directories for the documentation
- To run `javadoc` on a single class, give the following command from the directory containing the class file:
`javadoc ClassName.java`
- To run `javadoc` on all the classes in a directory, give the following command instead: `javadoc *.java`

Options for javadoc

Display 5.23 Options for java.doc

<code>-link <i>Link_To_Other_Docs</i></code>	Provides a link to another set of documentation. Normally, this is used with either a path name to a local version of the Java documentation or the URL of the Sun Web site with standard Java documentation.
<code>-d <i>Documentation_Directory</i></code>	Specifies a directory to hold the documentation generated. <i>Documentation_Directory</i> may be a relative or absolute path name.
<code>-author</code>	Includes author information (from <code>@author</code> tags). This information is omitted unless this option is set.
<code>-version</code>	Includes version information (from <code>@version</code> tags). This information is omitted unless this option is set.
<code>-classpath <i>List_of_Directories</i></code>	Overrides the CLASSPATH environment variable and makes <i>List_of_Directories</i> the CLASSPATH for the execution of this invocation of javadoc. Does not permanently change the CLASSPATH variable.
<code>-private</code>	Includes private members as well as public members in the documentation.