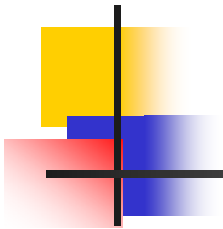
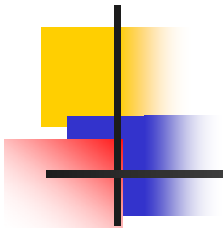
Classes 1/5



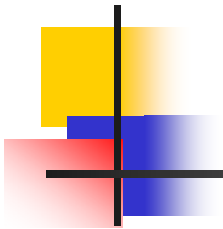
Outline

- Introduction
- Class Definitions
- A Class Is a Type
- The **new** Operator
- Instance Variables and Methods
- Information Hiding
- **public** and **private** Modifiers
- Accessor and Mutator Methods
- Encapsulation



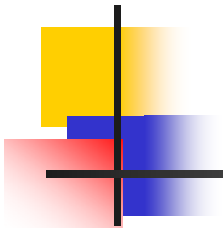
- Introduction

- Classes are the most important language feature that make *object-oriented programming* (*OOP*) possible
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- Classes are central to Java



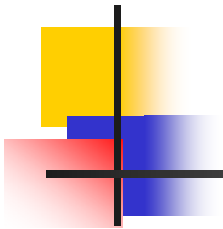
- Class Definitions

- You already know how to use classes and the objects created from them, and how to invoke their methods
  - For example, you have already been using the predefined **`String`** and **`Scanner`** classes
- Now you will learn how to define your own classes and their methods, and how to create your own objects from them



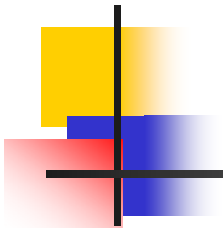
- A Class Is a Type

- A class is a special kind of programmer-defined type, and variables can be declared of a class type
- A value of a class type is called an object or *an instance of the class*
  - If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform



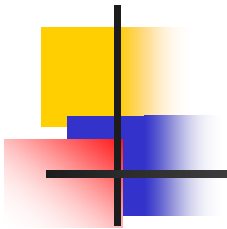
-- Primitive Type Values vs. Class Type Values

- A primitive type value is a single piece of data
- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value



-- The Contents of a Class Definition

- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition



- The **new** Operator

- An object of a class is named or declared by a variable of the class type:

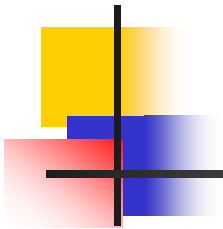
```
ClassName  classVar;
```

- The **new** operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- These can be combined as follows:

```
ClassName classVar = new ClassName();
```



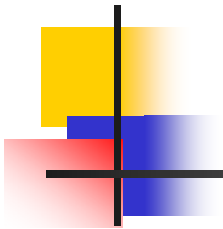
- Instance Variables and Methods ...

- Instance variables can be defined as in the following two examples
  - Note the **`public`** modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

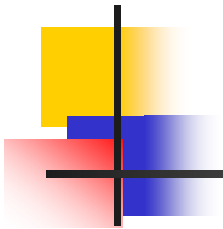
- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```



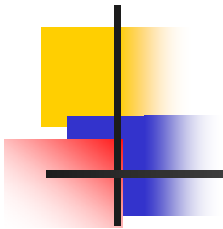
- Information Hiding and Encapsulation

- *Information hiding* is the practice of separating how to use a class from the details of its implementation
  - *Abstraction* is another term used to express the concept of discarding details in order to avoid information overload
- *Encapsulation* means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
  - In Java, hiding details is done by marking them **`private`**



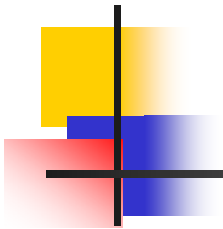
- **public** and **private** Modifiers ...

- The modifier **`public`** means that there are no restrictions on where an instance variable or method can be used
- The modifier **`private`** means that an instance variable or method cannot be accessed by name outside of the class
- It is considered good programming practice to make **all** instance variables **`private`**
- Most methods are **`public`**, and thus provide controlled access to the object
- Usually, methods are **`private`** only if used as helping methods for other methods in the class



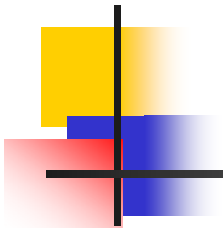
... - **public** and **private** Modifiers

- Within the definition of a class, private members of **any** object of the class can be accessed, not just private members of the calling object



- Accessor and Mutator Methods ...

- *Accessor* methods allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **`get`**
- *Mutator* methods allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **`set`**

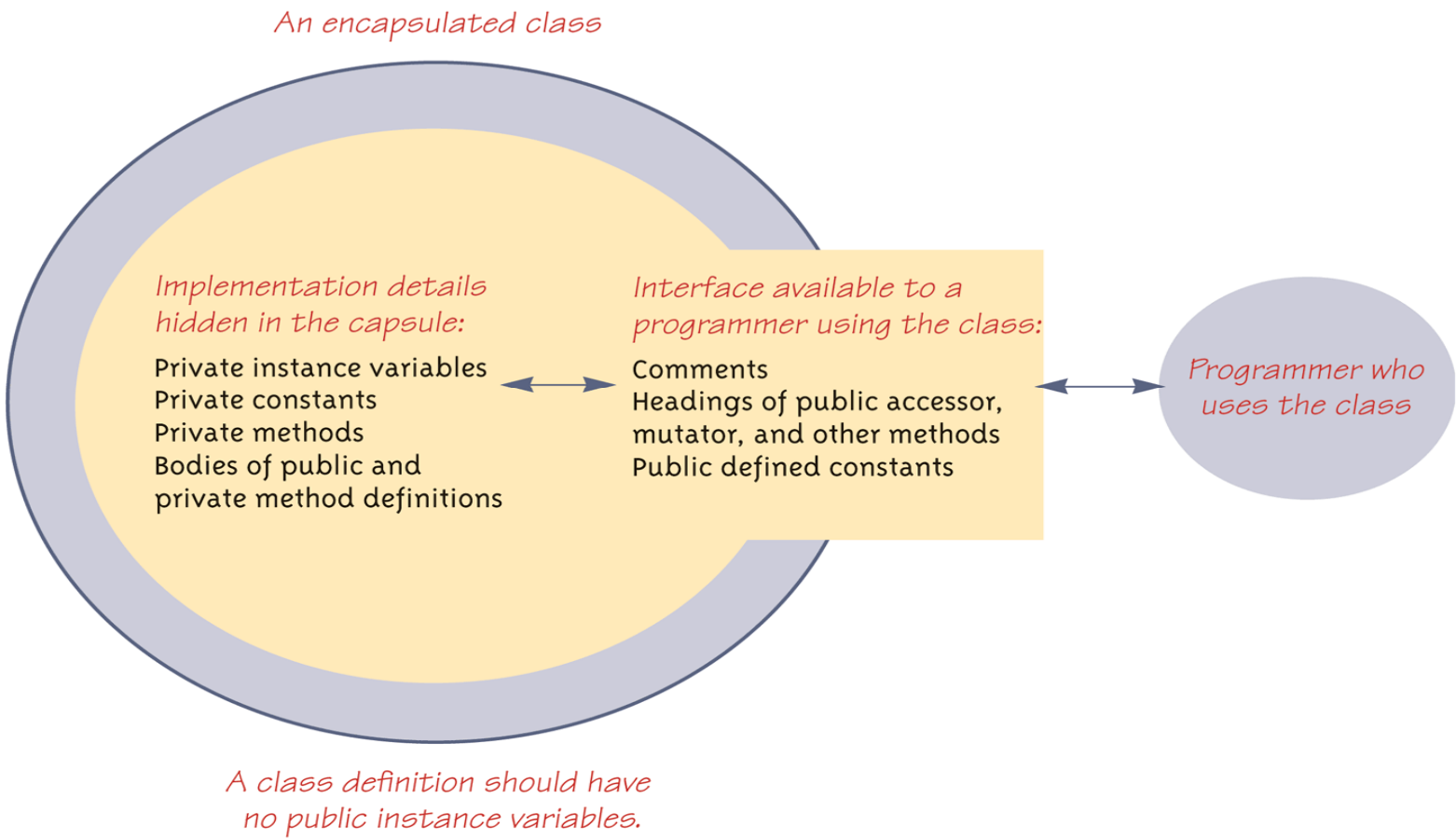


... - Accessor and Mutator Methods

- Some mutator methods issue an error message and end the program whenever they are given values that aren't sensible
- An alternative approach is to have the mutator test the values, but to never have it end the program
- Instead, have it return a boolean value, and have the calling program handle the cases where the changes do not make sense

- Encapsulation

Display 4.10 Encapsulation



- Example ...

```
import java.util.Scanner;

public class DateFifthTry
{
     private String month;
     private int day;
     private int year; //a four digit number.


     public void writeOutput()
     {
          System.out.println(month + " " + day + ", " + year);
     }

     public void readInput()
     {
          boolean tryAgain = true;
          Scanner keyboard = new Scanner(System.in);
          while (tryAgain)
          {
               System.out.println("Enter month, day, and year");
               System.out.println("as three integers:");
               System.out.println("do not use commas or other punctuations.");
               int monthInput = keyboard.nextInt();
               int dayInput = keyboard.nextInt();
               int yearInput = keyboard.nextInt();
               if (dateOK(monthInput, dayInput, yearInput) )
               {
                    setDate(monthInput, dayInput, yearInput);
                    tryAgain = false;
               }
               else
                    System.out.println("Illegal date. Reenter input.");
           }
     }
```

Note that this version of `readInput` *has the user enter the month as an integer rather than as a string. In this class, a month is an integer to the user, but is a string inside the class.*

Note that this version of `readInput` *checks to see that the input is reasonable.*

... - Example ...

```
public void setDate(int month, int day, int year)
{
     if (dateOK(month, day, year))
     {
          this.month = monthString(month);
          this.day = day;
          this.year = year;
     }
     else
     {
          System.out.println("Fatal Error");
          System.exit(0);
     }
}

public void setMonth(int monthNumber)
{
     if ((monthNumber <= 0) || (monthNumber > 12))
     {
          System.out.println("Fatal Error");
          System.exit(0);
     }
     else
          month = monthString(monthNumber);
}

public void setDay(int day)
{
     if ((day <= 0) || (day > 31))
     {
          System.out.println("Fatal Error");
          System.exit(0);
     }
     else
          this.day = day;
}
```

... - Example ...

```
public void setYear(int year)
{
    if ( (year < 1000) || (year > 9999) )
    {
        System.out.println("Fatal Error");
        System.exit(0);
    }
    else
        this.year = year;
}

public boolean equals(DateFifthTry otherDate)
{
    return ( (month.equalsIgnoreCase(otherDate.month))
             && (day == otherDate.day) && (year == otherDate.year) );
}
```

Within the definition of DateFifthTry, *you can directly access private instance variables of any object of type* DateFifthTry.

```
public boolean precedes(DateFifthTry otherDate)
{
    return ( (year < otherDate.year) ||
       (year == otherDate.year && getMonth() < otherDate.getMonth()) ||
       (year == otherDate.year && month.equals(otherDate.month)
                               && day < otherDate.day) );
}
```

Within the definition of DateFifthTry, *you can directly access private instance variables of any object of type* DateFifthTry.

definitions of the following methods are the same as in Display 4.2 and Display 4.7:
getMonth, getDay, getYear, and toString.>

```
private boolean dateOK(int monthInt, int dayInt, int yearInt)
{
    return ( (monthInt >= 1) && (monthInt <= 12) &&
             (dayInt >= 1) && (dayInt <= 31) &&
             (yearInt >= 1000) && (yearInt <= 9999) );
}
```

... - Example

```
private String monthString(int monthNumber)
{
    switch (monthNumber)
    {
    case 1:
        return "January";
    case 2:
        return "February";
    case 3:
        return "March";
    case 4:
        return "April";
    case 5:
        return "May";
    case 6:
        return "June";
    case 7:
        return "July";
    case 8:
        return "August";
     case 9:
        return "September";
    case 10:
        return "October";
    case 11:
        return "November";
    case 12:
        return "December";
    default:
        System.out.println("Fatal Error");
        System.exit(0);
        return "Error"; //to keep the compiler happy
    }
}
}
```

THE END