Programming and Software Development
COMP90041

Lecture 5

Classes and Methods II

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 20-02-2021

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Review: Week 4

- **Static methods and static variables**
  - **The Math class and wrapper classes**
  - **Automatic boxing and unboxing mechanism**

- **References and class parameters**
  - **Variables and Memory**
  - **Using and misusing references**

Outline

- **Static methods and static variables**
  - **The Math class and wrapper classes**
  - **Automatic boxing and unboxing mechanism**
- **References and class parameters**
  - **Variables and Memory**
  - **Using and misusing references**

Outline

- 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 returnedType myMethod(parameters)
{ . . . }
```

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

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

Static Methods

Display 5.1 Static Methods (part 1 of 2)

```
1  /**
2  Class with static methods for circles and spheres.
3  */
4  public class RoundStuff
5  {
6     public static final double PI = 3.14159;
7
8     /**
9       Return the area of a circle of the given radius.
10    */
11    public static double area(double radius)
12    {
13       return(PI*radius*radius);
14    }
15
16    /**
17     Return the volume of a sphere of the given radius.
18    */
19     public static double volume(double radius)
20    {
21       return((4.0/3.0)*PI*radius*radius*radius);
22    }
23 }
```

This is the file `RoundStuff.java`.

This is the file `RoundStuffDemo.java`.

```
1  import java.util.Scanner;
2  public class RoundStuffDemo
3  {
4      public static void main(String[] args)
5      {
6          Scanner keyboard = new Scanner(System.in);
7          System.out.println("Enter radius:");
8          double radius = keyboard.nextDouble();

9          System.out.println("A circle of radius"
10                                    + radius + "inches");
11         System.out.println("has an area of " +
12             RoundStuff.area(radius) + " square inches.");
13         System.out.println("A sphere of radius"
14                                    + radius + "inches");
15         System.out.println("has an volume of " +
16             RoundStuff.volume(radius) + "cubic inches.");
17     }
18 }
```

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

Example - RoundStuffDemo

- A static method cannot refer to an instance variable of the class, and it cannot invoke a non-static 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

Pitfall: Invoking a Non-static Method Within a Static Method

- 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

Tip: You Can Put a `main` in any Class

Display 5.3 Another Class with a `main` Added

```
1   import java.util.Scanner;

2   /**
3   Class for a temperature (expressed in degrees Celsius).
4   */
5   public class Temperature
6   {
7       private double degrees; //Celsius

8       public Temperature()
9       {
10          degrees = 0;
11      }

12      public Temperature(double initialDegrees)
13      {
14          degrees = initialDegrees;
15      }

16      public void setDegrees(double newDegrees)
17      {
18          degrees = newDegrees;
19      }
```

Note that this class has a `main` *method and both static and nonstatic methods.*

(continued)

Another Class with a `main` Added (Part 1 of 4)

Display 5.3 Another Class with a `main` Added

```
20      public double getDegrees()
21      {
22          return degrees;
23      }

24      public String toString()
25      {
26          return (degrees + " C");
27      }
28
29      public boolean equals(Temperature otherTemperature)
30      {
31          return (degrees == otherTemperature.degrees);
32      }
```

(continued)

Another Class with a `main` Added (Part 2 of 4)

Display 5.3 Another Class with a `main` Added

```
33      /**
34       Returns number of Celsius degrees equal to
35       degreesF Fahrenheit degrees.
36      */
37      public static double toCelsius(double degreesF)
38      {
39
40           return 5*(degreesF - 32)/9;
41      }

42      public static void main(String[] args)
43      {
44          double degreesF, degreesC;
45
46          Scanner keyboard = new Scanner(System.in);
47          System.out.println("Enter degrees Fahrenheit:");
48          degreesF = keyboard.nextDouble();
49
50          degreesC = toCelsius(degreesF);
51
```

Because this is in the definition of the class `Temperature`*, this is equivalent to* `Temperature.toCelsius(degreesF)`.

(continued)

Another Class with a `main` Added (Part 3 of 4)

Display 5.3 Another Class with a `main` Added

```
52          Temperature temperatureObject = new Temperature(degreesC);
53          System.out.println("Equivalent Celsius temperature is "
54                                  + temperatureObject.toString());
55      }
56  }
```

Because `main` is a static method, `toString` must have a specified calling object like `temperatureObject`.

SAMPLE DIALOGUE

```
Enter degrees Fahrenheit:
212
Equivalent Celsius temperature is 100.0 C
```

Another Class with a `main` Added (Part 4 of 4)

- 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 : Initialization

- 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
  - E.g. Human.java > populationCount


```java
public class Human
{
  private String name;
  private static int populationCount = 0;
  public Human(String aName)
  {
    name = aName;
    populationCount++;
  }
  public static int getPopulation()
  {
    return populationCount;
  }
}
```

Human.java

```java
public class HumanDemo
{
  public static void main(String[] args)
  {
    for (int i = 0; i < 10; i++)
    {
      Human newHuman = new Human("Person " + i);
      System.out.println("Current population: "
                         + Human.getPopulation());
    }
  }
}
```

HumanDemo.java

Example - Human.java

- 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 double PI = 3.14159;
```

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

```
int year = RoundStuff.PI;
```

Public Static Variables : Constants

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;
```

Display 5.6 Some Methods in the Class `Math`

The `Math` class is in the `java.lang` package, so it requires no `import` statement.

```
public static double pow(double base, double exponent)
```

Returns `base` to the power `exponent`.

EXAMPLE

`Math.pow(2.0,3.0)` returns `8.0`.

(continued)

Some Methods in the Class `Math` (Part 1 of 5)

Display 5.6 Some Methods in the Class `Math`

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the `argument`. (The method name `abs` is overloaded to produce four similar methods.)

EXAMPLE

`Math.abs(-6)` and `Math.abs(6)` both return 6. `Math.abs(-5.5)` and `Math.abs(5.5)` both return `5.5`.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

Returns the minimum of the arguments `n1` and `n2`. (The method name `min` is overloaded to produce four similar methods.)

EXAMPLE

`Math.min(3, 2)` returns 2.

(continued)

Display 5.6 Some Methods in the Class `Math`

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments n1 and n2. (The method name max is overloaded to produce four similar methods.)

EXAMPLE

`Math.max(3, 2)` returns 3.

```
public static long round(double argument)
public static int round(float argument)
```

Rounds its `argument`.

EXAMPLE

`Math.round(3.2)` returns 3; `Math.round(3.6)` returns 4.

(continued)

Some Methods in the Class `Math` (Part 3 of 5)

Display 5.6 Some Methods in the Class `Math`

```
public static double ceil(double argument)
```

Returns the smallest whole number greater than or equal to the `argument`.

EXAMPLE

`Math.ceil(3.2)` and `Math.ceil(3.9)` both return `4.0`.

(continued)

Some Methods in the Class `Math` (Part 4 of 5)

Display 5.6 Some Methods in the Class `Math`

`public static double floor(double argument)`

Returns the largest whole number less than or equal to the `argument`.

EXAMPLE

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

`public static double sqrt(double argument)`

Returns the square root of its `argument`.

EXAMPLE

`Math.sqrt(4)` returns `2.0`.

Some Methods in the Class `Math` (Part 5 of 5)

- *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 **`boolean`**, **`byte`**, **`short`**, **`long`**, **`float`**, **`double`**, and **`char`** are (in order) **`Boolean`**, **`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 **`Boolean`**, **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`** to their corresponding primitive type are (in order) **`booleanValue, byteValue`**, **`shortValue`**, **`intValue`**, **`longValue`**, **`floatValue`**, **`doubleValue`**, and **`charValue`**
  - None of these methods take an argument

```
int i = integerObject.intValue();
```

Wrapper Classes

- 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;
```

Automatic Boxing and Unboxing

- 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

Constants and Static Methods in Wrapper Classes

Display 5.8 Some Methods in the Class Character

The class `Character` is in the `java.lang` package, so it requires no `import` statement.

```
public static char toUpperCase(char argument)
```

Returns the uppercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return `'A'`.

```
public static char toLowerCase(char argument)
```

Returns the lowercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return `'a'`.

```
public static boolean isUpperCase(char argument)
```

Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLE

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Character.isUpperCase('%')` both return `false`.

(continued)

Display 5.8 Some Methods in the Class Character

`public static boolean isLowerCase(char argument)`

Returns `true` if its `argument` is a lowercase letter; otherwise returns `false`.

EXAMPLE

`Character.isLowerCase('a')` returns `true`. `Character.isLowerCase('A')` and `Character.isLowerCase('%')` both return `false`.

`public static boolean isWhitespace(char argument)`

Returns `true` if its `argument` is a whitespace character; otherwise returns `false`. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character (`'\t'`), and the line break character (`'\n'`).

EXAMPLE

`Character.isWhitespace(' ')` returns `true`. `Character.isWhitespace('A')` returns `false`.

(continued)

Some Methods in the Class Character (Part 2 of 3)

Display 5.8 Some Methods in the Class Character

```
public static boolean isLetter(char argument)
```

Returns `true` if its `argument` is a letter; otherwise returns `false`.

EXAMPLE

`Character.isLetter('A')` returns `true`. `Character.isLetter('%')` and `Character.isLetter('5')` both return `false`.

```
public static boolean isDigit(char argument)
```

Returns `true` if its `argument` is a digit; otherwise returns `false`.

EXAMPLE

`Character.isDigit('5')` returns `true`. `Character.isDigit('A')` and `Character.isDigit('%')` both return `false`.

```
public static boolean isLetterOrDigit(char argument)
```

Returns `true` if its `argument` is a letter or a digit; otherwise returns `false`.

EXAMPLE

`Character.isLetterOrDigit('A')` and `Character.isLetterOrDigit('5')` both return `true`. `Character.isLetterOrDigit('&')` returns `false`.

```
1  import java.util.Scanner;

2  /**
3  Illustrate the use of a static method from the class Character.
4  */
5
6  public class StringProcessor
7  {
8      public static void main (String[] args)
9      {
10         System.out.println("Enter a one line sentence:");
11         Scanner keyboard = new Scanner(System.in);
12         String sentence = keyboard.nextLine();
13
14         sentence = sentence.toLowerCase();
15         char firstCharacter = sentence.charAt(0);
16         sentence = Character.toUpperCase(firstCharacter)
17                    + sentence.substring(1);
18
19         System.out.println("The revised sentence is:");
20         System.out.println(sentence);
21     }
22 }
```

```
Enter a one line sentence:
is you is OR is you ain't my BABY?
The revised sentence is:
Is you is or is you ain't my baby?
```

Example

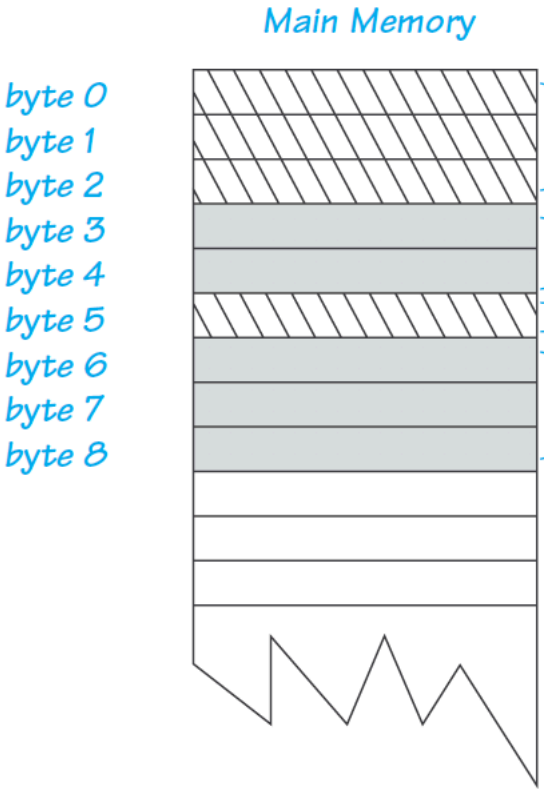
- Static methods and static variables
  - The Math class and wrapper classes
  - Automatic boxing and unboxing mechanism
- References and class parameters
  - Variables and Memory
  - Using and misusing references

Outline

- A computer has two forms of memory
- *Secondary memory* is used to hold files for "permanent" storage
- *Main memory (a.k.a Primary 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

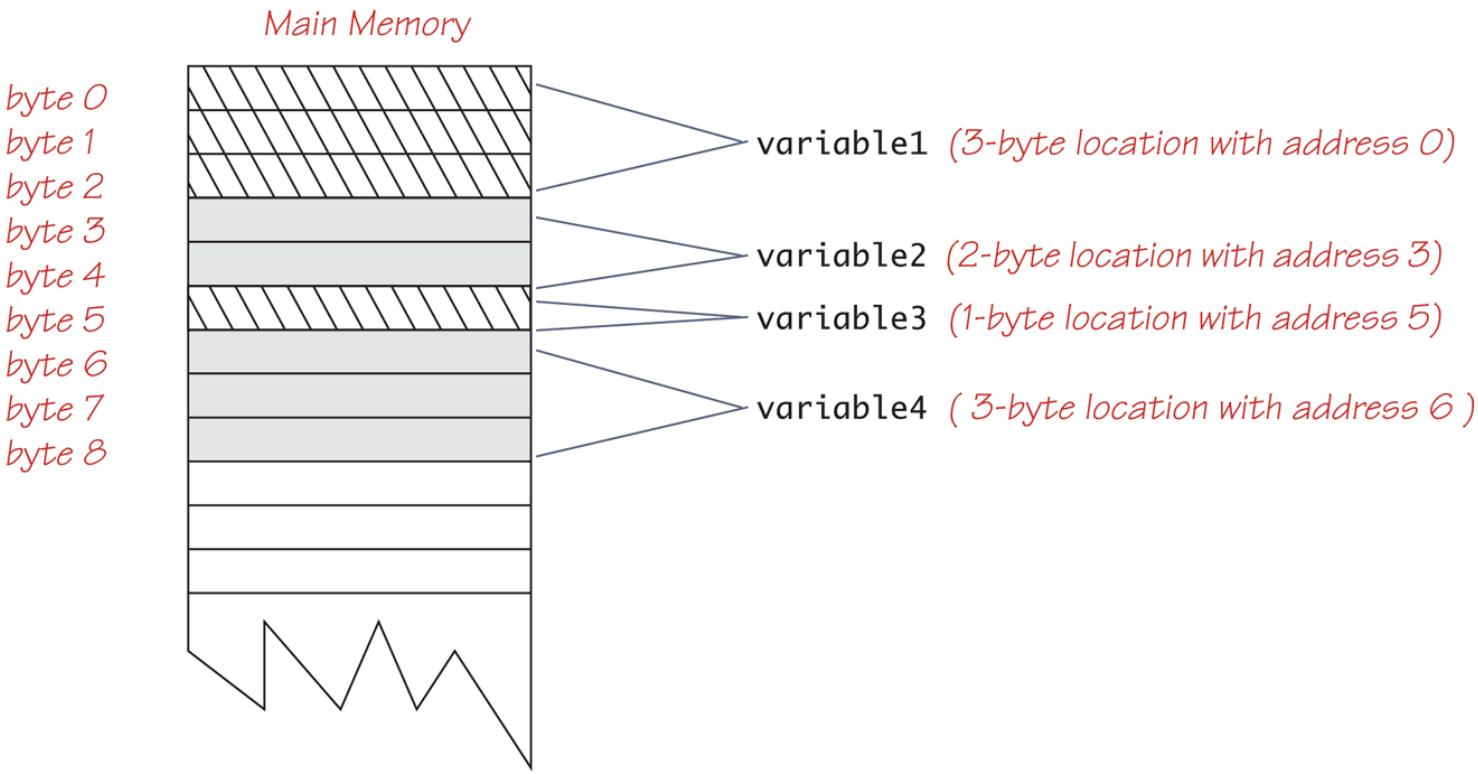
- 1Byte = 8 bits
- 1KB = 2^{10} = 1024 ~ 10^3
- 1MB = 2^{20} =1K x 1K ~ 10^6
- 1GB = 2^{30} =1K x 1M ~ 10^9
- 2^{32}B = 4 x 2^{30} = 4GB

Conversions

- 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 and Memory

Display 5.10 **Variables in Memory**



Variables in Memory

- 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 theses variables will produce a change to the object named by the other variable, since they are the same object

```
variable2 = variable1;
```

References

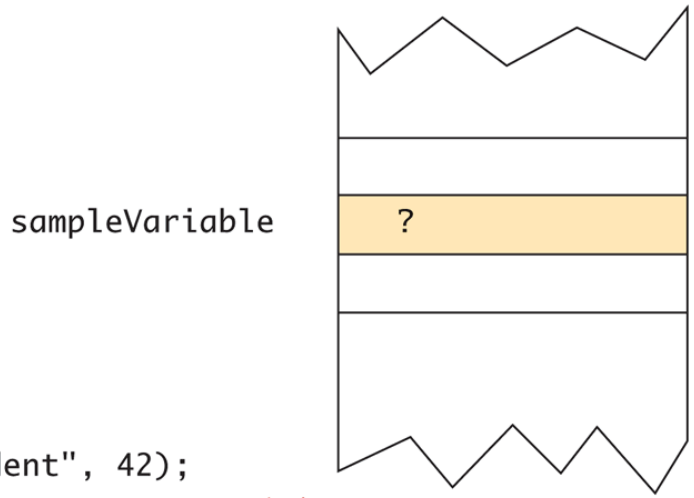
Display 5.12 Class Type Variables Store a Reference

```
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.*

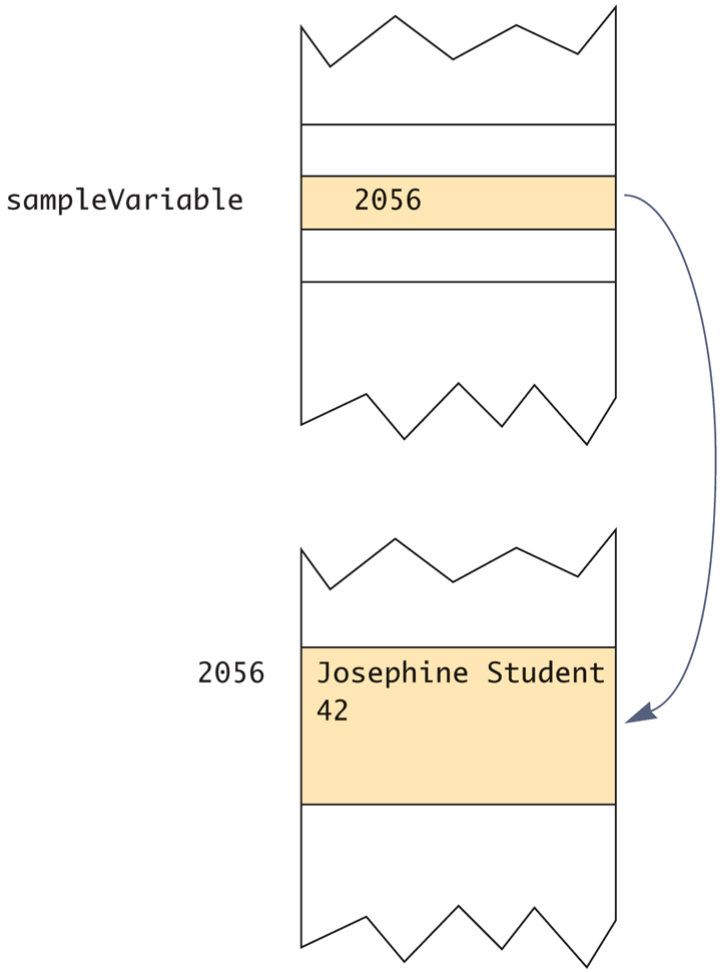



```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

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

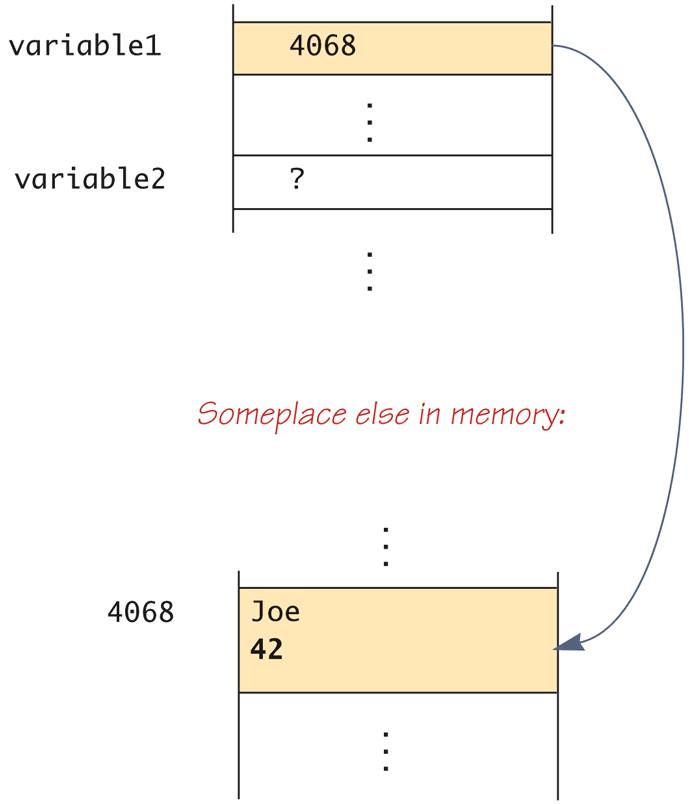


For emphasis, we made the arrow point to the memory location referenced.

2 of 2)

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

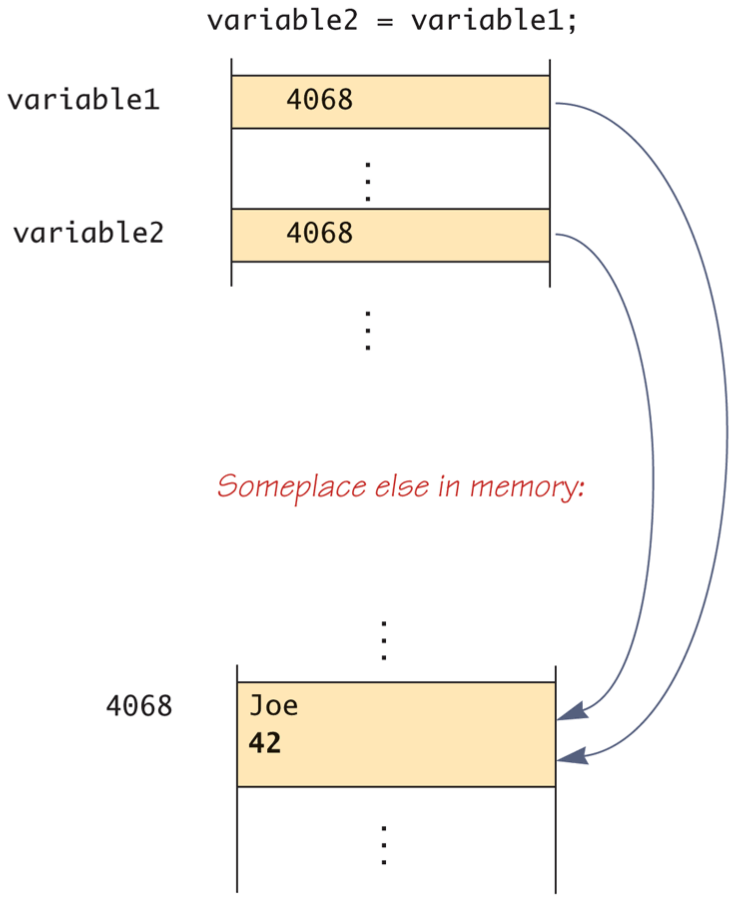
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

(Part 1 of 3)

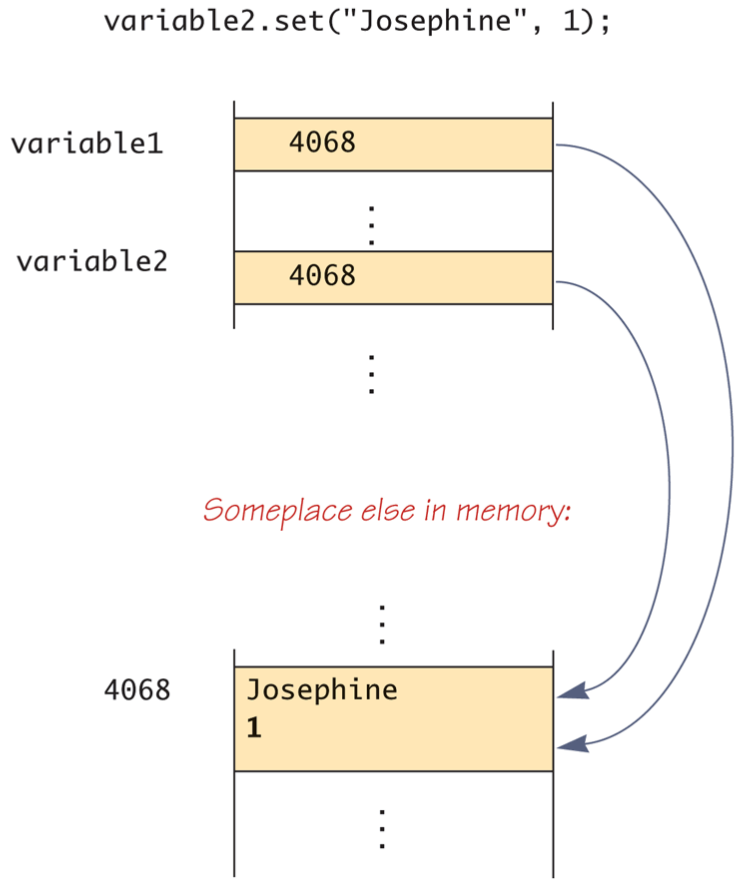
Display 5.13 Assignment Operator with Class Type Variables



(continued)

(Part 2 of 3)

Display 5.13 Assignment Operator with Class Type Variables



art 3 of 3)

- 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)

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

Class Parameters

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`.

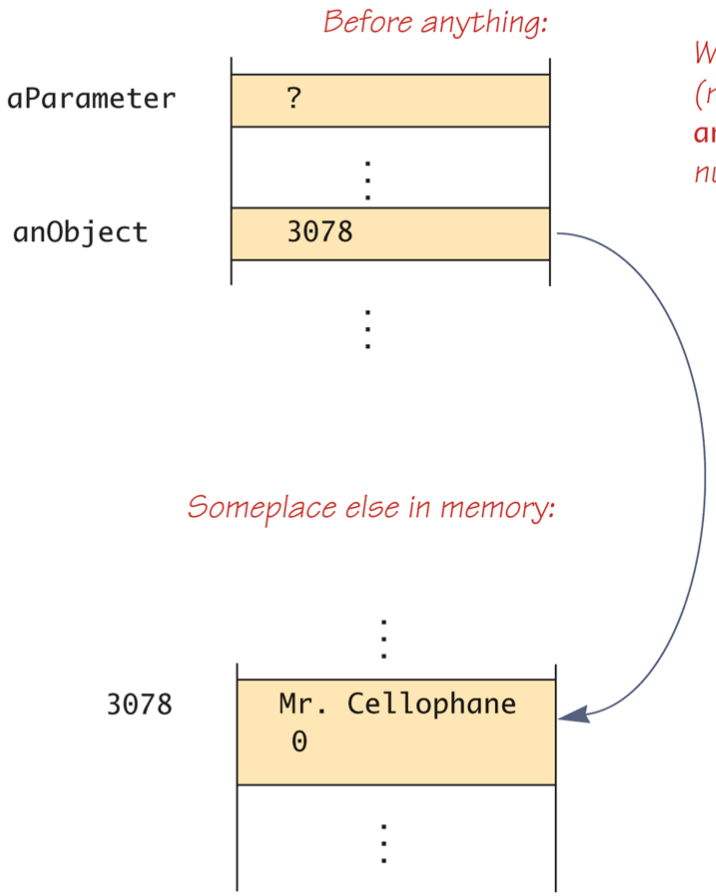
SAMPLE DIALOGUE

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

E.g. ClassParameterDemo

Parameters of a Class Type

Display 5.15 Memory Picture for Display 5.14

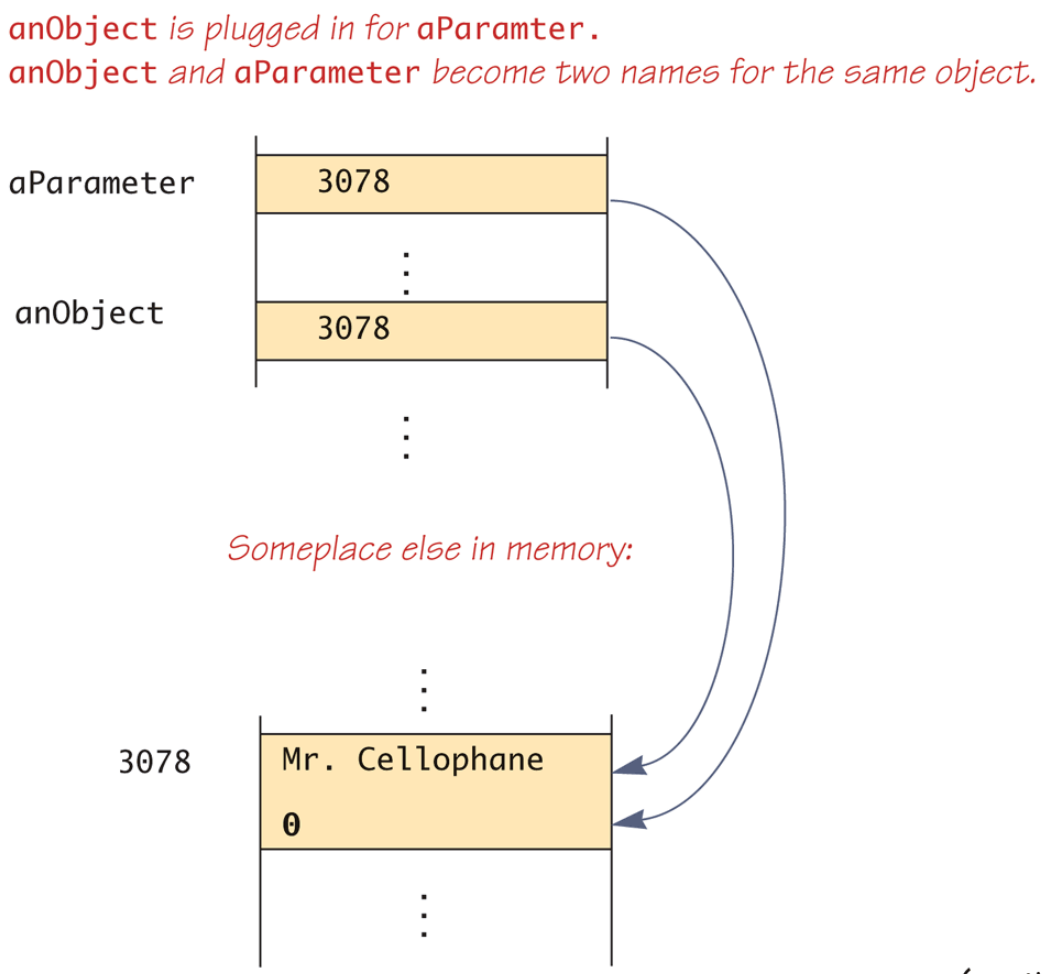


We do not know what memory address (reference) is stored in the variable `anObject`*. Let's say it is 3078. The exact number does not matter.*

(continued)

(Display 5.14 (1 of 3))

Display 5.15 Memory Picture for Display 5.14



(continued)

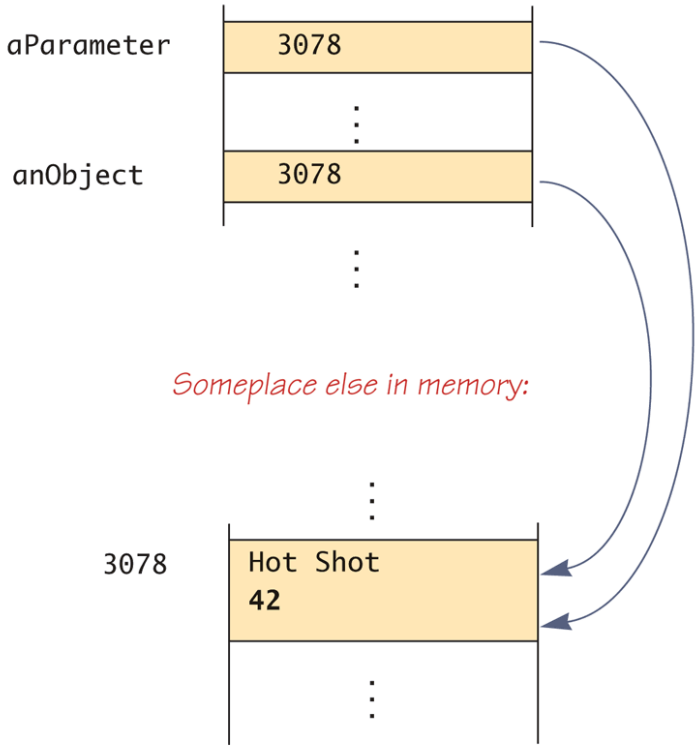
lay 5.14 t 2 of 3)

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.*



isplay 5.14 Part 3 of 3)

- 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

Differences Between Primitive and Class-Type Parameters

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)

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.

Comparing Parameters of a Class Type and a Primitive Type (Part 2 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 1 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      }

    <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
```

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

(Part 2 of 2)

- 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"

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

- **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 Constant null

- 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

Pitfall: Null Pointer Exception

- 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)))**

The new Operator and Anonymous Objects

Display 5.18 **Use of the method `Double.parseDouble`**

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

3  public class InputExample
4  {
5      public static void main(String[] args)
6      {
7          Scanner keyboard = new Scanner(System.in);

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

12         String delimiters = ", "; //Comma and blank space
13         StringTokenizer numberFactory =
14             new StringTokenizer(inputLine, delimiters);
```

(continued)

Another Approach to Keyboard Input Using `Double.parseDouble` (Part 1 of 3)

Display 5.18 **Use of the method `Double.parseDouble`**

```
15          String string1 = null;
16          String string2 = null;
17          if (numberFactory.countTokens() >= 2)
18          {
19              string1 = numberFactory.nextToken();
20              string2 = numberFactory.nextToken();
21          }
22          else
23          {
24              System.out.println("Fatal Error.");
25              System.exit(0);
26          }

27          double number1 = Double.parseDouble(string1);
28          double number2 = Double.parseDouble(string2);

29          System.out.print("You input ");
30          System.out.println(number1 + " and " + number2);
31      }
32  }
```

(continued)

Display 5.18 **Use of the method** `Double.parseDouble`

SAMPLE DIALOGUE

```
Enter two numbers on a line.
Place a comma between the numbers.
Extra blank space is OK.
    41.98,  42
You input is 41.98 and 42.0
```

Another Approach to Keyboard Input Using Double.parseDouble (Part 3 of 3)

- 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 insure that there will be no ***privacy leaks***
- For a class type instance variable, however, adding the **`private`** modifier alone is not sufficient

Using and Misusing References

- 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 is still alive
      . . .
```

Designing A Person Class: Instance Variables

- 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

Designing 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);
  }
}
```

A Person Class Constructor

- 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 Class Invariant

- 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` and `datesMatch` Methods

```
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 equals 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 datesMatch 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);
}
```

Designing a `Person` Class: the `toString` Method

- 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 Constructors

```
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 Primitive 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);
}
```

Copy Constructor for a Class with Class Type Instance Variables

- 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
}
```

Pitfall: Privacy Leaks

- 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

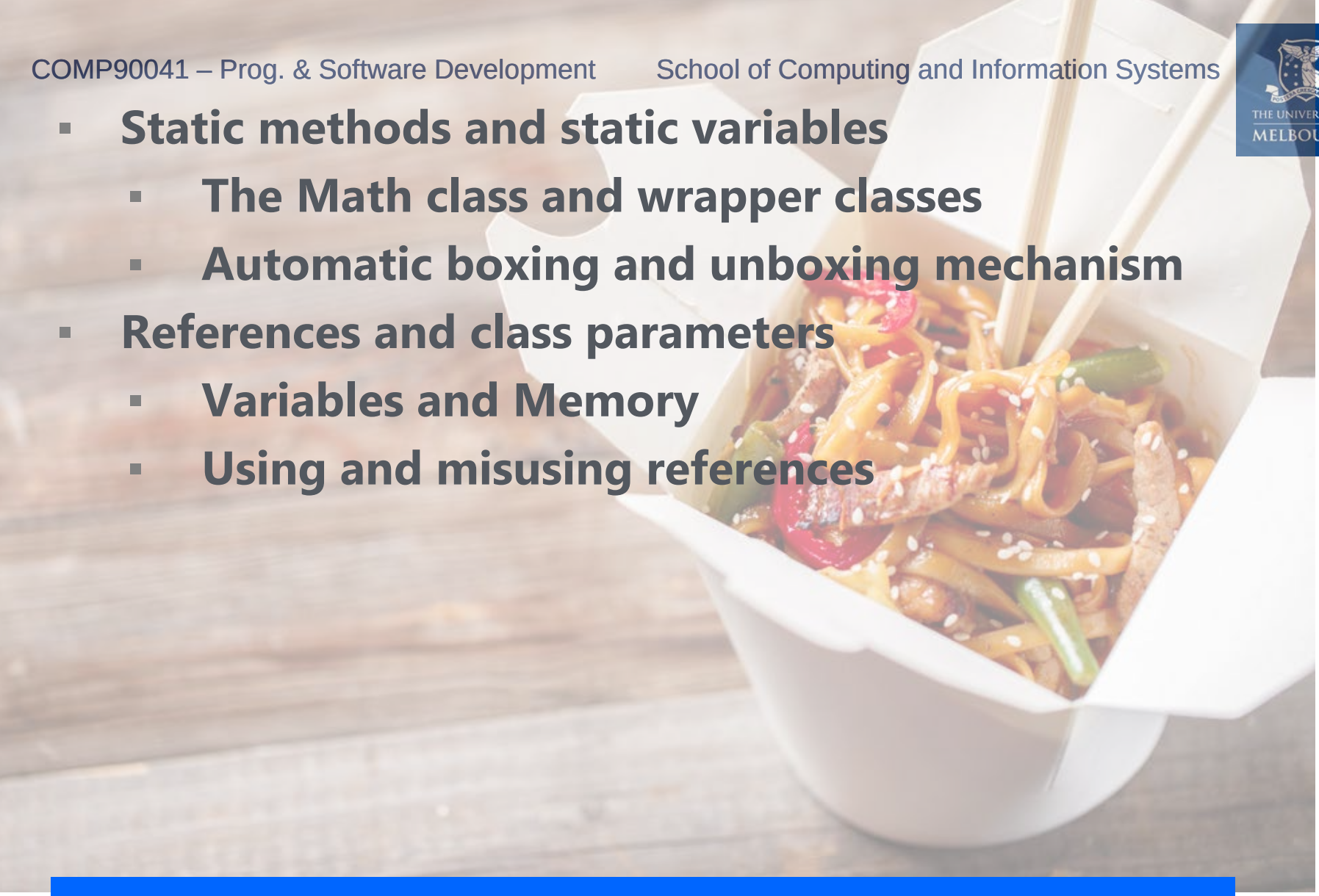
Mutable and Immutable Classes

- 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

Mutable and Immutable Classes

- 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



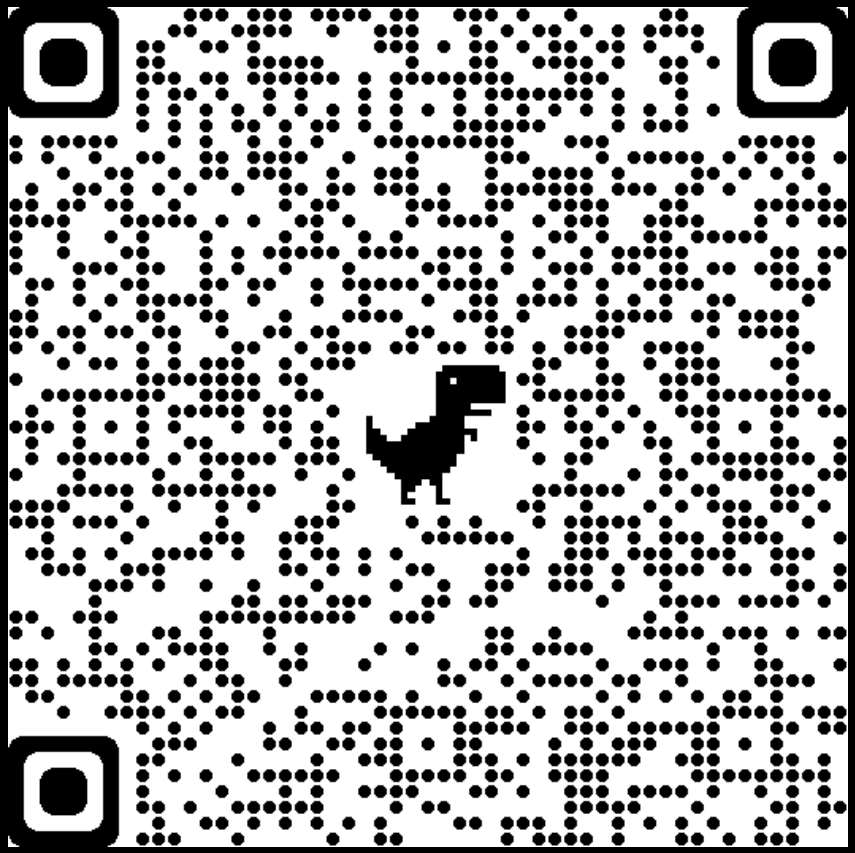
- Static methods and static variables
  - The Math class and wrapper classes
  - Automatic boxing and unboxing mechanism
- References and class parameters
  - Variables and Memory
  - Using and misusing references

Learning Outcomes

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections

Please fill in this microblog.



http://go.unimelb.edu.au/5o8i.

- Class structure
- Instance variables and methods
- Different types of methods and their invocation
- Information hiding & Encapsulation
- Overloading methods
- Class constructors

Learning Outcomes