AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 4: Object-Oriented Design

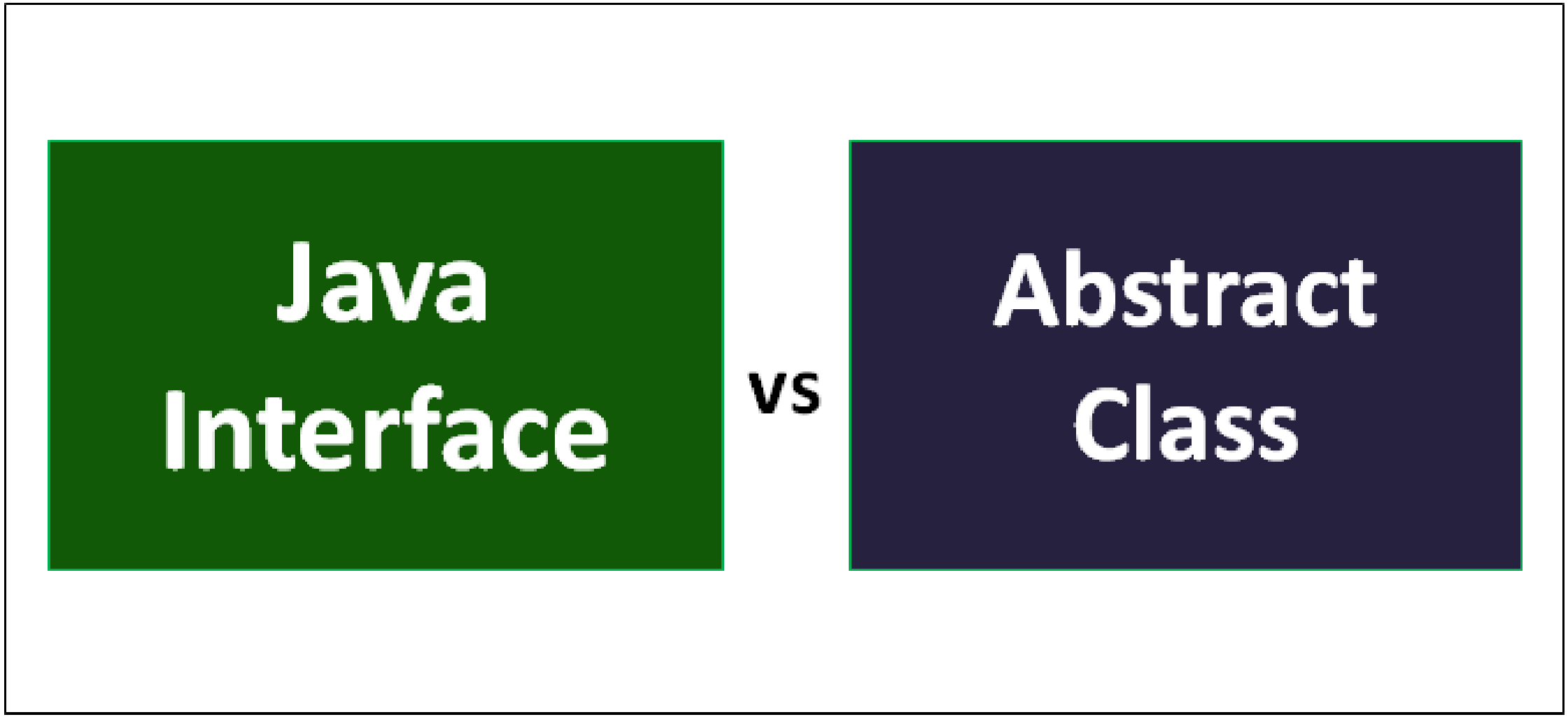
WEEK 8: CHAPTER 13 ABSTRACT CLASS AND INTERFACE

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Interface and Abstract Classes are special and reduced classes. (Also for enum, Junit classes)
- Abstract Classes
- Interfaces
- Clonable, Comparable, and Iterable Interfaces
- Design Patterns


Java
Interface
vs
Abstract
Class

Abstract Classes and Interfaces Templates and Shell of Data

Yes data fields No instantiation.

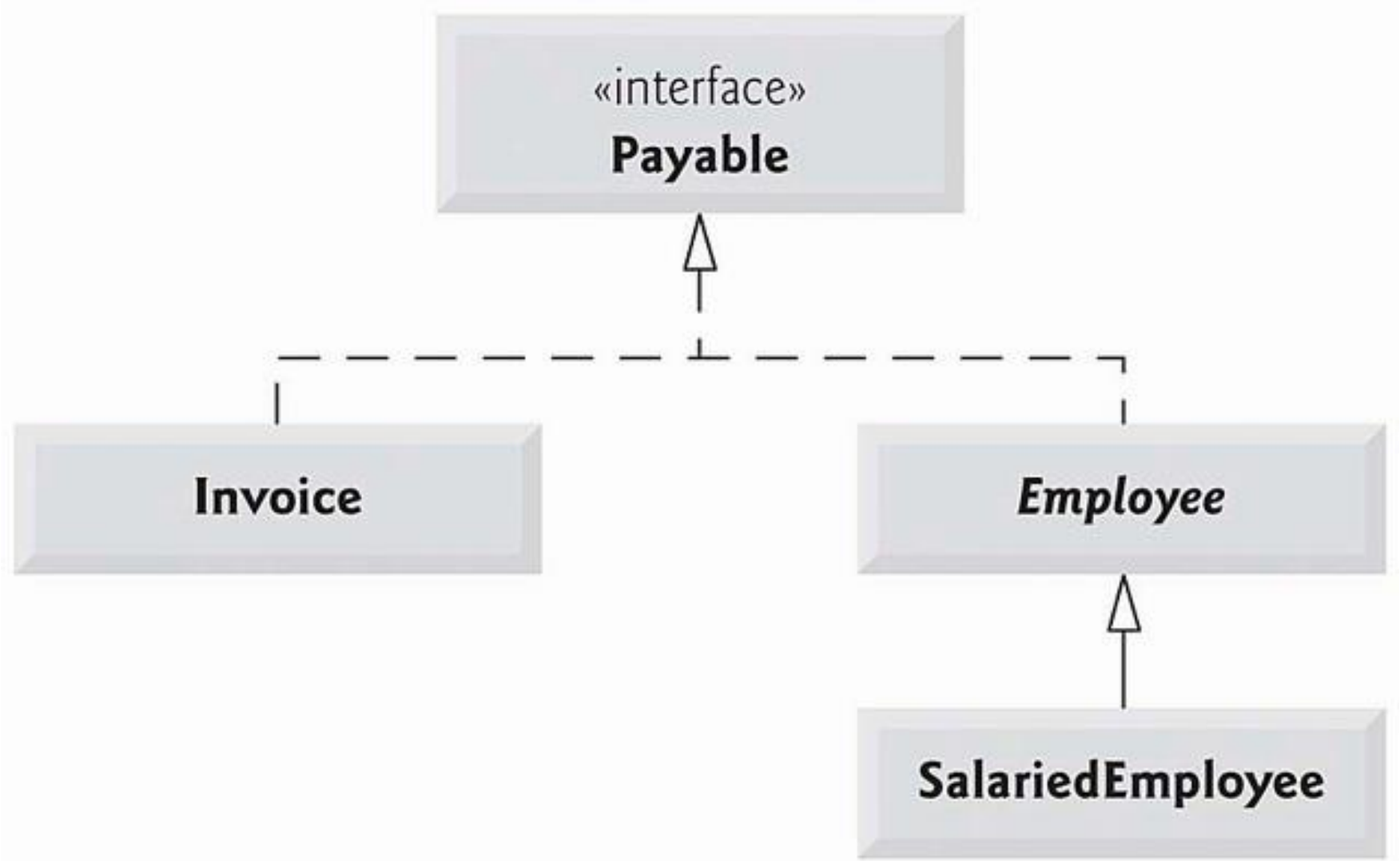
Abstract Class is CORE

Interface is Outer Skin

No data fields No instantiation.

OOPs interface vs abstract class

Interface	Abstract class
Interface support multiple implementations.	Abstract class does not support multiple inheritance.
Interface does not contain Data Member	Abstract class contains Data Member
Interface does not contain Constructors	Abstract class contains Constructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static


«interface»
Payable
Invoice
Employee
SalariedEmployee

Abstract Classes

LECTURE 1

Abstract Class

Pointer Yes, no objects, with data/methods (Conceptual Class)

A class can be declared with the abstract qualifier, e.g.:

public abstract class SuperClass

When this is added, it means that **no object** can be instantiated from this class and so must have subclasses for it to be useful.

Experiment with abstract

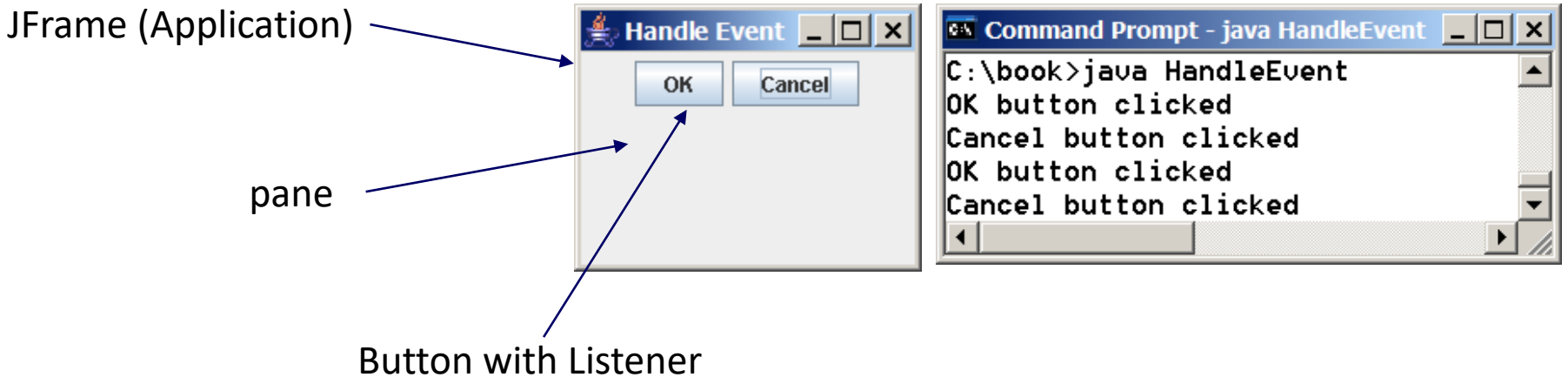
Using the classes from the basic package above:

1. Edit SuperClass, changing the declaration line to the following and save the changes:public **abstract** class SuperClass { When you save the changes, you'll see that the Driver class now has an error.
2. Edit Driver and observe NetBeans' response. Fix it by commenting out the flagged line:**//**new SuperClass(), Re-run the driver program observe the only change is reflected by the missing object.
3. Reset SuperClass and Driver back to their original states.

Note that the top member function in SuperClass is unaffected by making the class abstract. An abstract class is similar to an interface in that it must be extended to be used, but unlike an interface in that it usually **does** have functionality whereas an interface has no functionality, only prototypes.

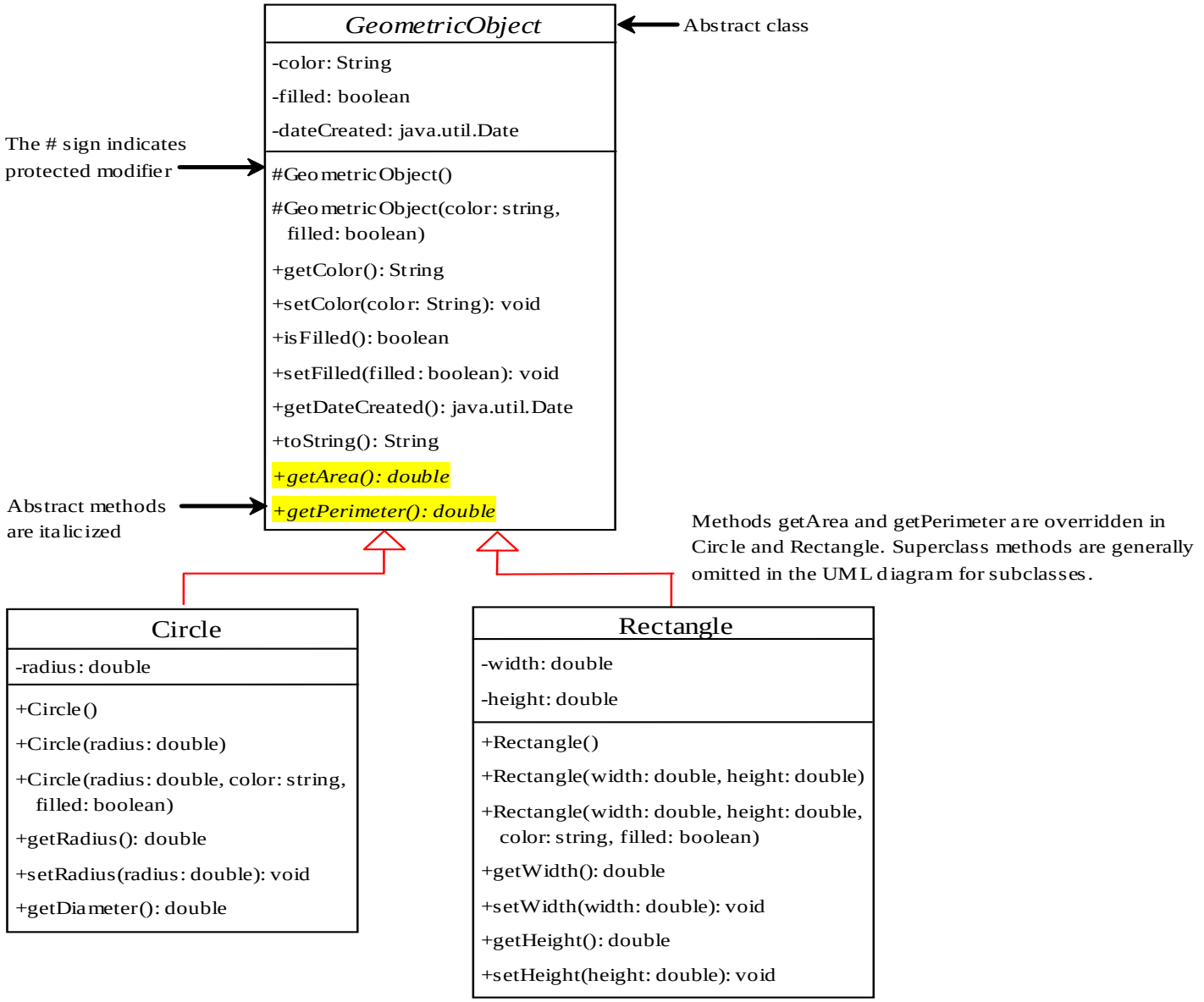
Motivations

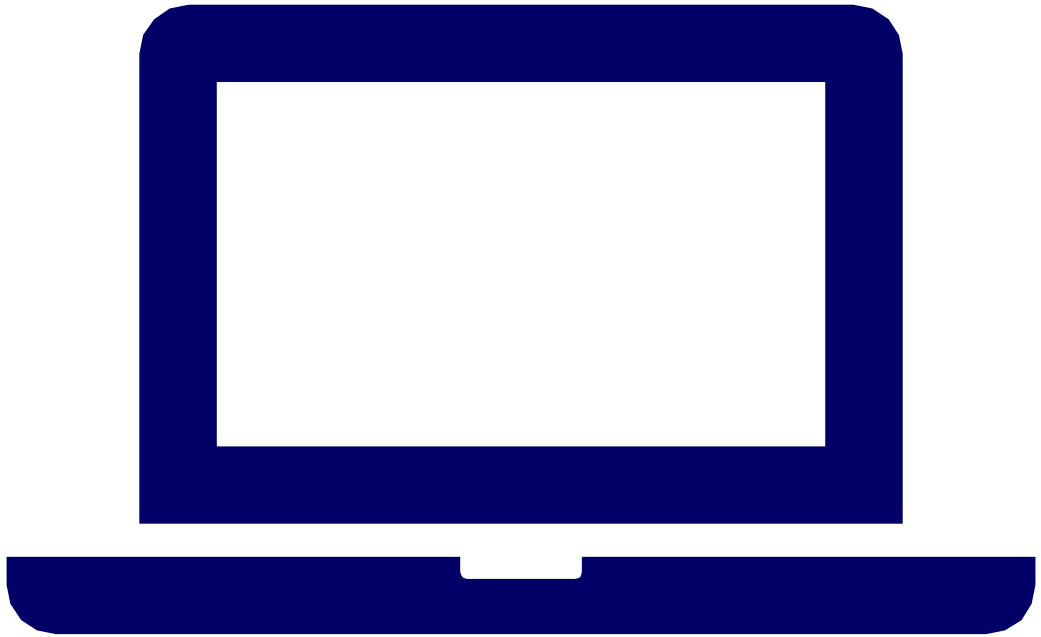
You learned how to write simple programs to display GUI components. Can you write the code to respond to user actions such as clicking a button?



Demo Program: HandleEvent.java

Abstract Classes and Abstract Methods





Demonstration Program

GEOMETRICOBJECT.JAVA

CIRCLE.JAVA

RECTANGLE.JAVA

TESTGEOMETRICOBJECT.JAVA

Abstract method in abstract class

- An **abstract method** cannot be contained in a **nonabstract** class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a nonabstract subclass extended from an abstract class, **all the abstract methods must be implemented**, even if they are not used in the subclass.

- **abstract** Methods undefined method (like headers of methods)

Object cannot be created from abstract class

- An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

Abstract class without abstract method

A class that contains abstract methods must be abstract. However, it is **possible** to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the **new** operator. This class is used as a base class for defining a new subclass.

(Template Class of Classes)

Superclass of Abstract Class May Be Concrete

- A subclass can be abstract even if its superclass is concrete. For example, the **Object** class is concrete, but its subclasses, such as **GeometricObject**, may be abstract.

Concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

(Disable concrete methods to set new definition in subclasses.)

Abstract Class as Type

You cannot create an instance from an abstract class using the **new** operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of **GeometricObject** type, is correct.

```
GeometricObject[] geo=new GeometricObject[10];
```

Final classes

Declaring a class final means that it cannot be extended; it is effectively the exact opposite of abstract. **Final classes** are the "**true leaves**" in an inheritance tree diagram. As a simple experiment with our basic example, add final onto the declaration line of SubClass1:

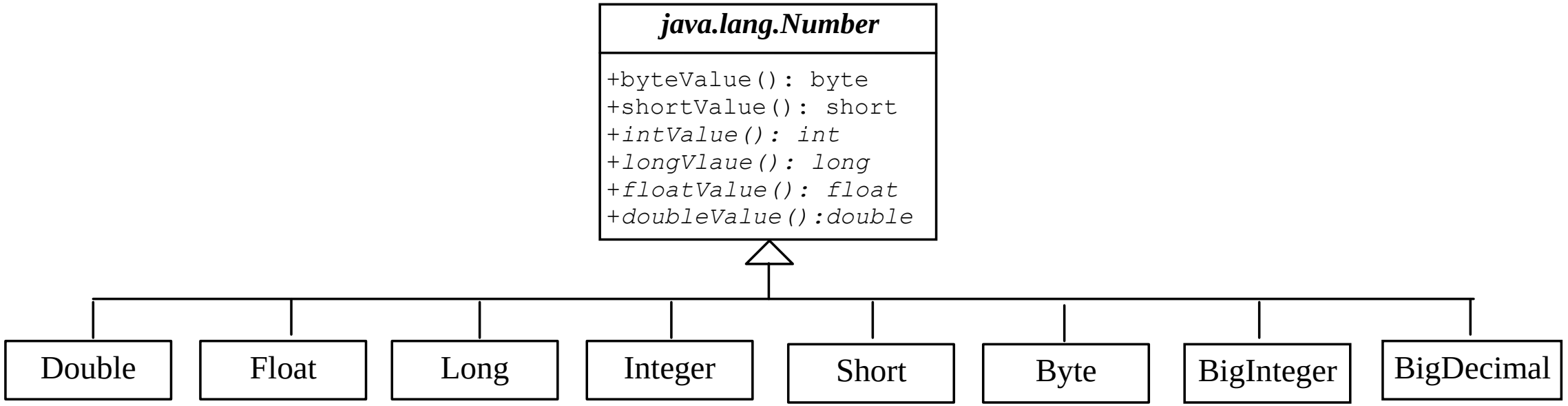
```
public final class SubClass1 {
```

Save the changes and observe that the class declaration in the file SubSubClass1 is now flagged with the obvious error message. Again, undo the change in SubClass1 to fix it.

Abstract Number Class

LECTURE 1

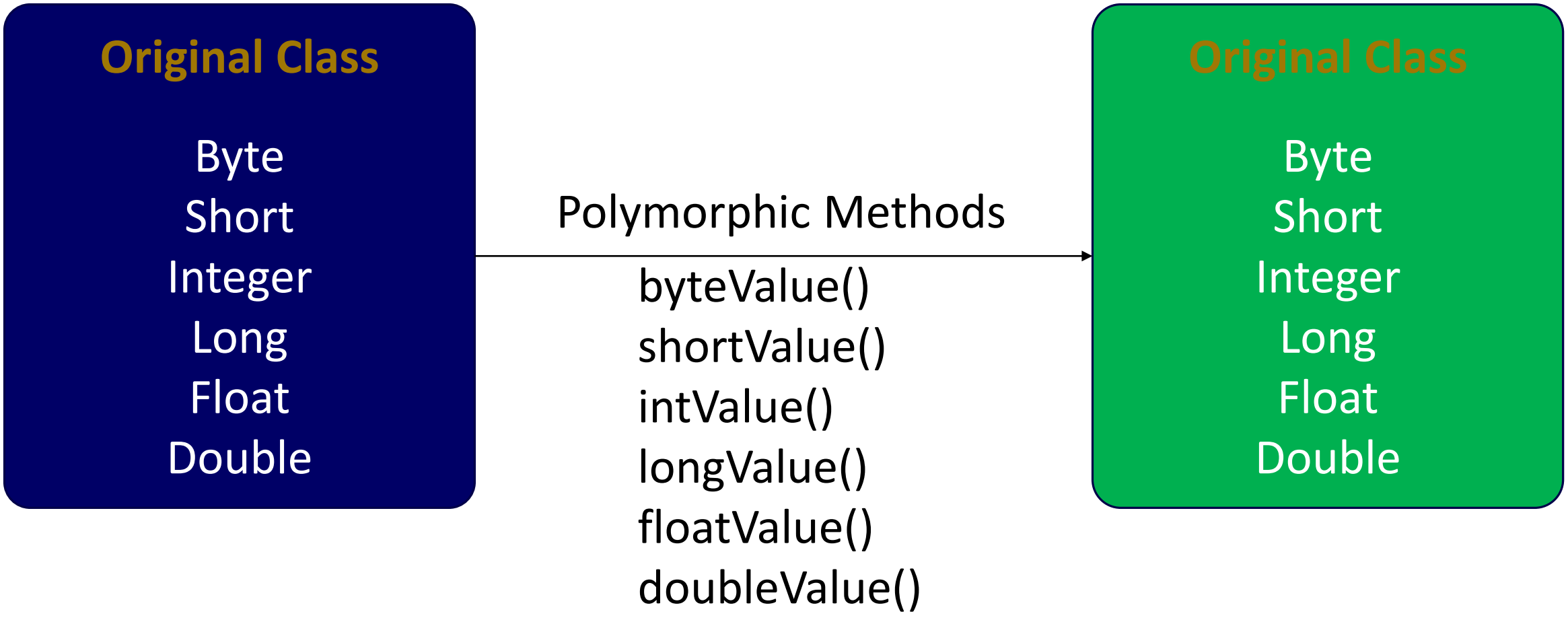
Case Study: the Abstract Number Class

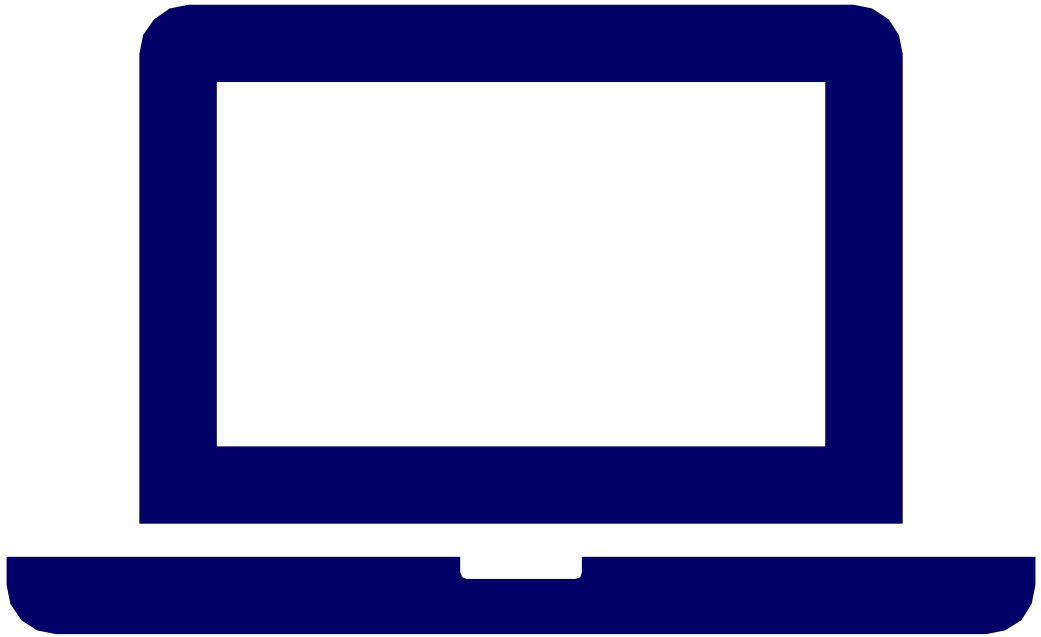


Abstract Class
Conceptual Class which allows polymorphism

- Object, Number, Stuff, Goods, Arrangement. These terms are abstract and conceptual.
- You know it is a number but not which type of number.
- You know it is an arrangement, but you don’t know whether it is a full-time hire or part-time contract.
- Abstract class allows programmer to design common data fields and methods to be shared among the concrete classes.

Conversion Among Number Sub-Classes using Polymorphic Methods





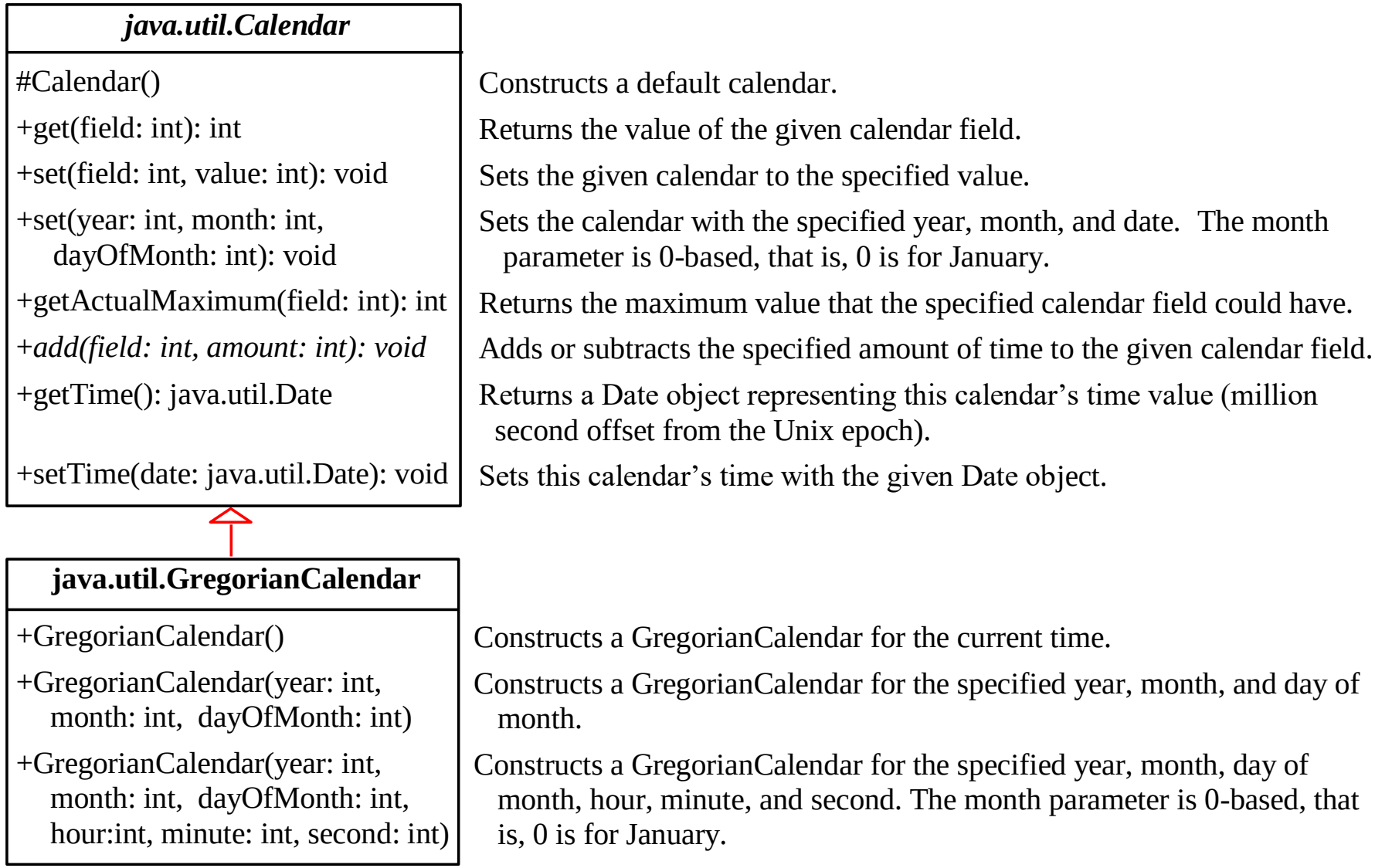
Demonstration Program

LARGENUMBERS.JAVA

Calendar and GregorianCalendar

LECTURE 1

The **Abstract** Calendar Class and Its GregorianCalendar subclass



Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based, that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a Date object representing this calendar's time value (million second offset from the Unix epoch).

Sets this calendar's time with the given Date object.

Constructs a GregorianCalendar for the current time.

Constructs a GregorianCalendar for the specified year, month, and day of month.

Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

Uses of Abstract Classes

(1) Class Methods (Static Methods)

(2) Inheritance by the Concrete Classes:

- Save implementation time. You do not need to implement some methods in different concrete sub-classes.

The Abstract Calendar Class and Its GregorianCalendar subclass

An instance of **java.util.Date** represents a specific instant in time with millisecond precision. **java.util.Calendar** is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a **Date** object. Subclasses of **Calendar** can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar. Currently, **java.util.GregorianCalendar** for the Gregorian calendar is supported in the Java API.

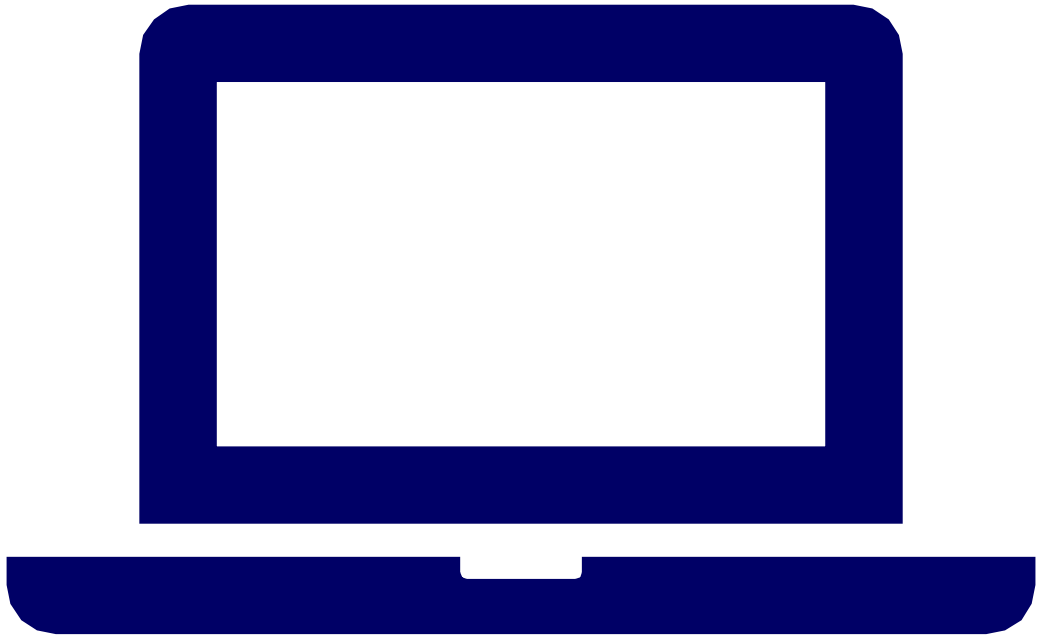
The GregorianCalendar Class

You can use **new GregorianCalendar()** to construct a default **GregorianCalendar** with the current time and use **new GregorianCalendar(year, month, date)** to construct a **GregorianCalendar** with the specified **year**, **month**, and **date**. The **month** parameter is 0-based, i.e., 0 is for January.

The get Method in Calendar Class

The get(int field) method defined in the Calendar class is useful to extract the date and time information from a Calendar object. The fields are defined as constants, as shown in the following.

Constant	Description
YEAR	The year of the calendar.
MONTH	The month of the calendar with 0 for January.
DATE	The day of the calendar.
HOUR	The hour of the calendar (12-hour notation).
HOUR_OF_DAY	The hour of the calendar (24-hour notation).
MINUTE	The minute of the calendar.
SECOND	The second of the calendar.
DAY_OF_WEEK	The day number within the week with 1 for Sunday.
DAY_OF_MONTH	Same as DATE.
DAY_OF_YEAR	The day number in the year with 1 for the first day of the year.
WEEK_OF_MONTH	The week number within the month.
WEEK_OF_YEAR	The week number within the year.
AM_PM	Indicator for AM or PM (0 for AM and 1 for PM).



Demonstration Program

TESTCALENDER.JAVA

Demo Program: absfinal package

LECTURE 1

Abstract Class and Final

Abstract Class: use to utilize polymorphism only. (No instantiation)

Final Variable: no update of value

Final Method: no overriding

Final Class: no subclass

Demo Program:
absfinal package in Inheritance Project

Go BlueJ!!!

SubClass\SuperClass foo()	Concrete	Abstract
Override	SubClass.foo()	SubClass.foo()
No Override	SuperClass.foo()	X
SubClass\SuperClass bar()	**final**	**Non-final**
Override	X	SubClass.bar()
No Override	SuperClass.bar()	SuperClass.bar()

Anonymous Class

LECTURE 1

Anonymous Class

An anonymous inner class is an inner class that is declared without using a class name at all – and that of course is why it's called an **anonymous class**. An anonymous inner class also has some pretty unusual syntax.

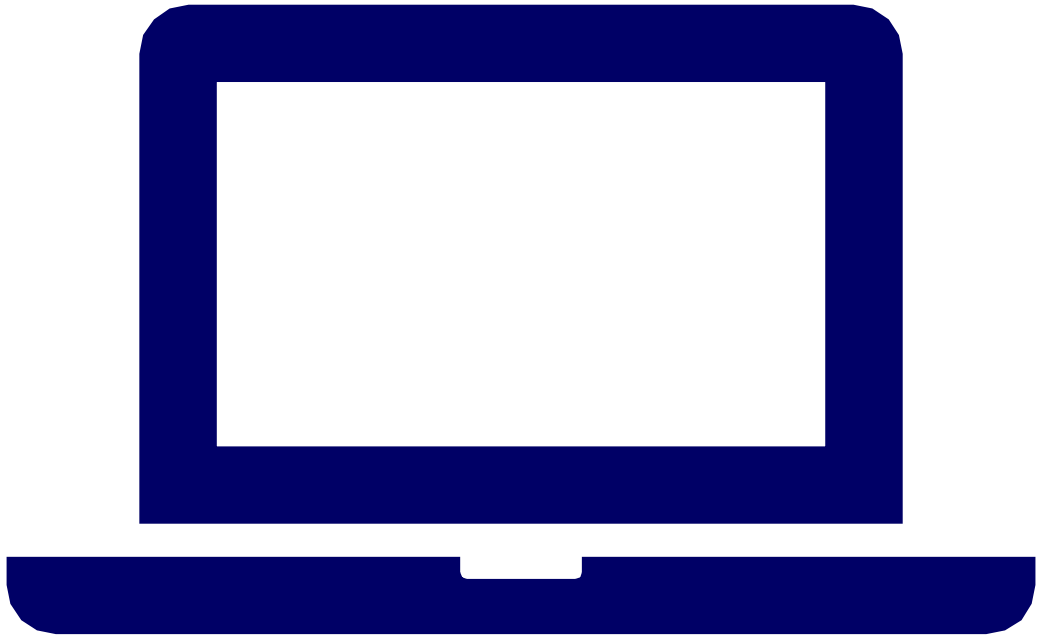
Anonymous class does not mean "No name". It really means in-line **object**. or embedded **object**. Just like anonymous array **new int[]{1, 2, 3};**

Anonymous inner class syntax in Java

```
/*Pay attention to the opening curly braces and the fact
that there's a semicolon at the very end, once the
anonymous class is created: */

ProgrammerInterview pInstance = new ProgrammerInterview() {
    //code here...
};
```

Anonymous inner classes and polymorphism

When using anonymous inner classes, polymorphism is actually at work as well. Taking another look at our example above, note that pInstance is actually a superclass reference type that refers to a subclass object. In plain English, that means pInstance is of type ProgrammerInterview (which is the superclass), but pInstance refers to a subclass (or child class) of the ProgrammerInterview class – and this is polymorphism at work. That subclass is the anonymous inner class with no name that is created inside the Website class.



Demonstration Program

ACTIONLISTENERTEST3.JAVA

Interfaces

LECTURE 1

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

An interface is a classlike construct that contains only **constants** and **abstract methods**. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify behavior for objects. For example, you can specify that the objects are **comparable**, **edible**, **cloneable** using appropriate interfaces.

No data fields and no concrete methods!

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
   constant declarations;
   method signatures;
}
```

Define an Interface

Example:

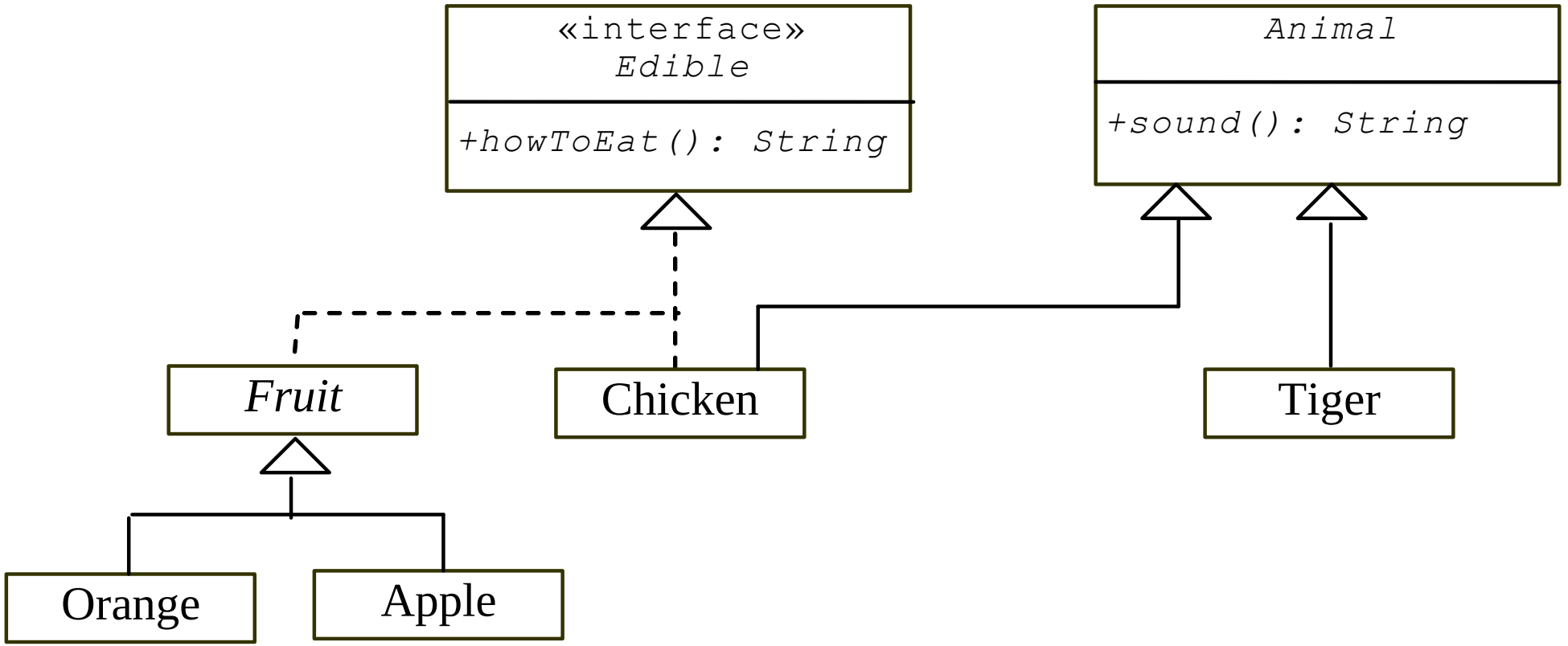
```
public interface Edible {
   /** Describe how to eat */
   public abstract String howToEat(); //to be overridden
}
```

Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the **new** operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Example

Objects of different hierarchical tree can share the same polymorphic methods



Example

Objects of different hierarchical tree can share the same polymorphic methods.

You can now use the **Edible** interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the **implements** keyword. For example, the classes **Chicken** and **Fruit** implement the **Edible** interface (See TestEdible).

Omitting Modifiers in Interfaces

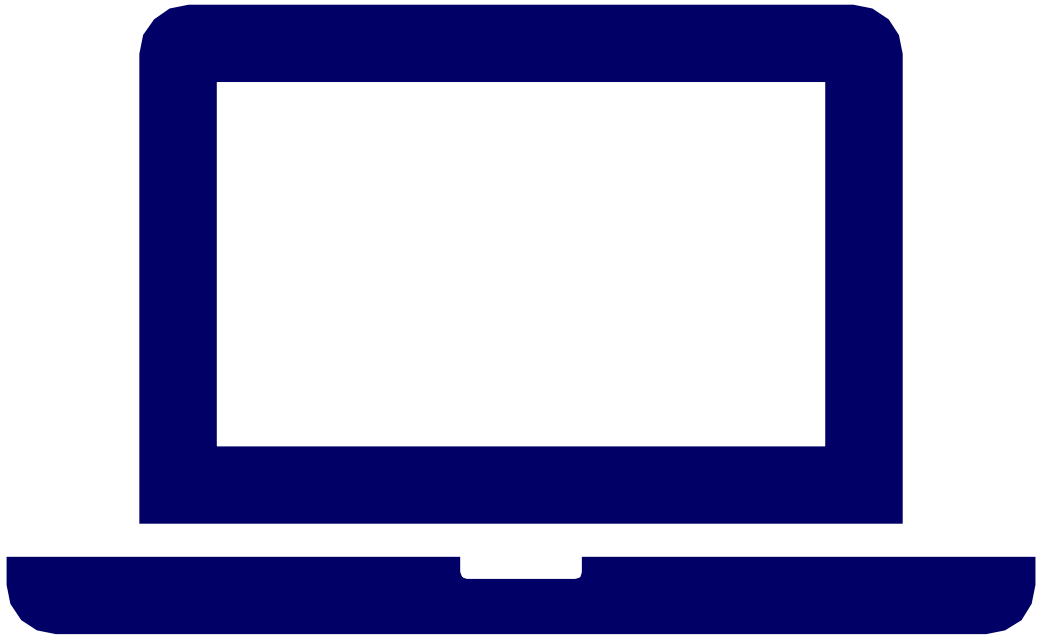
All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).



Demonstration Program

EDIBLE.JAVA TESTEDIBLE.JAVA

Comparable Interface

LECTURE 1

Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

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

Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

Example

```
1  System.out.println(new Integer(3).compareTo(new Integer(5)));
2  System.out.println("ABC".compareTo("ABE"));
3  java.util.Date date1 = new java.util.Date(2013, 1, 1);
4  java.util.Date date2 = new java.util.Date(2012, 1, 1);
5  System.out.println(date1.compareTo(date2));
```

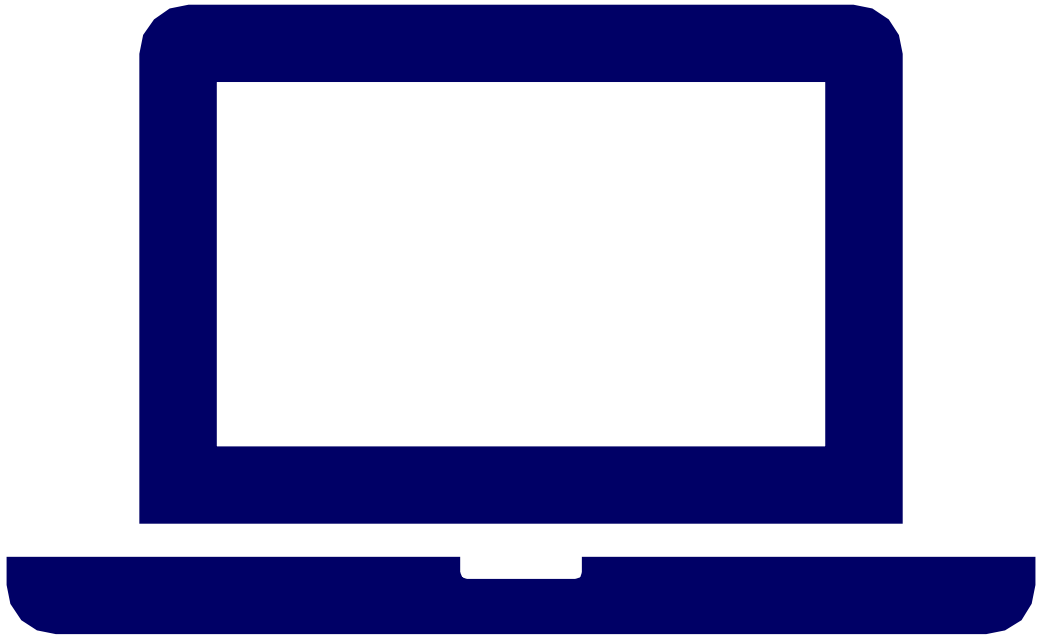
Generic max Method

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are **true**.

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

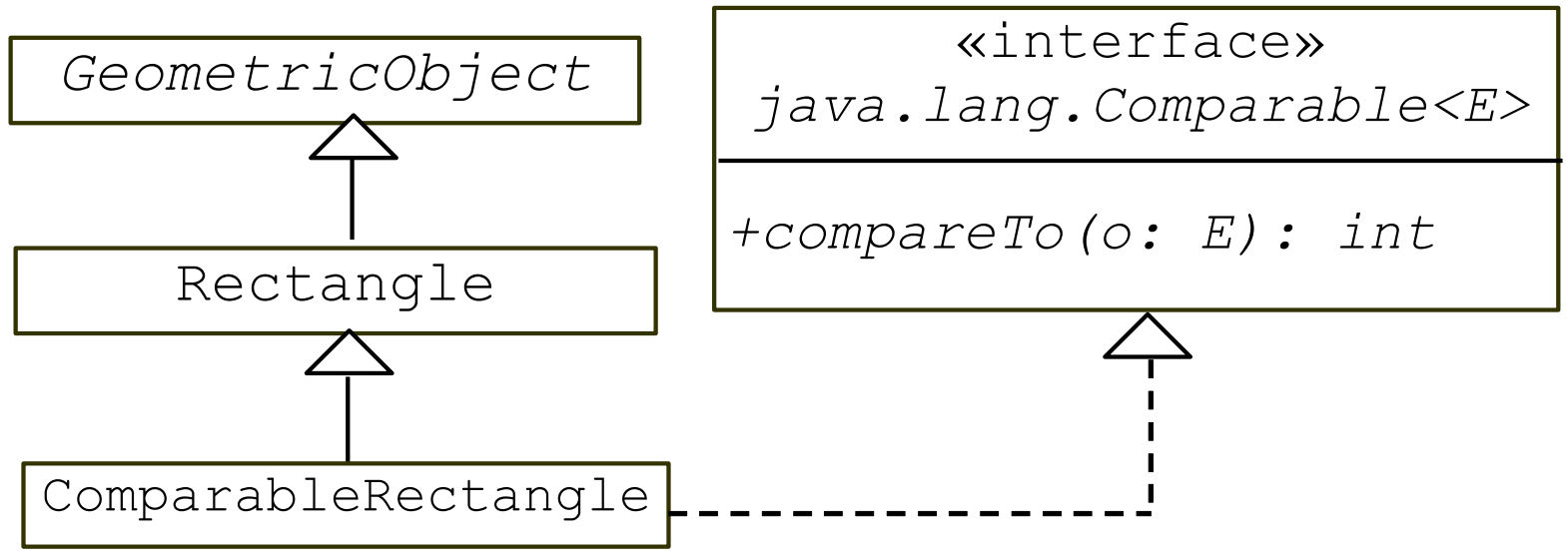


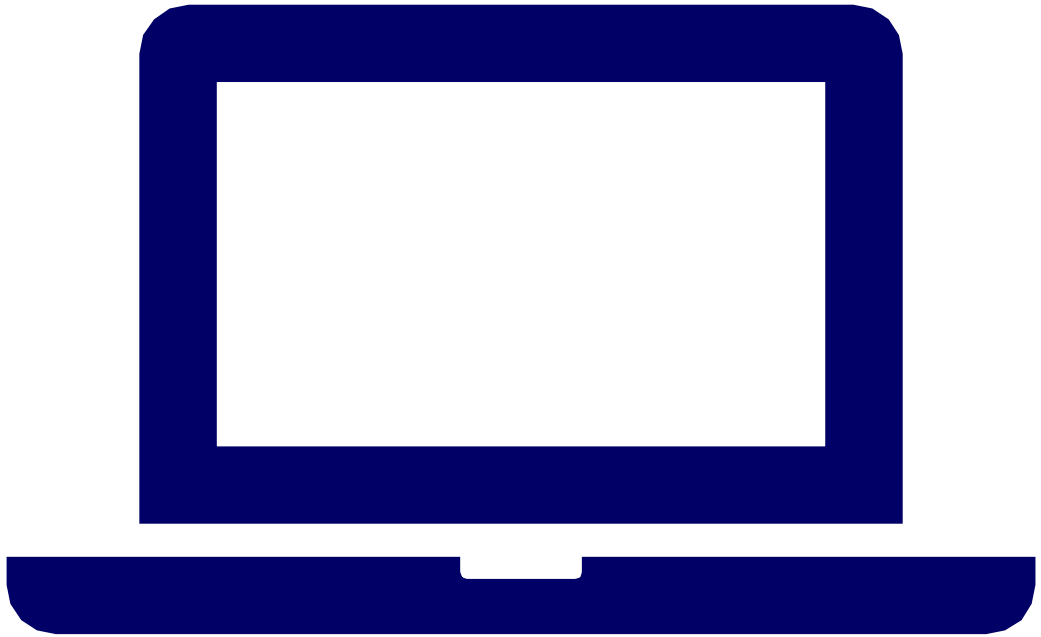
Demonstration Program

SORTCOMPARABLEOBJECTS.JAVA

Defining Classes to Implement Comparable

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.





Demonstration Program

COMPARABLERECTANGLE.JAVA,
SORTRECTANGLES.JAVA

Cloneable Interface

LECTURE 1

The Cloneable Interfaces

Marker Interface: An empty interface.

A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the **Cloneable** interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

```
package java.lang;
public interface Cloneable {
}
```

Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
  (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
  calendar.equals(calendarCopy));
```

Examples

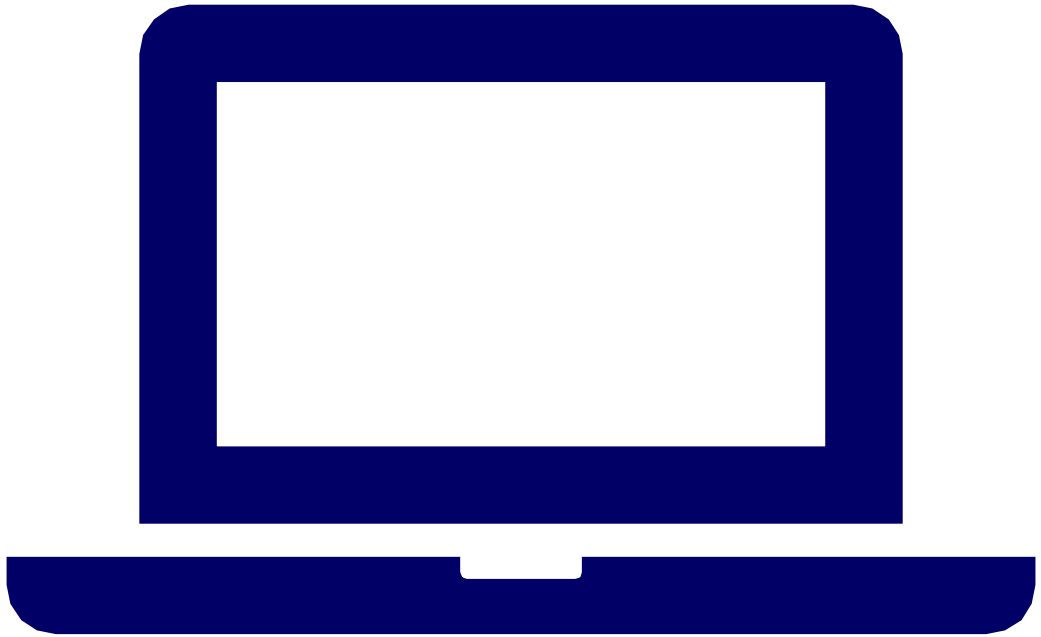
displays

`calendar == calendarCopy` is false

`calendar.equals(calendarCopy)` is true

Implementing Cloneable Interface

To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class. The following code defines a class named House that implements Cloneable and Comparable.



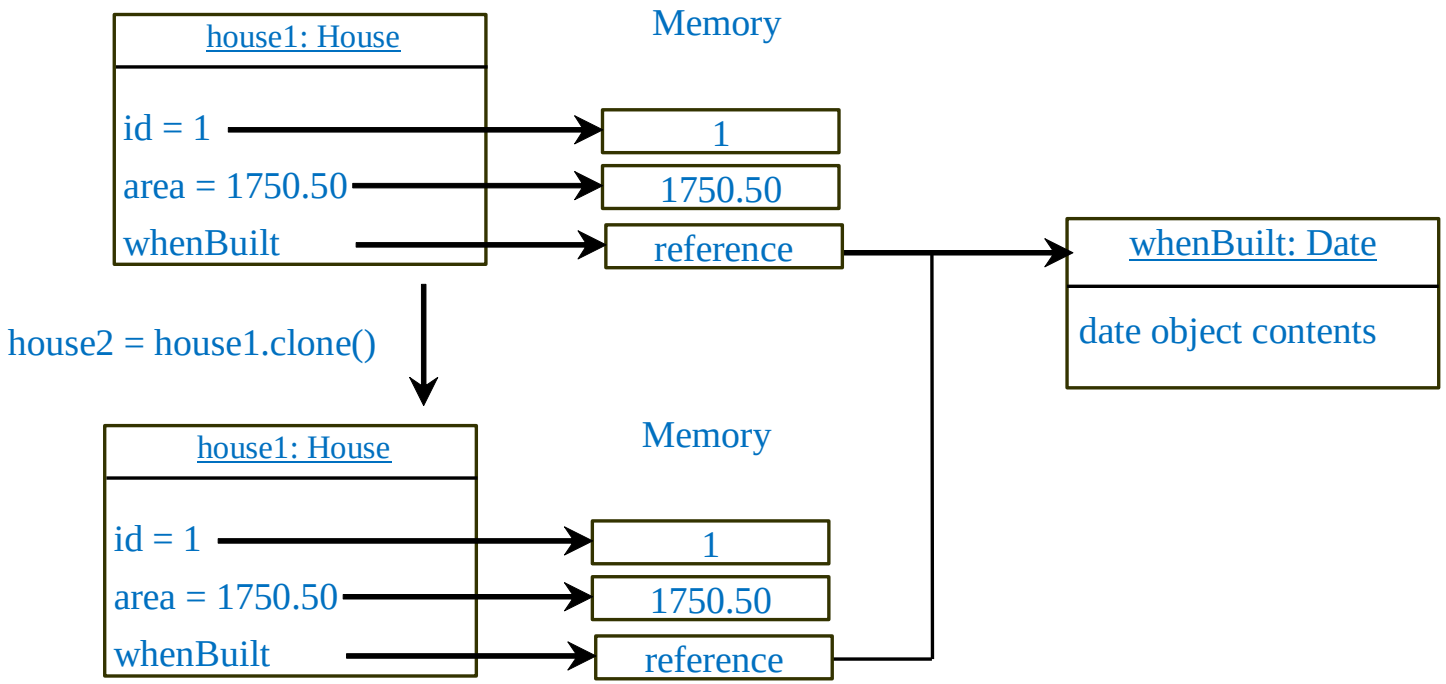
Demonstration Program

HOUSE.JAVA

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House) house1.clone();

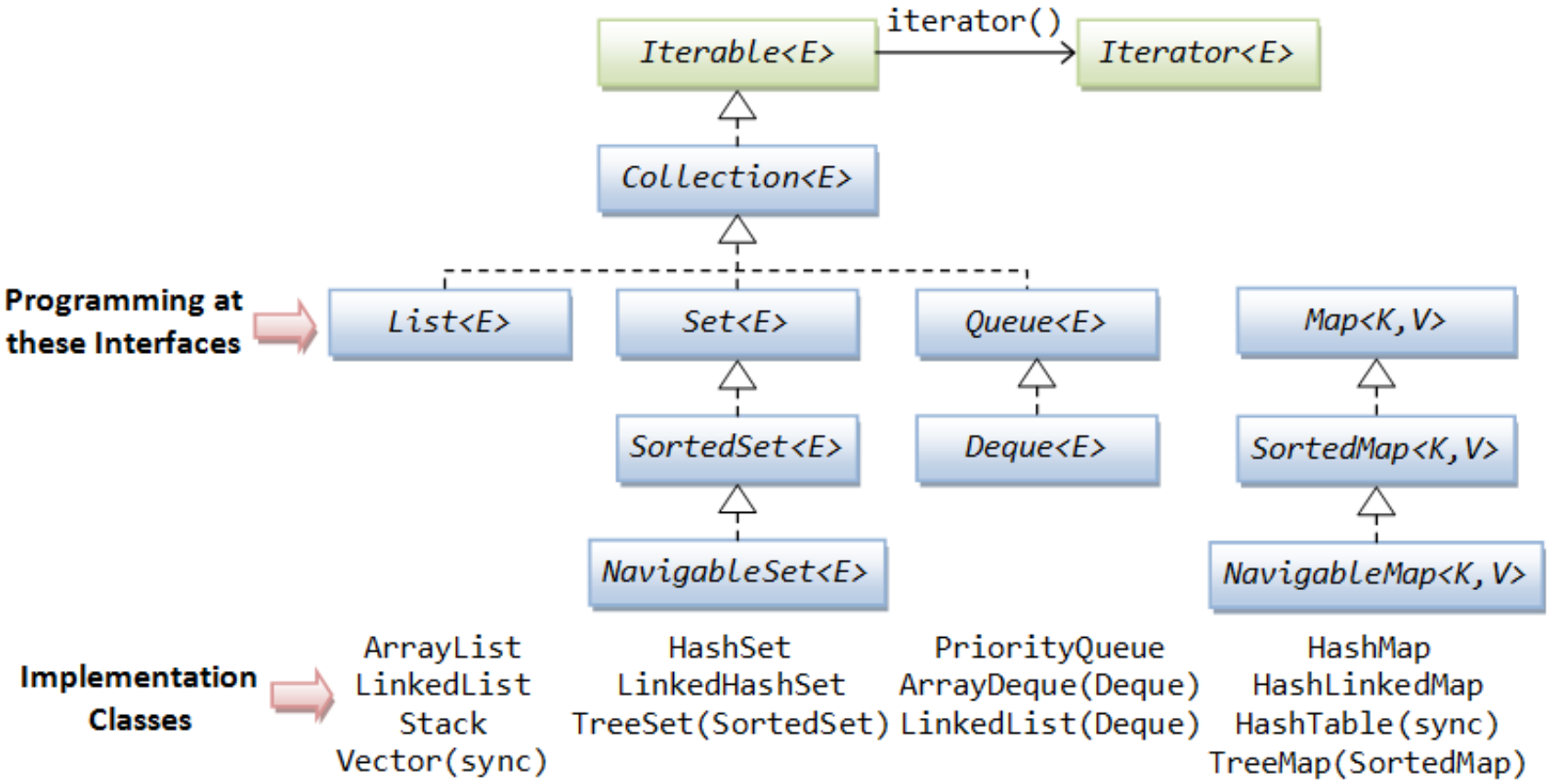


Iterable Interface (Non-AP Topic)

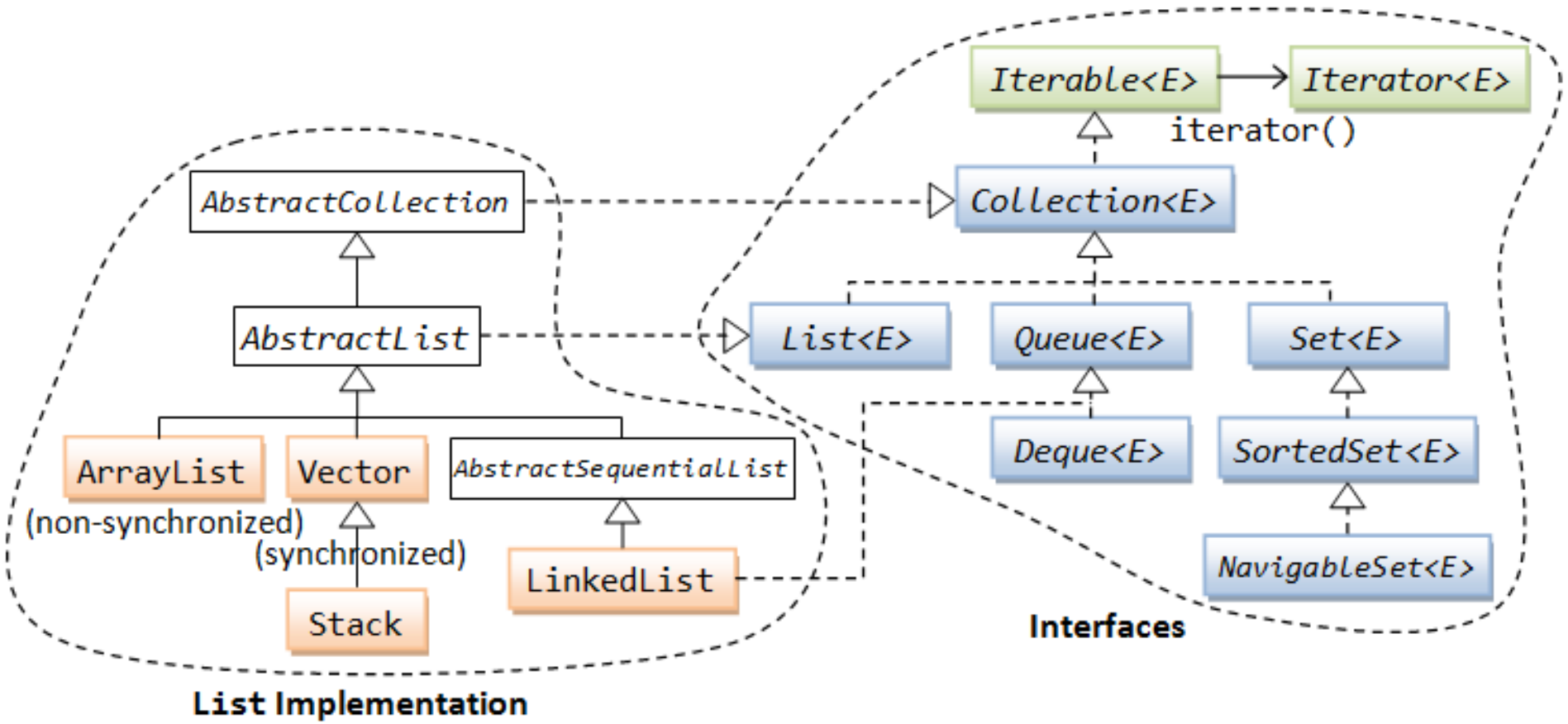
LECTURE 1

Interfaces Derived from Iterable Interface

Almost all data structures in Java are interable



Interface and Abstract Classes



Iterable Interface

Iterable is a feature all data structures under Collection have.

The **Iterable** interface (java.lang.Iterable) is one of the root interfaces of the Java collection classes. The Collection interface extends Iterable, so all subtypes of Collection also implement the **Iterable** interface.

A class that implements the Iterable can be used with the new for-loop. Here is such an example:

```
List list = new ArrayList();

for(Object o : list){
    //do something o;
}
```

iterator() method

iterator create a reference variable which can point to objects in a data structure

The Iterable interface has only one method:

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

How you implement this **Iterable** interface so that you can use it with the new for-loop, is explained in the text Implementing the **Iterable** Interface, in my Java Generics tutorial.

Using an Iterator

A Running Pointer

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class MainClass {
  public static void main(String[] a) {
    Collection c = new ArrayList();
    c.add("1");
    c.add("2");
    c.add("3");
    Iterator i = c.iterator();
    while (i.hasNext()) {
      System.out.println(i.next());
    }
  }
}
```

The Iterator interface has the following methods:

hasNext.

next.

remove.

Compared to an Enumeration, hasNext() is equivalent to hasMoreElements() and next() equates to nextElement().

Iterable interface: while loop and for loop

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class MainClass {
  public static void main(String[] args) {
    List list = Arrays.asList("A", "B", "C", "D");
     Iterator iterator = list.iterator();
     while (iterator.hasNext () ) {
         String element = (String) iterator.next ();
         System.out.println(element);
     }

  }
}
```

'for' statement for Iterable object in JDK 5 has been enhanced.

It can iterate over a Collection without the need to call the iterator method. The syntax is

for (Type identifier : expression) {

statement (s)

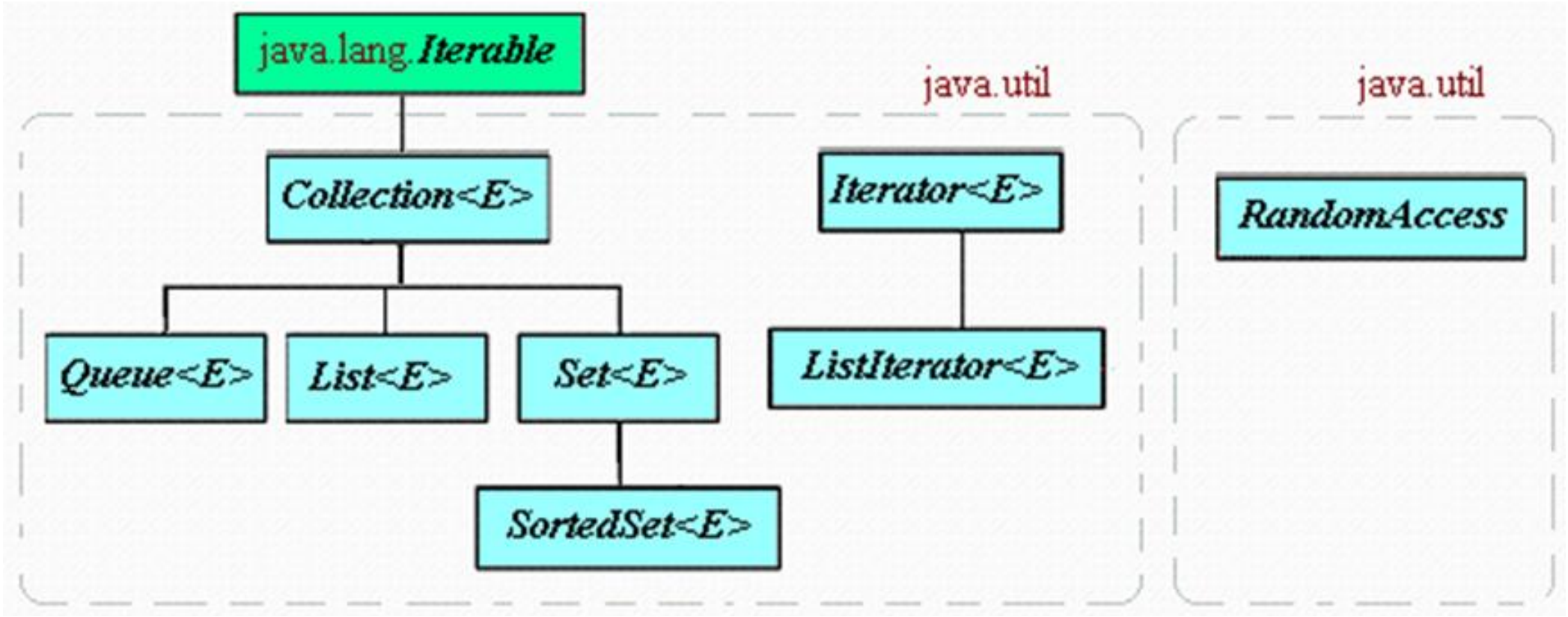
}

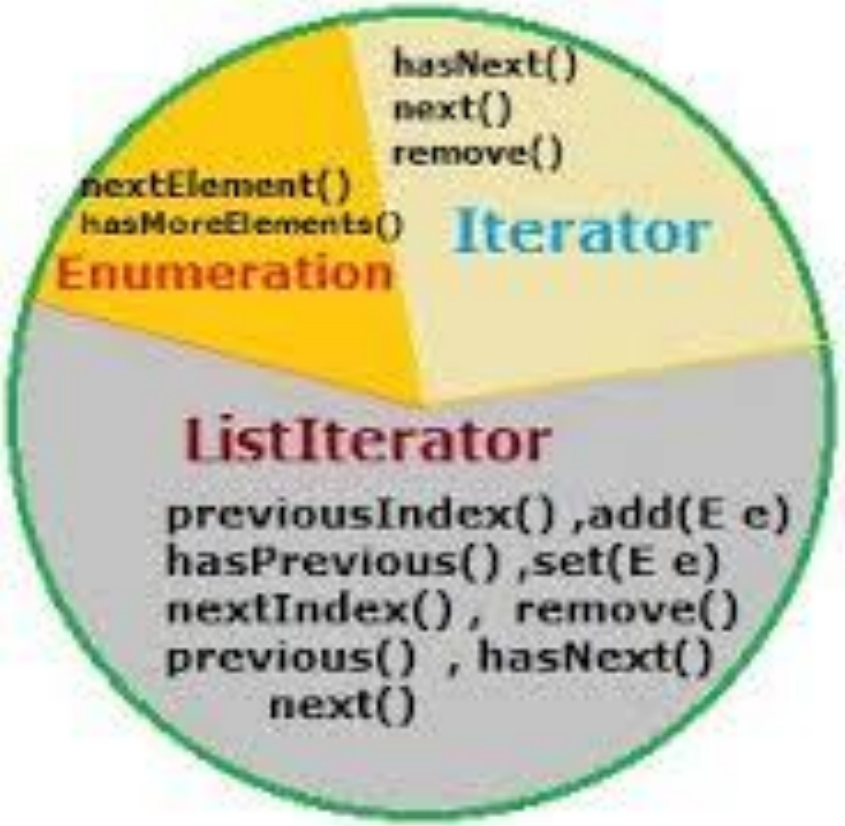
In which expression must be an Iterable. **(implements Iterable Interface)**

```
import java.util.Arrays;
import java.util.List;

public class MainClass {
  public static void main(String[] args) {
    List list = Arrays.asList("A", "B", "C", "D");

    for (Object object : list) {
      System.out.println(object);
    }

  }
}
```

Creating Iterable Objects: using a for-each for loop on an Iterable object

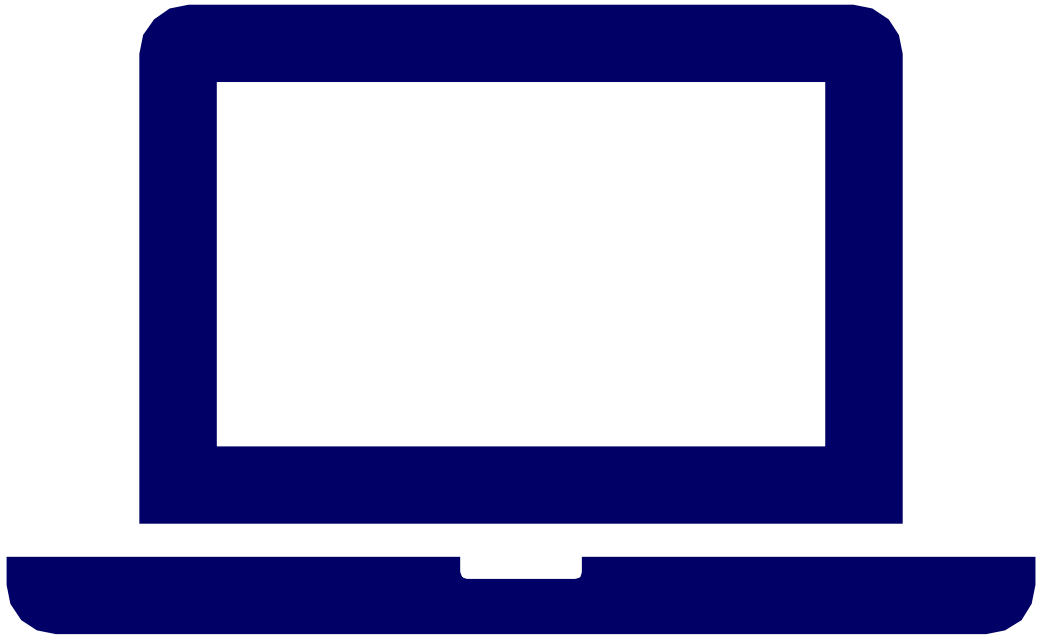
```
import java.util.Iterator;
import java.util.NoSuchElementException;

// This class supports iteration of the
// characters that comprise a string.
class IterableString implements Iterable<Character>, Iterator<Character> {
  private String str;

  private int count = 0;

  public IterableString(String s) {
    str = s;
  }
  // The next three methods implement Iterator.
  public boolean hasNext() {
    if (count < str.length()){
      return true;
    }
    return false;
  }
  public Character next() {
    if (count == str.length())
      throw new NoSuchElementException();

    count++;
    return str.charAt(count - 1);
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }

  // This method implements Iterable.
  public Iterator<Character> iterator() {
    return this;
  }
}

public class MainClass {
  public static void main(String args[]) {
    IterableString x = new IterableString("This is a test.");

    for (char ch : x){
      System.out.println(ch);
    }
  }
}
```


java.lang.Iterable
java.util
java.util
Collection<E>
Iterator<E>
RandomAccess
Queue<E>
List<E>
Set<E>
ListIterator<E>
SortedSet<E>


hasNext()
next()
remove()
nextElement()
hasMoreElements()
Enumeration
Iterator
ListIterator
previousIndex() ,add(E e)
hasPrevious() ,set(E e)
nextIndex() , remove()
previous() , hasNext()
next()

ListIterator Interface

Adds the specified object to the list.
Returns true if this list iterator has more elements when traversing backward.
Returns the index of the next element.
Returns the previous element in this list iterator.
Returns the index of the previous element.
Replaces the last element returned by the previous or next method with the specified element.



Demonstration Program

ITERABLE PACKAGE

Abstract Class Versus Interfaces

LECTURE 1

Interfaces V.S. Abstract Classes

A class can implement multiple interfaces, but it can only extend one superclass.

Interfaces vs. Abstract Classes

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

class implements interface
interface extends interface

Java allows only *single inheritance* for class extension but allows *multiple extensions* for interfaces. For example,

```
public class NewClass extends BaseClass
    implements Interface1, ..., InterfaceN {
  ...
}
```

An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a *subinterface*. For example, `NewInterface` in the following code is a subinterface of `Interface1`, . . . , and `InterfaceN`.

```
public interface NewInterface extends Interface1, ... , InterfaceN {
  // constants and abstract methods
}
```

Interfaces vs. Abstract Classes, cont.

- All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type.
- A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.

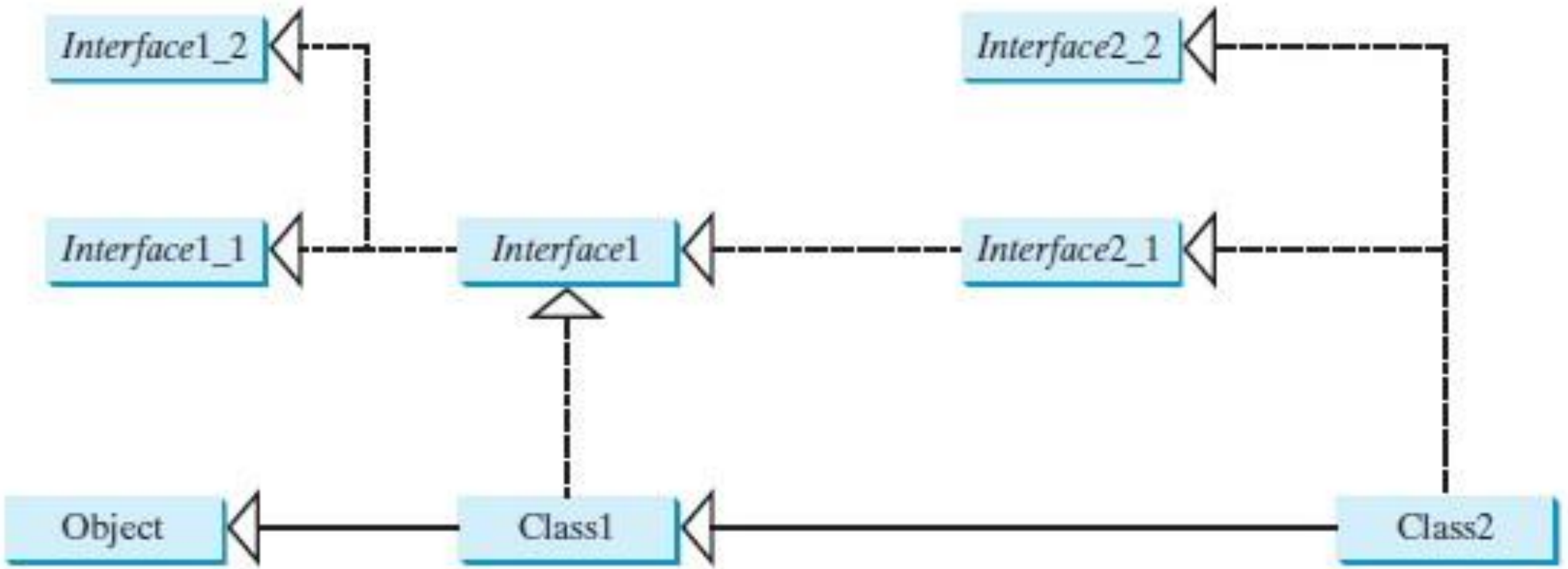


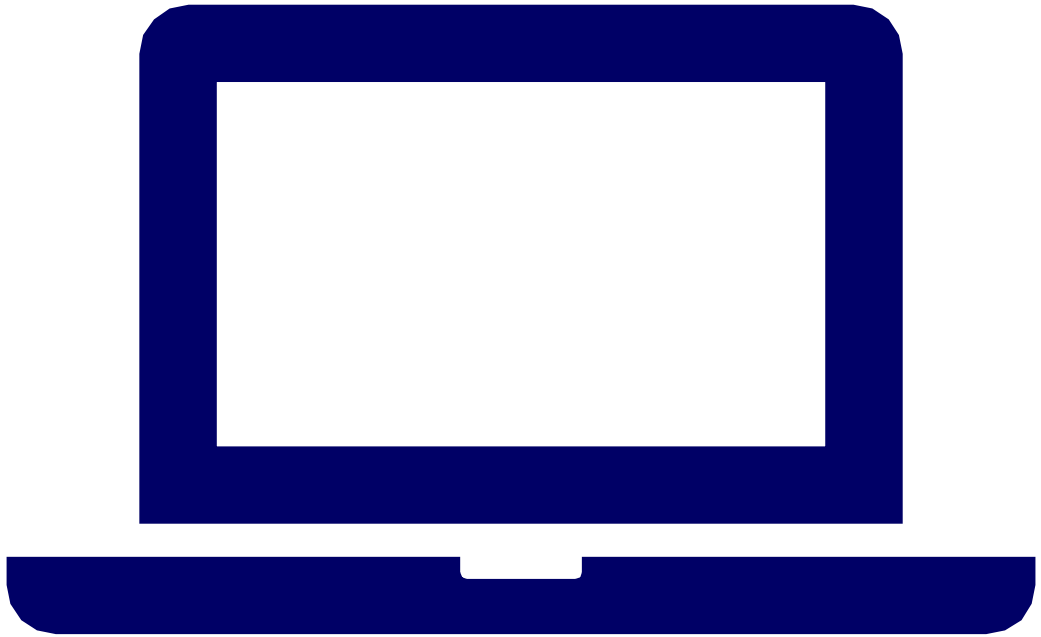
FIGURE 13.7 `Class1` implements `Interface1`; `Interface1` extends `Interface1_1` and `Interface1_2`. `Class2` extends `Class1` and implements `Interface2_1` and `Interface2_2`.

Caution: conflict interfaces

- In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). **This type of errors will be detected by the compiler.**

Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class? **In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.** For example, a staff member is a person. So their relationship should be modeled using class inheritance. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. **A weak is-a relationship can be modeled using interfaces.** For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface. **(is-a, capability)**

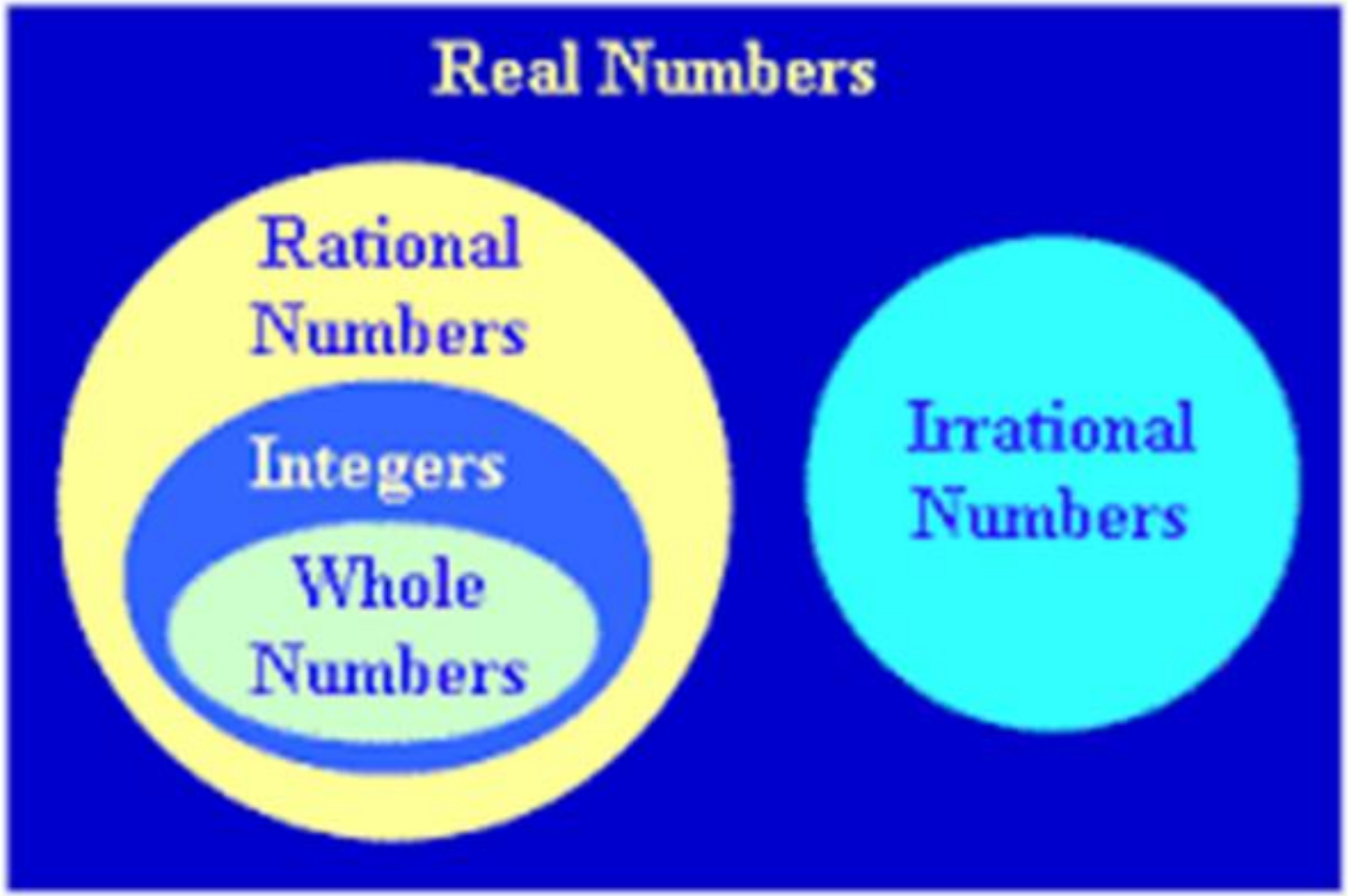


Demonstration Program

TESTANIMAL.JAVA TESTSTUFF.JAVA

Demo Program: The Rational Class

LECTURE 1


Real Numbers
Rational Numbers
Integers
Whole Numbers
Irrational Numbers

Rational Number

In mathematics, a rational number is any number that can be expressed as the quotient or fraction p/q of two integers, a numerator p and a non-zero denominator q. Since q may be equal to 1, every integer is a rational number.

1/1	1/2	1/3	1/4	1/5	1/6	1/7	1/8	1/9
2/1	2/2	2/3	2/4	2/5	2/6	2/7	2/8	2/9
3/1	3/2	3/3	3/4	3/5	3/6	3/7	3/8	3/9
4/1	4/2	4/3	4/4	4/5	4/6	4/7	4/8	4/9
5/1	5/2	5/3	5/4	5/5	5/6	5/7	5/8	5/9
6/1	6/2	6/3	6/4	6/5	6/6	6/7	6/8	6/9
7/1	7/2	7/3	7/4	7/5	7/6	7/7	7/8	7/9
8/1	8/2	8/3	8/4	8/5	8/6	8/7	8/8	8/9
9/1	9/2	9/3	9/4	9/5	9/6	9/7	9/8	9/9

a/b => (a, b)
Can be viewed as fractional number or vector.

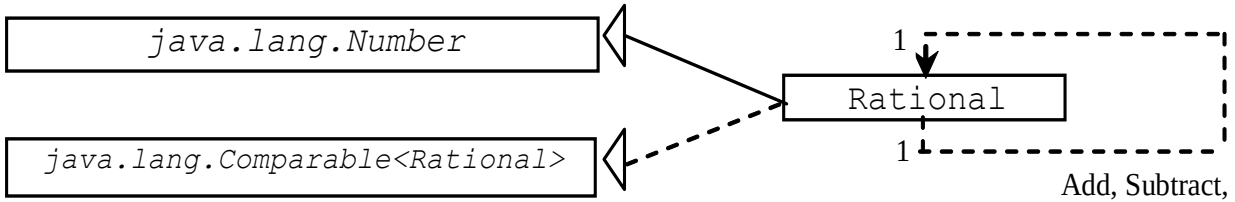
Rational Class

Inherits Number abstract class and implements Comparable Interface.

Treated like 2-D vector.

Value as double/float.

The Rational Class



Rational
-numerator: long -denominator: long
+Rational() +Rational(numerator: long, denominator: long) +getNumerator(): long +getDenominator(): long +add(secondRational: Rational): Rational +subtract(secondRational: Rational): Rational +multiply(secondRational: Rational): Rational +divide(secondRational: Rational): Rational +toString(): String

The numerator of this rational number.

The denominator of this rational number.

Creates a rational number with numerator 0 and denominator 1.

Creates a rational number with a specified numerator and denominator.

Returns the numerator of this rational number.

Returns the denominator of this rational number.

Returns the addition of this rational number with another.

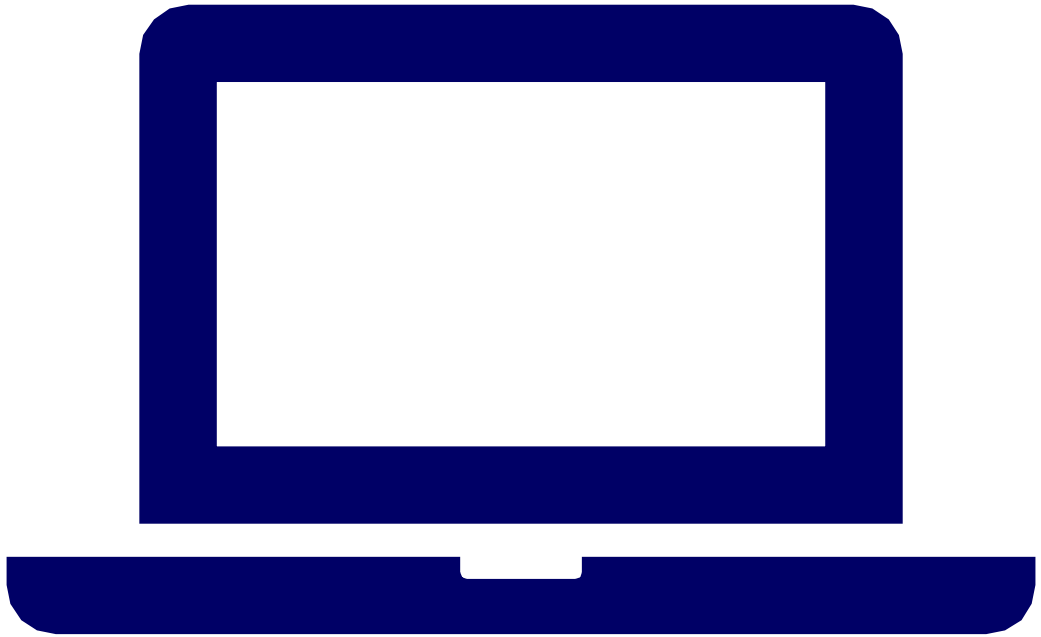
Returns the subtraction of this rational number with another.

Returns the multiplication of this rational number with another.

Returns the division of this rational number with another.

Returns a string in the form “numerator / denominator.” Returns the numerator if denominator is 1.

Returns the greatest common divisor of n and d.



Demonstration Program

RATIONAL.JAVA,
TESTRATIONAL.JAVA

Class Design Guidelines

LECTURE 1

Class Design Guidelines

- Instance vs. Static (9/10)
- Encapsulation (9/10)
- Cohesion (10)
- Consistency (10)
- Clarity (10)
- Completeness (10)
- Inheritance vs. Aggregation (10/11)
- Polymorphism (11)
- Abstract Class, Concrete Class vs Interface (13)

Abstract Class VS Interface

- Both interfaces and abstract classes can be used to specify common behavior for objects. How do you decide whether to use an interface or a class?
- In general, a strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes.
- For example, since an orange is a fruit, their relationship should be modeled using class inheritance.
- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces.

Combination of Abstract Class and Interface

- For example, all strings are comparable, so the **String** class implements the **Comparable** interface. A circle or a rectangle is a geometric object, so **Circle** can be designed as a subclass of **GeometricObject**. Circles are different and comparable based on their radii, so **Circle** can implement the **Comparable** interface. **Interfaces are more flexible than abstract classes, because a subclass can extend only one**
- **superclass but can implement any number of interfaces.** However, interfaces cannot contain concrete methods. **The virtues of interfaces and abstract classes can be combined by creating an interface with an abstract class that implements it.** Then you can use the interface or the abstract class, whichever is convenient.

Java Design Patterns (Non-AP Topic)

LECTURE 1

Design Patterns in Java

A design patterns are **well-proved solution** for solving the specific problem/task. Now, a question will be arising in your mind what kind of specific problem? Let me explain by taking an example.

Problem Given:
Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

Solution:
Singleton design pattern is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems. What are those specifications, you will see later in the types of design patterns.

Design Patterns in Java

But remember one-thing, design patterns are programming language **independent strategies** for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, **reusable** and **maintainable**. It is the most important part because java internally follows design patterns.

To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

Advantage of design pattern:

1.They are reusable in multiple projects.
2.They provide the solutions that help to define the system architecture.
3.They capture the software engineering experiences.
4.They provide transparency to the design of an application.
5.They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6.Design patterns don’t guarantee an absolute solution to a problem.
They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Categorization of design patterns:

Basically, design patterns are categorized into two parts:

1. Core java (or JSE) Design Patterns.
2. JEE Design Patterns.

Creational Design Patterns

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

Structural Design Patterns

1. **Adapter Pattern**
2. Composite Pattern
3. Proxy Pattern
4. Flyweight Pattern
5. Facade Pattern
6. Bridge Pattern
7. Decorator Pattern

Behavioral Design Patterns

1. Template Method Pattern
2. Mediator Pattern
3. Chain of Responsibility Pattern
4. Observer Pattern
5. Strategy Pattern
6. Command Pattern
7. State Pattern
8. Visitor Pattern
9. Interpreter Pattern
10. Iterator Pattern
11. Memento Pattern
12. **Front Controller Pattern**

Adapter Classes (Non-AP Topic)

LECTURE 1

Adapter Class

An adapter class provides the default implementation of all methods in an event listener interface. Adapter classes are very useful when you want to process only few of the events that are handled by a particular event listener interface. You can define a new class by extending one of the adapter classes and implement only those events relevant to you.

See more at: http://www.java2novice.com/java_interview_questions/adapter-class/#sthash.j74BBaA1.dpuf

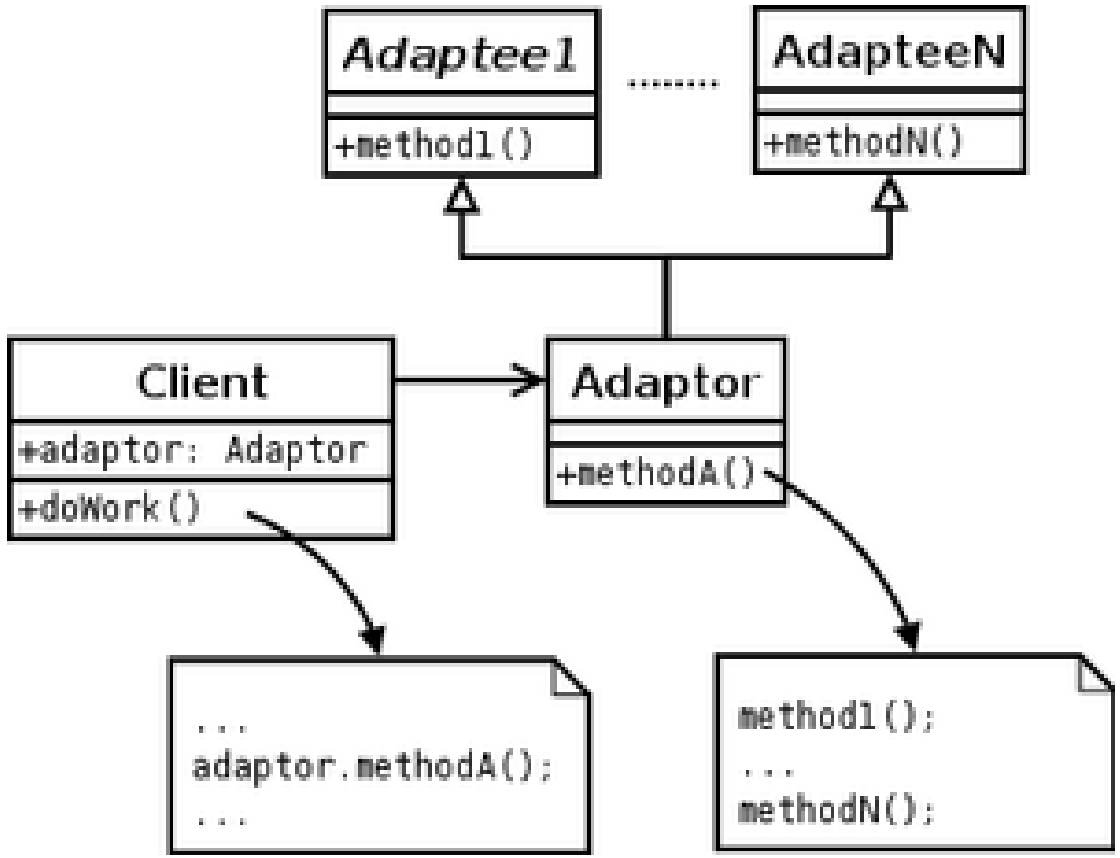
Adapter Classes

The AWT provides a number of *adapter* classes for the different EventListener interfaces. These are:

- ComponentAdapter
- ContainerAdapter
- FocusAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter

Adapter

(Collection of Methods from Classes and Interfaces)



Methods in Adapter Classes

Each adapter class implements the corresponding interface with a series of do-nothing methods. For example, MouseListener declares these five methods:

```
public abstract void mouseClicked(MouseEvent evt)
public abstract void mousePressed(MouseEvent evt)
public abstract void mouseReleased(MouseEvent evt)
public abstract void mouseEntered(MouseEvent evt)
public abstract void mouseExited(MouseEvent evt)
```

Therefore, MouseAdapter looks like this:

```
package java.awt.event;
import java.awt.*;
import java.awt.event.*;

public class MouseAdapter implements MouseListener  {
  public void mouseClicked(MouseEvent evt) {}
  public void mousePressed(MouseEvent evt) {}
  public void mouseReleased(MouseEvent evt) {}
  public void mouseEntered(MouseEvent evt) {}
  public void mouseExited(MouseEvent evt) {}
}
```

Use of Adapter Class

By subclassing MouseAdapter rather than implementing MouseListener directly, you avoid having to write the methods you don't actually need. You only override those that you plan to actually implement.

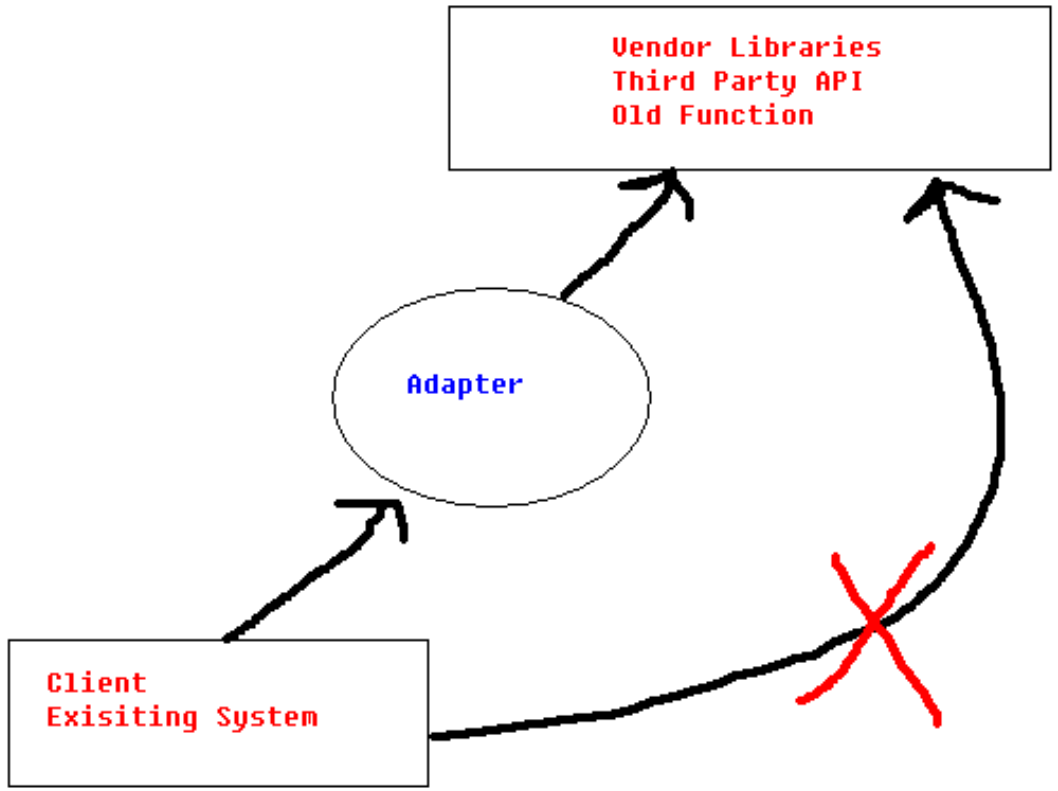
Adapter Pattern Demo (Non-AP Topic)

LECTURE 1

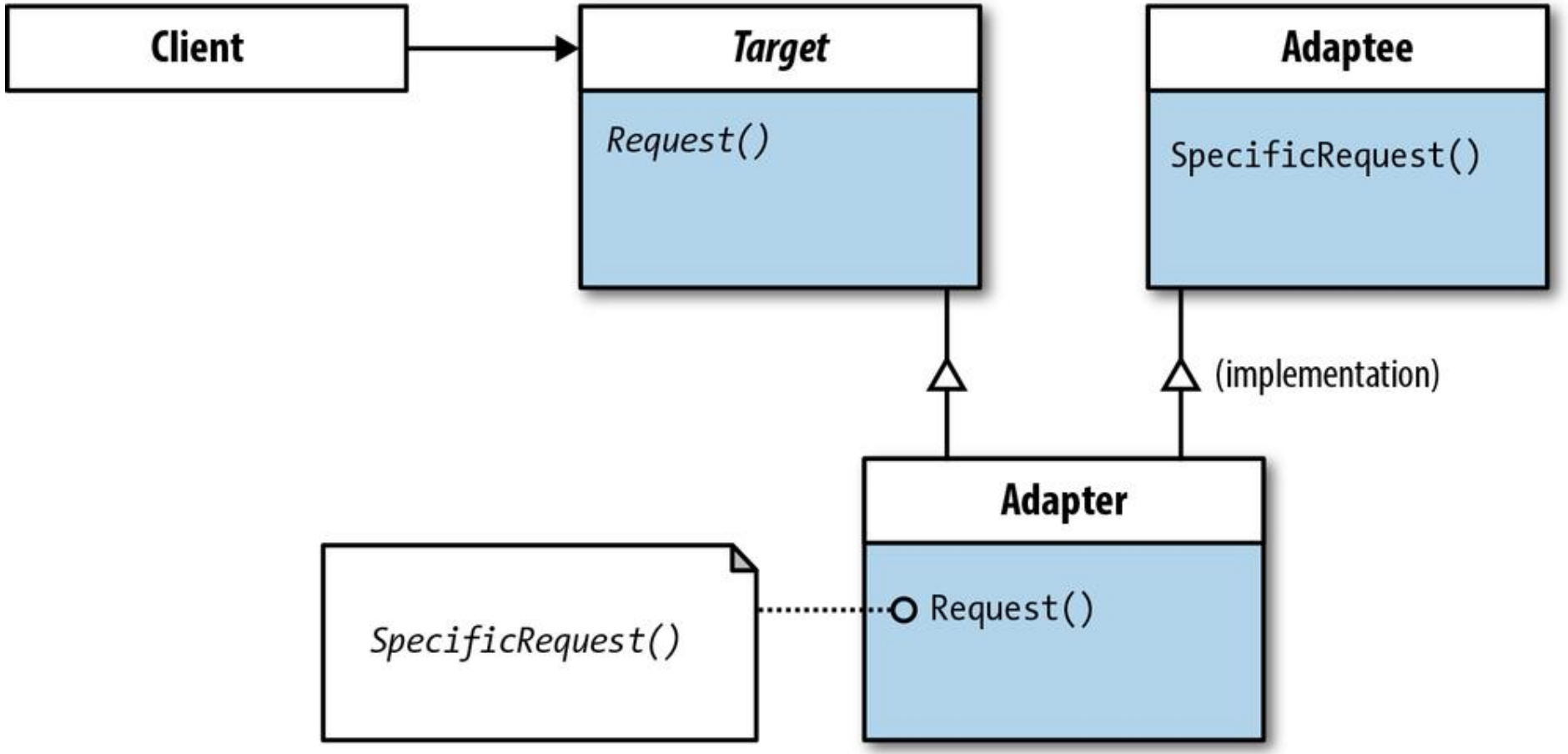
Adapter

(Adapter Class and Adapter Pattern)

Adapter , Adapter , Adapter ~ Actually Adapter design pattern can consider as a simple conversion program / class. It usually used to make two incompatible interfaces or classes to work together.



Adapter Design Pattern



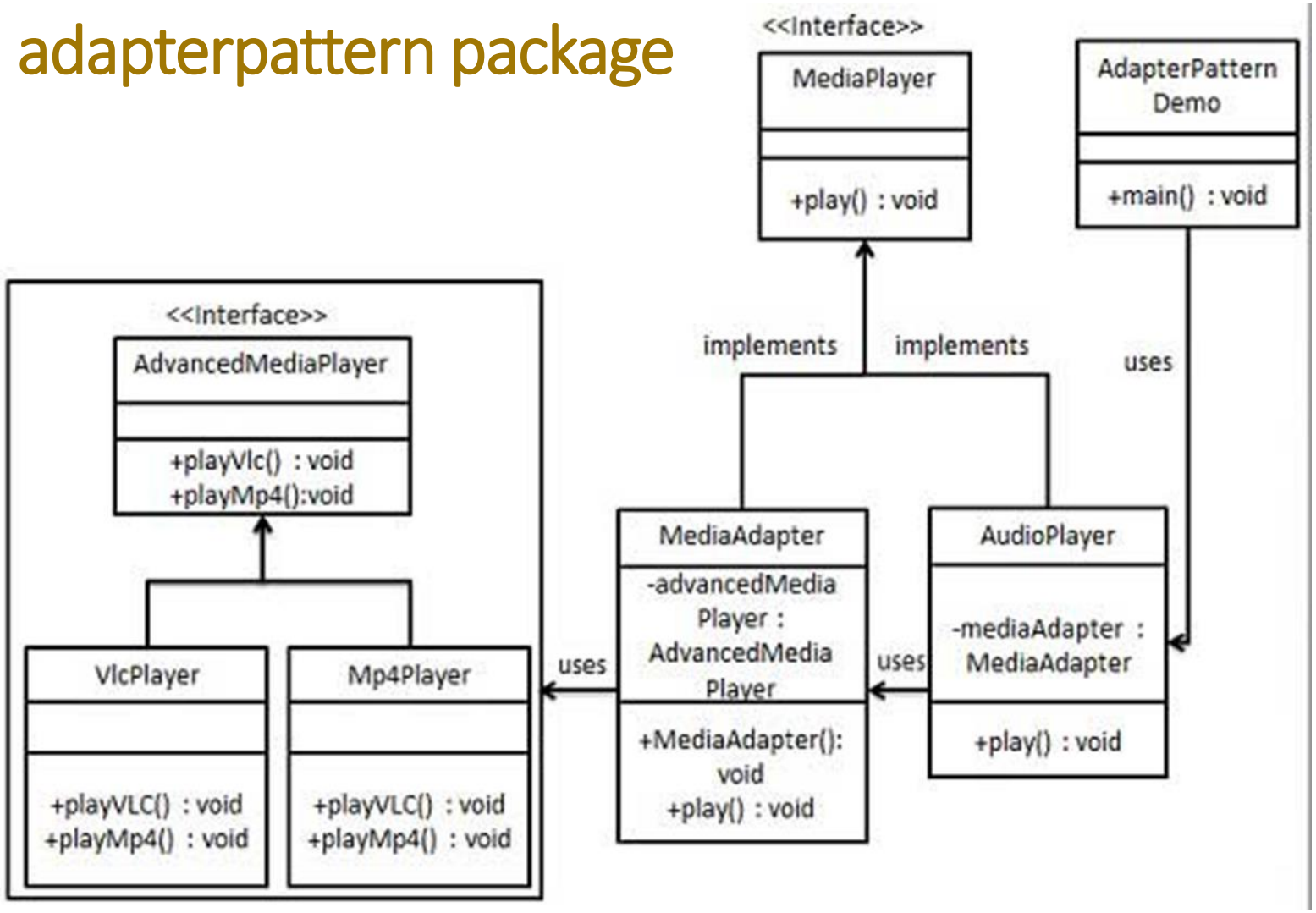
Adapter Pattern

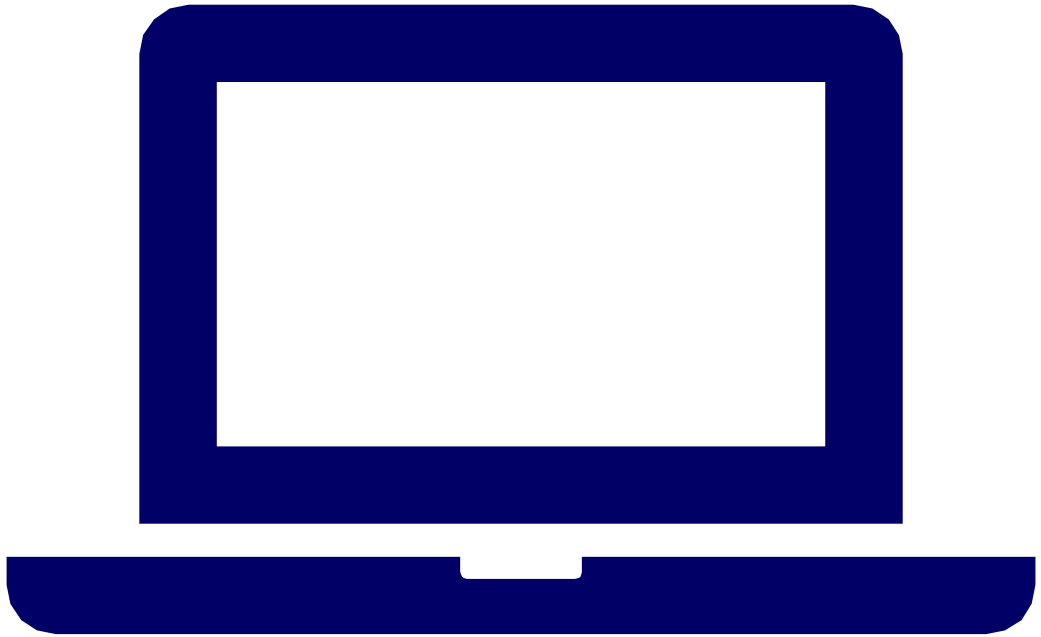
(Standard Ways of using Classes and Interfaces)

In software engineering, the **adapter pattern** is a **software design pattern** that allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

An adapter helps two **incompatible interfaces** to work together. This is the real world definition for an adapter. Interfaces may be incompatible but the inner functionality should suit the need. **The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.**

adapterpattern package





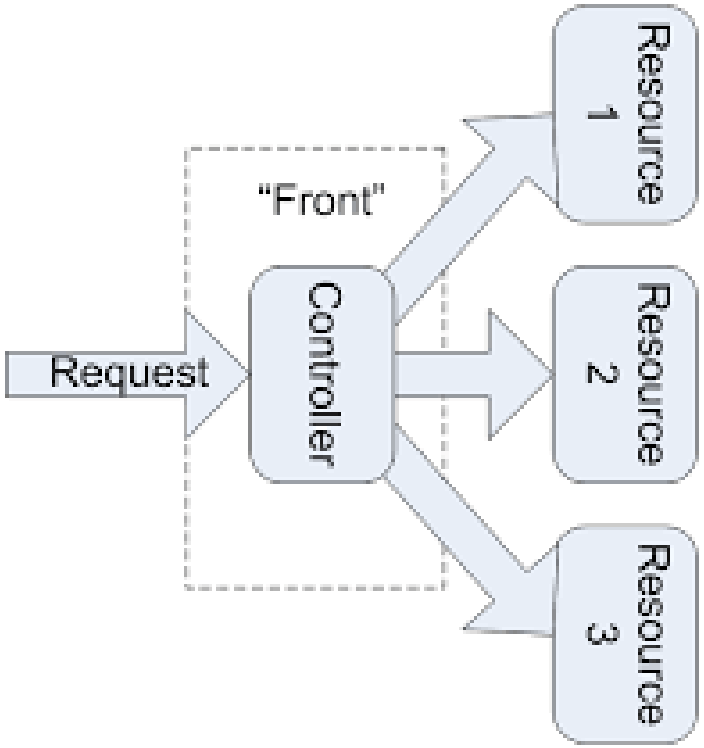
Demonstration Program

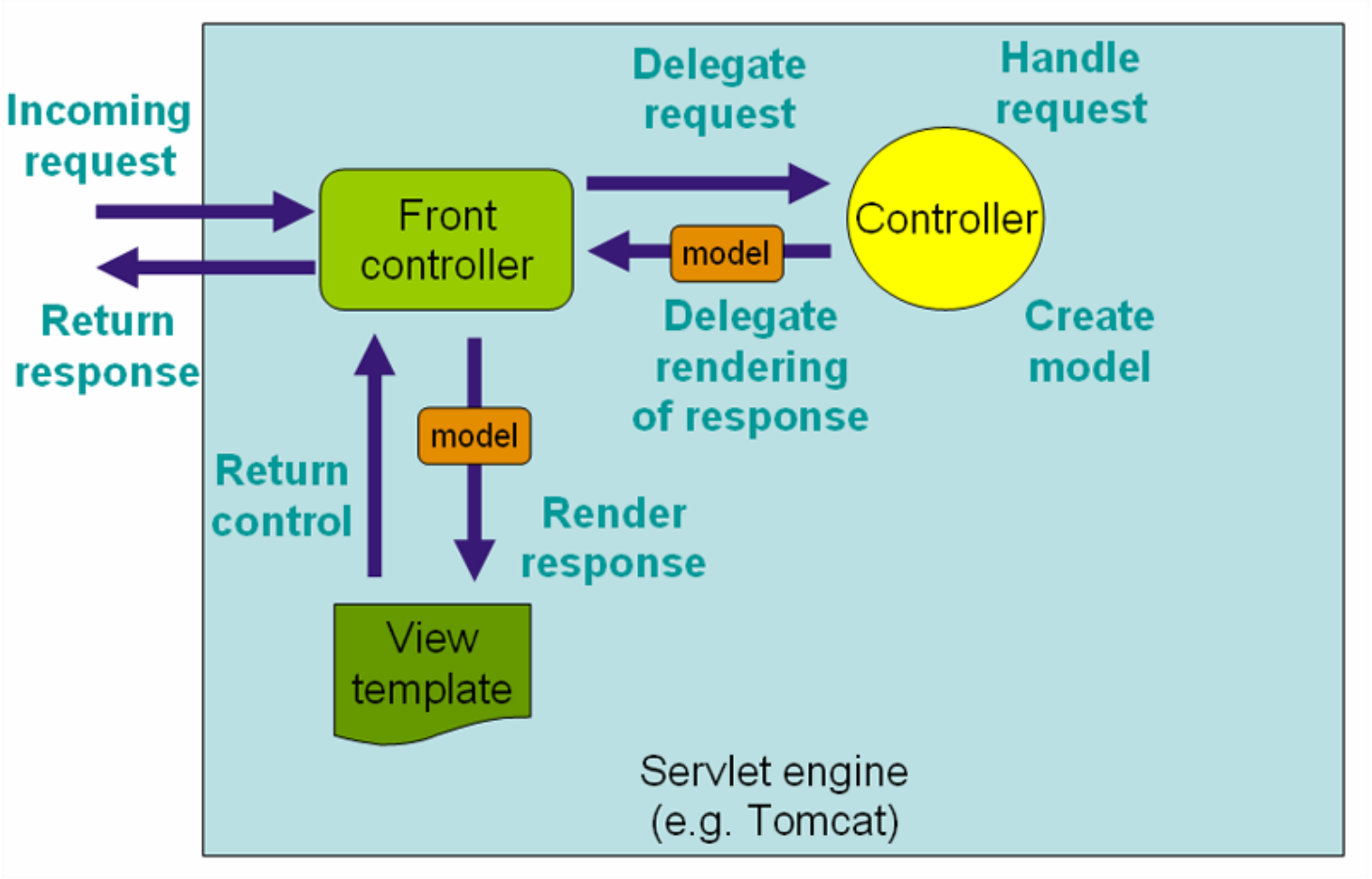
ADAPTERPATTERN PACKAGE

Front Controller Design Pattern (Non-AP Topic)

LECTURE 1

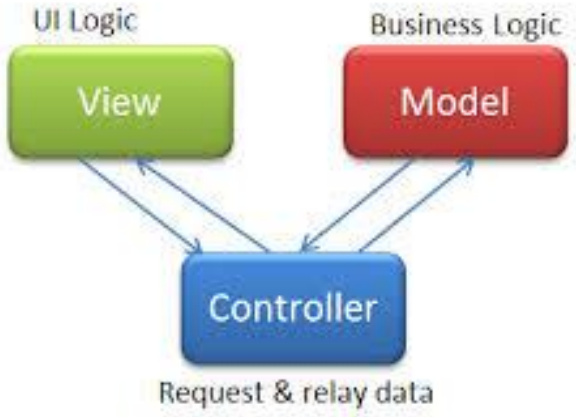
Front Controller (Single Dispatcher)

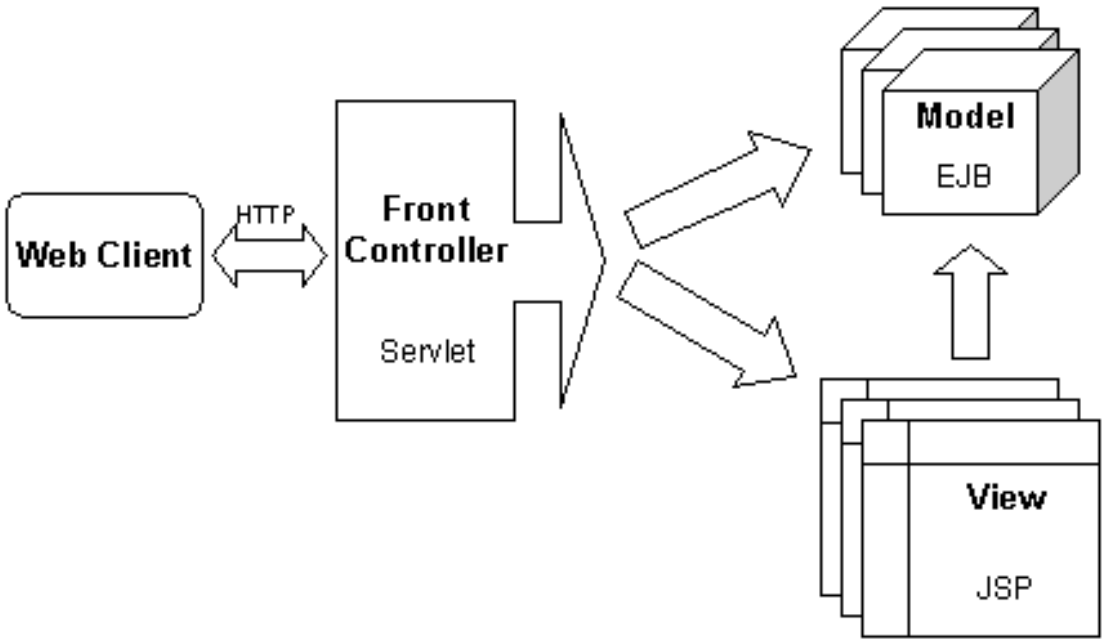


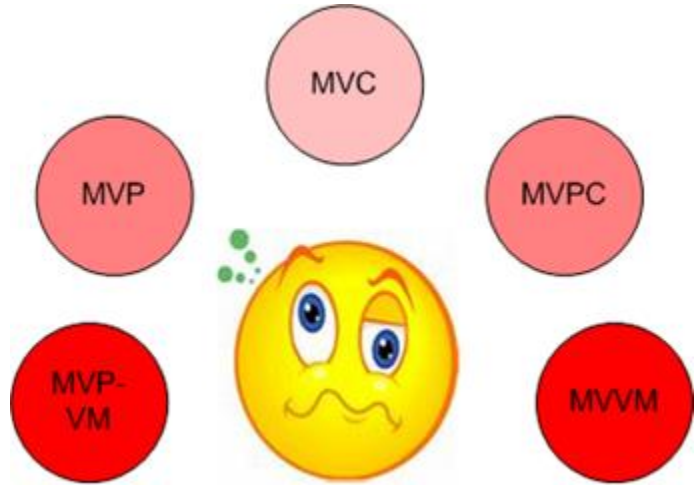


Front-Controller for Web Servlet

Simplest MVC Pattern





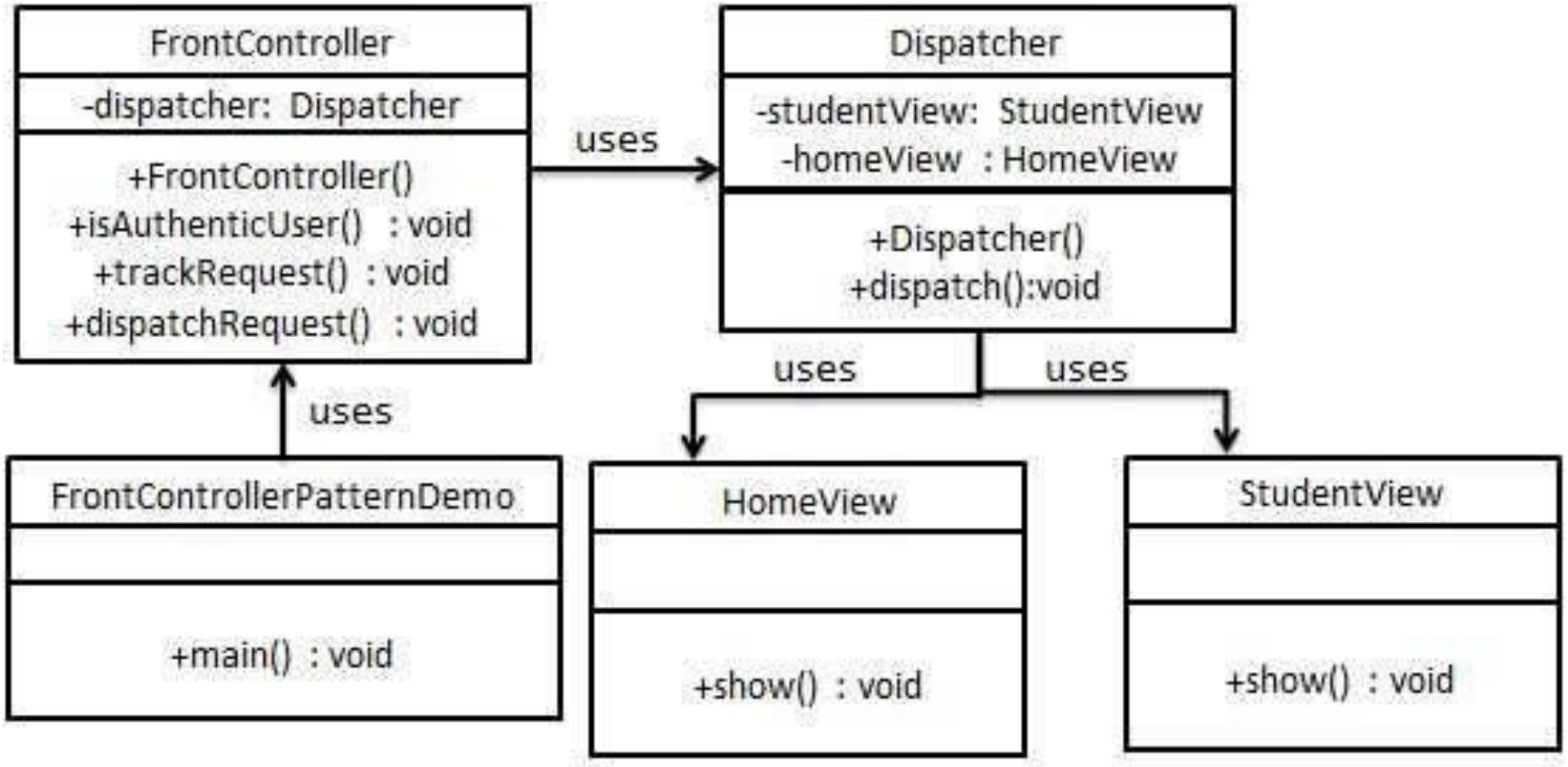


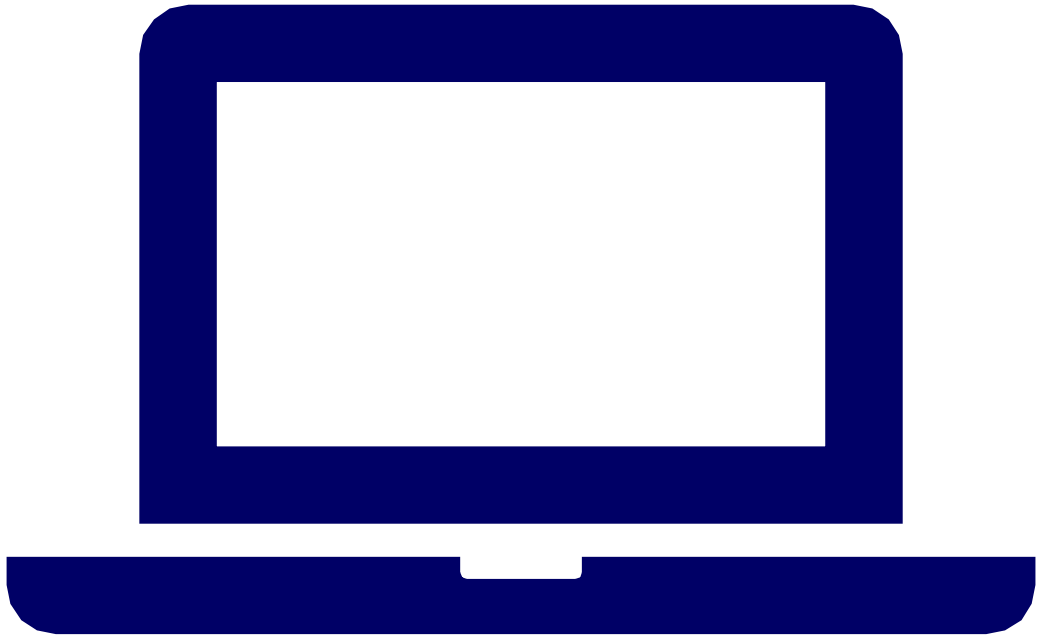
Front Controller Design Pattern

The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

- **Front Controller** - Single handler for all kinds of requests coming to the application (either web based/ desktop based).
- **Dispatcher** - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.
- **View** - Views are the object for which the requests are made.

Implementation





Demonstration Program

FRONTCONTROLLER PACKAGE