BIT 210: Object Oriented Programming II

Introduction to Java programming

- Java (Originally called Oak), was developed by Sun Microsystems in **1995**. It was intended for cable boxes and hand-held devices but was later enhanced so it could be used to deliver information on the World Wide Web.
- Java is everywhere, from computers to smartphones to parking meters. Three billion devices run Java!
- We are going to study java as a general purpose Programming Language
- The syntax of expression and assignment will be similar to that of C and C++ but details concerning the handling of String and Console output will be **new.**

Why Java

• Java enables users to develop and deploy applications on the Internet for servers, desktop computers and small hand-held devices. i.e Java is a general purpose programming language.

Characteristics of Java

i) Java is object-oriented Language

• it allows reuse of code

• OOP provides greater flexibility, modularity and reusability through encapsulation, inheritance and polymorphism.

ii) Java is distributed what does it mean by distributed?

• Distributed computing involves several computers working together on a network. Java is designed to make distributed computing easy.

iii) Java is robust

•It has a run time exception handling feature to provide programming support for robustness.

iv) Java is secure meaning of stay program?

• Java implements several security mechanisms to protect system against harm caused by stray programs

v) Java is Architecture-Neutral

• With a Java Virtual Machine, you can write one program that will run on any platform.

vi) Java is Dynamic

• Java was designed to adapt to an evolving environment. New code can be loaded on the fly without recompilation.

Simple Java Program

//This program prints Welcome to Java!

```
public class Welcome {
public static void main (String [] args)
{
System.out.println (“Welcome to Java!”);
}
}
```

- Message is printed to the console window
- Java is case sensitive
- Program file name should exactly match the class name.

when saving file, save using class name.

Anatomy of a Java Program

1) **Comments**

- Line comment which is preceded by 2 slashes (//) in a line
- Paragraph comment which is enclosed between /* and */ in one or multiple lines

2) **Reserved words**

• Are key words that have specific meaning to the compiler and cannot be used for other purposes in the program. E.g the word class which the compiler understands that the word after class is the name of the class. Other reserved words are public, static, void etc.

3) Modifiers

- Specific properties of data, methods and classes and how they can be used.

Example modifiers are public, static, private etc.

- A public datum, method or class can be accessed by other programs
- A private datum or method cannot be accessed by other programs. Only the method defined in the class can access and modify what is private.

4) Statements

- Represents an action or sequence of actions
- The statement System.out.println ("Welcome to Java!”) is a statement to display the greeting "Welcome to Java!”
- Every statement in Java ends with a semicolon (;)

5) Blocks

• A pair of braces in a program forms a block that groups components of a program.

6) Classes

- A class is a template or a blue print for objects. Example, “Welcome” is a class name
- For all class names, the first letter should be upper case.

7) Methods

e.g System.out.println

- It is a method: *a collection of statements that perform a sequence of operations to display a message to the console*.
- It is used by invoking a statement with a string argument
- The string argument is enclosed within parentheses. In this case, the argument is "***Welcome to Java!”***

8) main Method

- Provides control of program flow
- Java program executes application by invoking the main method

public static void main (String [] args) {

//Statements;

}

- public – means method can be viewed outside class by all programs
- static means any changes on the state of data inside the function will be effected
- void means the method returns nothing to the calling program
- main - is the name of the function
- String [] args- The function receives args which is a string as an argument.

  Arguments are data passed to the function to be used.
- System.out.println – Java stores content in a package. Content is contained in package called System. To output content, begin by calling the package.
- println is a function.
- System.out.println – prints and curser moves to next line
- System.out.print – prints and curser remains there

==========

Lab # 1(preparation of Java Programming Environment)

- ✓ Getting Started with Java Programming

Requirements
1. **Projector** 2. **Software**: NetBeans, JCreator

3. **Java Compiler:** sdk
4. **Laptop Computer and Smart Phones**

-java programming environment (NetBeans, JCreator and Java compiles JDK)

- Installation and configuration

-getting started with java

Data Types in Java

Primitive Data Types

• Java has eight primitive data types that are used to store data during a program's operation.

• Primitive data types are a special group of data types that do not use the keyword new when initialized.

• Java creates them as automatic variables that are not references, which are stored in memory with the name of the variable.

• The most common primitive types used in this course are int (integers) and double (decimals).

Primitive Data Type

Data Type	Size	Example Data	Data Description
boolean	1 bit	true, false	true, false
byte	1 byte (8 bits)	12, 128	Stores integers from -128 to 127
char	2 bytes	'A', '5', '#'	Stores a 16-bit Unicode character
short	2 bytes	6, -14, 2345	Stores integers from -32,768 to 32,767.

Data Type	Size	Example Data	Data Description
int	4 bytes	6, -14, 2345	Stores integers from: -2,147,483,648 to 2,147,483,647
long	8 bytes	3459111, 2	Stores integers from: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	3.145, .077	Stores a positive or negative decimal number from: 1.4023×10^{-45} to $3.4028 \times 10^{+38}$
double	8 bytes	.0000456, 3.7	Stores a positive or negative decimal number from: 4.9406×10^{-324} to $1.7977 \times 10^{+308}$

Declaring Variables and Using Literals

- The keyword new is not used when initializing a variable primitive type.
- Instead, a literal value should be assigned to each variable upon initialization.
- A literal can be any number, text, or other information that represents a value.

Examples of declaring a variable and assigning it a literal value:

boolean result = true;	short s = 10000;

char capitalC = 'C'; byte b = 100;	int i = 100000; long creditCardNumber = 1234_5678_9012_3456L;

Strings

- A String is an object that contains a sequence of characters.
- Declaring and instantiating a String is much like any other object variable.

However, there are differences:

- They can be instantiated (created) without using the new keyword.
- They are immutable.
- Once instantiated, they are final and cannot be changed.

String Operations Example

```
public class StringOperations    {
 public static void main(String[] args)
      {
String string1 = "Hello";
String string2 = "Caron";
String string3 = ""; //empty String or null
string3 = "How are you "+ string2.concat(string2);
 System.out.println("string3: "+ string3);
//get length
System.out.println("Length: "+ string1.length());
//get substring beginning with character 0, up to, but not
//including character 5
System.out.println("Sub: "+ string3.substring(0,5)); //uppercase
System.out.println("Upper: "+string3.toUpperCase());
      }
  }
```

compareTo Method

- There are methods to use when comparing Strings.
- Method: s1.compareTo(s2);
- Should be used when trying to find the lexicographical order of two strings.

Returns an integer.

- If s1 is less than s2, an int < 0 is returned.
- If s1 is equal to s2, 0 is returned.
- If s1 is larger than s2, an int > 0 is returned.

equals Method

•Method: s1.equals(s2)

•Should be used when you only wish to find if the two strings are equal.

•Returns a boolean value.

–If true is returned, s1 is equal to s2

–If false is returned, s1 is not equal to s2.

Scanner

To read in the input from the Keyboard to the program, we always use the Java object Scanner. You will have to use an import statement to access the class **java.util.Scanner**

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

- To initialize a Scanner object (create a Scanner object), write:

```
Scanner in = new Scanner(System.in);
```

- (in) is an object of the type Scanner.
- To read the next string from the console:

```
String input = in.next();
```

- To read the next integer:

```
int answer = in.nextInt();
```

Example:

Relational Operators

•Java has six relational operators used to test primitive or literal numerical values.

•Relational operators are used to evaluate if-else and loop conditions.

Relational Operator	Definition
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Logic Operators

• Java has three logic operators used to combine boolean expressions into complex tests.

Logic Operator	Meaning
&&	And
||	Or
!	Not

If Conditional Statement

To build an if-else statement, remember the following rules: An if-else statement needs a condition or method that is tested for true/false.

For example:

```
if(x==5)                    if(y >= 17)                    if(s1.equals(s2))
```

if-else Statements with the int Data Type

```
import java.util.Scanner;
public class ValueChecker   {
public static void main(String[] args){
Scanner in = new Scanner(System.in);
int value = 0;
System.out.println("Enter a number:");
value = in.nextInt();
if( value == 7)          {   System.out.println("That's lucky!");       }
else if( value == 13)    {   System.out.println("That's unlucky!");  }
else {   System.out.println("That is neither lucky nor unlucky!"); }
}
}
```

While Loop

With a while loop, Java uses the syntax:

```
while(condition is true)          { //logic }
```

Similar to if statements, the while loop parameters can be boolean types or can equate to a boolean value.

• Conditional statements (<, >, <=, >=, !=, ==) equate to boolean values.

• Examples:

–while (num1 < num2)

–while (isTrue)

–while (n !=0)

The do-while Loop

• The do-while loop:

- Is a post-test loop.
- Is a modified while loop that allows the program to run through the loop once before testing the boolean condition.
- Continues until the condition becomes false.

• If you do not allow for a change in the condition, the loop will run forever as an infinite loop.

```
do{ //statements to repeat go here }
 while(condition);
```

The for Loop

- The for loop repeats code a preset number of times.
- for loop syntax contains three parts:
- Initializing the loop counter.
- Conditional statement, or stopping condition.
- Updating the counter (going to the next value).

- Think of i as a counter starting at 0 and incrementing until
- i=timesToRun.

```
    for(int i=0; i < timesToRun; i++)
           { //logic }
```

Array

- An array is a collection of values of the same data type stored in a container object.
- Can be any number of values.
- Length of the array is set when the array is declared.
- Size is fixed once the array is declared.
- Array examples:

```
• String[] myBouquet = new String[6];
• int[] myArray = {7, 24, 352, 2, 37};
```

Classes

•A Java class is a template/blueprint that defines the features of an object.

•A class can be thought of as a category used to define groups of things.

•Classes:

–Class variables

–Define and implement methods.

- Writing a class is called implementinga class: declaration & body

–Implement methods from implemented interfaces.

Objects

An object is an instance of a class.

- A program may have many objects.
- An object stores data in the class variables to give it state.
- This state will differentiate it from other objects of the same class.
- Each object has a state, a set of characteristics, and a behavior represented by methods

Object classes:

Are classes that define objects to be used in a driver class.

Can be found in the Java API, or created by you.

Examples: String, BankAccount, Student, Rectangle

Student Class Example

package javaexample;	Package or folder where the class is stored
public class Student {	Class Declaration
private int studentId; private String name; private String ssn; private double gpa; public final int SCHCODE = 34958;	Fields/Variables
public Student(){ }	Constructor
public int getStudentId() { return studentId; } public void setStudentId(int x) { studentId = x; }	Methods
}	

Constructor

• A constructor is a method that creates an object.

• In Java, constructors are methods with the same name as their class used to create an instance of an object.

• Constructors are invoked using the new keyword.

• You can declare more than one constructor in a class declaration.

• You do not have to declare a constructor. In fact, Java will provide a default (blank) constructor for you.

• Example creating an object using Student constructor:

```
Student stu = new Student();
```

Overloading Constructors

• Constructors assign initial values to instance variables of a class.

• Constructors inside a class are declared like methods.

• Overloading a constructor means having more than one constructor with the same name.

• However the number of arguments would be different, and/or the data types of the arguments would differ.

Constructor with Parameters

A constructor with parameters is used when you want to initialize the private variables to values other than the default values.

```
public Student(String n, String ssn)
{ name = n; this.ssn = ssn; }
```

```
package copy;
class Box {
double width; double height;  double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
{ width = w; height = h; depth = d; }
// compute and return volume
double volume() { return width * height * depth; }
}
```

To instantiate a Student instance using the constructor with parameters, write:

```
Student student1 = new Student("Zina", "3003456");
```

```
package copy;
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Methods

Methods represent operations on data and also hold the logic to determine those operations

Using methods offer two main advantages:

• A method may be executed (called) repeatedly from different points in the program: decrease the program size, the effort to maintain the code and the probability for an error

• Methods help make the program logically segmented, or modularized: less error prone, and easier to maintain

A method is a self-contained block of code that performs specific operations on the data by using some logic

- **Method declaration:**

•Name

•Parameter(s)

•Argument(s)

•Return type

•Access modifier

Components of a Method

Method components include:

- **Return type**:
  - ✓ This identifies what type of object if any will be returned when the method is invoked (called).
  - ✓ If nothing will be returned, the return type is declared as void.
- **Method name:** Used to make a call to the method.
- **Parameter(s):**

- ✓ The programmer may choose to include parameters depending on the purpose and function of the method.
- ✓ Parameters can be of any primitive or type of object, but the parameter type used when calling the method must match the parameter type specified in the method definition.

```
public String getName(String firstName, String lastName)
   { return( firstName + " " + lastName ); }
```

Class Methods

• Every class will have a set of methods associated with it which allow functionality for the class.

• **Accessor method**

- Often called "getter" method.
- Returns the value of a specific private variable.
- Accessor methods access and return the value of a specific private variable of the class.
- Non-void return type corresponds to the data type variable you are accessing.
- Include a return statement.
- Usually have no parameters

```
public String getName()    { return name; }
public int getStudentId()  { return studentId; }
```

- **Mutator method**
  - Often called "setter" method.
  - Changes or sets the value of a specific private variable.
  - Mutator methods set or modify the value of a specified private variable of the class.
  - **Void** return type.
  - Parameter with a type that corresponds to the type of the variable being set.

```
public String setName(String name)   { this.name = name; }
public int setStudentId(int id)      { studentId = id; }
```

• **Functional method**

- Returns or performs some sort of functionality for the class.
- Void or non-void return type.
- Parameters are optional and used depending on what is needed for the method's function.

Overloading Methods

• Like overloading constructors, overloading a method occurs when the type and/or number of parameters differ.

• Below is an example of a situation where a method would need to be overloaded.

• Create the Dog class, then create an instance of Dog in a Driver Class. Call (use) both bark() methods.

```
public class Dog{
private int weight; private int loudness; private String BarkNoise;
public void bark(String b)  { System.out.println(b);      }
public void bark()  { System.out.println("Woof"); }
}
```

Main Method

• To run a Java program you must define a main method in a Driver Class.

• The main method is automatically called when the class is called.

- When executing a Java application, the JVM loads the class, and invokes the main() method of this class
- The method main() must be declared public, static, and void
- A source file may have one or more classes. Only one class (matching the file name) at most may be declared public

• Example:

```
public class StudentTester {
public static void main(String args[]) { }  }
```

Access Modifiers

• Access modifiers specify accessibility to changing variables, methods, and classes.

• There are four access modifiers in Java:

Access Modifier	Description
Public	Allows access from anywhere.
Protected	Allows access only inside the package containing the modifier.
Private	Only allows access from inside the same class.
Default (not specified/blank)	Allows access inside the class, subclass, or other classes of the same package as the modifier.

• The static methods and variables are shared by all the instances of a class

• The staticmodifier may be applied to a variable, a method, and a block of code inside a method

• Because a static element of a class is visible to all the instances of the class, if one instance makes a change to it, all the instances see that change.

Inheritance: Superclass versus Subclass

• Classes can derive from or evolve out of parent classes, which means they contain the same methods and fields as their parents, but can be considered a more specialized form of their parent classes.

• The difference between a subclass and a superclass is as follows:

Superclass:

- The more general class from which other classes derive their methods and data
- Contain methods and fields that are passed down to all of their subclasses.

Subclass :

- The more specific class that derives or inherits from another class (the superclass)
- Inherit methods and fields from their superclasses.
  –May define additional methods or fields that the superclass does not have.
  –May redefine (override) methods inherited from the superclass.

extends Keyword

• In Java, you have the choice of which class you want to inherit from by using the keyword extends.

• The keyword extends allows you to designate the superclass that has methods you want to inherit, or whose methods and data you want to extend.

• For example, to inherit methods from the Shape class, use extends when the Rectangle class is created.

```
public class Rectangle extends Shape { //code }
```

More about Inheritance

• Inheritance is a one-way street.

• Subclasses inherit from superclasses, but superclasses cannot access or inherit methods and data from their subclasses.

• This is just like how parents don't inherit genetic traits like hair color or eye color from their children.

Object:

- Is considered the highest and most general component of any hierarchy. It is the only class that does not have a superclass.
- Contains very general methods which every class inherits.

The super keyword

The super keyword is similar to this keyword. Following are the scenarios where the super keyword is used.

It is used to differentiate the members of superclass from the members of subclass, if they have same names.
It is used to invoke the superclass constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable super.method();
```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class Animal {
}
```

```
public class Mammal extends Animal {
}
```

```
public class Reptile extends Animal {
}
```

```
public class Dog extends Mammal {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true –

Animal is the superclass of Mammal class.
Animal is the superclass of Reptile class.
Mammal and Reptile are subclasses of Animal class.
Dog is the subclass of both Mammal and Animal classes.
Now, if we consider the IS-A relationship, we can say –

Mammal IS-A Animal
Reptile IS-A Animal
Dog IS-A Mammal
Hence: Dog IS-A Animal as well
With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator
Example

```
class Animal {
}
class Mammal extends Animal {
}
class Reptile extends Animal {
}
public class Dog extends Mammal {
   public static void main(String args[]) {
      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();
      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result −
Output

```
true
true
true
```

Since we have a good understanding of the extends keyword, let us look into how the implements keyword is used to get the IS-A relationship.

Generally, the implements keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {
}
public class Mammal implements Animal {
}
public class Dog extends Mammal {
}
```

The instanceof Keyword

Let us use the instanceof operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

```
 interface Animal{}
class Mammal implements Animal{}
public class Dog extends Mammal {
   public static void main(String args[]) {
      Mammal m = new Mammal();
      Dog d = new Dog();
```

```
    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

This will produce the following result −

Output

```
true
true
true
```

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class HAS-A certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example

Example

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle {
  private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class Animal {
}
```

```
public class Mammal extends Animal {
}
```

```
public class Reptile extends Animal {
}
```

```
public class Dog extends Mammal {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true −

Animal is the superclass of Mammal class.
Animal is the superclass of Reptile class.
Mammal and Reptile are subclasses of Animal class.
Dog is the subclass of both Mammal and Animal classes.
Now, if we consider the IS-A relationship, we can say −

Mammal IS-A Animal
Reptile IS-A Animal
Dog IS-A Mammal
Hence: Dog IS-A Animal as well
With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator
Example

```
class Animal {
}
class Mammal extends Animal {
}
class Reptile extends Animal {
}
public class Dog extends Mammal {
   public static void main(String args[]) {
      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();
      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result −
Output

```
true
true
true
```

Since we have a good understanding of the extends keyword, let us look into how the implements keyword is used to get the IS-A relationship.

Generally, the implements keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {
}
public class Mammal implements Animal {
}
public class Dog extends Mammal {
}
```

The instanceof Keyword

Let us use the instanceof operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

```
 interface Animal{}
class Mammal implements Animal{}
public class Dog extends Mammal {
   public static void main(String args[]) {
      Mammal m = new Mammal();
      Dog d = new Dog();
      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result –

Output

```
true
true
true
```

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class HAS-A certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example

Example

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle {
   private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type

Example
Let us look at an example

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples –

A Deer IS-A Animal
A Deer IS-A Vegetarian
A Deer IS-A Deer
A Deer IS-A Object
When we apply the reference variable facts to a Deer object reference, the following declarations are legal –

Example

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Example

```
/* File name : Employee.java */
public class Employee {
```

```
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number) {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
    }

    public void mailCheck() {
    System.out.println("Mailing a check to " + this.name + " " + this.address);
    }

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

    public String getName() {
    return name;
    }

    public String getAddress() {
    return address;
    }

    public void setAddress(String newAddress) {
    address = newAddress;
    }

    public int getNumber() {
    return number;
    }
     }
```

Now suppose we extend Employee class as follows −

```
/* File name : Salary.java */
    public class Salary extends Employee {
    private double salary; // Annual salary
    public Salary(String name, String address, int number, double salary) {
    super(name, address, number);
    setSalary(salary);
    }
    public void mailCheck() {
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName()
```

```
    + " with salary " + salary);
    }
    public double getSalary() {
    return salary;
    }
    public void setSalary(double newSalary) {
    if(newSalary >= 0.0) {
    salary = newSalary;
    }
    }
    public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary/52;
    }
    }
```

Now, you study the following program carefully and try to determine its output –

```
 /* File name : VirtualDemo.java */
public class VirtualDemo {
 public static void main(String [] args) {
     Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
     Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
     System.out.println("Call mailCheck using Salary reference --");
     s.mailCheck();
     System.out.println("\n Call mailCheck using Employee reference--");
     e.mailCheck();
   }
}
```

This will produce the following result –

Output

```
Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
ailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
ailing check to John Adams with salary 2400.0
```

Here, we instantiate two Salary objects. One using a Salary reference s, and the other using an Employee reference e.

While invoking s.mailCheck(), the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() on e is quite different because e is an Employee reference. When the compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

Encapsulation

• Encapsulation is a fundamental concept in object oriented programming.

- ✓ **Encapsulation means** *to enclose something into a capsule or container, such as putting a letter in an envelope. In object-oriented programming, encapsulation encloses, or wraps, the internal workings of a Java instance/object.*

How Encapsulation Works

- In object-oriented programming, encapsulation encloses, or wraps, the internal workings of a Java instance/object.
- Data variables or fields are hidden from the user of the object.
- Methods can provide access to the private data, but methods hide the implementation.
- Encapsulating your data prevents it from being modified by the user or other classes so that the data is not corrupted.

What Classes Can and Cannot Do

•Classes can be instantiated by:

–A public or protected constructor.

–A public or protected static method or nested class.

Classes cannot:

–Override inherited methods when the method is final.

When Classes Can be Subclassed or Made Immutable

•A class can be subclassed when:

–The class is not declared final.

–The methods are public or protected.

•Strategy for making a class immutable:

–Make it final.

–Limit instantiation to the class constructors.

–Eliminate any methods that change instance variables.

–Make all fields final and private

Immutable Using Final

- Declaring a class as final means that it cannot be extended.
- Example: You may have a class that has a method to allow users to login by using some secure call.

- You would not want someone to later extend it and remove the security.

```
public final class ImmutableClass {
public static boolean logOn(String username, String password)
 {
//call to public boolean someSecureAuthentication(username, password);
return someSecureAuthentication (username, password);
}
}
```

Immutable by Limiting Instantiation to the Class Constructor

- By removing any method that changes instance variables and limiting their setting to the constructor, the class variables will be made immutable.
- Example: When we create an instance of the ImmutableClass, the immutableInt variable cannot be changed.

```
public final class ImmutableClass {
private final int immutableInt;
public ImmutableClass (int mutableIntIn) {  immutableInt = mutableIntIn; }
private int getImutableInt() { return immutableInt; }
}
```

Interface

- An interface is a Java construct that helps define the roles that an object can assume.
- It is implemented by a class or extended by another interface.
- An interface looks like a class with abstract methods (no implementation), but we cannot create an instance of it.
- Interfaces define collections of related methods without implementations.
- All methods in a Java interface are abstract.

Why Use Interfaces

- When implementing a class from an interface we force it to implement all of the abstract methods.
- The interface forces separation of what a class can do, to how it actually does it.
- So a programmer can change how something is done at any point, without changing the function of the class.
- This facilitates the idea of polymorphism as the methods described in the interface will be implemented by all classes that implement the interface.

What an Interface Can Do

•An interface:

–Can declare public constants.

–Define methods without implementation.

–Can only refer to its constants and defined methods.

–Can be used with the instanceof operator.

While a class can only inherit a single superclass, a class can implement more than one interface.

•Example of SavingsAccount implementing two interfaces – **Account** and **Transactionlog**

```
public class className implements interfaceName
{
...class implementation...
}
```

implements is a keyword in Java that is used when a class inherits an interface.

```
public class SavingsAccount implements Account,Transactionlog
{...class implementation...}
```

Interface Method

•An interface method:

–Each method is public even when you forget to declare it as public.

–Is implicitly abstract but you can also use the abstract keyword.

Declaring an Interface

•To declare a class as an interface you must replace the keyword class with the keyword interface.

•This will declare your interface and force all methods to be abstract and make the default access modifier public.

1	***public interface InterfaceBankAccount***	Replace class with interface.
2	***{***	
3	***public final String bank= "JavaBank";***	
4	***public void deposit(int amt);***	
5	***public void withdraw(int amt);***	
6	***public int getbalance();***	
7	***}***	

Bank Example

public interface InterfaceBankAccount ***{*** ***public final String bank= "JavaBank";*** ***public void deposit(int amt);*** ***public void withdraw(int amt);*** ***public int getbalance();*** ***}***	Classes that extend InterfaceBankAccount will have to provide working methods for methods defined here.
public class Account implements InterfaceBankAccount ***{*** ***private String bankname;*** ***public Account()*** ***{this.bankname = InterfaceBankAccount.bank; }*** ***public void deposit(int amt) { /* deposit code */ }*** ***public void withdraw(int amt) {/* withdraw code */ }*** ***public int getbalance() { /* getBalance code */ }***	Classes implementing an interface can access the interface constants.

Bank Example Explained

The keyword ***fina***l means that the variable bank is a constant in the interface because you can only define constants and method stubs here.

```
this.bankname = InterfaceBankAccount.bank
```

•The assignment of the constant from an interface uses the same syntax that you use when assigning a static variable to another variable.

The interface defined 3 methods – **deposit, withdraw** and **getbalance.**

•These must be implemented in our class.

```
public class Account implements InterfaceBankAccount{
        private String bankname;
         int balance;
public Account()
          { this.bankname = InterfaceBankAccount.bank; }
public void deposit(int amt)
          { balance = balance + amt;}
public void withdraw(int amt)
          { balance = balance – amt ;}
public int getbalance() { return balance; }
}
```

Why use interfaces with Bank Example?

- You may be wondering why you would want to create a class that has no implementation.
- One reason could be to force all classes that implement the interface to have specific qualities.
- In our bank example we would know that all classes that implement the interface InterfaceBankAccount must have methods for deposit, withdraw and getbalance.
- Classes can only have one superclass, but can implement multiple interfaces.

Store Example

•A store owner wants to create a website that displays all items in the store.

•We know:

- Each item has a name.
- Each item has a price.
- Each item is organized by department.

•It would be in the store owner's best interest to create an interface for what defines an item.

•This will serve as the blueprints for all items in the store, requiring all items to at least have the defined qualities above.

Adding a New Item to Store Example

•The owner adds a new item to his store named cookie:

- Each costs 1 US dollar.
- Cookies can be found in the Bakery department.
- Each cookie is identified by a type.

•The owner may create a Cookie class that implements the Item interface such as shown on the below, adding methods or fields that are specific to cookies.

Item Interface

- Possible Item interface

```
public interface Item
{
public String getItemName();
public int getPrice();
public void setPrice(int price);
public String getDepartment();
}
```

- We now force any class or interface that implements Item interface to implement the methods defined above.

Create Cookie Class

•The owner may create a Cookie class that implements the Item interface, such as shown below, adding a method or two specific to cookie items.

```
public class Cookie implements Item{
        public String cookieType;
        private int price;
public Cookie(String type)
    {
      cookieType = type;
      price = 1;
     }
public String getItemName() { return "Cookie";}
public int getPrice() {return price;}
public void setPrice(int price){this.price = price;}
public String getDepartment() {return "Bakery";}
public String getType() {return cookieType;}
}
```

==========

Abstract Classes

An abstract class provides a base class from which other classes extend.

• Abstract classes can provide:

- Implemented methods: Those that are fully implemented and their child classes may use them.
- Abstract methods: Those that have no implementation and require child classes to implement them.

• Abstract classes do not allow construction directly but you can create subclasses or static nested classes to instantiate them.

An abstract class:

- Must be defined by using the abstract keyword.
- Cannot be instantiated into objects.
- Can have local declared variables.
- Can have method definitions and implementations.
- Must be subclassed rather than implemented.

More Information about Abstract Classes

• If a concrete class uses an abstract class, it must inherit it. Inheritance precludes inheriting from another class.

• Abstract classes are important when all derived classes should share certain methods.

• The alternative is to use an interface, then define at least one concrete class that implements the interface.

• This means you can use the concrete class as an object type for a variable in any number of programs which provides much greater flexibility.

Abstract Class or Interface

• There is no golden rule on whether to use Interfaces, Abstract classes or both.

• An abstract class usually has a stronger relationship between itself and the classes that will be derived from it than interfaces.

• Classes and Interfaces can implement multiple interfaces. Where a class can only be a sub class of one abstract class.

• Abstract classes allow methods to be defined.

Bank Account as Abstract

•Previously we had implemented an interface for the Account class.

```
public interface InterfaceBankAccount{
public final String bank= "JavaBank";
public void deposit(int amt);
public void withdraw(int amt);
public int getbalance();
}
```

- We could have used an abstract class instead.

Bank Account as Abstract

```
public abstract class AbstractBankAccount {
public final String bank= "JavaBank";
public abstract void deposit(int amt);
public abstract void withdraw(int amt);
public abstract int getbalance();
}
```

- In this implementation there is no advantage of using an abstract class over an interface.

- We would be better using an interface especially if all bank accounts had to implement only those defined methods.
- The interface approach would also allows us to use other interfaces in our class design.
- If we wanted some base functionality to be defined then we would use an abstract class.

• **An example is shown below.**

```
public abstract class AbstractBankAccount {
String accountname;
int accountnum;
int balance;
public void deposit(int amt)
{ balance=balance+amt; }
public void withdraw(int amt)
{ balance=balance-amt; }
public void setaccountname(String name)
{ accountname = name; }
public void setaccountnum(int num)
{ accountnum = num; }
public abstract int getbalance();
public abstract void print();
}
```

The instanceof Operator

• The instanceof operator allows you to determine the type of an object.

• It takes an object on the left side of the operator and a type on the right side of the operator and returns a boolean value indicating whether the object belongs to that type or not.

• For example:

- (String instanceof Object) would return true.
- (String instanceof Integer) would return false.

- This could be useful when you desire to have different object types call methods specific to their type.

```
public class CustomerAccounts {
private InterfaceBankAccount[] bankaccount = new
InterfaceBankAccount[10];
public CustomerAccounts() {
InterfaceBankAccount[] b = {
new Account("Sanjay Gupta",11556,300),
new CreditAccount("Sally Walls",66778,1000,500)
};
this.bankaccount = b;
}
}
```

Example:

In this example, the instanceof operator is used to find all of the CreditAccounts in an array and add them to a String called list.

```
public String getAllCreditAccounts() {
String list = "";
for (int i = 0; i < this.bankaccount.length; i++) {
if (this.bankaccount[i] instanceof CreditAccount) {
if (list.length() == 0) { list += this.bankaccount[i]; }
else { list += ", " + this.bankaccount[i];}
```

```
}
}
return list;
}
```

In the previous instanceof operator example:

- The array is defined using the InterfaceBankAccount interface. It can hold any class that solely implements its methods and extends them with other specialization methods and variables.

• As the program reads through the array, it assigns the string value from the toString() method to the list variable.

• You should always consider overridding the toString(), hash(), and equals() methods in your user-defined classes.

Virtual Method Invocation

• Virtual method invocation is the process where Java identifies the type of subclass and calls its implementation of a method.

• When the Java virtual machine invokes a class method, it selects the method to invoke based on the type of the object reference, which is always known at compile time.

• On the other hand, when the virtual machine invokes an instance method, it selects the method to invoke based on the actual class of the object, which may only be known at runtime.

Virtual Method Invocation Example

```
...
public String toString() {
String list = "";
for (int i = 0; i < this.bankaccount.length; i++) {
if (list.length() == 0) { list += this.bankaccount[i]; }
else { list += "\n : " +this.bankaccount[i];}
}
return super.toString() + "\n" + list + "\n"
}
```

```
...
```

- The toString() method uses virtual method invocation by calling the toString() method of the different subclasses.
- The decision is made by JVM to call the toString method implemented in the subclasses rather than the toString method in the superclass or the Object class.

Upcasting and Downcasting

- Casting is the principal of changing the object type during assignment.
- Upcasting loses access to specialized methods in the subclassed object instance.
- Downcasting gains access to specialized methods of the subclass.
- Casting objects works somewhat like casting primitives.
- For example, when downcasting a double to an int, the values to the right of the decimal point are lost.
- This is a loss of precision.
- For that reason, you must explicitly specify the downcast type.
- For example:

```
double complexNumber = 45.75L;
int simpleNumber = 34;
simpleNumber = (int) complexNumber;
```

- Upcasting does not risk the loss of precision nor require you to specify the new data type in parentheses.

- Even though CreditAccount is an Account, it was upcast to Account so it lost access to its Credit Account specific methods and fields.
- Attempts to access subclass methods fail at compile time.

```
...
Account account =
(Account) new CreditAccount("Sally Walls",66778,1000,500);
...
account.getcreditlimit();
...
```

==========

Generics

Problem: Often in programming we want to write code which can be used by more than one type with the same underlying behavior.

If we wanted a very simple class to get and set a string value we could define this as:

```
public class Cell {
private String data;
public void set(String celldata) { data = celldata; }
public String get() { return data; }
}
```

Simple Driver Class

•Using a simple driver class we could set and retrieve a string value.

```
public class CellDriver {
public static void main(String[] args) {
Cell cell = new Cell();
cell.set("Test");
System.out.println(cell.get());
}
}
```

- Although this is a very simple class without much coding, if it had been more complex we may wish to reuse the algorithms with other data types.

Flexible Class

We could change the String primitive type to Object.

```
public class Cell {
private Object data;
public void set(Object celldata) {  data = celldata; }
public Object get() { return data; }
}
```

This would then give us the flexibility to use other datatypes.

Flexible Driver Class

Now our driver class can set the type of data we wish to store.

```
public class CellDriver {
public static void main(String[] args) {
Cell cell = new Cell();
cell.set(1);
int num = (int)cell.get();
System.out.println(num);
}
}
```

The problem with this is if we pass a String in the set method and try to cast as int then we will receive a casting error at runtime.

• **A generic class** is a special type of class that associates one or more non-specified Java types upon instantiation.

• This removes the risk of the runtime exception "ClassCastException" when casting between different types.

• Generic types are declared by using angled brackets - <> around a holder return type. E.g. <E>

– **We can modify our Cell class to make it generic.**

```
public class Cell<T> {
private T t;
public void set(T celldata)
{ t = celldata; }
public T get() { return t; }
}
```

Generic Cell Driver Class

We can now set the type at creation.

```
public class CellDriver {
public static void main(String[] args) {
Cell<Integer> integerCell = new Cell<Integer>();
Cell<String> stringCell = new Cell<String>();
integerCell.set(1);
stringCell.set("Test");
int num = integerCell.get();
String str = stringCell.get();
}
}
```

Initializing a Generic Object

• How to initialize a Generic object with one type, Example:

```
Example<String> showMe = new Example<String>();
```

•With two types:

```
Example<String, Integer> showMe = new Example<String, Integer>();
```

– The only difference between creating an object from a regular class versus a generics class is <String, Integer>.
– This is how to tell the Example class what type of types you are using with that particular object.
– In other words, Type1 is a String type, and Type2 is an Integer type.
– The benefit to having a generic class is that you can identify multiple objects of type Example with different types given for each one, so we could initialize another object Example with <Double, String>.

Type Parameter Names

The most commonly used type parameter names are:

– E - Element (used extensively by the Java Collections Framework)
– K – Key
– N – Number
– T – Type
– V – Value

- S,U,V etc. - 2nd, 3rd, 4th types

Working with Generic Types

• When working with generic types, remember the following:

• The types must be identified at the instantiation of the class.

• Your class should contain methods that set the types inside the class to the types passed into the class upon creating an object of the class.

• One way to look at generic classes is by understanding what is happening behind the code.

Generic Classes Code Example

• This code can be interpreted as a class that creates two objects, Type1 and Type2.

• Type1 and Type2 are not the type of objects required to be passed in upon initializing an object.

• They are simply placeholders, or variable names, for the actual type that is to be passed in.

```
public class Example<Type1, Type2>{
private Type1 t1;
private Type2 t2;
…}
```

These placeholders allow for the class to include any Java type: They become whatever type is initially used at the object creation.

• Inside of the generic class, when you create an object of Type1 or Type2, you are actually creating objects of the types initialized when an Example object is created.

Generic Methods

• So far we have created Generic classes, but we can also create generic methods outside of a generic class.

• Just like type declarations, method declarations can be generic—that is, parameterized by one or more type parameters

• A type interface diamond is used to create a generic method.

Type Interface Diamond

• A type interface diamond enables you to create a generic method as you would an ordinary method, without specifying a type between angle brackets.

• Why a diamond?

- The angle brackets are often referred to as the diamond <>.
- Typically if there is only one type inside the diamond, we use <T> where T stands for Type.
- For two types we would have <K,T>

• You can use any non reserved word as the type holder instead of using <T>. We could have used <T1>.

• By convention, type parameter names are single, uppercase letters.

• This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

– To define a generic method printArray for returning the contents of an array we would declare it as.

```
public class GenericMethodClass {
public static <T> void printArray(T[] array){
for ( T arrayitem : array ){
System.out.println( arrayitem );
```

This would allow printing of multiple array types.

```
Integer[] integerArray = { 1, 2, 3 };
String[] stringArray = { "This","is","fun" };
printArray( integerArray );
printArray( stringArray );
```

•Output

```
1
2
3
This
is
fun
```

Generic Wildcards

• Wildcards with generics allows us greater control of the types we use.

• They fall into two categories:

– Bounded
  • <? extends type>
  • <? super type>
– Unbounded
  • <?>

Unbounded Wildcards

• <?> denotes an unbounded wildcard

• It can be used to represent any type

• Example – arrayList<?> represents an arrayList of unknown type.

```
ArrayList<?> array1 = new ArrayList<Integer>( );
array1 = new ArrayList<Double>( );
```

We are going to create a method called printArrayList. Its goal is to print an arrayList of any type.

```
public static void printList(List<?> list) {
for (Object elem: list)
System.out.println(elem);
System.out.println();
}
```

– We could then pass any type of arrayList.

```
ArrayList<Integer> li = new ArrayList<Integer>();
li.add(1);
```

```
li.add(2);
ArrayList<String> ls = new ArrayList<String>();
ls.add("one");
ls.add("two");
printList(li);
printList(ls);
```

Upper Bounded Wildcard

• <? extends Type> denotes an Upper Bounded Wildcard.

• Sometimes we want to relax restrictions on a variable.

• Lets say we wished to create a method that works only on ArrayLists of numbers

– ArrayList<Integer>, ArrayList<Double>, ArrayList<Float>

• We could use an upper bounded wildcard:

```
public static double sumOfList(ArrayList<? extends Number> arrayList) {
double s = 0.0;
for (Number n : arrayList)
s += n.doubleValue();
return s;
}
```

Lower Bounded Wildcard

• <? super Type> denotes a Lower Bounded Wildcard.

• A lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

• Say you want to write a method that puts Integer objects into an ArrayList.

• To maximize flexibility, you would like the method to work on ArrayList<Integer>, ArrayList<Number>, and ArrayList<Object> — anything that can hold Integer values.

```
public static void addNumbers(ArrayList<? super Integer> arrayList) {
for (int i = 1; i <= 10; i++) {
arrayList.add(i);
}
}
```

Enumerations

• Enumerations (or enums) are a specification for a class where all instances of the class are created within the class.

• Enums are a datatype that contains a fixed set of constants.

• Enums are good to use when you already know all possibilities of the values or instances of the class.

• If you use enums instead of strings or integers you increase the checks at compile time.

For example, say we wish to store the type of bank account within our Account Class.

• We could have Current, Savings, and Deposit as possible options.

• As long as we specify that the class is of type enum, we can create these account types inside the class itself as if each was created outside of the class.

Enumerations Bank Code Simple Example

public enum AccountType { ***Current,*** ***Savings,*** ***Deposit*** }	• This keyword enum initializes the class AccountType as an enum type. • These are the initializations of all the Account Types

We could assign any one of these to a field in our class.

```
AccountType type = AccountType.Deposit;
```

Enumerations Iterate

We could print out our enums by using a for loop.

```
for (AccountType at : AccountType.values()) System.out.println(at+", Value: "+at.name()+", ord:"+ at.ordinal());
```

Would produce:

```
Current, Value: Current, ord:0
Savings, Value: Savings, ord:1
Deposit, Value: Deposit, ord:2
```

Our bank account type might also have an internal code that is used by the bank.

```
        public enum AccountType {
        Current("CU"),
        Savings("SA"),
        Deposit("DP");
        private String code;
        private AccountType(String code){
        this.code=code;
        }
        public String getCode() { return code; }
}
```

We can now access the code value from the enum.

```
AccountType type = AccountType.Deposit;
String code = type.getCode();
System.out.println(code);
```

Collections

- We have seen previously that we can use arrays to store items of the same type.
- The main issue with arrays is that they have to be manually sized before use.
- We could create our own simple data structure to store multiple Cell occurrences.

```
public class CellCollection {
Cell[] cells;
int index;
public void add(Cell c) { Cells[index]=c;  index++; }
public Cell get(int i) { return cells[i]; }

 //more methods….. }
```

- We would have to create more methods, such as insert, sort, delete and a constructor.
- We could try to create a generic collection similar to our non generic example.
- We could try to modify it to use generics.

```
public class CellGenericCollection<T> {
T[] cells;
int index;
public void add(T c) {
cells[index]=c; index++; }
public T get(int i) { return cells[i]; }
}
```

- The problem is we should not create generic arrays in java.

```
public GenericCell(int size)
{
 cells = new T[size];
}
```

- This would produce an error.
- Java has better ways of handling generic collections.

Collections - Introduction

• Data structures are used a lot in programming and as such java comes with pre-built collection classes for you to utilize.

• These save you and every other developer the task of having to create similar storage structures.

• They come in a few guises, each with its own pros and cons.

• You should check to see if any of the pre-built collections meet your needs before building your own.

A collection is an interface in the java.util package that is used to define a group, or collection of objects.

• It includes sets and lists.

• It is a very important part of data storing.

• Because it is in the java.util package, it will be necessary to import the java.util package into any programs you write using a collection.

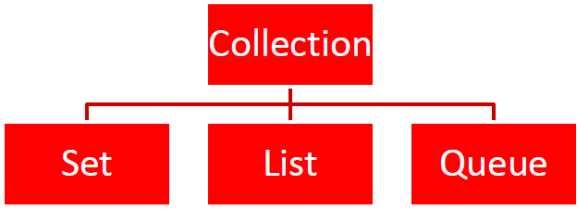
• This can be done by typing the following as the first line in your program:

```
import java.util.*;
```

Collections Class Hierarchy

Java defines three main interface collections.

• This forces any classes that use the collection interface to define its methods such as add().



Lists

• A list, in Java, is an ordered collection that may contain duplicate elements.

• In other words, List extends Collections.

• Important qualities of Lists are:

- They grow and shrink as you add and remove objects.
- They maintain a specific order.
- They allow duplicate elements.

ArrayLists

Until now we have been using arrays inside a collections class to create a collections interface.
• With arrays, you are restricted to the size of the array that you initiate inside the constructor.
• ArrayLists are very similar to arrays, except that you do not need to know the size of the ArrayList when you initialize it, like you would with an array.
• You may alter the size of the ArrayList by adding or removing elements.
• The only information you need when initializing an ArrayList is the object type that it stores.

The following code initializes an ArrayList of Accounts.

```
ArrayList<Account> account= new ArrayList<Account>();
```

ArrayList Methods

• The ArrayList class already exists in Java. It contains many methods, including the following:

Method	Method Description
boolean add(E e)	Appends the specified element to the end of the list.
void add(int index, E element)	Inserts the specified element at the specified position
E get(int index)	Returns the element at the specified position.
E set(int index, E element)	Replaces the element at the specified position in the list with the specified element.
E remove(int index)	Removes the element at the specified position in the list
boolean remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.

ArrayList Activity

•Try the following activity:

- Get into groups and initialize an ArrayList of Strings called GroupNames.
- Add each of your names into the ArrayList.
- Collections contains a method called sort that takes in a List as a parameter and lexicographically sorts the list.
- An ArrayList is a List.
- Use Collections sort method on your ArrayList of names.
- What does your ArrayList look like after sorting it?

ArrayList Activity: Sample Answer 1

•Get into groups and initialize an ArrayList of Strings called GroupNames.
•Add each of your names into the ArrayList.

```
ArrayList<String> GroupNames;
GroupNames = new ArrayList<String>();
GroupNames.add("Khaoya");
GroupNames.add("Ng'etich");
GroupNames.add("Kilwake");
```

- Collections contains a method called sort that takes in a List as a parameter and lexicographically sorts the list.

•An ArrayList is a List.
• Use Collections sort method on your ArrayList of names.
• What does your ArrayList look like after sorting it?

```
Collections.sort(GroupNames);
GroupNames would look like:
 Khaoya, Kilwake, Ng'etich
```

Sets

• A set is a collection of elements that does not contain duplicates.

• For example, a set of integers 1, 2, 2, 3, 5, 3, 7 would be:

  – {1, 2, 3, 5, 7}

• All elements of a set must be of the same type.

•For example, you can not have a set that includes integers and Strings, but you could have a set of integers and a separate set of Strings.

Implementing Sets with a HashSet

• Lists, which are a collection that may contain duplicates, are implemented through ArrayLists.

• Similarly, Sets are commonly implemented with a HashSet.

• A HashSet:

  – Is similar to an ArrayList, but does not have any specific ordering.
  – Is good to use when order is not important.

  • For example, imagine you have 35 coins in a bag.

  • There is no special order to these coins, but we can search the bag to see if it contains a certain coin.

  • We can add to or remove coins from the bag, and we always know how many different coins are in the bag.

  • Think of the bag as the HashSet.

  • There is no ordering and therefore no indexes of the coins, so we cannot increment through or sort HashSets.

    – Even if we incremented through or sorted the coins, with one little shake of the bag, the order is lost.

  •HashSets have no guarantee that the order will be the same throughout time.

  • The code below demonstrates the initialization of HashSet bagOfCoins.

  • The HashSet bagOfCoins is a set of Coin objects.

  • This assumes that the class Coin has already been created.

```
HashSet<Coin> bagOfCoins = new HashSet<Coin>();
```

Searching Through HashSets

• Although HashSets do not have ordering, we can search through them, just like we could search through the bag of coins to see if the coin we are looking for is in the bag.

• For example, to search for a coin in the HashSet bagOfCoins, use HashSet's contains(element) as demonstrated below:

```
bagOfCoins.contains(quarter); //returns true if bagOfCoins
contains the coin
```

More HashSet Methods

Method	Method Description
boolean add(E e)	Adds the specified element to this set if it is not already present.
boolean remove(Object o)	Removes the specified element from the list if present.
int size()	Returns the number of elements in the set.

Maps

• A map is a collection that links a key to a value.

• Similar to how an array links an index to a value, a map links a key (one object) to a value (another object).

• Maps, like sets, cannot contain duplicates.

• This means each key can only exist once and can only link to a single value.

• Since Map is an interface, you must use one of the classes that implement Map such as HashMap to instantiate a map.

HashMaps

• HashMaps are maps that link a Key to a Value.

• The Key and Value can be of any type, but their types must be consistent for every element in the HashMap.

• Below is a generic breakdown of how to initialize a HashMap.

```
HashMap<KeyType,ValueType> mapName = new HashMap<KeyType,ValueType>( );
```

• For example, we wish to group together many different fruits and wish to be able to store and later retrieve their color.

• The first step to do this is to initialize a HashMap.

```
HashMap<String,String> fruitBowl = new HashMap<String,String>( );
```

– To add fruits to our fruitBowl, simply use the put(Key,Value) function of HashMaps.

• Add a few fruits to the fruitBowl.

• Each code segment adds the key (fruit name) and value (color) to the HashMap.

```
fruitBowl.put("Apple", "Red");
fruitBowl.put("Orange", "Orange");
fruitBowl.put("Banana", "Yellow");
```

get(Key) Method of HashMap

• The Key of a HashMap can be thought of as the index linked to the element, even though it does not necessarily have to be an integer.

• Getting the value stored is easy once we understand that the key is the index: Use the get(Key) method of HashMap.

• To get the color of the Banana in the fruit bowl, use this method which searches through the HashMap until it finds a Key match to the parameter ("Banana") and returns the Value for that Key ("Yellow").

```
String bananaColor = fruitBowl.get("Banana");
```

More HashMap Methods

Method	Method Description
boolean containsKey(Object Key)	Returns true if the HashMap contains the specified Key.
boolean containsValue(Object Value)	Returns true if this map maps one or more keys to the specified value.
Set<K> keySet()	Returns a set of the keys contained in the HashMap.
Collection<V> values()	Returns a collection of the values contained in the HashMap.
V remove(Object Key)	Removes the mapping for the specified key from this map if present.
int size()	Returns the number of key-value mappings in the HashMap.

Queues

• A Queue is a list of elements with a first in first out ordering.

• When you enqueue an element, it adds it to the end of the list.

• When you dequeue an element, it returns the element at the front of the list and removes that element from the list.

• For example, picture a line at the movie theater.

• The first person there is the first person to get their ticket (First In First Out, also known as FIFO).

Stacks

• Stacks are Queues that have reverse ordering to the standard Queue.

• Instead of FIFO ordering (like a queue or line at the theater), the ordering of a stack is last in first out.

• This can be represented by the acronym LIFO.

Stack of Pancakes Example

• For example, you have a pile of pancakes.

• Typically this would be called a "stack" of pancakes because the pancakes are added on top of the previous leaving the most recently added pancake at the top of the stack.

• To remove a pancake, you would have to take off the one that was most recently added: The pancake on the top of the stack.

• If you tried to remove the pancake that was added first, you would most likely make a very large mess.

Implementing a Stack: Deque

• One way to implement a Stack is by using a Double-Ended Queue (or deque, pronounced “deck”, for short).

• These allow us to insert and remove elements from either end of the queue using methods inside the Deque class.

• Deques like building blocks, allow you to put pieces on the bottom of your structure or on the top, and likewise pull pieces off from the bottom or top.

• Deques can be implemented by LinkedLists.

LinkedLists

• A LinkedList is a list of dynamically-stored elements.

• Like an ArrayList, it changes size and has an explicit ordering, but it doesn’t use an array to store information.

• It uses an object known as a Node.

• Nodes are like roadmaps: they tell you where you are (the element you are looking at), and where you can go (the previous element and the next element).

Adding Nodes to LinkedLists

• Ultimately, we have a list of Nodes, which point to other Nodes and have an element attached to them.

• To add a Node, set its left Node to the one on its left, and its right Node to the one on its right.

• Do not forget to change the Nodes around it as well.

• A fourth node was added to the end of this linked list:

Initializing a LinkedList

• A LinkedList is initialized in the same way as ArrayList.

• The following code shows how to initialize a LinkedList of pancakes.

```
LinkedList<Pancake> myStack = new LinkedList<Pancake>();
```

LinkedList Methods

FIFO LinkedList Methods	LIFO LinkedList Methods
add(E e) Appends the given element to the end of the list.	add(E e) Appends the given element to the end of the list.
removeFirst() Removes the first element from the list and returns it.	removeLast() Removes the last element from the list and returns it.
getFirst() Returns the first element of the list.	getLast() Returns the last element of the list.

Collection.sort

• We saw earlier to sort a collection with a simple element that we can use Collections.sort().

• In a previous example we saw an ArrayList of strings called groupNames being sorted by**:**

```
Collections.sort(groupNames);
```

- This will sort our ArrayList in its natural order.
- This is fine with simple elements but what about our Cell class?
- What if we had additional fields, then which field should it order on?

```
public class Cell {
                private String data
                 private String data2;
                 }
```

Comparable Interface

• For our classes to have a natural order we can implement the interface java.lang.Comparable.

```
public interface Comparable<T> {
        int compareTo(T o);
          }
```

So if we implement this interface we must write the code for compareTo for our class.

compareTo

• The compareTo method will return an integer based on the following:

• Return a negative value if this object is smaller than the other object

• Return 0 (zero) if this object is equal to the other object.

• Return a positive value if this object is larger than the other object.

```
import java.util.Comparator;
public class Cell implements Comparable<Cell> {
private String data;
public void set(String celldata) { data = celldata; }
```

```
public String get() { return data; }
public int compareTo(Cell c2) {
if(data.compareTo(c2.get()) < 0 ) return -1;
if (data.compareTo(c2.get()) == 0) return 0;
return 1;
    }
}
```

• Our compareTo method implementation could have used any or multiple fields from our class.

• We have chosen to simply use the String field called data.

```
public class Cell implements Comparable<Cell> {
private String data;
<snip>
public int compareTo(Cell c2) {
 if(data.compareTo(c2.get()) < 0 ) return -1;
if (data.compareTo(c2.get()) == 0) return 0; return 1; } }
```

Cell Driver

• Simple driver class.

```
Cell c1 = new Cell();
c1.set("Alice");
Cell c2 = new Cell();
c2.set("Brian");
ArrayList<Cell> list = new ArrayList<Cell>();
list.add(c2);
list.add(c1);
for (Cell c : list) { System.out.println(c.get()); }
Collections.sort(list);
 System.out.println("*Sorted*");
for (Cell c : list) { System.out.println(c.get()); }
```

```
Output
Brian
Alice
*Sorted*
Alice
Brian
```

GUI: Graphical User Interface

INTRODUCTION

The *Java AWT (Abstract Window Toolkit)* package is the original Java package for doing GUIs. A GUI (graphical user interface) is a windowing system that interacts with the user. The Swing package is an improved version of the *AWT* package. However, it does not completely replace the *AWT*. Some *AWT* classes are replaced by Swing classes, but other AWT classes are needed when using Swing. Swing GUIs are designed using a form of object-oriented programming known as *event-driven programming*.

Events

Event-driven programming is a programming style that uses a *signal-and-response* approach to programming. An *event* is an object that acts as a signal to another object know as a *listener*. The sending of an event is called *firing* the event. The object that fires the event is often a GUI component, such as a *button* that has been clicked.

Listeners

A listener object performs some action in response to the event. A given component may have any number of listeners. Each listener may respond to a different kind of event, or multiple listeners may respond to the same events

Exception Objects

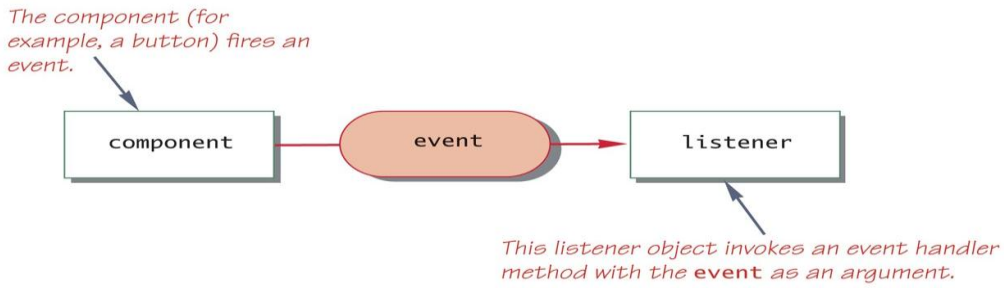
An exception object is an event. The throwing of an exception is an example of firing an event. The listener for an exception object is the **catch** block that catches the event.

Event Handlers

A listener object has methods that specify what will happen when events of various kinds are received by it. These methods are called *event handlers.* The programmer using the listener object will define or redefine these event-handler methods

Event Firing and an Event Listener

Display 17.1 **Event Firing and an Event Listener**



Event-Driven Programming

Event-driven programming is very different from most programming seen up until now. So far, programs have consisted of a list of statements executed in order. When that order changed, whether or not to perform certain actions (such as repeat statements in a loop, branch to another statement, or invoke a method) was controlled by the logic of the program.

In event-driven programming, objects are created that can fire events, and listener objects are created that can react to the events. The program itself no longer determines the order in which things can happen. Instead, the events determine the order.

In an event-driven program, the next thing that happens depends on the next event. In particular, *methods are defined that will never be explicitly invoked in any program.* Instead, methods are invoked automatically when an event signals that the method needs to be called.

A Simple Window

A simple window can consist of an object of the **JFrame** class. A **JFrame** object includes a border and the usual three buttons for minimizing, changing the size of, and closing the window. The **JFrame** class is found in the **javax.swing** package.

```
JFrame firstWindow = new JFrame();
```

A **JFrame** can have components added to it, such as buttons, menus, and text labels. These components can be programmed for action.

```
firstWindow.add(endButton);
```

It can be made visible using the **setVisible** method

```
firstWindow.setVisible(true);
```

Display 17.2 A First Swing Demonstration Program

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;

3  public class FirstSwingDemo
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public static void main(String[] args)
8      {
9          JFrame firstWindow = new JFrame();
10         firstWindow.setSize(WIDTH, HEIGHT);
```

This program is not typical of the style we will use in Swing programs.

(continued)

Display 17.2 A First Swing Demonstration Program

```
11         firstWindow.setDefaultCloseOperation(
12                                 JFrame.DO_NOTHING_ON_CLOSE);

13         JButton endButton = new JButton("Click to end program.");
14         EndingListener buttonEar = new EndingListener();
15         endButton.addActionListener(buttonEar);
16         firstWindow.add(endButton);

17         firstWindow.setVisible(true);
18     }
19 }
```

This is the file `FirstSwingDemo.java`.

(continued)

A First Swing Demonstration (Part 3 of 4)

Display 17.2 A First Swing Demonstration Program

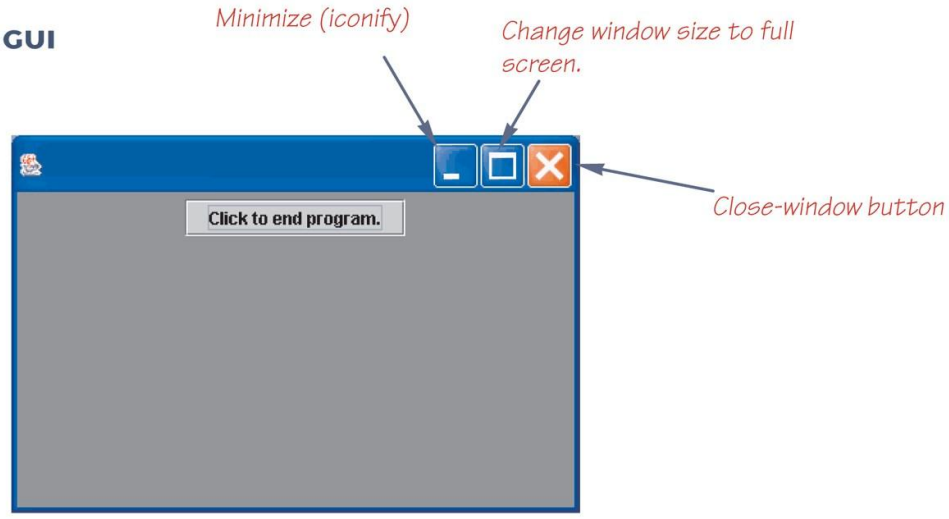
```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;

3  public class EndingListener implements ActionListener
4  {
5      public void actionPerformed(ActionEvent e)
6      {
7          System.exit(0);
8      }
9  }
```

This is the file `EndingListener.java`.

(continued)

Display 17.2 **A First Swing Demonstration Program**

RESULTING GUI



Display 17.3 Some Methods in the Class `JFrame`

The class `JFrame` is in the `javax.swing` package.

`public JFrame()`

Constructor that creates an object of the class `JFrame`.

`public JFrame(String title)`

Constructor that creates an object of the class `JFrame` with the title given as the argument.

(continued)

Display 17.3 Some Methods in the Class `JFrame`

`public void setDefaultCloseOperation(int operation)`

Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:
`JFrame.DO_NOTHING_ON_CLOSE`: Do nothing. The `JFrame` does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained in Chapter 19.)
`JFrame.HIDE_ON_CLOSE`: Hide the frame after invoking any registered `WindowListener` objects.
`JFrame.DISPOSE_ON_CLOSE`: Hide and *dispose* the frame after invoking any registered window listeners. When a window is **disposed** it is eliminated but the program does not end. To end the program, you use the next constant as an argument to `setDefaultCloseOperation`.
`JFrame.EXIT_ON_CLOSE`: Exit the application using the `System exit` method. (Do not use this for frames in applets. Applets are discussed in Chapter 18.)
If no action is specified using the method `setDefaultCloseOperation`, then the default action taken is `JFrame.HIDE_ON_CLOSE`.
Throws an `IllegalArgumentException` if the argument is not one of the values listed above.[2]
Throws a `SecurityException` if the argument is `JFrame.EXIT_ON_CLOSE` and the Security Manager will not allow the caller to invoke `System.exit`. (You are not likely to encounter this case.)

`public void setSize(int width, int height)`

Sets the size of the calling frame so that it has the `width` and `height` specified. Pixels are the units of length used.

(continued)

Display 17.3 Some Methods in the Class `JFrame`

`public void setTitle(String title)`

Sets the title for this frame to the argument string.

`public void add(Component componentAdded)`

Adds a component to the `JFrame`.

`public void setLayout(LayoutManager manager)`

Sets the layout manager. Layout managers are discussed later in this chapter.

`public void setJMenuBar(JMenuBar menubar)`

Sets the menubar for the calling frame. (Menus and menu bars are discussed later in this chapter.)

`public void dispose()`

Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program. (The method `dispose` is discussed in Chapter 19.)

Pixels and the Relationship between Resolution and Size

A *pixel* is the smallest unit of space on a screen. Both the size and position of Swing objects are measured in pixels. The more pixels on a screen, the greater the screen resolution. A highresolution screen of fixed size has many pixels. Therefore, each one is very small. A lowresolution screen of fixed size has fewer pixels. Therefore, each one is much larger. Therefore, a two-pixel figure on a low-resolution screen will look larger than a two-pixel figure on a high-resolution screen.

Pitfall: Forgetting to Program the Close-Window Button

The following lines from the **FirstSwingDemo** program ensure that when the user clicks the *close-window button*, nothing happens:

firstWindow.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

If this were not set, the default action would be JFrame.HIDE_ON_CLOSE. This would make the window invisible and inaccessible, but would not end the program. Therefore, given this scenario, there would be no way to click the "Click to end program" button. Note that the close-window and other two accompanying buttons are part of the **JFrame** object, and not separate buttons.

Buttons

A *button* object is created from the class **JButton** and can be added to a **JFrame.** The argument to the **JButton** constructor is the string that appears on the button when it is displayed.

JButton endButton = new JButton("Click to end program."); firstWindow.add(endButton);

Action Listeners and Action Events

Clicking a button fires an event. The event object is "sent" to another object called a listener. This means that a method in the listener object is invoked automatically. Furthermore, it is invoked with the event object as its argument. In order to set up this relationship, a GUI program must do two things:

1. It must specify, for each button, what objects are its listeners, i.e., it must register the listeners
2. It must define the methods that will be invoked automatically when the event is sent to the listener

EndingListener buttonEar = new EndingListener(); endButton.addActionListener(buttonEar);

Above, a listener object named **buttonEar** is created and registered as a listener for the button named **endButton .** Note that a button fires events known as *action events*, which are handled by listeners known as *action listeners.*

Different kinds of components require different kinds of listener classes to handle the events they fire. An action listener is an object whose class implements the **ActionListener** interface. The **ActionListener** interface has one method heading that must be implemented:

public void actionPerformed(ActionEvent e) public void actionPerformed(ActionEvent e)

```
{
  System.exit(0);
}
```

The **EndingListener** class defines its **actionPerformed** method as above. When the user clicks the **endButton**, an action event is sent to the action listener for that button. The **EndingListener** object **buttonEar** is the action

listener for **endButton.** The action listener **buttonEar** receives the action event as the parameter **e** to its **actionPerformed** method, which is automatically invoked. Note that **e** must be received, even if it is not used.

Pitfall: Changing the Heading for actionPerformed

When the **actionPerformed** method is implemented in an action listener, its header must be the one specified in the **ActionListener** interface. It is already determined, and may not be changed. Not even a throws clause may be added:

```
public void actionPerformed(ActionEvent e)
```

The only thing that can be changed is the name of the parameter, since it is just a placeholder. Whether it is called **e** or something else does not matter, as long as it is used consistently within the body of the method

Tip: Ending a Swing Program

GUI programs are often based on a kind of infinite loop. The windowing system normally stays on the screen until the user indicates that it should go away. If the user never asks the windowing system to go away, it will never go away. In order to end a GUI program, **System.exit** must be used when the user asks to end the program. It must be explicitly invoked, or included in some library code that is executed. Otherwise, a Swing program will not end after it has executed all the code in the program

A Better Version of Our First Swing GUI

A better version of **FirstWindow** makes it a subclass of the class **JFrame.** This is the normal way to define a windowing interface. The constructor in the new **FirstWindow** class starts by calling the constructor for the parent class using **super();.** This ensures that any initialization that is normally done for all objects of type **JFrame** will be done. Almost all initialization for the window **FirstWindow** is placed in the constructor for the class. Note that this time, an anonymous object is used as the action listener for the **endButton.**

Display 17.4 The Normal Way to Define a JFrame

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;


3  public class FirstWindow extends JFrame
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public FirstWindow()
8      {
9          super();
10         setSize(WIDTH, HEIGHT);

11         setTitle("First Window Class");
```

(continued)

Display 17.4 The Normal Way to Define a JFrame

```
12          setDefaultCloseOperation(
13                                  JFrame.DO_NOTHING_ON_CLOSE);

14          JButton endButton = new JButton("Click to end program.");
15          endButton.addActionListener(new EndingListener());
16          add(endButton);
17      }
18  }
```

The class `EndingListener` *is defined in Display 17.2.*

This is the file `FirstWindow.java`.

(continued)

Display 17.4 The Normal Way to Define a JFrame

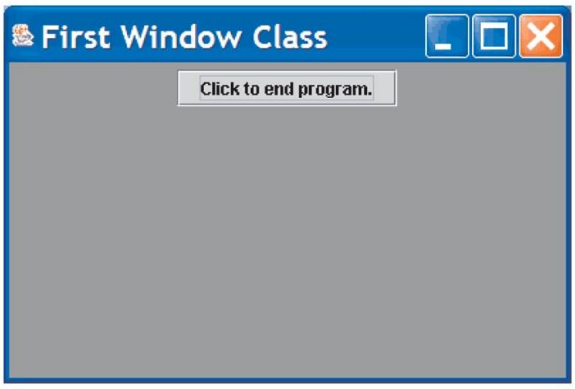
This is the file `DemoWindow.java`.

```
1   public class DemoWindow
2   {
3       public static void main(String[] args)
4       {
5           FirstWindow w = new FirstWindow();
6           w.setVisible(true);
7       }
8   }
```

(continued)

Display 17.4 The Normal Way to Define a JFrame

RESULTING GUI



Labels

A *label* is an object of the class **JLabel.** Text can be added to a **JFrame** using a label. The text for the label is given as an argument when the **JLabel** is created. The label can then be added to a **JFrame:**

JLabel greeting = new JLabel("Hello"); add(greeting);

Color

In Java, a *color* is an object of the class **Color.** The class **Color** is found in the **java.awt** package. There are constants in the **Color** class that represent a number of basic colors. A **JFrame** can not be colored directly. Instead, a program must color something called the *content pane* of the **JFrame.** Since the content

pane is the "inside" of a **JFrame**, coloring the content pane has the effect of coloring the inside of the **JFrame.** Therefore, the background color of a **JFrame** can be set using the following code:

getContentPane().setBackground(*Color*);

The Color Constants

Display 17.5 The Color Constants

`Color.BLACK`	`Color.MAGENTA`
`Color.BLUE`	`Color.ORANGE`
`Color.CYAN`	`Color.PINK`
`Color.DARK_GRAY`	`Color.RED`
`Color.GRAY`	`Color.WHITE`
`Color.GREEN`	`Color.YELLOW`
`Color.LIGHT_GRAY`	

The class `Color` is in the `java.awt` package.

A JFrame with Color (Part 1 of 4)

Display 17.6 A JFrame with Color

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.Color;

4  public class ColoredWindow extends JFrame
5  {
6      public static final int WIDTH = 300;
7      public static final int HEIGHT = 200;

8      public ColoredWindow(Color theColor)
9      {
10         super("No Charge for Color");
11         setSize(WIDTH, HEIGHT);
12         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

Display 17.6 A JFrame with Color

```
13         getContentPane().setBackground(theColor);

14         JLabel aLabel = new JLabel("Close-window button works.");
15         add(aLabel);
16     }

17     public ColoredWindow()
18     {
19         this(Color.PINK);
20     }
21 }
```

This is an invocation of the other constructor.

This is the file `ColoredWindow.java`.

(continued)

Display 17.6 A JFrame with Color

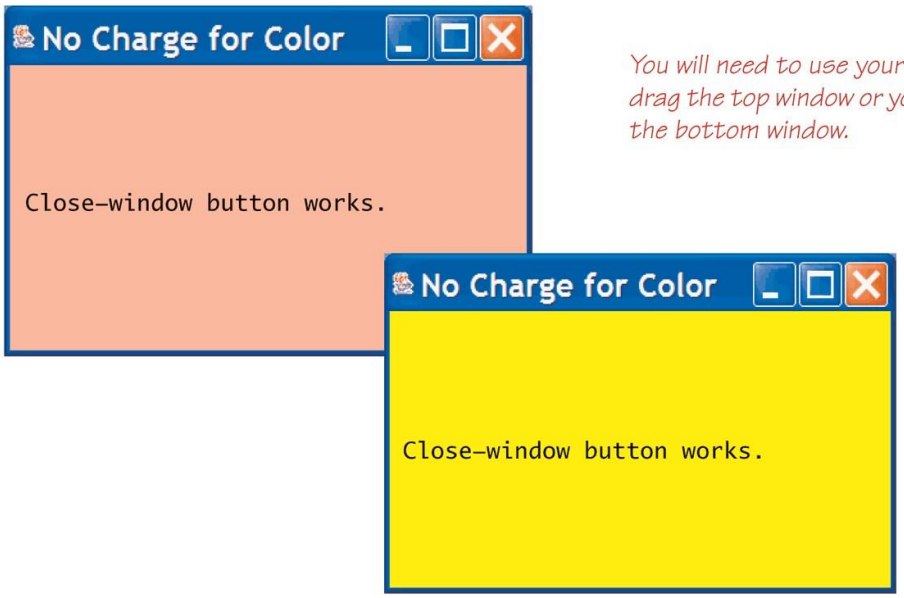
This is the file `ColoredWindow.java`.

```
1  import java.awt.Color;

2  public class DemoColoredWindow
3  {
4      public static void main(String[] args)
5      {
6          ColoredWindow w1 = new ColoredWindow();
7          w1.setVisible(true);

8          ColoredWindow w2 = new ColoredWindow(Color.YELLOW);
9          w2.setVisible(true);
10     }
11 }
```

(continued)

Display 17.6 **A JFrame with Color**

RESULTING GUI



You will need to use your mouse to drag the top window or you will not see the bottom window.

Containers and Layout Managers

Multiple components can be added to the content pane of a **JFrame** using the **add** method. However, the **add** method does not specify how these components are to be arranged. To describe how multiple components are to be arranged, a *layout manager* is used. There are a number of layout manager classes such as **BorderLayout**, **FlowLayout**, and **GridLayout.** If a layout manager is not specified, a default layout manager is used

Border Layout Managers

A **BorderLayout** manager places the components that are added to a **JFrame** object into five regions. These regions are: BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, and BorderLayout.Center

A **BorderLayout** manager is added to a **JFrame** using the **setLayout** method. For example:

```
setLayout(new BorderLayout());
```

The BorderLayout Manager (Part 1 of 4)

Display 17.7 The BorderLayout Manager

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.BorderLayout;

4  public class BorderLayoutJFrame extends JFrame
5  {
6      public static final int WIDTH = 500;
7      public static final int HEIGHT = 400;

8      public BorderLayoutJFrame()
9      {
10         super("BorderLayout Demonstration");
11         setSize(WIDTH, HEIGHT);
12         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

Display 17.7 The BorderLayout Manager

```
13         setLayout(new BorderLayout());

14         JLabel label1 = new JLabel("First label");
15         add(label1, BorderLayout.NORTH);

16         JLabel label2 = new JLabel("Second label");
17         add(label2, BorderLayout.SOUTH);

18         JLabel label3 = new JLabel("Third label");
19         add(label3, BorderLayout.CENTER);
20     }
21 }
```

This is the file `BorderLayoutJFrame.java`.

(continued)

Display 17.7 The BorderLayout Manager

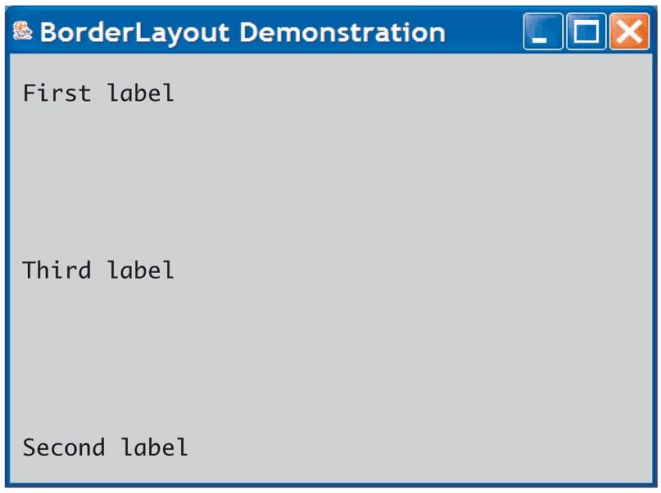
This is the file `BorderLayoutDemo.java`.

```
1  public class BorderLayoutDemo
2  {
3      public static void main(String[] args)
4      {
5          BorderLayoutJFrame gui = new BorderLayoutJFrame();
6          gui.setVisible(true);
7      }
8  }
```

(continued)

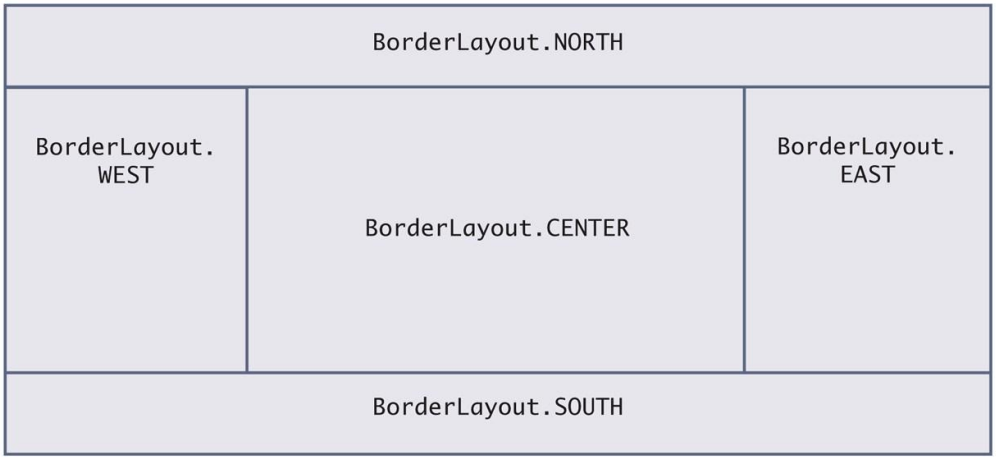
Display 17.7 **The BorderLayout Manager**

RESULTING GUI



BorderLayout Regions

Display 17.8 **BorderLayout Regions**



Border Layout Managers

The previous diagram shows the arrangement of the five border layout regions. Note: None of the lines in the diagram are normally visible. When using a **BorderLayout** manager, the location of the component being added is given as a second argument to the **add** method

```
add(label1, BorderLayout.NORTH);
```

Components can be added in any order since their location is specified.

Flow Layout Managers

The **FlowLayout** manager is the simplest layout manager.

```
setLayout(new FlowLayout());
```

It arranges components one after the other, going from left to right. Components are arranged in the order in which they are added. Since a location is not specified, the **add** method has only one argument when using the **FlowLayoutManager.**

add.(label1);

Grid Layout Managers

A **GridLayout** manager arranges components in a two-dimensional grid with some number of rows and columns.

setLayout(new GridLayout(rows, columns));

Each entry is the same size. The two numbers given as arguments specify the number of rows and columns. Each component is stretched so that it completely fills its grid position.

Note: None of the lines in the diagram are normally visible.

When using the **GridLayout** class, the method **add** has only one argument

add(label1);

Items are placed in the grid from left to right. The top row is filled first, then the second, and so forth. Grid positions may not be skipped. Note the use of a **main** method in the GUI class itself in the following example. This is often a convenient way of demonstrating a class.

Display 17.9 The GridLayout Manager

```
13      public GridLayoutJFrame(int rows, int columns )
14      {
15          super();
16          setSize(WIDTH, HEIGHT);
17          setTitle("GridLayout Demonstration");
18          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19          setLayout(new GridLayout(rows, columns ));

20          JLabel label1 = new JLabel("First label");
21          add(label1);
```

(continued)

Display 17.9 The GridLayout Manager

```
1   import javax.swing.JFrame;
2   import javax.swing.JLabel;
3   import java.awt.GridLayout;

4   public class GridLayoutJFrame extends JFrame
5   {
6       public static final int WIDTH = 500;
7       public static final int HEIGHT = 400;

8       public static void main(String[] args)
9       {
10          GridLayoutJFrame gui = new GridLayoutJFrame(2, 3);
11          gui.setVisible(true);
12      }
```

(continued)

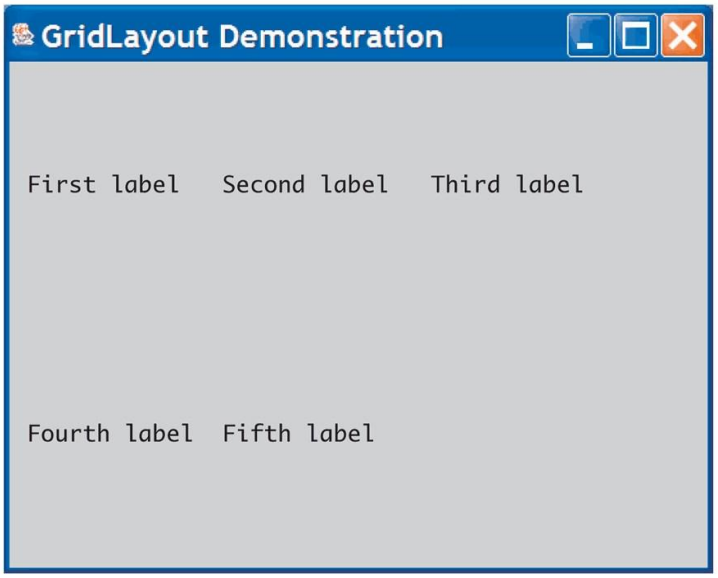
Display 17.9 The GridLayout Manager

```
22            JLabel label2 = new JLabel("Second label");
23            add(label2);

24            JLabel label3 = new JLabel("Third label");
25            add(label3);

26            JLabel label4 = new JLabel("Fourth label");
27            add(label4);

28            JLabel label5 = new JLabel("Fifth label");
29            add(label5);
30        }
31    }
```

(continued)

Display 17.9 The GridLayout Manager

RESULTING GUI



Display 17.10 **Some Layout Managers**

LAYOUT MANAGER	DESCRIPTION
These layout manager classes are in the `java.awt` package.	
`FlowLayout`	Displays components from left to right in the order in which they are added to the container.
`BorderLayout`	Displays the components in five areas: north, south, east, west, and center. You specify the area a component goes into in a second argument of the add method.
`GridLayout`	Lays out components in a grid, with each component stretched to fill its box in the grid.

Panels

A GUI is often organized in a hierarchical fashion, with containers called *panels* inside other containers. A panel is an object of the **JPanel** class that serves as a simple container. It is used to group smaller objects into a larger component (the panel). One of the main functions of a **JPanel** object is to subdivide a **JFrame** or other container. Both a **JFrame** and each panel in a **JFrame** can use different layout managers. Additional panels can be added to each panel, and each panel can have its own layout manager. This enables almost any kind of overall layout to be used in a GUI

```
setLayout(new BorderLayout()); JPanel somePanel = new JPanel();
somePanel.setLayout(new FlowLayout());
```

Note in the following example that panel and button objects are given color using the **setBackground** method without invoking **getContentPane.** The **getContentPane** method is only used when adding color to a **JFrame.**

Display 17.11 Using Panels

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.BorderLayout;
4  import java.awt.GridLayout;
5  import java.awt.FlowLayout;
6  import java.awt.Color;
7  import javax.swing.JButton;
8  import java.awt.event.ActionListener;
9  import java.awt.event.ActionEvent;

10 public class PanelDemo extends JFrame implements ActionListener
11 {
12     public static final int WIDTH = 300;
13     public static final int HEIGHT = 200;
```

In addition to being the GUI class, the class `PanelDemo` *is the action listener class. An object of the class* `PanelDemo` *is the action listener for the buttons in that object.*

(continued)

Display 17.11 Using Panels

```
14     private JPanel redPanel;
15     private JPanel whitePanel;
16     private JPanel bluePanel;

17     public static void main(String[] args)
18     {
19         PanelDemo gui = new PanelDemo();
20         gui.setVisible(true);
21     }

22     public PanelDemo()
23     {
24         super("Panel Demonstration");
25         setSize(WIDTH, HEIGHT);
26         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27         setLayout(new BorderLayout());
```

We made these instance variables because we want to refer to them in both the constructor and the method `actionPerformed`.

(continued)

Display 17.11 Using Panels

```
28         JPanel biggerPanel = new JPanel();
29         biggerPanel.setLayout(new GridLayout(1, 3));

30         redPanel = new JPanel();
31         redPanel.setBackground(Color.LIGHT_GRAY);
32         biggerPanel.add(redPanel);

33         whitePanel = new JPanel();
34         whitePanel.setBackground(Color.LIGHT_GRAY);
35         biggerPanel.add(whitePanel);
```

(continued)

Display 17.11 Using Panels

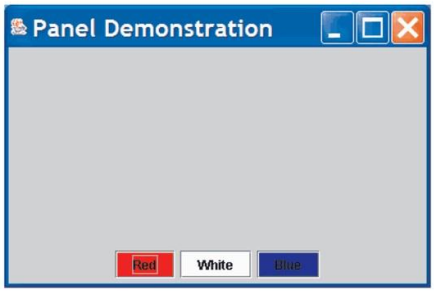
```
36          bluePanel = new JPanel();
37          bluePanel.setBackground(Color.LIGHT_GRAY);
38          biggerPanel.add(bluePanel);

39          add(biggerPanel, BorderLayout.CENTER);

40          JPanel buttonPanel = new JPanel();
41          buttonPanel.setBackground(Color.LIGHT_GRAY);
42          buttonPanel.setLayout(new FlowLayout());

43          JButton redButton = new JButton("Red");
44          redButton.setBackground(Color.RED);
45          redButton.addActionListener(this);
46          buttonPanel.add(redButton);

47          JButton whiteButton = new JButton("White");
48          whiteButton.setBackground(Color.WHITE);
49          whiteButton.addActionListener(this);
50          buttonPanel.add(whiteButton);

51          JButton blueButton = new JButton("Blue");
52          blueButton.setBackground(Color.BLUE);
53          blueButton.addActionListener(this);
54          buttonPanel.add(blueButton);

55          add(buttonPanel, BorderLayout.SOUTH);
56      }

57      public void actionPerformed(ActionEvent e)
58      {
59          String buttonString = e.getActionCommand();

60          if (buttonString.equals("Red"))
61               redPanel.setBackground(Color.RED);
62          else if (buttonString.equals("White"))
63               whitePanel.setBackground(Color.WHITE);
64          else if (buttonString.equals("Blue"))
65               bluePanel.setBackground(Color.BLUE);
66          else
67              System.out.println("Unexpected error.");
68      }
69  }
```

An object of the class `PanelDemo` *is the action listener for the buttons in that object.*

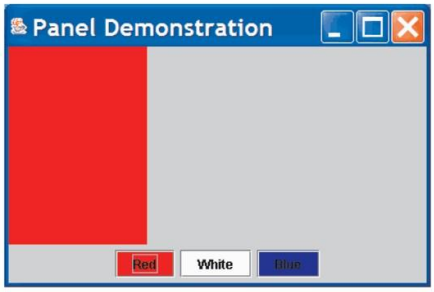
(continued)

Display 17.11 **Using Panels**

RESULTING GUI (When first run)



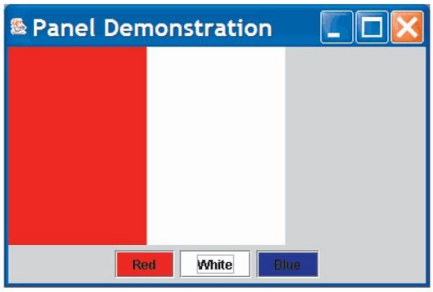
RESULTING GUI (After clicking Red button)



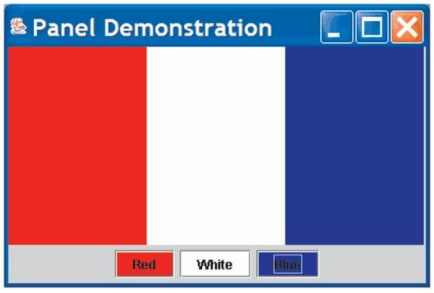
(continued)

Display 17.11 **Using Panels**

RESULTING GUI (After clicking White button)



RESULTING GUI (After clicking Blue button)



The Container Class

Any class that is a descendent class of the class Container is considered to be a container class. The **Container** class is found in the **java.awt** package, not in the Swing library. Any object that belongs to a class derived from the **Container** class (or its descendents) can have components added to it. The classes **JFrame** and **JPanel** are descendent classes of the class **Container.** Therefore they and any of their descendents can serve as a container.

The JComponent Class

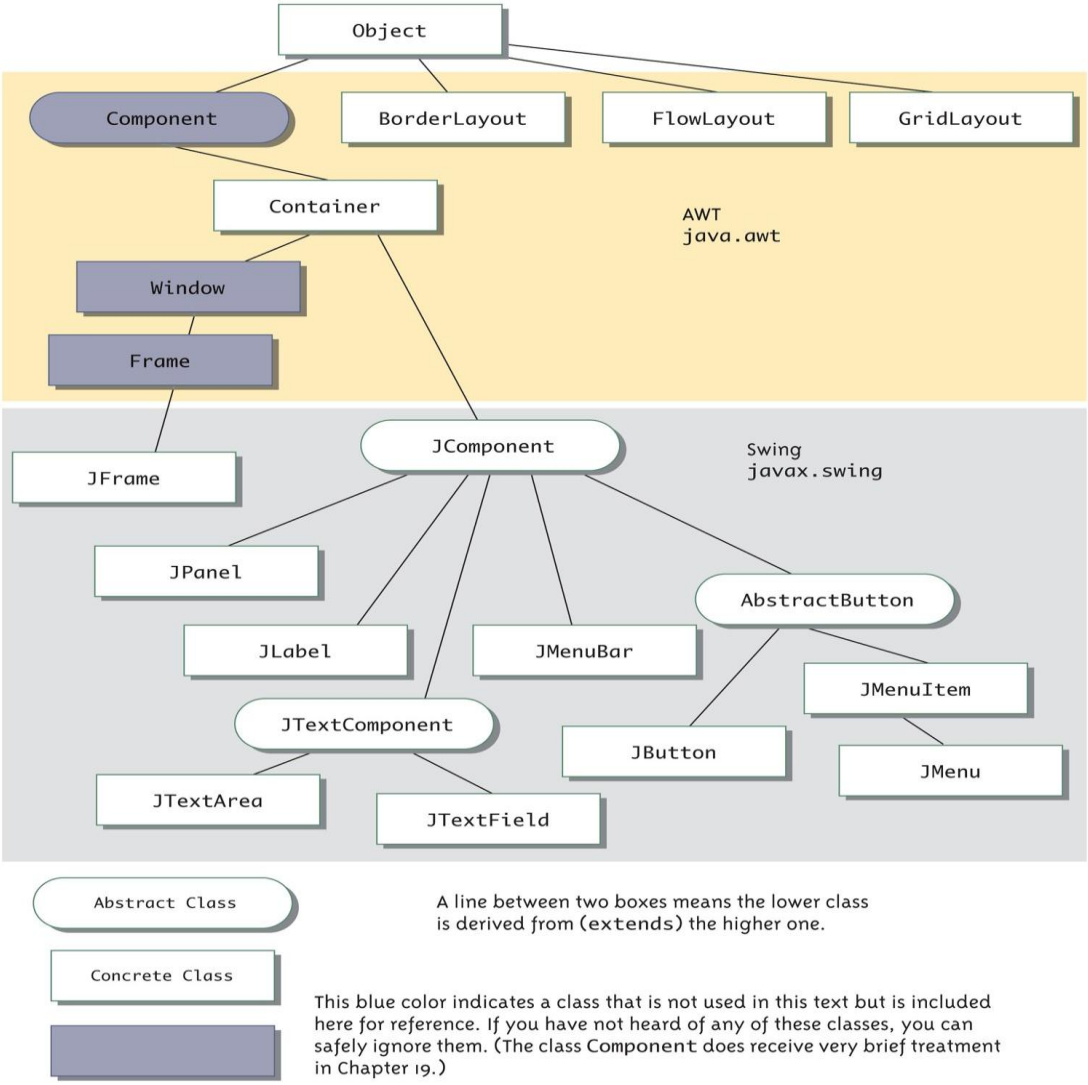
Any descendent class of the class **JComponent** is called a *component class.* Any **JComponent** object or *component* can be added to any container class object. Because it is derived from the class **Container**, a **JComponent** can also be added to another **JComponent.**

Objects in a Typical GUI

Almost every GUI built using Swing container classes will be made up of three kinds of objects:

1. The container itself, probably a panel or window-like object
2. The components added to the container such as labels, buttons, and panels
3. A layout manager to position the components inside the container

Display 17.12 **Hierarchy of Swing and AWT Classes**



Tip: Code a GUI's Look and Actions Separately

The task of designing a Swing GUI can be divided into two main subtasks:

1. Designing and coding the appearance of the GUI on the screen
2. Designing and coding the actions performed in response to user actions

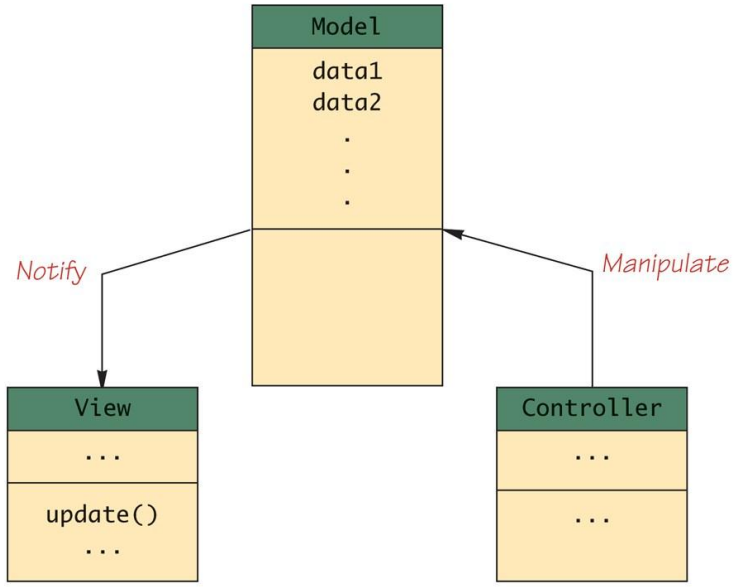
In particular, it is useful to implement the **actionPerformed** method as a *stub*, until the GUI looks the way it should

```
public void actionPerformed(ActionEvent e){}
```

This philosophy is at the heart of the technique used by the *Model-View-Controller* pattern.

The Model-View-Controller Pattern

Display 17.13 **The Model-View-Controller Pattern**



Menu Bars, Menus, and Menu Items

A *menu* is an object of the class **JMenu.** A choice on a menu is called a *menu item*, and is an object of the class **JMenuItem.** A menu can contain any number of menu items. A menu item is identified by the string that labels it, and is displayed in the order to which it was added to the menu. The **add** method is used to add a menu item to a menu in the same way that a component is added to a container object.

The following creates a new menu, and then adds a menu item to it

```
JMenu diner = new JMenu("Daily Specials"); JMenuItem lunch = new
JMenuItem("Lunch Specials"); lunch.addActionListener(this); diner.add(lunch);
```

Note that the **this** parameter has been registered as an action listener for the menu item.

Nested Menus

The class **JMenu** is a descendent of the **JMenuItem** class. Every **JMenu** can be a menu item in another menu. Therefore, menus can be nested. Menus can be added to other menus in the same way as menu items

Menu Bars and JFrame

A *menu bar* is a container for menus, typically placed near the top of a windowing interface. The **add** method is used to add a menu to a menu bar in the same way that menu items are added to a menu.

```
JMenuBar bar = new JMenuBar(); bar.add(diner);
```

The menu bar can be added to a **JFrame** in two different ways

1. Using the **setJMenuBar** method

   ```
   setJMenuBar(bar);
   ```

2. Using the **add** method – which can be used to add a menu bar to a **JFrame** or any other container

A GUI with a Menu

Display 17.14 A GUI with a Menu

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.GridLayout;
4  import java.awt.Color;
5  import javax.swing.JMenu;
6  import javax.swing.JMenuItem;
7  import javax.swing.JMenuBar;
8  import java.awt.event.ActionListener;
9  import java.awt.event.ActionEvent;
```

(continued)

Display 17.14 A GUI with a Menu

```
10 public class MenuDemo extends JFrame implements ActionListener
11 {
12     public static final int WIDTH = 300;
13     public static final int HEIGHT = 200;

14     private JPanel redPanel;
15     private JPanel whitePanel;
16     private JPanel bluePanel;

17     public static void main(String[] args)
18     {
19         MenuDemo gui = new MenuDemo();
20         gui.setVisible(true);
21     }

22     public MenuDemo()
23     {
24         super("Menu Demonstration");
25         setSize(WIDTH, HEIGHT);
26         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27         setLayout(new GridLayout(1, 3));

28         redPanel = new JPanel();
29         redPanel.setBackground(Color.LIGHT_GRAY);
30         add(redPanel);

31         whitePanel = new JPanel();
32         whitePanel.setBackground(Color.LIGHT_GRAY);
33         add(whitePanel);
```

(continued)

```
34          bluePanel = new JPanel();
35          bluePanel.setBackground(Color.LIGHT_GRAY);
36          add(bluePanel);

37          JMenu colorMenu = new JMenu("Add Colors");

38          JMenuItem redChoice = new JMenuItem("Red");
39          redChoice.addActionListener(this);
40          colorMenu.add(redChoice);

41          JMenuItem whiteChoice = new JMenuItem("White");
42          whiteChoice.addActionListener(this);
43          colorMenu.add(whiteChoice);
```

(continued)

Display 17.14 A GUI with a Menu

```
44          JMenuItem blueChoice = new JMenuItem("Blue");
45          blueChoice.addActionListener(this);
46          colorMenu.add(blueChoice);

47          JMenuBar bar = new JMenuBar();
48          bar.add(colorMenu);
49          setJMenuBar(bar);
50      }
```

The definition of `actionPerformed` *is identical to the definition given in Display 17.11 for a similar GUI using buttons instead of menu items.*

(continued)

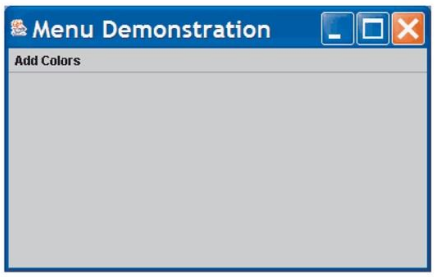
Display 17.14 A GUI with a Menu

```
51      public void actionPerformed(ActionEvent e)
52      {
53          String buttonString = e.getActionCommand();

54          if (buttonString.equals("Red"))
55               redPanel.setBackground(Color.RED);
56          else if (buttonString.equals("White"))
57              whitePanel.setBackground(Color.WHITE);
58          else if (buttonString.equals("Blue"))
59              bluePanel.setBackground(Color.BLUE);
60          else
61              System.out.println("Unexpected error.");
62      }
63  }
```

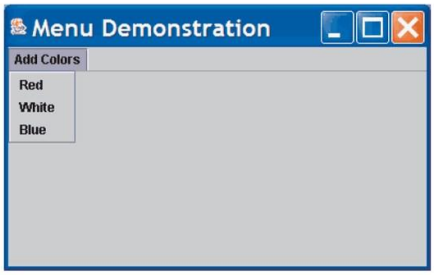
(continued)

Display 17.14 **A GUI with a Menu**

RESULTING GUI



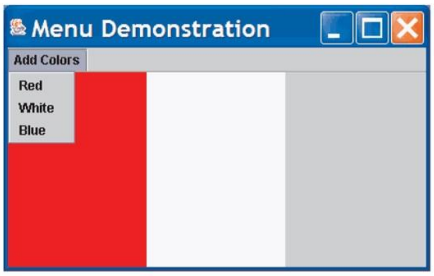
RESULTING GUI (after clicking Add `Colors` in the menu bar)



(continued)

Display 17.14 **A GUI with a Menu**

RESULTING GUI (after choosing Red and White on the menu)



RESULTING GUI (after choosing all the colors on the menu)

The AbstractButton and Dimension Classes

The classes **JButton** and **JMenuItem** are derived classes of the abstract class named

AbstractButton. All of their basic properties and methods are inherited from the class **AbstractButton.** Objects of the **Dimension** class are used with buttons, menu items, and other objects to specify a size. The **Dimension** class is in the package **java.awt**

Dimension(int width, int height)

Note: **width** and **height** parameters are in pixels.

The setActionCommand Method

When a user clicks a button or menu item, an event is fired that normally goes to one or more action listeners. The action event becomes an argument to an **actionPerformed** method. This action event includes a **String** instance variable called the *action command* for the button or menu item. The default value for this string is the string written on the button or the menu item. This string can be retrieved with the **getActionCommand** method.

e.getActionCommand()

The **setActionCommand** method can be used to change the action command for a component. This is especially useful when two or more buttons or menu items have the same default action command strings

JButton nextButton = new JButton("Next"); nextButton.setActionCommand("Next Button"); JMenuItem choose = new JMenuItem("Next"); choose.setActionCommand("Next Menu Item");

Some Methods in the Class AbstractButton

Display 17.15 Some Methods in the Class `AbstractButton`

The abstract class `AbstractButton` is in the `javax.swing` package.
All of these methods are inherited by both of the classes `JButton` and `JMenuItem`.

```
public void setBackground(Color theColor)
```

Sets the background color of this component.

```
public void addActionListener(ActionListener listener)
```

Adds an `ActionListener`.

```
public void removeActionListener(ActionListener listener)
```

Removes an `ActionListener`.

```
public void setActionCommand(String actionCommand)
```

Sets the action command.

(continued)

Display 17.15 Some Methods in the Class `AbstractButton`

```
public String getActionCommand()
```

Returns the action command for this component.

```
public void setText(String text)
```

Makes text the only text on this component.

```
public String getText()
```

Returns the text written on the component, such as the text on a button or the string for a menu item.

```
public void setPreferredSize(Dimension preferredSize)
```

Sets the preferred size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to use the preferred size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setPreferredSize(
                        new Dimension(int width, int height))
```

(continued)

Display 17.15 Some Methods in the Class `AbstractButton`

```
public void setMaximumSize(Dimension maximumSize)
```

Sets the maximum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this maximum size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMaximumSize(
                        new Dimension(int width, int height))
```

```
public void setMinimumSize(Dimension minimumSize)
```

Sets the minimum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this minimum size.
Although we do not discuss the `Dimension` class, the following special case is intuitively clear and will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMinimumSize(
                       new Dimension(int width, int height))
```

Listeners as Inner Classes

Often, instead of having one action listener object deal with all the action events in a GUI, a separate **ActionListener** class is created for each button or menu item. Each button or menu item has its own unique action listener. There is then no need for a multiway if-else statement. When this approach is used, each class is usually made a private inner class. **Listeners as Inner Classes**

Display 17.16 Listeners as Inner Classes

<Import statements are the same as in Display 17.14.>

```
1  public class InnerListenersDemo extends JFrame
2  {
3      public static final int WIDTH = 300;
4      public static final int HEIGHT = 200;

5      private JPanel redPanel;
6      private JPanel whitePanel;
7      private JPanel bluePanel;
```

(continued)

Display 17.16 Listeners as Inner Classes

```
 8      private class RedListener implements ActionListener
 9      {
10          public void actionPerformed(ActionEvent e)
11          {
12              redPanel.setBackground(Color.RED);
13          }
14      } //End of RedListener inner class

15      private class WhiteListener implements ActionListener
16      {
17          public void actionPerformed(ActionEvent e)
18          {
19              whitePanel.setBackground(Color.WHITE);
20          }
21      } //End of WhiteListener inner class
```

(continued)

Display 17.16 Listeners as Inner Classes

```
22      private class BlueListener implements ActionListener
23      {
24          public void actionPerformed(ActionEvent e)
25          {
26              bluePanel.setBackground(Color.BLUE);
27          }
28      } //End of BlueListener inner class

29      public static void main(String[] args)
30      {
31          InnerListenersDemo gui = new InnerListenersDemo();
32          gui.setVisible(true);
33      }
```

(continued)

Display 17.16 Listeners as Inner Classes

The resulting GUI is the same as in Display 17.14.

```
34       public InnerListenersDemo()
35       {
36           super("Menu Demonstration");
37           setSize(WIDTH, HEIGHT);
38           setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39           setLayout(new GridLayout(1, 3));

40           redPanel = new JPanel();
41           redPanel.setBackground(Color.LIGHT_GRAY);
42           add(redPanel);

43           whitePanel = new JPanel();
44           whitePanel.setBackground(Color.LIGHT_GRAY);
45           add(whitePanel);
```

(continued)

Display 17.16 Listeners as Inner Classes

```
46          bluePanel = new JPanel();
47          bluePanel.setBackground(Color.LIGHT_GRAY);
48          add(bluePanel);

49          JMenu colorMenu = new JMenu("Add Colors");

50          JMenuItem redChoice = new JMenuItem("Red");
51          redChoice.addActionListener(new RedListener());
52          colorMenu.add(redChoice);
```

(continued)

Display 17.16 Listeners as Inner Classes

```
53          JMenuItem whiteChoice = new JMenuItem("White");
54          whiteChoice.addActionListener(new WhiteListener());
55          colorMenu.add(whiteChoice);

56          JMenuItem blueChoice = new JMenuItem("Blue");
57          blueChoice.addActionListener(new BlueListener());
58          colorMenu.add(blueChoice);

59          JMenuBar bar = new JMenuBar();
60          bar.add(colorMenu);
61          setJMenuBar(bar);
62      }

63  }
```

Text Fields

A *text field* is an object of the class **JTextField.** It is displayed as a field that allows the user to enter a single line of text.

```
private JTextField name;
. . .
name = new JTextField(NUMBER_OF_CHAR);
```

In the text field above, at least NUMBER**_OF_CHAR** characters can be visible. There is also a constructor with one additional **String** parameter for displaying an initial **String** in the text field.

JTextField name = new JTextField("Enter name here.", 30);

A Swing GUI can read the text in a text field using the **getText** method.

```
String inputString = name.getText();
```

The method **setText** can be used to display a new text string in a text field.

```
name.setText("This is some output");
```

Display 17.17 A Text Field

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextField;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import java.awt.GridLayout;
7  import java.awt.BorderLayout;
8  import java.awt.FlowLayout;
9  import java.awt.Color;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;
```

(continued)

Display 17.17 A Text Field

```
12 public class TextFieldDemo extends JFrame
13                            implements ActionListener
14 {
15     public static final int WIDTH = 400;
16     public static final int HEIGHT = 200;
17     public static final int NUMBER_OF_CHAR = 30;

18     private JTextField name;


19     public static void main(String[] args)
20     {
21         TextFieldDemo gui = new TextFieldDemo();
22         gui.setVisible(true);
23     }
```

(continued)

Display 17.17 A Text Field

```
24     public TextFieldDemo()
25     {
26         super("Text Field Demo");
27         setSize(WIDTH, HEIGHT);
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29         setLayout(new GridLayout(2, 1));

30         JPanel namePanel = new JPanel();
31         namePanel.setLayout(new BorderLayout());
32         namePanel.setBackground(Color.WHITE);

33         name = new JTextField(NUMBER_OF_CHAR);
```

(continued)

Display 17.17 A Text Field

```
34          namePanel.add(name, BorderLayout.SOUTH);
35          JLabel nameLabel = new JLabel("Enter your name here:");
36          namePanel.add(nameLabel, BorderLayout.CENTER);

37          add(namePanel);

38          JPanel buttonPanel = new JPanel();
39          buttonPanel.setLayout(new FlowLayout());
40          buttonPanel.setBackground(Color.PINK);
41          JButton actionButton = new JButton("Click me");
42          actionButton.addActionListener(this);
43          buttonPanel.add(actionButton);

44          JButton clearButton = new JButton("Clear");
45          clearButton.addActionListener(this);
46          buttonPanel.add(clearButton);
```

(continued)

Display 17.17 A Text Field

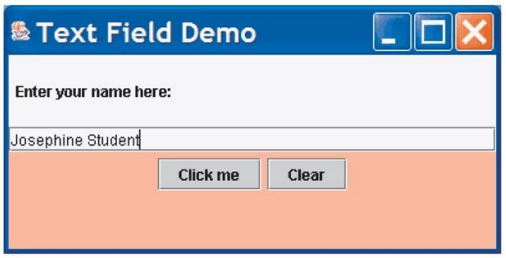
```
47          add(buttonPanel);
48      }


49      public void actionPerformed(ActionEvent e)
50      {
51          String actionCommand = e.getActionCommand();

52          if (actionCommand.equals("Click me"))
53              name.setText("Hello " + name.getText());
54          else if (actionCommand.equals("Clear"))
55              name.setText("");
56          else
57              name.setText("Unexpected error.");
58      }

59  }
```

This sets the text field equal to the empty string, which makes it blank.

(continued)

Display 17.17 A Text Field

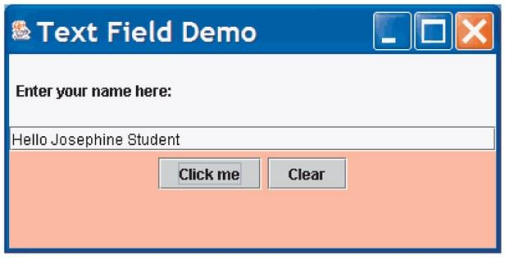
RESULTING GUI (When program is started and a name entered)



(continued)

Display 17.17 **A Text Field**

RESULTING GUI (After clicking the "Click me" button)



Text Areas

A *text area* is an object of the class **JTextArea.** It is the same as a text field, except that it allows multiple lines. Two parameters to the **JTextArea** constructor specify the minimum number of lines, and the minimum number of characters per line that are guaranteed to be visible.

```
JTextArea theText = new JTextArea(5,20);
```

Another constructor has one addition **String** parameter for the string initially displayed in the text area.

```
JTextArea theText = new JTextArea("Enter\ntext here." 5, 20);
```

The line-wrapping policy for a **JTextArea** can be set using the method **setLineWrap.** The method takes one **boolean** type argument, If the argument is **true**, then any additional characters at the end of a line will appear on the following line of the text area. If the argument is **false**, the extra characters will remain on the same line and not be visible.

```
theText.setLineWrap(true);
```

Text Fields and Text Areas

A **JTextField** or **JTextArea** can be set so that it cannot be changed by the user.

```
theText.setEditable(false);
```

This will set **theText** so that it can only be edited by the GUI program, not the user. To reverse this, use true instead (this is the default)

```
theText.setEditable(true);
```

Tip: Labeling a Text Field

In order to label one or more text fields:

- Use an object of the class **JLabel**
- Place the text field(s) and label(s) in a **JPanel**
- Treat the **JPanel** as a single component

Numbers of Characters Per Line

The number of characters per line for a **JTextField** or **JTextArea** object is the number of *em* spaces. An *em space* is the space needed to hold one uppercase letter **M.** The letter **M** is the widest letter in the alphabet. A line specified to hold 20 **M** 's will almost always be able to hold more than 20 characters.

Tip: Inputting and Outputting Numbers

When attempting to input numbers from any Swing GUI, input text must be converted to numbers. If the user enters the number **42** in a **JTextField**, the program receives the string **"42"** and must convert it to the integer **42.** The same thing is true when attempting to output a number. In order to output the number **42**, it must first be converted to the string **"42".**

The Class JTextComponent

Both **JTextField** and **JTextArea** are derived classes of the abstract class **JTextComponent.**

Most of their methods are inherited from **JTextComponent** and have the same meanings. Except for some minor redefinitions to account for having just one line or multiple lines.

Some Methods in the Class JTextComponent

Display 17.18 Some Methods in the Class `JTextComponent`

```
public void setBackground(Color theColor)
```

Sets the background color of this text component.

```
public void setEditable(boolean argument)
```

If `argument` is `true`, then the user is allowed to write in the text component. If `argument` is `false`, then the user is not allowed to write in the text component.

```
public void setText(String text)
```

Sets the text that is displayed by this text component to be the specified `text`.

Display 17.18 Some Methods in the Class `JTextComponent`

All these methods are inherited by the classes `JTextField` and `JTextArea`.
The abstract class `JTextComponent` is in the package `javax.swing.text`. The classes `JTextField` and `JTextArea` are in the package `javax.swing`.

```
public String getText()
```

Returns the text that is displayed by this text component.

```
public boolean isEditable()
```

Returns `true` if the user can write in this text component. Returns `false` if the user is not allowed to write in this text component.

(continued)

A Swing Calculator

A GUI for a simple calculator keeps a running total of numbers. The user enters a number in the text field, and then clicks either + or –. The number in the text field is then added to or subtracted from the running total, and displayed in the text field. This value is kept in the instance variable **result.** When the GUI is first run, or when the user clicks the **Reset** button, the value of **result** is set to zero.

If the user enters a number in an incorrect format, then one of the methods throws a **NumberFormatException.** The exception is caught in the catch block inside the **actionPerformed** method. Note that when this exception is thrown, the value of the instance variable **result** is not changed.

A Simple Calculator (Part 1 of 11)

Display 17.19 A Simple Calculator

```
 1  import javax.swing.JFrame;
 2  import javax.swing.JTextField;
 3  import javax.swing.JPanel;
 4  import javax.swing.JLabel;
 5  import javax.swing.JButton;
 6  import java.awt.BorderLayout;
 7  import java.awt.FlowLayout;
 8  import java.awt.Color;
 9  import java.awt.event.ActionListener;
10  import java.awt.event.ActionEvent;
```

(continued)

Display 17.19 A Simple Calculator

```
11  /**
12   A simplified calculator.
13   The only operations are addition and subtraction.
14  */
15  public class Calculator extends JFrame
16                          implements ActionListener
17  {
18      public static final int WIDTH = 400;
19      public static final int HEIGHT = 200;
20      public static final int NUMBER_OF_DIGITS = 30;

21      private JTextField ioField;
22      private double result = 0.0;

23      public static void main(String[] args)
24      {
25          Calculator aCalculator = new Calculator();
26          aCalculator.setVisible(true);
27      }
```

(continued)

Display 17.19 A Simple Calculator

```
28      public Calculator()
29      {
30          setTitle("Simplified Calculator");
31          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32          setSize(WIDTH, HEIGHT);
33          setLayout(new BorderLayout());

34          JPanel textPanel = new JPanel();
35          textPanel.setLayout(new FlowLayout());
36          ioField =
37               new JTextField("Enter numbers here.", NUMBER_OF_DIGITS);
38          ioField.setBackground(Color.WHITE);
39          textPanel.add(ioField);
40          add(textPanel, BorderLayout.NORTH);
```

(continued)

Display 17.19 A Simple Calculator

```
41          JPanel buttonPanel = new JPanel();
42          buttonPanel.setBackground(Color.BLUE);
43          buttonPanel.setLayout(new FlowLayout());

44          JButton addButton = new JButton("+");
45          addButton.addActionListener(this);
46          buttonPanel.add(addButton);
47          JButton subtractButton = new JButton("-");
48          subtractButton.addActionListener(this);
49          buttonPanel.add(subtractButton);
50          JButton resetButton = new JButton("Reset");
51          resetButton.addActionListener(this);
52          buttonPanel.add(resetButton);

53         add(buttonPanel, BorderLayout.CENTER);
54      }
```

(continued)

Display 17.19 A Simple Calculator

```
55      public void actionPerformed(ActionEvent e)
56      {
57          try
58          {
59              assumingCorrectNumberFormats(e);
60          }
61          catch (NumberFormatException e2)
62          {
63              ioField.setText("Error: Reenter Number.");
64          }
65      }
```

A `NumberFormatException` *does not need to be declared or caught in a* `catch` *block.*

(continued)

Display 17.19 A Simple Calculator

```
66      //Throws NumberFormatException.
67      public void assumingCorrectNumberFormats(ActionEvent e)
68      {
69          String actionCommand = e.getActionCommand();

70          if (actionCommand.equals("+"))
71          {
72              result = result + stringToDouble(ioField.getText());
73              ioField.setText(Double.toString(result));
74          }
75          else if (actionCommand.equals("-"))
76          {
77              result = result - stringToDouble(ioField.getText());
78              ioField.setText(Double.toString(result));
```

(continued)

Display 17.19 A Simple Calculator

```
79          }
80          else if (actionCommand.equals("Reset"))
81          {
82              result = 0.0;
83              ioField.setText("0.0");
84          }
85          else
86              ioField.setText("Unexpected error.");
87      }
```

(continued)

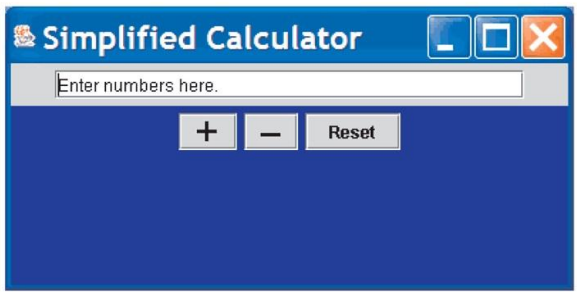
Display 17.19 A Simple Calculator

```
88      //Throws NumberFormatException.
89      private static double stringToDouble(String stringObject)
90      {
91          return Double.parseDouble(stringObject.trim());
92      }

93  }
```

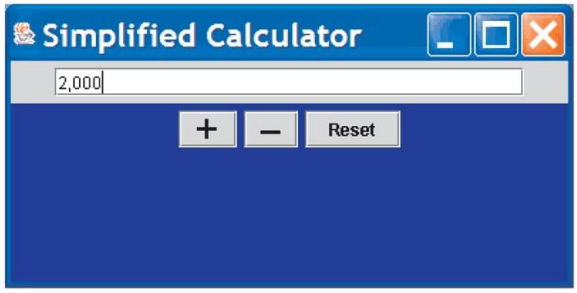
(continued)

Display 17.19 A Simple Calculator

RESULTING GUI (When started)



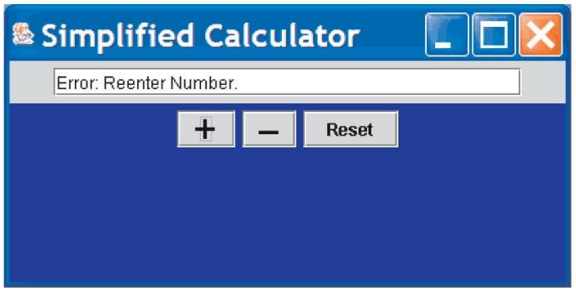
RESULTING GUI (After entering 2,000)



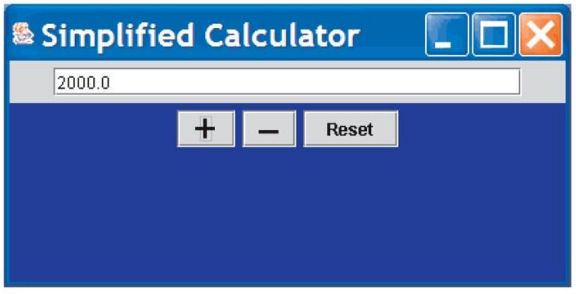
(continued)

Display 17.19 A Simple Calculator

RESULTING GUI (After clicking +)



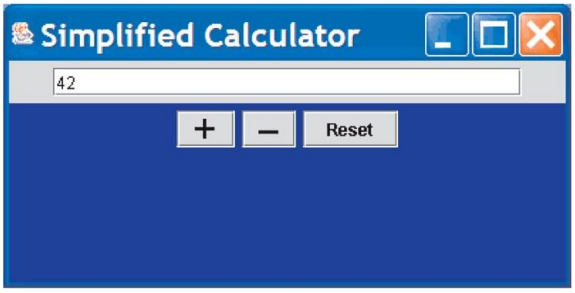
RESULTING GUI (After entering 2000 and clicking +)



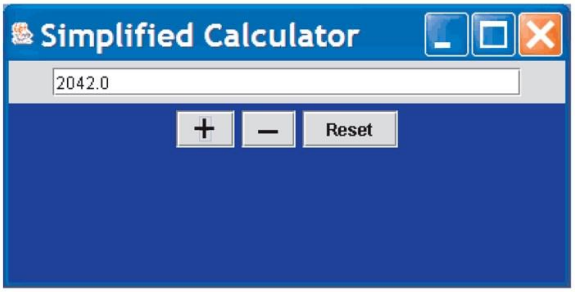
(continued)

Display 17.19 **A Simple Calculator**

RESULTING GUI (After entering 42)



RESULTING GUI (After clicking +)



Uncaught Exceptions

In a Swing program, throwing an uncaught exception does not end the GUI. However, it may leave it in an unpredictable state. It is always best to catch any exception that is thrown even if all the catch block does is output an error message, or ask the user to reenter some input.

More notes on:

- ✓ Programming Guideline
- ✓ Exception Handling
- ✓ Input and output stream
- ✓ Threads

DO IT YOURSELF

1. **Write a complete Java application to prompt the user for the double radius of a sphere, and call method sphereVolumeto calculate and display the volume of the sphere. Use the following statement to calculate the volume:** `(4.0/3.0)* Math.PI * Math.pow (radius,3)`

```
// File: Question6.java
// Author: Khaoya Muyobo

import java.util.Scanner;  // include scanner utility for accepting keyboard input

public class Question6 {

public static double sphereVolume(double radius) { // begin sphereVolume method

return (4.0/3.0)* Math.PI * Math.pow (radius,3); // return the volume after calculation
}

public static void main(String[] args) { // begin the main method

Scanner input=new Scanner (System.in); //create a new Scanner object to use

double radius=0.0, volume=0.0; // initialize variables

System.out.printf("Enter Radius: ");

radius=input.nextInt();// store next integer in radius

// display the Volume by calling the sphereVolume method

System.out.printf("Volume = %.3f", sphereVolume(radius));
}
}
```

2. **Write statements that perform the following one-dimensional-array operations:**
   - **Set the 10 elements of integer array counts to zero.**
   - **Add one to each of the 15 elements of integer array bonus.**
   - **Display the five values of integer array bestScores in column format.**

```
    a. File: Question1.java
    b. Author: Khaoya Muyobo
public class Question1 { // begin class
public static void main(String args[]) { // begin the main method
// part a
int array[]={0,0,0,0,0,0,0,0,0,0}; // declaring and setting 10 elements in the array with zero
// part b
int bonus[];
bonus=new int[15];  // declaring array bonus with 15 elements
for(int i=0;i<15;i++){ // adding 1 to each element
bonus[i]+=1;
}
// part c
int bestScores[]={10,20,30,40,50}; // declaring the array bestScores of 5 elements
for (int j=0;j<5;j++){
System.out.printf("%d\t", bestScores[j]); // displaying them in a column format
}
}
}
```

3. **Write a Java program that reads a string from the keyboard, and outputs the string twice in a row, first all uppercase and next all lowercase. If, for instance, the string “Hello" is given, the output will be “HELLOhello"**

```
    a. File: Question2.java
    b. Author: khaoya Muyobo
import java.util.Scanner; // include scanner utility for accepting input
public class Question2 { // begin class
public static void main(String[] args) {// begin the main method
Scanner input=new Scanner(System.in); //create a new Scanner object to use
String str; //declaring a string variable str
System.out.printf("Enter String: ");
str=input.nextLine();// store next line in str
// display the same string in both uppercase and lowercase
System.out.printf("%s%s",str.toUpperCase(),str.toLowerCase());
}
}
```

4. **Write a Java application that allows the user to enter up to 20 integer grades into an array. Stop the loop by typing in -1. Your main method should call an Average method that returns the average of the grades. Use the DecimalFormat class to format the average to 2 decimal places.**

```
    a. File: Question3.java
    b. Author: Khaoya Muyobo
import java.util.Scanner; // include scanner utility for accepting input
public class Question3 {// begin class
public static double Average(int grades[], int max ) {// begin Average method
int sum=0; // initialize variables
double average=0.0;
for (int i=1; i<max; i++){ // loop and calculate the Average
sum+=grades[i];
average=sum/(i);
}
return average; // return the average after calculation
}
public static void main(String[] args) {// begin the main method
Scanner input=new Scanner(System.in); //create a new Scanner object to use
int i, grades[]; // initialize variables
grades=new int[20];
for (i=0; i<20; i++){ // start to loop 20 times
System.out.printf("Enter Grade: ");
grades[i]=input.nextInt();// store next integer in grades[i]
if (grades[i]==-1)
break;
}
System.out.printf("%.2f", Average(grades, i-1));
}
}
```

5. **Modify class Account (in the example used in our lesson) to provide a method called debit that withdraws money from an Account. Ensure that the debit amount does not exceed the Account's balance. If it does, the balance should be left unchanged and the method should print a message indicating —Debit amount exceeded account balance. Modify class AccountTest (in the example) to test method debit.**

```
//filename: Account.java
// Accout class
public class Account {
 private double balance;
public Account(double initialBalance) {
if (initialBalance > 0.0) // initialize if initial balance is not negative
balance=initialBalance;
}
public void credit(double amount){
balance=balance+amount;
}
public void debit(double amount){
 balance=balance-amount;
}
public double getBalance(){
return balance;
}
}
```

```
//filename: AccountTest.java
// Accout testing class with the main() method
import java.util.Scanner;
public class AccountTest {
public static void main (String args[ ]){
        Account account1 = new Account (50.00);
        Account account2 = new Account (-7.53);
System.out.printf("Account1 Balance: KES %.2f\n", account1.getBalance());
System.out.printf("Account2 Balance: KES %.2f\n\n" , account2.getBalance());
Scanner input = new Scanner ( System.in );
 double depositAmount;
double debitAmount;
System.out.print( "Enter deposit amount for account1: " ); // prompt
depositAmount = input.nextDouble(); // obtain user input
System.out.printf( "\nadding %.2f to account1 balance\n\n", depositAmount );
 account1.credit( depositAmount ); // add to account1 balance
// display balances
System.out.printf( "Account1 balance: KES %.2f\n", account1.getBalance() );
System.out.printf( "Account2 balance: KES %.2f\n\n" , account2.getBalance() );
System.out.print( "Enter deposit amount for account2: " ); // prompt
depositAmount = input.nextDouble(); // obtain user input
System.out.printf( "\nAdding %.2f to account2 balance\n\n", depositAmount );
account2.credit( depositAmount ); // add to account2 balance
// display balances
System.out.printf( "Account1 balance: KES %.2f\n", account1.getBalance() );
System.out.printf( "Account2 balance: KES %.2f\n", account2.getBalance() );
System.out.print( "Enter debit amount for account1: " );
debitAmount = input.nextDouble();
System.out.printf( "\nSubtracting %.2f from account1 balance\n\n", debitAmount );
if (account1.getBalance()>=debitAmount) {
account1.debit( debitAmount );
System.out.printf( "Account1 balance: KES %.2f\n", account1.getBalance() );
System.out.printf( "Account2 balance: KES %.2f\n\n" , account2.getBalance() );
```

```
}
else {
System.out.printf("!!! Debit amount exceeded account balance!!!\n\n");
}
// display balances
System.out.print( "Enter debit amount for account2: " );
debitAmount = input.nextDouble();
System.out.printf( "\nSubtracting %.2f from account2 balance\n\n",
debitAmount );
 if (account1.getBalance()>=debitAmount) {
account1.debit( debitAmount );
System.out.printf( "Account1 balance: KES %.2f\n", account1.getBalance() );
System.out.printf( "Account2 balance: KES %.2f\n\n" , account2.getBalance() );
}
else {
System.out.printf("!!!Debit amount exceeded account balance!!!\n\n");
}
}
}
```

6. **Create a class called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four pieces of information as instance variables- a part number (type String),a part description(type String),a quantity of the item being purchased (type int) and a price per item (double). Your class should have a constructor that initializes the four instance variables. Provide a set and a get method for each instance variable. In addition, provide a method named getInvoice Amount that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a double value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0.0. Write a test application named InvoiceTest that demonstrates class Invoice's capabilities.**

```
//filename: Invoice.java
// Invoice class
public class Invoice {
        private String partNumber;
        private String partDescription;
         private int quantity;
         private double price;
public Invoice(String pNum, String pDesc, int qty, double prc) {
if (pNum != null)// assign if the value of part Number is not equal to zero
 partNumber=pNum;
 else partNumber="0";// part number is set to zero
if (pDesc != null)
partDescription=pDesc;
else partDescription="0";
if (qty > 0) quantity=qty;
else quantity=0;
if (prc > 0.0) price=prc;
else price=0;
}
public String getPartNum(){      return partNumber;        }
public String getPartDesc(){     return partDescription;   }
public int getQuantity(){        return quantity;          }
public double getPrice(){        return price;             }
```

```
public void setPartNum(String pNum){
if (pNum != null) {  partNumber=pNum;   } else {   partNumber="0";   }
}
public void setPartDesc(String pDesc){
if (pDesc != null) {   partDescription=pDesc;   } else {   partDescription="0";    }
}
public void setQuantity  (int qty)
{
if (qty > 0) { quantity=qty;  }
else {   quantity=0;     }
}
public void setPrice(double prc){
if (prc > 0.0) {   price=prc;   }
 else     {   price=0.0;    }
}
public double getInvoiceAmount() { return double)quantity*price; }
}
```

```
//filename: InvoiceTest.java
// Invoice testing class with the main() method
public class InvoiceTest {
public static void main (String args[]){
Invoice invoice1=new Invoice ("A5544", "Big Black Book", 500, 250.00);
Invoice invoice2=new Invoice ("A5542", "Big Pink Book", 300, 50.00);
System.out.printf("Invoice 1: %s\t%s\t%d\t$%.2f\n", invoice1.getPartNum(),
invoice1.getPartDesc(), invoice1.getQuantity(), invoice1.getPrice());
System.out.printf("Invoice 2: %s\t%s\t%d\t$%.2f\n", invoice2.getPartNum(),
invoice2.getPartDesc(), invoice2.getQuantity(), invoice2.getPrice());
}
}
```

7. **Create a class called Employee that includes three pieces of information as instance variables—a first name (type String), a last name (type String) and a monthly salary (type double). Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. If the monthly salary is not positive, set it to 0.0. Write a test application named EmployeeTest that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10% raise and display each Employee's yearly salary again.**

```
//filename: Employee.java
// Employee class
public class Employee {
 private String firstName;
 private String lastName;
private double salary;
public Employee(String fName, String lName, double sal) {
if (fName != null) firstName =fName;
      if (lName != null) lastName = lName;
      if (sal > 0.0) {    salary=sal;     }
      else { salary=0.0;  }
}
//get methods
public String getFirstName()   { return firstName;     }
public String getLastName()   {  return lastName;     }
public double getSalary()       { return salary;          }
//set methods
public void setFirstName(String fName){
if (fName != null)
firstName = fName;
}
public void setLastName(String lName){
if (lName != null)
lastName = lName;
}
public void setSalary(double sal){
if (sal > 0.0){    salary = sal;   }
```

```
else {salary = 0.0; }
}
}
```

```
//filename: EmployeeTest.java
// Employee testing class with the main() method
public class EmployeeTest {
public static void main (String args[]){
Employee employee1=new Employee ("Khaoya", "Muyobo", 150000.00);
Employee employee2=new Employee ("Kilwake", "Humphery", 200000.00);
System.out.printf("\nNO:\t NAME\t\t\tYEARLY SALARY\n" );
System.out.printf("--\t ----\t\t\t----------\n");
System.out.printf("1:\t %s %s\t\t$%.2f\n",employee1.getFirstName(),employee1.getLastName(),
employee1.getSalary( ));
System.out.printf("2: \t %s %s\t\t$%.2f\n", employee2.getFirstName(),employee2.getLastName(),
employee2.getSalary( ));
//set raise by  10%
employee1.setSalary((.1*employee1.getSalary())+employee1.getSalary( ));
employee2.setSalary( (.1*employee2.getSalary())+employee2.getSalary());

System.out.printf("\n10 Percent Salary Raised!! Yoohooooo!\n"); System.out.printf("\nNO:\t
NAME\t\t\tYEARLY SALARY\n" );
 System.out.printf("--\t ----\t\t\t----------\n");
System.out.printf("1:\t %s %s\t\t$%.2f\n", employee1.getFirstName(),employee1.getLastName(),
employee1.getSalary());
System.out.printf("2:\t %s %s\t\t$%.2f\n", employee2.getFirstName(),employee2.getLastName(),
employee2.getSalary());
}
}
```

8. **Create a class called Date that includes three pieces of information as instance variables—a month (type int), a day (type int) and a year (type int). Your class should have a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a set and a get method for each instance variable. Provide a method displayDate that displays the month, day and year separated by forward slashes(/). Write a test application named DateTest that demonstrates classDate's capabilities.**

```
//filename: Date.java
// Date class
public class Date {
        private int month;
         private int day;
         private int year;
public Date(int myMonth, int myDay, int myYear) // constructor
{
month = myMonth;
day = myDay;
year = myYear;
}
//setters and  getters
public void setMonthDate (int myMonth) {       month = myMonth; }
public int getMonthDate( )      {   return month;     }
public void setDayDate(int myDay  {    day = myDay;   }
public int getDayDate( )        {     return month;     }
public void setYearDate(int myYear) { year = myYear; }
public int getYearDate( ) {  return year; }

public void displayDate() // method to display date
{
System.out.printf("%d/%d/%d", month,day,year);
}
}
```

```
//filename: DateTest.java
// Date testing class with the main() method
import java.util.*;
public class DateTest {
public static void main(String[] args) {
Scanner input = new Scanner(System.in);
Date myDate = new Date(25, 03, 2020);
System.out.println("Enter The Month");
int myMonth = input.nextInt( );
myDate.setMonthDate(myMonth);
System.out.println("Enter the Date");
 int myDay = input.nextInt(); myDate.setDayDate(myDay);
System.out.println("Enter the Year");
int myYear = input.nextInt(); myDate.setYearDate(myYear);
myDate.displayDate();
}
}
```

9. **Create class SavingsAccount. Use a static variable annualInterestRate to store the annual interest rate for all account holders. Each object of the class contains a private instance variable savingsBalance indicating the amount the saver currently has on deposit. Provide method calculateMonthlyInterest to calculate the month interest by multiplying the savingsBalance by annualInterestRate divided by 12 this interest should be added to savingsBalance. Provide a static method modifyInterestRate that sets the annualInterestRate to a new value. Write a program to test class SavingsAccount. Instantiate two savingsAccount objects, saver1 and saver2, with balances of KES 200,000.00 and KES 300,000.00, respectively. Set annualInterestRate to 4%, then calculate the monthly interest and print the new balances for both savers. Then set the annualInterestRate to 5%, calculate the next month's interest and print the new balances for both savers.**

```
//filename: SavingAccount.java
// SavingAccount class
public class SavingsAccount {
      public static double annualInterestRate;
      private double savingsBalance;
public SavingsAccount( )
{
       annualInterestRate =0.0;
       savingsBalance = 0.0;
}
public SavingsAccount(double intRate, double savBal)
{
        annualInterestRate = intRate;
        savingsBalance = savBal;
}
public double calculateMonthlyInterest( ) {
double intRate = (savingsBalance * annualInterestRate/12);
 savingsBalance = savingsBalance + intRate;
          return  intRate;
}
public static void modifyInterestRate(double newInteresRate)
{
       annualInterestRate = newInteresRate;
}
```

```
public void setSavingsBalance(double newBal)
{
        savingsBalance = newBal;
}
public double getSavingsBalance( )
{
       return savingsBalance;
}
public double getAnnualInterestRate( )
{
       return annualInterestRate;
}
}
```

```
//filename: SavingsAccountTest.java
// SavingsAccount testing class with the main() method
public class SavingsAccountTest {
public static void main(String[] args) {
     SavingsAccount saver1 = new SavingsAccount( );
      SavingsAccount saver2 = new SavingsAccount( );
            saver1.setSavingsBalance(200000.00);
            saver2.setSavingsBalance(300000.00);
SavingsAccount.modifyInterestRate(0.04);
            saver1.calculateMonthlyInterest( );
             saver2.calculateMonthlyInterest( );
System.out.printf("New Balance for Saver1=%f\n",saver1.getSavingsBalance());
System.out.printf("New Balance for Saver2=%f\n",saver2.getSavingsBalance( ));
 SavingsAccount.modifyInterestRate(0.05);
          saver1.calculateMonthlyInterest( );
          saver2.calculateMonthlyInterest( );
System.out.printf("New Balance for Saver1=%f\n",saver1.getSavingsBalance( ));
System.out.printf("New Balance for Saver2=%f\n",saver2.getSavingsBalance( ));
}
```

```
}
```

10. **Create a class called Book to represent a book. A Book should include four pieces of information as instance variables-a book name, an ISBN number, an author name and a publisher. Your class should have a constructor that initializes the four instance variables. Provide a mutator method and accessor method (query method) for each instance variable. In addition, provide a method named getBookInfo that returns the description of the book as a String (the description should include all the information about the book). You should use this keyword in member methods and constructor. Write a test application named BookTest to create an array of object for 30 elements for class Book to demonstrate the class Book's capabilities.**

```
//filename: Book.java
// Book class
public class Book {
private String Name;
private String ISBN;
private String Author;
private String Publisher;

public Book( )     // Default Constructor
{
Name = "NULL";
ISBN = "NULL";
Author = "NULL";
Publisher = "NULL";
}
public Book(String name, String isbn, String author, String publisher) // Parameterized Constructor
{
Name = name;
ISBN = isbn;
Author = author;
Publisher = publisher;
}

public void setName(String Name)
{
        this.Name = Name;
}
public String getName( )
{
          return Name;
}
public void setISBN(String ISBN)
{
           this.ISBN = ISBN;
}
public String getISBN( )
{
      return ISBN;
}
public void setAuthor(String Author)
```

```
{
        this.Author = Author;
}
public String getAuthor( )
{
         return Author;
}
public void setPublisher (String Publisher)
 {
           this.Publisher = Publisher;
}
public String getPublisher( )
{
  return Publisher;
}
public void getBookInfo( )
 {
System.out.printf ("%s %s %s %s", Name, ISBN, Author, Publisher);
}
}
```

```
//filename: SavingsAccountTest.java
// SavingsAccount testing class with the main() method
public class BookTest {
public static void main(String[] args) {
Book test[] = new Book[30];
test[1] = new Book();
test[1].getBookInfo();
}
}
```

11.

a. Create a super class called Car. The Car class has the following fields and methods.

- **int speed;**
- **double regularPrice;**
- **String color;**
- **double getSalePrice();**

```
//filename: Car.java
//Car class
public class Car {
        private int speed;
        private double regularPrice;
        private String color;
   public Car (int Speed, double regularPrice, String color)
{
         this.speed = Speed;
         this.regularPrice = regularPrice;
         this.color = color;
}
public double getSalePrice()
{
 return regularPrice;
}
}
```

b. Create a sub class of Car class and name it as Truck. The Truck class has the following fields and methods.

- **int weight;**
- **doublegetSalePrice(); such that If weight>2000,10%**

discount.Otherwise,20%discount.

```
//filename: Truck.java
// Truck class, subclass of Car
public class Truck extends Car {
        private int weight;
public Truck (int Speed, double regularPrice,String color, int weight)
{
super(Speed,regularPrice,color);
this.weight = weight;
}

public double getSalePrice() {
if (weight > 2000)
        {
      return super.getSalePrice() - (0.1 * super.getSalePrice());
        }
else {
return super.getSalePrice();
        }
}
}
```

c. **Create a subclass of Car class and name it as Ford. The Ford class has the following fields and methods:**
   - **year (int);**
   - **manufacturerDiscount(int);**
   - **getSalePrice():double;//From the saleprice computed from Car class, subtract them anufacturerDiscount.**

```
//filename: Ford.java
// Ford class, subclass of Car
public class Ford extends Car {
        private int year;
        private int manufacturerDiscount;
public Ford (int Speed, double regularPrice, String color, int year, int
manufacturerDiscount)
{
super(Speed,regularPrice, color);
this.year = year;
this.manufacturerDiscount = manufacturerDiscount;
}
public double getSalePrice()
{
return (super.getSalePrice() - manufacturerDiscount);
        }
}
```

d. Create a subclass of Car class and name it as Sedan. The Sedan class has the following fields and methods.

- **int length;**
- **double getSalePrice();//If length>20 feet, 5% discount, Otherwise, 10%discount.**

```
//filename: Sedan.java
// Sedan class, subclass of Car
public class Sedan extends Car {
```

```
        private int length;
public Sedan (int Speed,double regularPrice,String color, int length)
{
      super (Speed,regularPrice,color);
       this.length = length;
}
public double getSalePrice( )
{
 if (length > 20)
       {
return super.getSalePrice() - (0.05 * super.getSalePrice());
}
else {
return super.getSalePrice() - (0.1 * super.getSalePrice());
}
}
}
```

e. **Create MyOwnAutoShop class which contains the main() method. Perform the following within the main() method.**
   - **Create an instance of Sedan class and initialize all the fields with appropriate values. Use super (...) method in the constructor for initializing the fields of the superclass.**
   - **Create two instances of the Ford class and initialize all the fields with appropriate values. Use super (...) method in the constructor for initializing the fields of the super class.**
   - **Create an instance of Car class and initialize all the fields with appropriate values. Display the sale prices of all instance.**

```
//filename: MyOwnAutoShop.java
// Testing class with the main() method
public class MyOwnAutoShop {
(int Speed, double regularPrice, String color, int year, int manufacturerDiscount)
public static void main(String[] args) {
Sedan mySedan = new Sedan(160, 20000, "Red", 10);
 Ford myFord1 = new Ford (156,4452.0,"Black",2005, 10);
Ford myFord2 = new Ford (155,5000.0,"Pink",1998, 5);
Car myCar - new Car (555, 56856.0, "Red");
System.out.printf("MySedan Price %.2f", mySedan.getSalePrice());
System.out.printf("MyFord1 Price %.2f", myFord1.getSalePrice());
System.out.printf("MyFord2 Price %.2f", myFord2.getSalePrice());
System.out.printf("MyCar Price %.2f", myCar.getSalePrice());
}
}
```