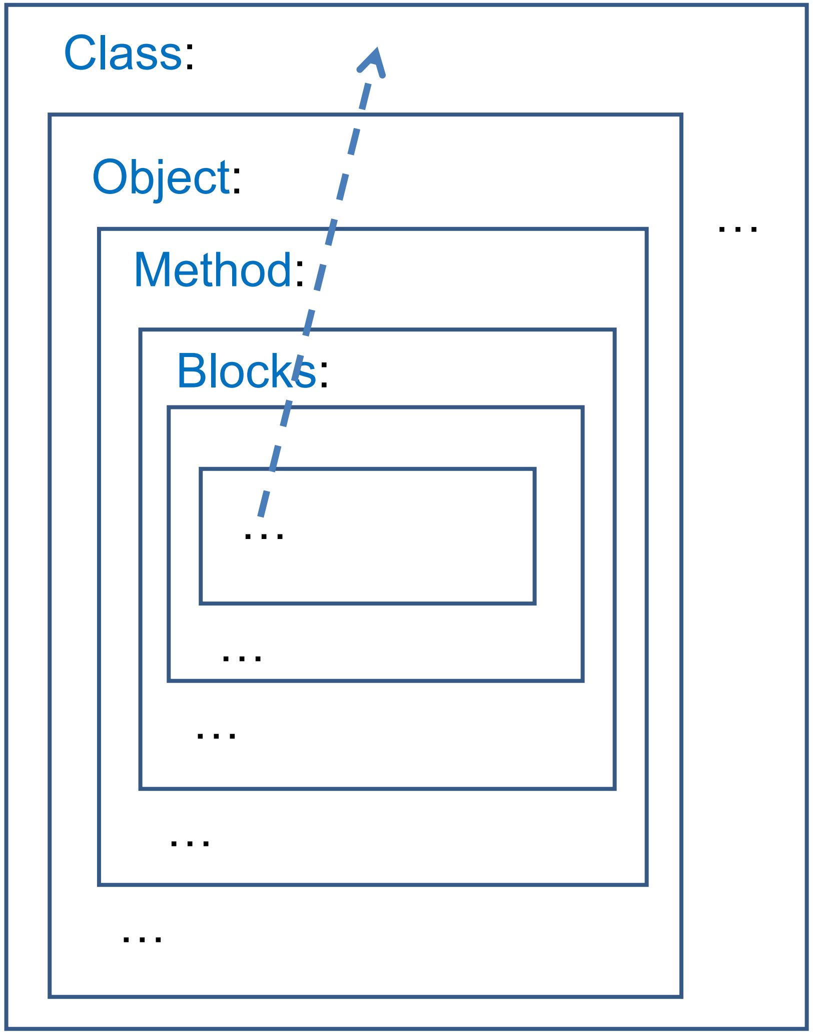
Class Design – Part II

CIS*2430 (Fall 2021)

Scoping Rules



(1) Define before use (or else get “undefined” errors)

(2) Can’t redefine a variable in the same scope (or else get “redefined” errors)

(3) An outside scope can’t see the inside scopes, but an inside scope can see the outside scopes level-by-level in the order of inside-out, and the search stops as soon as a desired name is found in the process.

An Example for Scoping Rules

```
public class Die {
    public static final int COMMON_MAX_FACES=6;

    private int maxFaces;
    private int faceValue;

    public void setFaceValue(int value) {
        faceValue = value;
    }
    ….
}
```

- At the class level, only "COMMON_MAX_FACES" is visible.
- At the object level, "maxFaces" and "faceValue" are visible, but "COMMON_MAX_FACES" is also visible.
- In method "setFaceValue", "value" is visible, but "maxFaces", "faceValue", and "COMMON_MAX_FACES" are also visible.

Static Methods

- A *static method* belongs to a class and can be invoked using the class name in place of a calling object:

  returnedValue = MyClass.myMethod(arguments);

- A static method can't refer to an instance variable, nor can it invoke an instance 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.
  - However, a static method can refer to a static variable and invoke another static method.

Tip: Add main in Any Class

- Although the main method is often defined in a class separate from the other classes of a program, it can also be contained within a regular class definition:

  - Such a class 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 is especially useful for diagnostic purpose.

Arguments for main

- Here is a program that expects three string arguments:

```
public class SomeProgram {
    public static void main(String[] args) {
        if (args.length > 2)
           System.out.println(args[0] + " "  + args[2] + args[1]);
    }
}
```

- Arguments for the **main** method must be provided from the command line when the program is run:

```
java SomeProgram Hi ! there
```

Static Variables

- A *static variable* belongs to the class, not to any specific object:
  - Only one copy of a static variable per class, unlike an instance variable 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.

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.

- 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 a constant:
  - The value of a static defined constant cannot be altered; therefore, it is safe to make it **public**

```
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 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** (the base of the natural logarithm system) and **PI** (3.1415):

```
area = Math.PI * radius * radius;
```

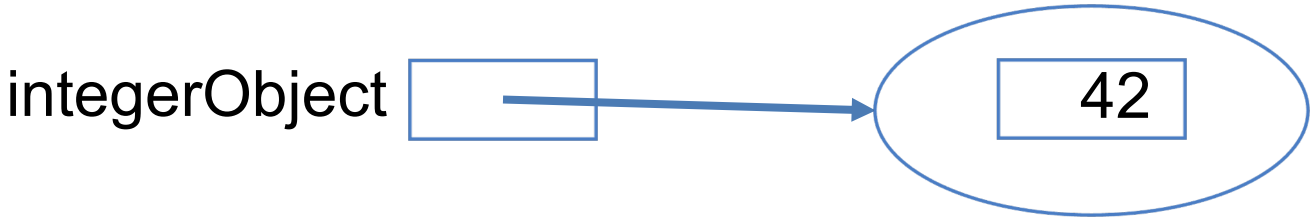
Wrapper Classes

- *Wrapper classes* provide a class type corresponding to each of the primitive types:
  - The wrapper classes for the primitive types **byte**, **short**, **long**, **float**, **double**, and **char** are **Byte**, **Short**, **Long**, **Float**, **Double**, and **Character**, respectively.
- Wrapper classes also contain quite a few useful predefined constants and static methods.

Wrapper Classes

- *Boxing*: the process of converting a value of a primitive type to an object of its wrapper class.

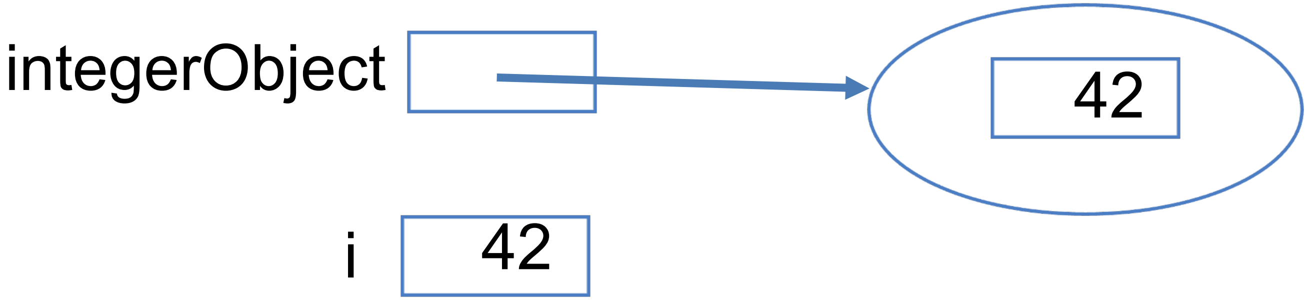
```
Integer integerObject = new Integer(42);
```



Wrapper Classes

- *Unboxing*: converting methods from objects of wrapper classes **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, and **Character** to their corresponding primitive type are **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue**, **doubleValue**, and **charValue**.

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



Automatic Boxing and Unboxing

- Starting with version 5.0, Java can automatically do boxing and unboxing.

- For example,

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

  is simplified to:

  ```
  Integer integerObject = 42;
  ```

- Similarly,

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

  is reduced to:

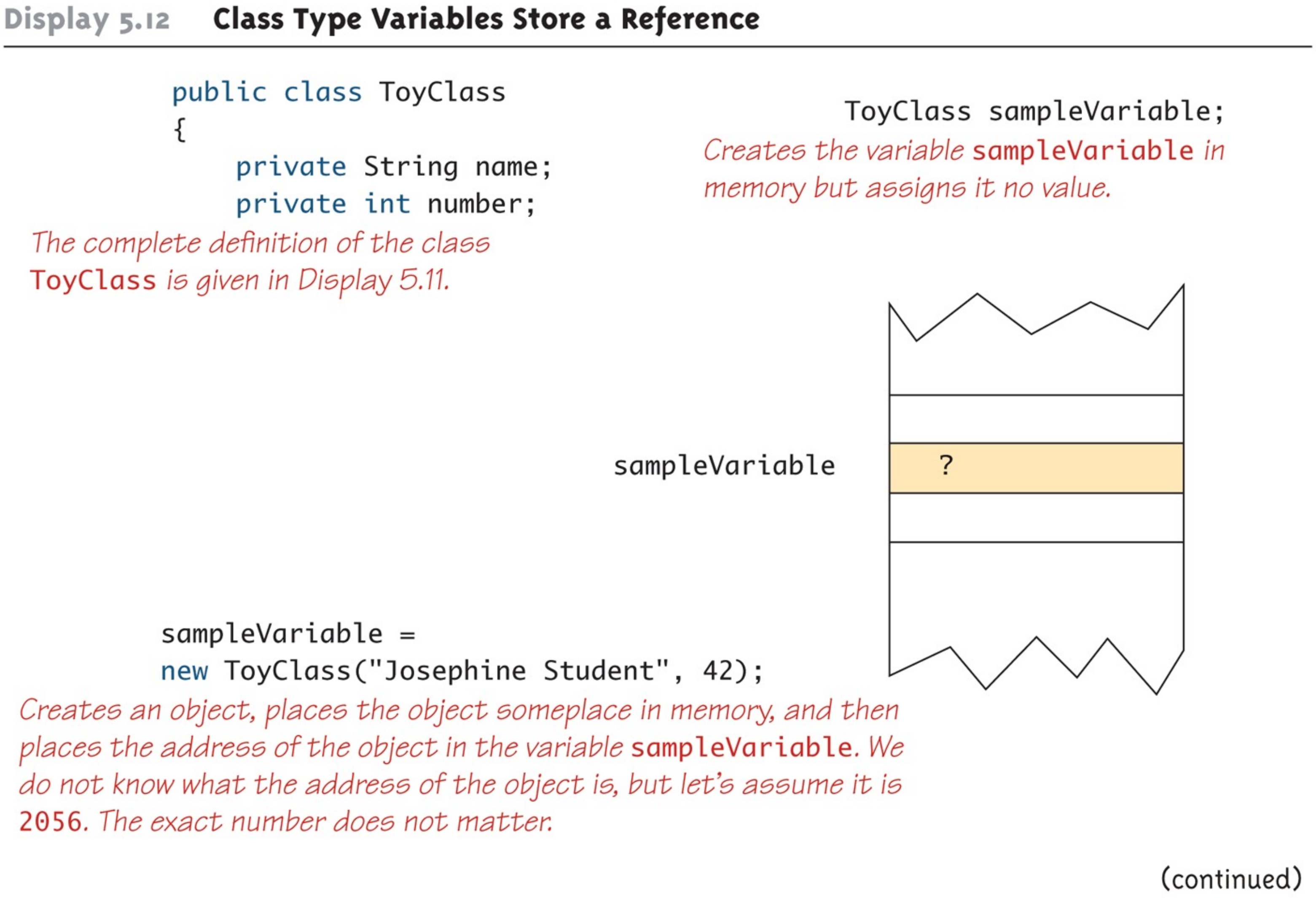
  ```
  int i = integerObject;
  ```

Static Constants and Methods

- Wrapper classes include useful constants that provide the largest and smallest values for any of the primitive number types:
  - E.g., **Integer.MAX_VALUE**, **Integer.MIN_VALUE**, **Double.MAX_VALUE**, **Double.MIN_VALUE**, …

- 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 **int**, **long**, **float**, and **double**.

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
?

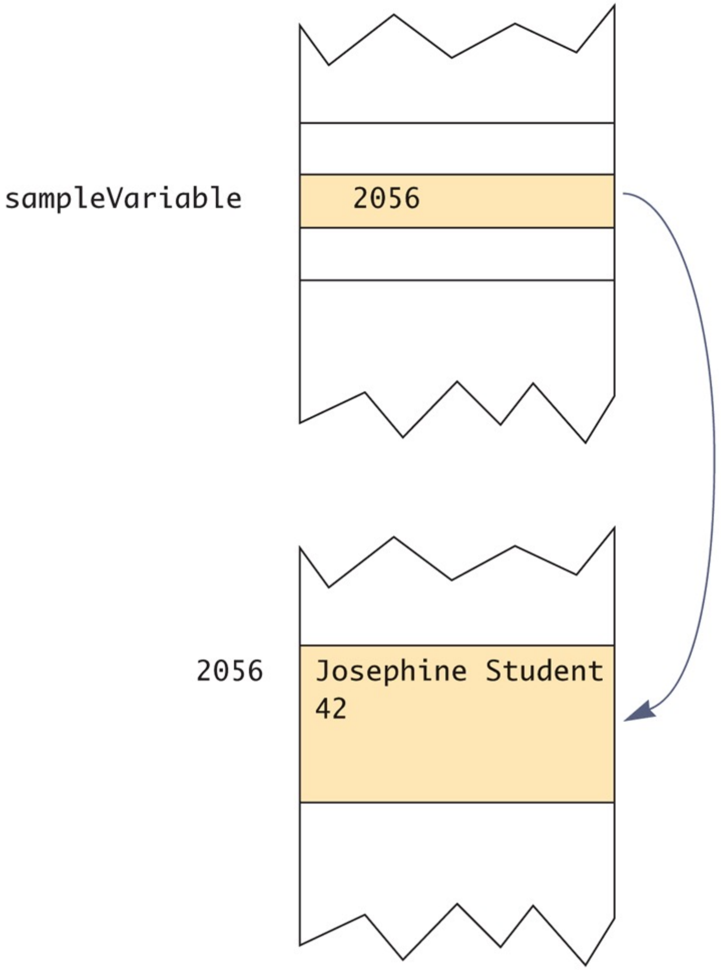
```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

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)

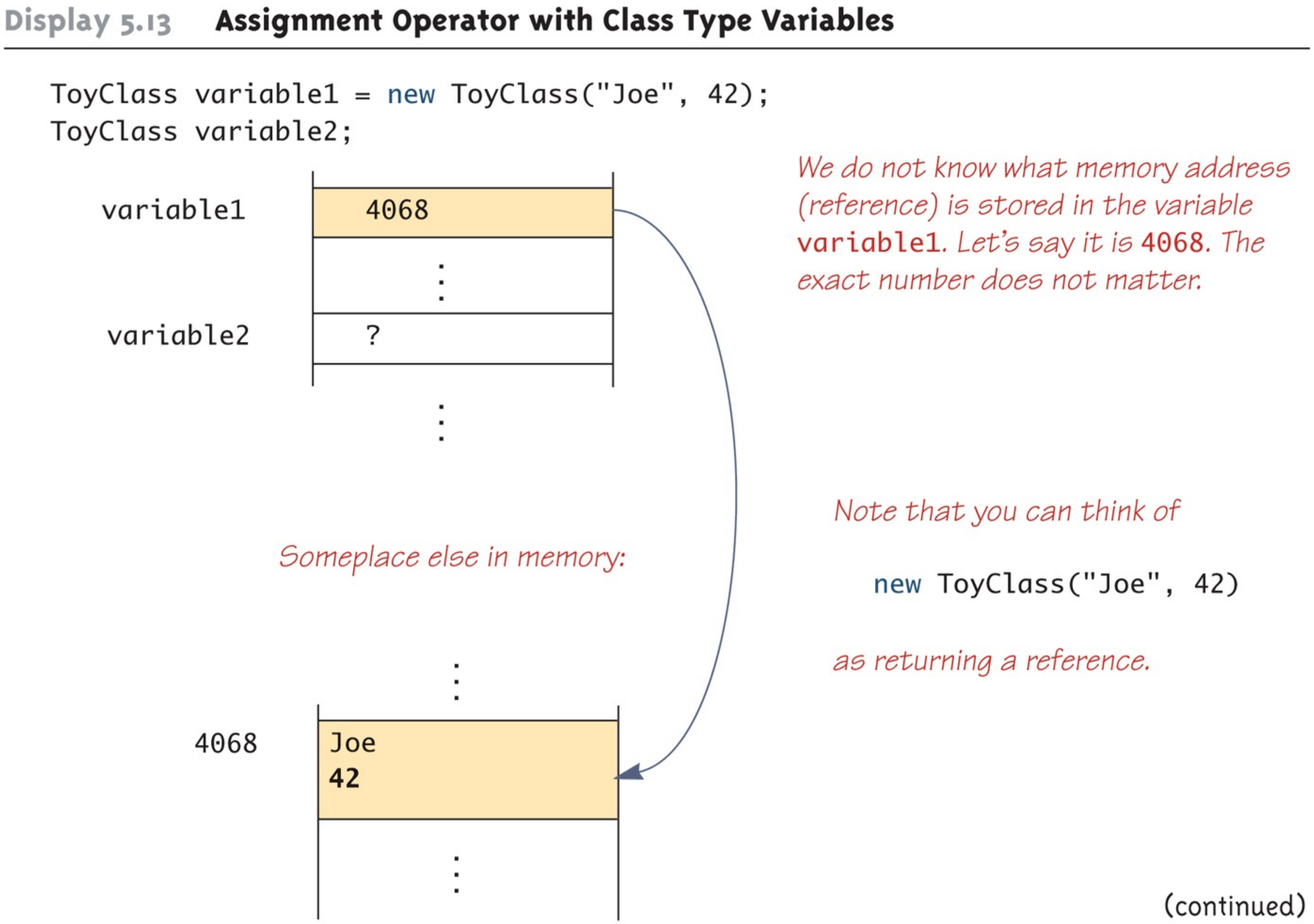
References

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



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

Assignments with References



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.*

Someplace else in memory:

Note that you can think of

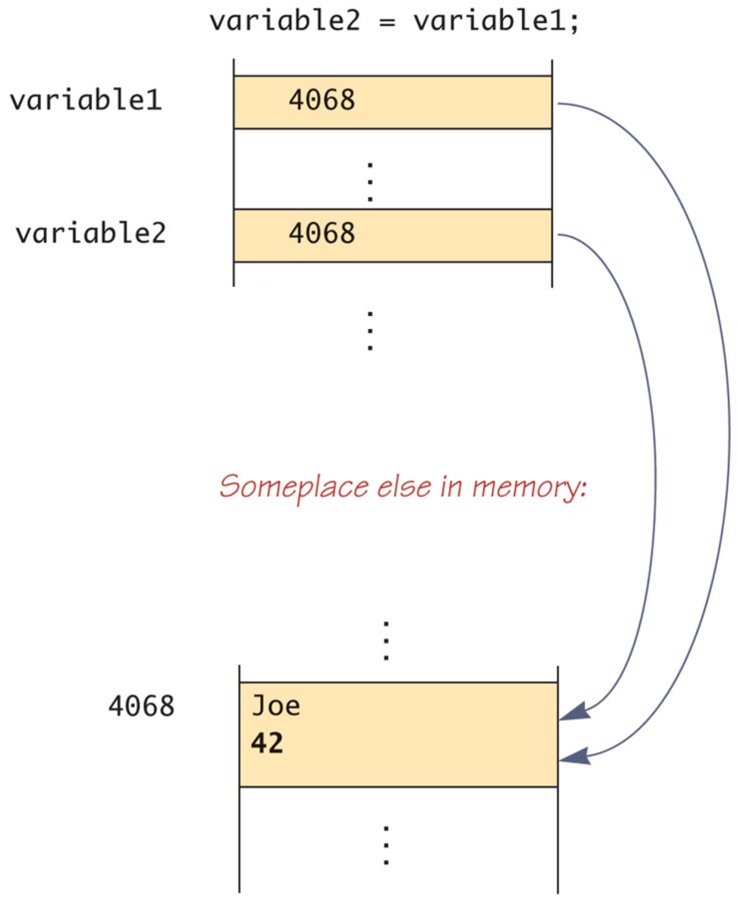
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

Assignments with References

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



```
variable2 = variable1;
```

variable1

4068

variable2

4068

Someplace else in memory:

4068

Joe

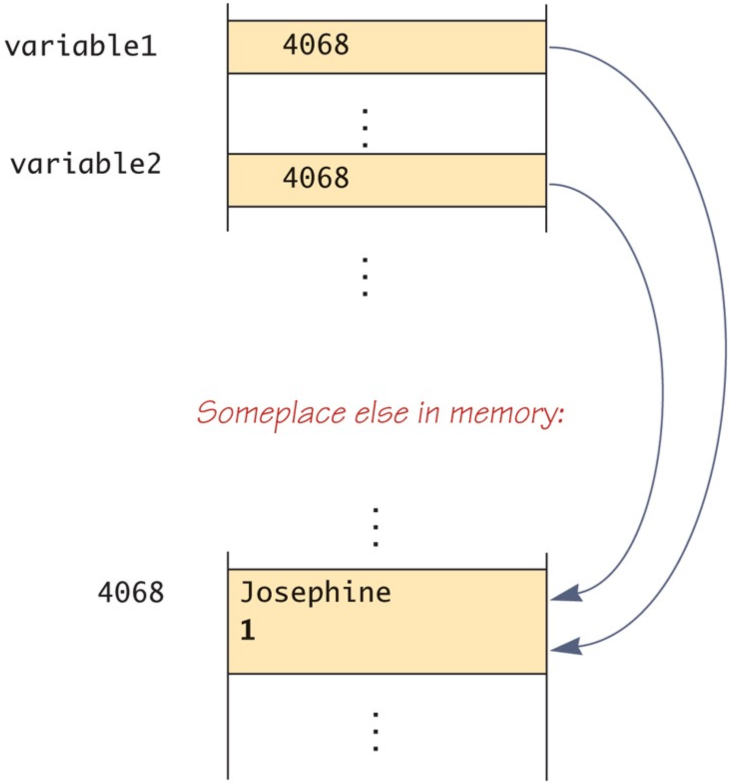
42

(continued)

Assignments with References

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

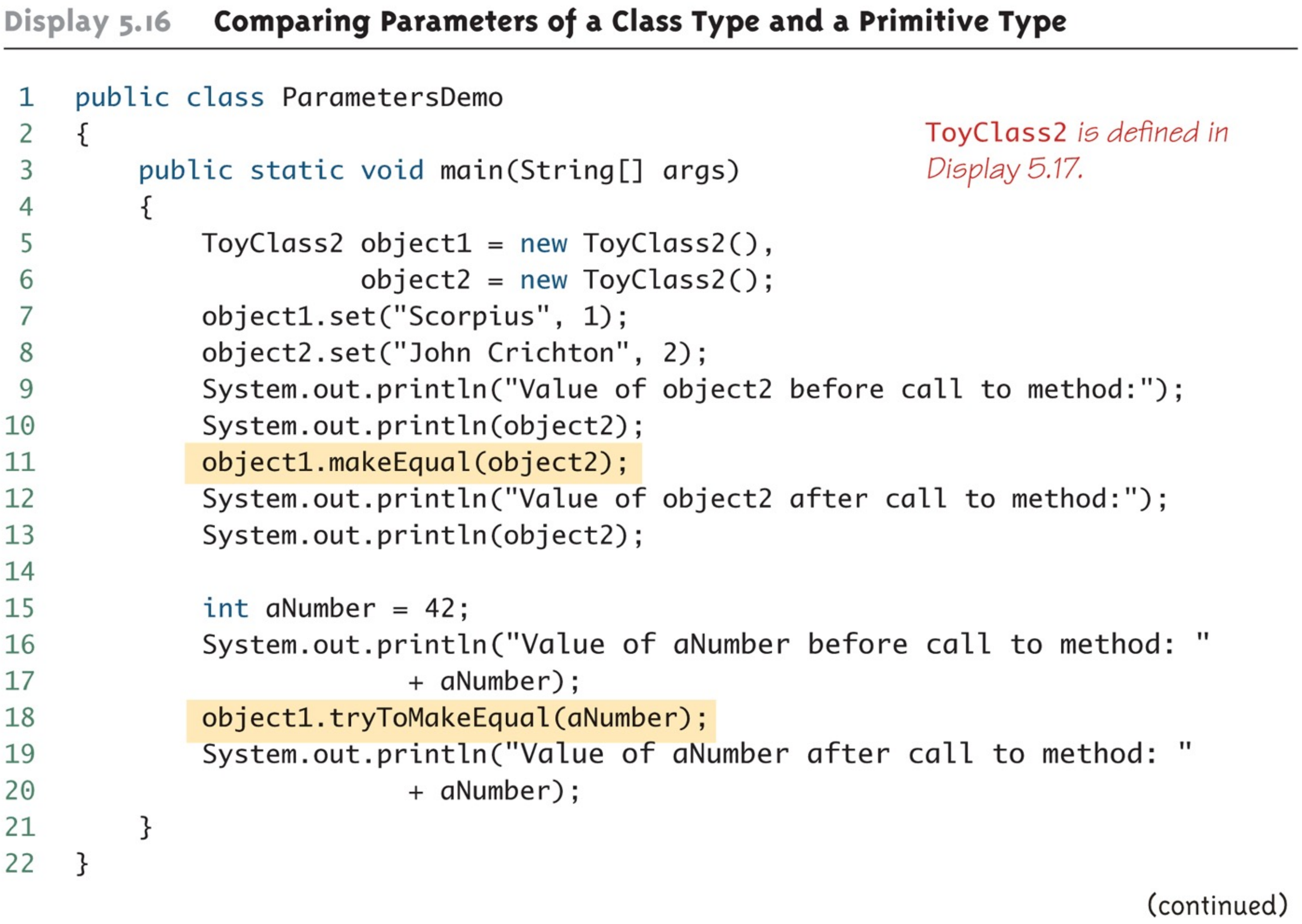
```
variable2.set("Josephine", 1);
```



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 behave differently from primitive type parameters:
  - They appear like parameters in languages that have the *call-by-reference* parameter passing mechanism.

Class vs. Primitive Parameters



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

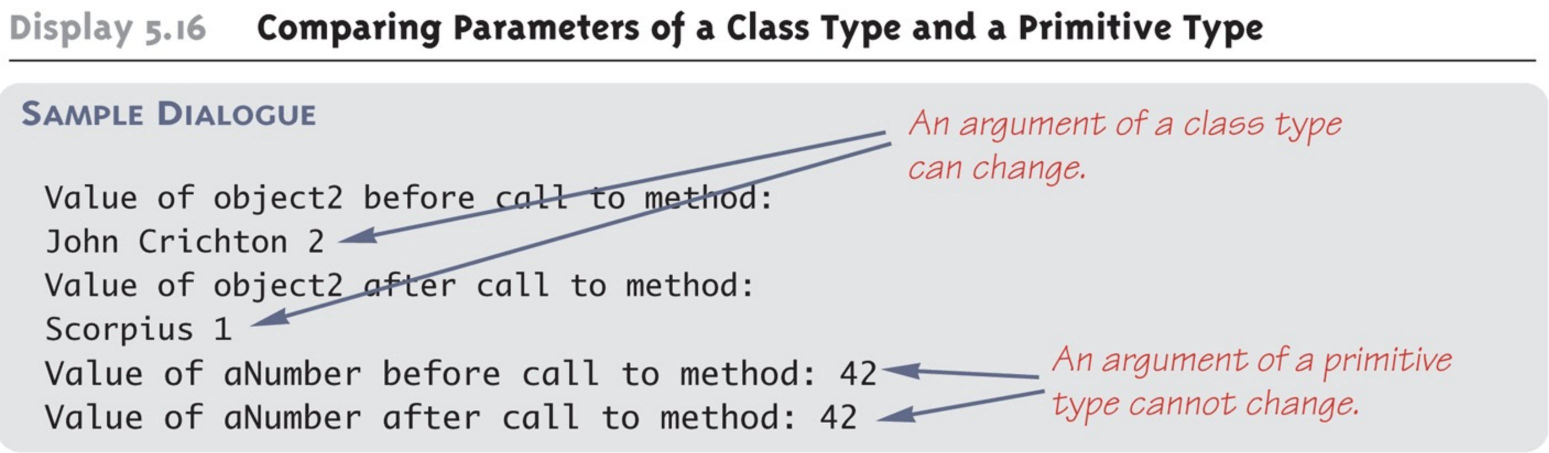
```
public class ParametersDemo
{
    public static void main(String[] args)
    {
        ToyClass2 object1 = new ToyClass2(),
                  object2 = new ToyClass2();
        object1.set("Scorpius", 1);
        object2.set("John Crichton", 2);
        System.out.println("Value of object2 before call to method:");
        System.out.println(object2);
        object1.makeEqual(object2);
        System.out.println("Value of object2 after call to method:");
        System.out.println(object2);

        int aNumber = 42;
        System.out.println("Value of aNumber before call to method: "
                             + aNumber);
        object1.tryToMakeEqual(aNumber);
        System.out.println("Value of aNumber after call to method: "
                             + aNumber);
    }
}
```

`ToyClass2` *is defined in Display 5.17.*

(continued)

Class vs. Primitive Parameters



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

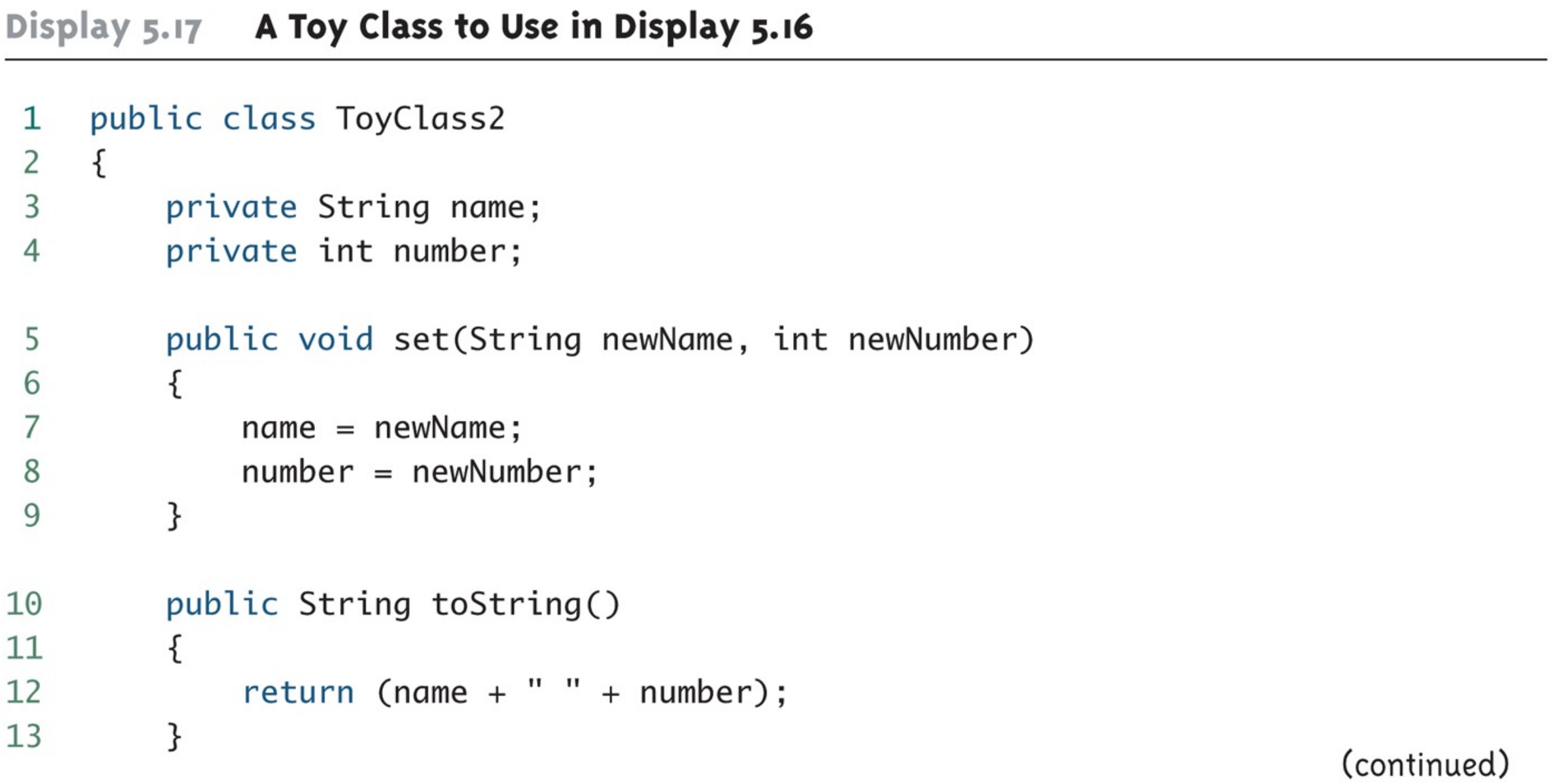
SAMPLE DIALOGUE

```
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.

Toy Class (1/2)



Display 5.17 **A Toy Class to Use in Display 5.16**

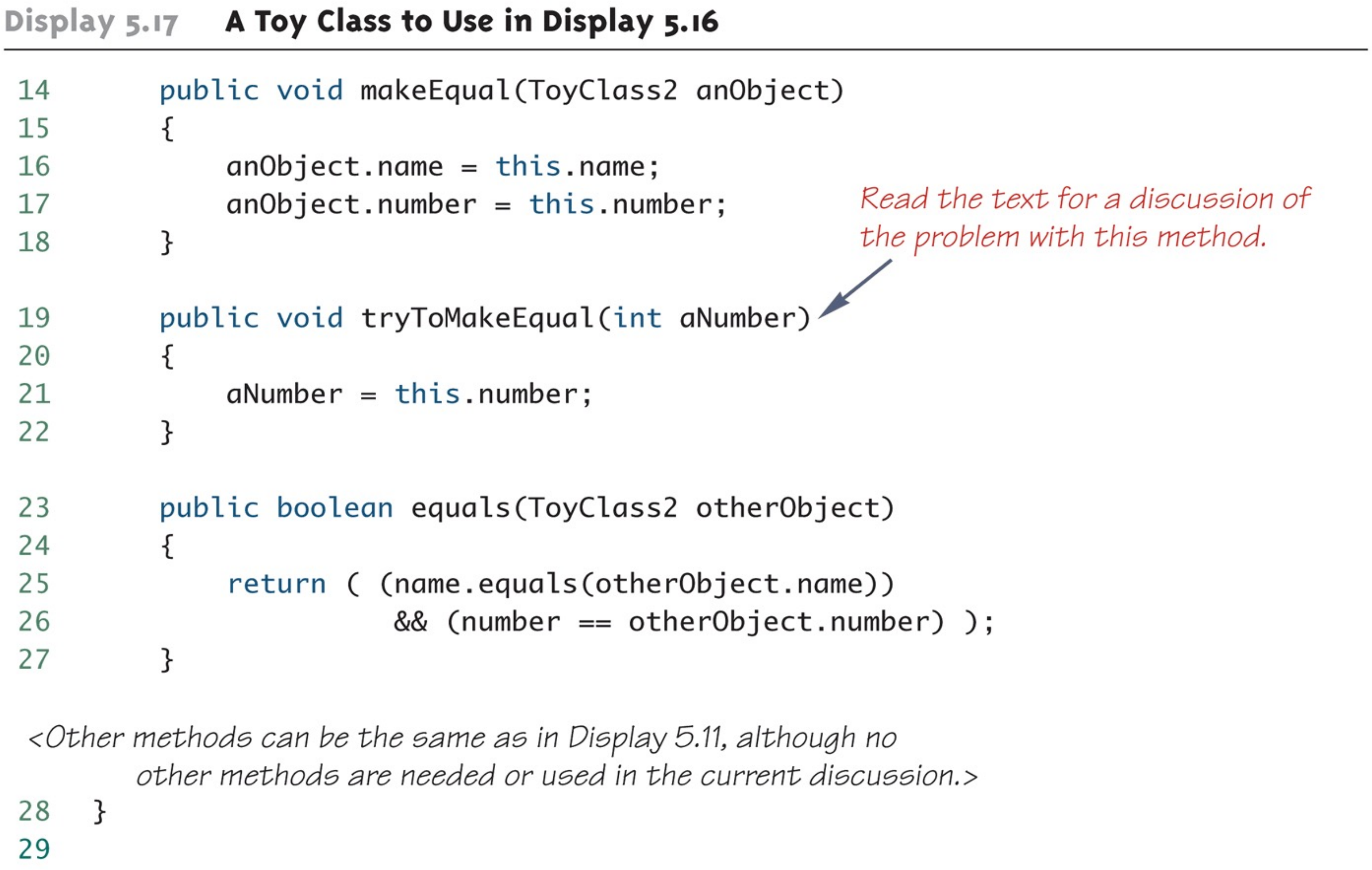
```
public class ToyClass2
{
    private String name;
    private int number;

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

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

(continued)

Toy Class (2/2)



Display 5.17 A Toy Class to Use in Display 5.16

```
    public void makeEqual(ToyClass2 anObject)
    {
        anObject.name = this.name;
        anObject.number = this.number;
    }

    public void tryToMakeEqual(int aNumber)
    {
        aNumber = this.number;
    }

    public boolean equals(ToyClass2 otherObject)
    {
        return ( (name.equals(otherObject.name))
                  && (number == otherObject.number) );
    }

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

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

Pitfall with = and ==

- With variables of a class type, the assignment operator (=) produces two references to the same object:
  - Different from how it behaves with primitive type variables.
- The equality (==) also behaves differently for class type variables:
  - The == operator only checks if two class type variables have the same memory address>
  - Two objects in two different locations whose instance variables have the same values would still test as being "not equal”.

Null-Pointer Exception

- Although 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**.
- Any attempt to do this will result in a "Null Pointer Exception" error message.

Person Class (1/4)

- For privacy, each of the instance variables are declared **private**:

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

- Class invariant: a statement that is true for all objects of the class:
  - 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.
  - Make no sense to have a no-argument constructor.

Person Class (2/4)

```
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);
  }
}
```

Person Class (3/4)

```
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));
}
```

Person Class (4/4)

```
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));
}

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

Copy Constructor

- The copy constructor should create a separate, independent object:

```
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;
 }
```

Unsafe Copy Constructor

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

Safe Copy Constructor

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

Pitfall: Privacy Leaks

- As illustrated in the **Person** class, an incorrectly defined constructor can result in a *privacy leak.*

- A similar problem can occur with incorrectly defined mutator or accessor methods:

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

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

Mutable 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.

Immutable Classes

- A class that contains no methods (other than constructors) that change any instance variables of its objects 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.

Packages

- Java uses *packages* to form libraries of classes.

- A package is a group of classes placed in a directory or folder, which can be imported to another program:
  - The import statement must be located at the start of a program: only blank lines, comments, and package statements may precede it.
  - The program can be in a different directory from the package.

The import Statements

- It is possible to make all the classes in a package available instead of just one class:

  import java.util.*;

  - Note that there is no additional overhead for importing the entire package.

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

The package Statement

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

  package package_name;

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

Package Names & Directories

- A package name is the path name for the directory that contains the related classes.

- To find the full path for a package, Java needs to know both the name of the package and the value of the **CLASSPATH** variable:
  - The **CLASSPATH** variable contains a list of directories (including the current directory, ".") in which Java looks for packages on a particular computer.
  - Java searches the list of directories in order and uses the first directory on the list in which the package is found.

Pitfall for Subdirectories

- When a package is stored in a subdirectory of another directory, importing the top package does not automatically import the subdirectory package:

```
import utilities.numericstuff.*;
import utilities.numericstuff.statistical.*;
```

import both the utilities.numericstuff and utilities.numericstuff.statistical packages.

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: the current directory must be included in the **CLASSPATH** variable; otherwise, Java may not even find the **.class** files for the program itself.

Name Clashes

- In addition to keeping class libraries organized, packages provide a way to deal with *name clashes*:
  - 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.