P. A. EDUCATIONAL TRUST'S (PAET)

P.A. COLLEGE OF ENGINEERING

MANGALURU -574153 , KARNATAKA (INDIA)

www.pace.edu.in

Approved by A.I.C.T.E. New Delhi, Recognized by Government of Karnataka,

Affiliated to Visvesvarya Technological University, Belagavi INDIA

OBJECT ORIENTED CONCEPTS
IV SEMESTER 2016-17
[AS PER CHOICE BASED CREDIT SYSTEM (CBCS) SCHEME]

Module III

Classes, Inheritance, Exceptions, Packages and Interfaces

Text: Java - The Complete Reference by Herbert Schildt, 7th Edition, Tata McGraw Hill, 2007. (Chapters 6,8,9,10)

OVERVIEW

- **Classes:** Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.
- **Inheritance:** inheritance basics, using super, creating multi level hierarchy, method overriding.
- **Packages:** Access Protection, Importing Packages.
- **Interfaces**
- **Exception handling:** Exception handling in Java.

Chapter 6
Introducing Classes

Text: Java - The Complete Reference by Herbert Schildt, 7th Edition, Tata McGraw Hill, 2007

WHAT IS AN OBJECT?

- Real world objects are things that have:
  1) state
  2) behavior

- Example: your dog:
  1) state – name, color, breed
  2) behavior – sitting, barking, waging tail, running
- A software object is a bundle of variables (state) and methods (operations).

WHAT IS A CLASS?

- A class is a blueprint that defines the variables and methods common to all objects of a certain kind.
- Example: ‘your dog’ is a object of the class Dog.
- An object holds values for the variables defines in the class.
- An object is called an instance of the Class

OBJECT AND CLASS

- Object: state and behavior
- Class: Attribute and method
- Object as an instance of class
- Class Box

  Attribute: width, height, depth

  Method: getVolume
- Object box,

  State: width = 1, height = 2, depth = 3

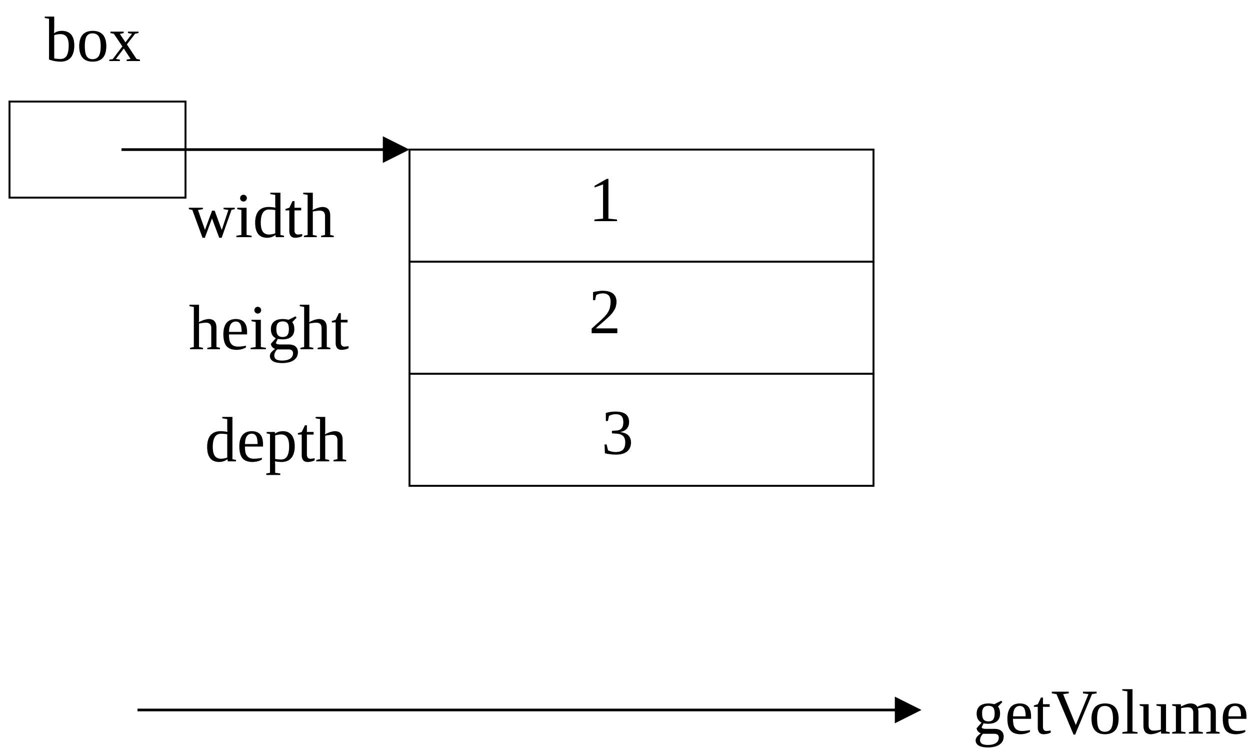
  Behavior: volume = 6

CLASS BOX

```
class Box {
  double width;
  double height;
  double depth;

  double getVolume() {
    return width*height*depth;
 }
}
```

OBJECT BOX



POINTER VS. REFERENCE

- Pointer is not used in Java for security reason
- Reference is adopted in Java
- Every object has a reference: `this`
- Reference `null`

OBJECTS

- Everything in Java is an object.
- Well ... almost.

- Object lifecycle:

  1) creation

  2) usage

  3) destruction

OBJECT CREATION

- A variable s is declared to refer to the objects of type/class String:

```
String s;
```

- The value of s is null; it does not yet refer to any object.
- A new String object is created in memory with initial “abc” value:

```
String s = new String(“abc”);
```

- Now s contains the address of this new object.

OBJECT USAGE

- Objects are used mostly through variables.
- Four usage scenarios:
  1) one variable, one object
  2) two variables, one object
  3) two variables, two objects
  4) one variable, two objects

ONE VARIABLE, ONE OBJECT

- One variable, one object:

```
String s = new String(“abc”);
```

- What can you do with the object addressed by s?
  1) Check the length: `s.length() == 3`
  2) Return the substring: `s.substring(2)`
  3) etc.
- Depending on what is allowed by the definition of String.

TWO VARIABLES, ONE OBJECT

- Two variables, one object:

```
  String s1 = new String("abc");
  String s2;
```

- Assignment copies the address, not value:

```
s2 = s1;
```

- Now s1 and s2 both refer to one object. After

```
s1 = null;
```

- s2 still points to this object.

TWO VARIABLES, TWO OBJECTS

- Two variables, two objects:

```
String s1 = new String("abc");
String s2 = new String("abc");
```

- s1 and s2 objects have initially the same values:

```
s1.equals(s2) == true
```

- But they are not the same objects:

```
(s1 == s2) == false
```

- They can be changed independent of each other.

ONE VARIABLE, TWO OBJECTS

- One variable, two objects:

```
String s = new String(“abc”);
s = new String(“cba”);
```

- The “abc” object is no longer accessible through any variable.

OBJECT DESTRUCTION

- A program accumulates memory through its execution.
- Two mechanisms to free memory that is no longer need by the program:
  1) manual – done in C/C++
  2) automatic – done in Java
- In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.
- Garbage collector is parts of the Java Run-Time Environment.

CLASS

- A basis for the Java language.
- Each concept we wish to describe in Java must be included inside a class.
- A class defines a new data type, whose values are objects:
  1) a class is a template for objects
  2) an object is an instance of a class

CLASS DEFINITION

- A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.
- General form of a class:

```
class classname {
  type instance-variable-1;
  …
  type instance-variable-n;
  type method-name-1(parameter-list) { … }
  type method-name-2(parameter-list) { … }
  …
  type method-name-m(parameter-list) { … }
}
```

EXAMPLE CLASS

- A class with three variable members:

```
class Box {
  double width;
  double height;
  double depth;
}
```

- A new Box object is created and a new value assigned to its width variable:

```
Box myBox = new Box();
myBox.width = 100;
```

EXAMPLE: CLASS USAGE

```
class BoxDemo {
 public static void main(String args[]) {
     Box mybox = new Box();
     double vol;
     mybox.width = 10;
     mybox.height = 20;
     mybox.depth = 15;
     vol = mybox.width * mybox.height *
          mybox.depth;
     System.out.println("Volume is " + vol);
 }
}
```

COMPILATION AND EXECUTION

- Place the Box class definitions in file Box.java:

```
class Box { … }
```

- Place the BoxDemo class definitions in file BoxDemo.java:

```
  class BoxDemo {
    public static void main(…) { … }
  }
```

- Compilation and execution:

```
  > javac BoxDemo.java
  > java BoxDemo
```

DECLARING AN OBJECT

- Obtaining objects of a class is a two-stage process:

  1) Declare a variable of the class type:

```
Box myBox;
```

  The value of `myBox` is a reference to an object, if one exists, or null. At this moment, the value of myBox is null.

  2) Acquire an actual, physical copy of an object and assign its address to the variable. How to do this?

OPERATOR NEW

- Allocates memory for a Box object and returns its address:

```
Box myBox = new Box();
```

- The address is then stored in the `myBox` reference variable.
- `Box()` is a class constructor - a class may declare its own constructor or rely on the default constructor provided by the Java environment.

METHODS

- General form of a method definition:

```
type name(parameter-list) {
     … return value; …
}
```

- Components:

1. type - type of values returned by the method. If a method does not return any value, its return type must be void
2. name is the name of the method
3. parameter-list is a sequence of type-identifier lists separated by commas
4. return value indicates what value is returned by the method

EXAMPLE: METHOD

- Classes declare methods to hide their internal data structures, as well as for their own internal use
- Within a class, we can refer directly to its member variables:

```
class Box {
 double width, height, depth;
 void volume() {
     System.out.print("Volume is ");
     System.out.println(width * height * depth);
 }
}
```

PARAMETERIZED METHODS

- Parameters increase generality and applicability of a method:
  1. method without parameters

     ```
     int square() { return 10*10; }
     ```
  2. method with parameters

     ```
     int square(int i) { return i*i; }
     ```
- Parameter: a variable receiving value at the time the method is invoked.
- Argument: a value passed to the method when it is invoked.

CONSTRUCTOR

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.
  1. it is syntactically similar to a method
  2. it has the same name as the name of its class
  3. it is written without return type; the default return type of a class constructor is the same class

- **When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.**

EXAMPLE: CONSTRUCTOR

```
class Box {
 double width;
 double height;
 double depth;
 Box() {
     System.out.println("Constructing Box");
     width = 10; height = 10; depth = 10;
 }
 double volume() {
     return width * height * depth;
 }
}
```

EXAMPLE: CONSTRUCTOR

```
class BoxDemo6 {
 public static void main(String args[]) {
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;
      vol = mybox1.volume();
      System.out.println("Volume is " + vol);
      vol = mybox2.volume();
      System.out.println("Volume is " + vol);
 }
}
```

EXAMPLE: PARAMETERIZED CONSTRUCTOR

- So far, all boxes have the same dimensions.
- We need a constructor able to create boxes with different dimensions:

```
class Box {
 double width;
 double height;
 double depth;
 Box(double w, double h, double d) {
     width = w; height = h; depth = d;
 }
 double volume() { return width * height * depth; }
}
```

EXAMPLE: PARAMETERIZED CONSTRUCTOR

```
class BoxDemo7 {
 public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(3, 6, 9);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
 }
}
```

FINALIZE() METHOD

- A constructor helps to initialize an object just after it has been created.
- In contrast, the finalize method is invoked just before the object is destroyed:

  1) implemented inside a class as:

  ```
  protected void finalize() { … }
  ```

  2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out
- How is the finalize method invoked?

GARBAGE COLLECTION

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:
  1. The garbage collector keeps track of how many references an object has.
  2. It removes an object from memory when it has no longer any references.
  3. Thereafter, the memory occupied by the object can be allocated again.
  4. **The garbage collector invokes the finalize method.**

KEYWORD **THIS**

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object

```
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

Chapter 8 Inheritance

Text: Java - The Complete Reference by Herbert Schildt, 7th Edition, Tata McGraw Hill, 2007

INHERITANCE

- One of the pillars of object-orientation.
- A new class is derived from an existing class:

  1) existing class is called super-class

  2) derived class is called sub-class
- A sub-class is a specialized version of its super-class:

  1) has all non-private members of its super-class

  2) may provide its own implementation of super-class methods
- Objects of a sub-class are a special kind of objects of a super-class.

INHERITANCE SYNTAX

- Syntax:

```
class sub-class extends super-class {
  …
}
```

- Each class has at most one super-class; no multi-inheritance in Java.
- No class is a sub-class of itself.

EXAMPLE: SUPER-CLASS

```
class A {
  int i;
  void showi() {
        System.out.println("i: " + i);
  }
}
```

EXAMPLE: SUB-CLASS

```
class B extends A {
 int j;
 void showj() {
       System.out.println(“j: " + j);
 }
 void sum() {
       System.out.println("i+j: " + (i+j));
 }
}
```

EXAMPLE: TESTING CLASS

```
class SimpleInheritance {
  public static void main(String args[]) {
        A a = new A();
        B b = new B();
        a.i = 10;
        System.out.println("Contents of a: ");
        a.showi();
        b.i = 7; b.j = 8;
        System.out.println("Contents of b: ");
        b.showi();
        b.showj();
        System.out.println("Sum of i and j in b:");
        b.sum();
  }
}
```

INHERITANCE AND PRIVATE MEMBERS 1

- A class may declare some of its members private.
- A sub-class has no access to the private members of its super-class:

```
class A {
  int i;
  private int j;
  void setij(int x, int y) {
      i = x; j = y;
  }
}
```

INHERITANCE AND PRIVATE MEMBERS 2

- Class B has no access to the A's private variable j.
- This program will not compile:

```
class B extends A {
  int total;
  void sum() {
      total = i + j;
  }
}
```

EXAMPLE: BOX CLASS

```
// The basic Box class with width, height and depth
class Box {
  double width, height, depth;
  Box(double w, double h, double d) {
        width = w; height = h; depth = d;
  }
  Box(Box b) {
        width = b.width;
        height = b.height;
        depth = b.depth;
  }
  double volume() {
        return width * height * depth;
  }
}
```

EXAMPLE: BOXWEIGHT SUB-CLASS

- BoxWeight class extends Box with the new weight variable:

```
class BoxWeight extends Box {
 double weight;
 BoxWeight(double w, double h, double d, double m) {
     width = w; height = h; depth = d; weight = m;
 }
 BoxWeight(Box b, double w) {
     super(b); weight = w;
 }
}
```

- Box is a super-class, BoxWeight is a sub-class.

EXAMPLE: BOXWEIGHT DEMO

```
class DemoBoxWeight {
  public static void main(String args[]) {
       BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
       BoxWeight mybox2 = new BoxWeight(mybox1);
       double vol;
       vol = mybox1.volume();
       System.out.println("Volume of mybox1 is " + vol);
       System.out.print("Weight of mybox1 is ");
       System.out.println(mybox1.weight);
       vol = mybox2.volume();
       System.out.println("Volume of mybox2 is " + vol);
       System.out.print("Weight of mybox2 is ");
       System.out.println(mybox2.weight);
  }
}
```

ANOTHER SUB-CLASS

- Once a super-class exists that defines the attributes common to a set of objects, it can be used to create any number of more specific sub-classes.
- The following sub-class of Box adds the color attribute instead of weight:

```
class ColorBox extends Box {
  int color;
  ColorBox(double w, double h, double d, int c) {
    width = w; height = h; depth = d;
    color = c;
  }
}
```

REFERENCING SUB-CLASS OBJECTS

- A variable of a super-class type may refer to any of its sub-class objects:

```
class SuperClass { … }
class SubClass extends SuperClass { … }
SuperClass o1;
SubClass o2 = new SubClass();
o1 = o2;
```

- However, the inverse is illegal:

```
o2 = o1;
```

EXAMPLE: SUB-CLASS OBJECTS 1

```
class RefDemo {
 public static void main(String args[]) {
      BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
      Box plainbox = new Box(5, 5, 5);
      double vol;
      vol = weightbox.volume();
      System.out.println("Volume of weightbox is “ + vol);
      System.out.print("Weight of weightbox is “);
      System.out.println(weightbox.weight);
      plainbox = weightbox;
      vol = plainbox.volume();
      System.out.println("Volume of plainbox is " + vol);
  }
}
```

SUPER-CLASS VARIABLE ACCESS

- plainbox variable now refers to the WeightBox object.
- Can we then access this object's weight variable through plainbox?
- No. The type of a variable, not the object this variable refers to, determines which members we can access!
- This is illegal:

```
System.out.print("Weight of plainbox is “);
System.out.println(plainbox.weight);
```

SUPER AS A CONSTRUCTOR

- Calling a constructor of a super-class from the constructor of a sub-class:

```
super(parameter-list);
```

- Must occur as the very first instruction in the sub-class constructor:

```
class SuperClass { … }
class SubClass extends SuperClass {
  SubClass(…) {
        super(…);
        …
  }
  …
}
```

EXAMPLE: SUPER CONSTRUCTOR 1

- BoxWeight need not initialize the variables for the Box super-class, only the added weight variable:

```
class BoxWeight extends Box {
  double weight;
  BoxWeight(double w, double h, double d, double m) {
      super(w, h, d); weight = m;
  }
  BoxWeight(Box b, double w) {
      super(b); weight = w;
  }
}
```

EXAMPLE: SUPER CONSTRUCTOR 2

```
class DemoSuper {
 public static void main(String args[]) {
      BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
      BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
      double vol;
      vol = mybox1.volume();
      System.out.println("Volume of mybox1 is " + vol);
      System.out.print("Weight of mybox1 is “);
      System.out.println(mybox1.weight);
      vol = mybox2.volume();
      System.out.println("Volume of mybox2 is " + vol);
      System.out.print("Weight of mybox2 is “);
      System.out.println(mybox2.weight);
  }
}
```

REFERENCING SUB-CLASS OBJECTS

- Sending a sub-class object:

```
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
```

- to the constructor expecting a super-class object:

```
BoxWeight(Box b, double w) {
    super(b); weight = w;
}
```

USES OF SUPER

- Two uses of super:
  1. to invoke the super-class constructor
     ```
     super();
     ```
  2. to access super-class members
     ```
     super.variable;
     super.method(…);
     ```

  (1) was discussed, consider (2).

SUPER AND HIDING

- Why is super needed to access super-class members?
- When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {
  int i = 1;
}


class B extends A {
  int i = 2;
  System.out.println("i is " + i);
}
```

- The re-declared variables/methods hide those of the super-class.

EXAMPLE: SUPER AND HIDING 1

```
class A {
  int i;
}
class B extends A {
  int i;
  B(int a, int b) {
        super.i = a; i = b;
  }
  void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
  }
}
```

EXAMPLE: SUPER AND HIDING 2

- Although the i variable in B hides the i variable in A, super allows access to the hidden variable of the super-class:

```
class UseSuper {
 public static void main(String args[]) {
     B subOb = new B(1, 2);
     subOb.show();
  }
}
```

MULTI-LEVEL CLASS HIERARCHY 1

- The basic Box class:

```
class Box {
  private double width, height, depth;
  Box(double w, double h, double d) {
       width = w; height = h; depth = d;
  }
  Box(Box ob) {
       width = ob.width;
       height = ob.height; depth = ob.depth;
  }
  double volume() {
       return width * height * depth;
  }
}
```

MULTI-LEVEL CLASS HIERARCHY 2

- Adding the weight variable to the Box class:

```
class BoxWeight extends Box {
  double weight;
  BoxWeight(BoxWeight ob) {
        super(ob); weight = ob.weight;
  }
  BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); weight = m;
  }
}
```

MULTI-LEVEL CLASS HIERARCHY 3

- Adding the cost variable to the BoxWeight class:

```
class Ship extends BoxWeight {
  double cost;
  Ship(Ship ob) {
      super(ob); cost = ob.cost;
  }
  Ship(double w, double h, double d, double m, double c) {
      super(w, h, d, m); cost = c;
  }
}
```

MULTI-LEVEL CLASS HIERARCHY 4

```
class DemoShip {
  public static void main(String args[]) {
        Ship ship1 = new Ship(10, 20, 15, 10, 3.41);
        Ship ship2 = new Ship(2, 3, 4, 0.76, 1.28);
        double vol;
        vol = ship1.volume();
        System.out.println("Volume of ship1 is " + vol);
        System.out.print("Weight of ship1 is”);
        System.out.println(ship1.weight);
        System.out.print("Shipping cost: $");
        System.out.println(ship1.cost);
        vol = ship2.volume();
        System.out.println("Volume of ship2 is " + vol);
        System.out.print("Weight of ship2 is “);
        System.out.println(ship2.weight);
        System.out.print("Shipping cost: $“);
        System.out.println(ship2.cost);
  }
}
```

CONSTRUCTOR CALL-ORDER

- Constructor call-order:

  1) first call super-class constructors

  2) then call sub-class constructors
- In the sub-class constructor, if super(…) is not used explicitly, Java calls the default, parameter-less super-class constructor.

EXAMPLE: CONSTRUCTOR CALL-ORDER 1

```
// A is the super-class, B and C are sub-classes of A:
class A {
      A() { System.out.println("Inside A's constructor."); }
}
class B extends A {
      B() { System.out.println("Inside B's constructor."); }
}
class C extends B {
      C() { System.out.println("Inside C's constructor."); }
}
// CallingCons creates a single object of the class C:
class CallingCons {
            public static void main(String args[]) {
                  C c = new C();
            }
}
```

EXERCISE: INHERITANCE 1

- 1) Define a Class Building for building objects. Each building has a door as one of its components.

  a) In the class Door, model the fact that a door has a color and three states, "open" , "closed", "locked“ and “unlocked". To avoid illegal state changes, make the state private, write a method (getState) that inspects the state and four methods (open, close, lock and unlock) that change the state. Initialize the state to "closed“ in the constructor. Look for an alternative place for this initialization.

  b) Write a method enter that visualizes the process of entering the building (unlock door, open door, enter, ...) by printing adequate messages, e.g. to show the state of the door.

  c) Write a corresponding method quit that visualizes the process of leaving the house. Don't forget to close and lock the door.

  d) Test your class by defining an object of type Building and visualizing the state changes when entering and leaving the building.

EXERCISE: INHERITANCE 2

3) Extend question 1 by introducing a subclass HighBuilding that contains an elevator and the height of the building in addition to the components of Building. Override the method enter to reflect the use of the elevator. Define a constructor that takes the height of the building as a parameter.

4) Define a subclass Skyscraper of HighBuilding, where the number of floors is stored with each skyscraper.

   What happens, if you don't define a constructor for class Skyscraper (Try it)?

   Write a constructor that takes the number of floors and the height as a parameter. Test the class by creating a skyscraper with 40 floors and using the inherited method enter.

INHERITANCE AND REUSE

- Reuse of code: every time a new sub-class is defined, programmers are reusing the code in a super-class.
- All non-private members of a super-class are inherited by its sub-class:

  1) an attribute in a super-class is inherited as-such in a sub-class

  2) a method in a super-class is inherited in a sub-class:

     a) as-such, or
     b) is substituted with the method which has the same name and parameters (overriding) but a different implementation

POLYMORPHISM: BEHAVIOR

- Suppose we have a hierarchy of classes:
    1. The top class in the hierarchy represents a common interface to all classes below. This class is the base class.
    2. All classes below the base represent a number of forms of objects, all referred to by a variable of the base class type.
- What happens when a method is invoked using the base class reference?
- The object responds in accordance with its true type.
- What if the user pointed to a different form of object using the same reference? The user would observe different behavior.
- This is polymorphism.

METHOD OVERRIDING

- When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is overridden.
- When an overridden method is called from within the sub-class:

  1) it will always refer to the sub-class method

  2) super-class method is hidden

EXAMPLE: HIDING WITH OVERRIDING 1

```
class A {
 int i, j;
 A(int a, int b) {
     i = a; j = b;
 }
 void show() {
     System.out.println("i and j: " + i + " " + j);
 }
}
```

EXAMPLE: HIDING WITH OVERRIDING 2

```
class B extends A {
 int k;
 B(int a, int b, int c) {
      super(a, b);
      k = c;
 }
 void show() {
      System.out.println("k: " + k);
 }
}
```

EXAMPLE: HIDING WITH OVERRIDING 3

- When show() is invoked on an object of type B, the version of show() defined in B is used:

```
class Override {
 public static void main(String args[]) {
     B subOb = new B(1, 2, 3);
     subOb.show();
 }
}
```

- The version of show() in A is hidden through overriding.

SUPER AND METHOD OVERRIDING

- The hidden super-class method may be invoked using super:

```
class B extends A {
  int k;
  B(int a, int b, int c) {
       super(a, b);
       k = c;
  }
  void show() {
       super.show();
       System.out.println("k: " + k);
  }
}
```

- The super-class version of show() is called within the sub-class's version.

OVERRIDING VERSUS OVERLOADING 1

- Method overriding occurs only when the names and types of the two methods (super-class and sub-class methods) are identical. If not identical, the two methods are simply overloaded:

```
class A {
  int i, j;
  A(int a, int b) {
    i = a; j = b;
  }
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

OVERRIDING VERSUS OVERLOADING 2

- The show() method in B takes a String parameter, while the show() method in A takes no parameters:

```
class B extends A {
  int k;
  B(int a, int b, int c) {
        super(a, b); k = c;
  }
  void show(String msg) {
        System.out.println(msg + k);
  }
}
```

OVERRIDING VERSUS OVERLOADING 3

- The two invocations of show() are resolved through the number of arguments (zero versus one):

```
class Overload {
  public static void main(String args[]) {
       B subOb = new B(1, 2, 3);
       subOb.show("This is k: ");
       subOb.show();
  }
}
```

DYNAMIC METHOD INVOCATION

- Overriding is a lot more than the namespace convention.
- Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.
- Method overriding allows for dynamic method invocation:
  1) an overridden method is called through the super-class variable
  2) Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
  3) when different types of objects are referred, different versions of the overridden method will be called.

EXAMPLE: DYNAMIC INVOCATION 1

- A super-class A:

```
class A {
  void callme() {
        System.out.println("Inside A's callme method");
  }
}
```

EXAMPLE: DYNAMIC INVOCATION 2

```
// Two sub-classes B and C
// B and C override the A’s callme() method.
class B extends A {
  void callme() {
        System.out.println("Inside B's callme method");
  }
}
class C extends A {
  void callme() {
        System.out.println("Inside C's callme method");
  }
}
```

EXAMPLE: DYNAMIC INVOCATION 3

- Overridden method is invoked through the variable of the super-class type. Each time, the version of the callme() method executed depends on the type of the object being referred to at the time of the call:

```
class Dispatch {
  public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        A r;
        r = a; r.callme();
        r = b; r.callme();
        r = c; r.callme();
  }
}
```

POLYMORPHISM AGAIN

- One interface, many behaviors:
  1) super-class defines common methods for sub-classes
  2) sub-class provides specific implementations for some of the methods of the super-class
- A combination of inheritance and overriding – sub-classes retain flexibility to define their own methods, yet they still have to follow a consistent interface.

EXAMPLE: POLYMORPHISM 1

```
// A class that stores the dimensions of various 2-dimensional objects:
class Figure {
  double dim1;
  double dim2;
  Figure(double a, double b) {
        dim1 = a; dim2 = b;
  }
  double area() {
        System.out.println("Area is undefined.");
        return 0;
  }
}
```

EXAMPLE: POLYMORPHISM 2

```
// Rectangle is a sub-class of Figure:
class Rectangle extends Figure {
 Rectangle(double a, double b) {
       super(a, b);
 }
 double area() {
       System.out.println("Inside Area for Rectangle.");
       return dim1 * dim2;
 }
}
```

EXAMPLE: POLYMORPHISM 3

```
// Triangle is a sub-class of Figure:
class Triangle extends Figure {
 Triangle(double a, double b) {
      super(a, b);
 }
 double area() {
      System.out.println("Inside Area for Triangle.");
      return dim1 * dim2 / 2;
 }
}
```

EXAMPLE: POLYMORPHISM 4

- Invoked through the Figure variable and overridden in their respective subclasses, the area() method returns the area of the invoking object:

```
class FindAreas {
  public static void main(String args[]) {
      Figure f = new Figure(10, 10);
      Rectangle r = new Rectangle(9, 5);
      Triangle t = new Triangle(10, 8);
      Figure figref;
      figref = r; System.out.println(figref.area());
      figref = t; System.out.println(figref.area());
      figref = f; System.out.println(figref.area());
  }
}
```

EXERCISE: POLYMORPHISM

1) Define a relationship among the following Building, HighBuilding and Skyscraper classes.
2) Define a class Visits that stores an array of 10 buildings (representing a street).
3) Define a method that enters all the buildings in the street using the method enter, one after another.
4) Fill the array with mixed objects from the classes Building, HighBuilding and Skyscraper.

Make sure, that the output of your program visualizes the fact that different method implementations are used depending on the type of the actual object.

Chapter 9
Packages and Interfaces

Text: Java - The Complete Reference by Herbert Schildt, 7th Edition, Tata McGraw Hill, 2007

NAME SPACE MANAGEMENT

- Classes written so far all belong to a single name space: a unique name has to be chosen for each class to avoid name collision.
- Some way to manage the name space is needed to:
  1) ensure that the names are unique
  2) provide a continuous supply of convenient, descriptive names
  3) ensure that the names chosen by one programmer will not collide with those chosen by another programmers
- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is a **package**.

PACKAGE

- A **package** is both a naming and a visibility control mechanism:

  1) divides the name space into disjoint subsets
- It is possible to define classes within a package that are not accessible by code outside the package.

  2) controls the visibility of classes and their members
- It is possible to define class members that are only exposed to other members of the same package.
- Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages.

PACKAGE DEFINITION

- A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- means that all classes in this file belong to the myPackage package. The package statement creates a name space where such classes are stored.
- **When the package statement is omitted, class names are put into the default package which has no name.**

MULTIPLE SOURCE FILES

- Other files may include the same package instruction:

```
// A Source File
package myPackage;
class MyClass1 { … }
class MyClass2 { … }

// Another Source File
package myPackage;
class MyClass3{ … }
```

- A package may be distributed through several source files.

PACKAGES AND DIRECTORIES

- Java uses file system directories to store packages.

- Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The bytecode files MyClass1.class and MyClass2.class must be stored in a directory myPackage.
- Case is significant! Directory names must match package names exactly.

PACKAGE HIERARCHY

- To create a package hierarchy, separate each package name with a dot:

package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be stored accordingly in the file system:
  1) Unix myPackage1/myPackage2/myPackage3
  2) Windows myPackage1\myPackage2\myPackage3
  3) Macintosh myPackage1:myPackage2:myPackage3
- You cannot rename a package without renaming its directory!

FINDING PACKAGES

- As packages are stored in directories, how does the Java run-time system know where to look for packages?
- Two ways:
  1) The current directory is the default start point – if packages are stored in the current directory or sub-directories, they will be found.
  2) Specify a directory path or paths by setting the CLASSPATH environment variable.

CLASSPATH VARIABLE

- CLASSPATH - environment variable that points to the root directory of the system's package hierarchy.
- Several root directories may be specified in CLASSPATH, e.g. the current directory and the C:\myJava directory:

```
.;C:\myJava
```

- Java will search for the required packages by looking up subsequent directories described in the CLASSPATH variable.

FINDING PACKAGES

- Consider this package statement:

```
package myPackage;
```

- In order for a program to find myPackage, one of the following must be true:
    1) program is executed from the directory immediately above myPackage (the parent of myPackage directory)
    2) CLASSPATH must be set to include the path to myPackage

EXAMPLE: PACKAGE 1

```
package MyPack;

class Balance {
  String name;
  double bal;
  Balance(String n, double b) {
        name = n; bal = b;
  }
  void show() {
        if (bal<0) System.out.print("-->> ");
        System.out.println(name + ": $" + bal);
  }
}
```

EXAMPLE: PACKAGE 2

```
class AccountBalance {
 public static void main(String args[]) {
      Balance current[] = new Balance[3];
      current[0] = new Balance("K. J. Fielding", 123.23);
      current[1] = new Balance("Will Tell", 157.02);
      current[2] = new Balance("Tom Jackson", -12.33);
      for (int i=0; i<3; i++) current[i].show();
 }
}
```

EXAMPLE: PACKAGE 3

Save, compile and execute:

1) call the file AccountBalance.java
2) save the file in the directory MyPack
3) compile; AccountBalance.class should be also in MyPack
4) set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
5) run:

```
java MyPack.AccountBalance
```

Make sure to use the package-qualified class name.

IMPORTING OF PACKAGES

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.
- The **import** statement allows to use classes or whole packages directly.
- Importing of a concrete class:

```
import myPackage1.myPackage2.myClass;
```

- Importing of all classes within a package:

```
import myPackage1.myPackage2.*;
```

ACCESS CONTROL

- Classes and packages are both means of encapsulating and containing the name space and scope of classes, variables and methods:
  1) packages act as a container for classes and other packages
  2) classes act as a container for data and code
- Access control is set separately for classes and class members.

ACCESS CONTROL: CLASSES

- Two levels of access:

1) A class available in the whole program:

```
public class MyClass { … }
```

2) A class available within the same package only:

```
class MyClass { … }
```

ACCESS CONTROL: MEMBERS

- Four levels of access:

  1) a member is available in the whole program:

```
public int variable;
public int method(…) { … }
```

  2) a member is only available within the same class:

```
private int variable;
private int method(…) { … }
```

ACCESS CONTROL: MEMBERS

3) a member is available within the same package (default access):

```
int variable;
int method(…) { … }
```

4) a member is available within the same package as the current class, or within its sub-classes:

```
protected int variable;
protected int method(…) { … }
```

- The sub-class may be located inside or outside the current package.

ACCESS CONTROL SUMMARY

- Complicated?
- Any member declared public can be accessed from anywhere.
- Any member declared private cannot be seen outside its class.
- When a member does not have any access specification (default access), it is visible to all classes within the same package.
- To make a member visible outside the current package, but only to subclasses of the current class, declare this member protected.

TABLE: ACCESS CONTROL

	private	default	protected	public
same class	**yes**	**yes**	**yes**	**yes**
same package subclass	**no**	**yes**	**yes**	**yes**
same package non-sub-class	**no**	**yes**	**yes**	**yes**
different package sub-class	**no**	**no**	**yes**	**yes**
different package non-sub-class	**no**	**no**	**no**	**yes**

EXAMPLE: ACCESS 1

- Access example with two packages p1 and p2 and five classes.
- A public Protection class is in the package p1. It has four variables with four possible access rights:

```
package p1;
public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;
```

EXAMPLE: ACCESS 2

```
  public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
  }
}
```

- The rest of the example tests the access to those variables.

EXAMPLE: ACCESS 3

- Derived class is in the same p1 package and is the sub-class of
- Protection. It has access to all variables of Protection except the private n_pri:

```
package p1;
class Derived extends Protection {
  Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
  }
}
```

EXAMPLE: ACCESS 4

- SamePackage is in the p1 package but is not a sub-class of Protection.
- It has access to all variables of Protection except the private n_pri: package p1;

```
class SamePackage {
  SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
  }
}
```

EXAMPLE: ACCESS 5

- Protection2 is a sub-class of p1.Protection, but is located in a different package – package p2.
- Protection2 has access to the public and protected variables of Protection. It has no access to its private and default-access variables:

```
package p2;
class Protection2 extends p1.Protection {
  Protection2() {
        System.out.println("derived other package");
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
  }
}
```

EXAMPLE: ACCESS 6

- OtherPackage is in the p2 package and is not a sub-class of p1.Protection.
- OtherPackage has access to the public variable of Protection only. It has no access to its private, protected or default-access variables:

```
class OtherPackage {
  OtherPackage() {
       p1.Protection p = new p1.Protection();
       System.out.println("other package constructor");
       System.out.println("n_pub = " + p.n_pub);
  }
}
```

EXAMPLE: ACCESS 7

- A demonstration to use classes of the p1 package:

```
package p1;
public class Demo {
 public static void main(String args[]) {
      Protection ob1 = new Protection();
      Derived ob2 = new Derived();
      SamePackage ob3 = new SamePackage();
 }
}
```

EXAMPLE: ACCESS 8

- A demonstration to use classes of the p2 package:

```
package p2;
public class Demo {
 public static void main(String args[]) {
      Protection2 ob1 = new Protection2();
      OtherPackage ob2 = new OtherPackage();
 }
}
```

IMPORT STATEMENT

- The import statement occurs immediately after the package statement and before the class statement:

```
package myPackage;
import otherPackage1.otherPackage2.otherClass;
class myClass { … }
```

- The Java system accepts this import statement by default:
  `import java.lang.*;`

- This package includes the basic language functions. Without such functions, Java is of no much use.

NAME CONFLICT 1

- Suppose a same-named class occurs in two different imported packages:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass { … otherClass … }

package otherPackage1;
class otherClass { … }

package otherPackage2;
class otherClass { … }
```

NAME CONFLICT 2

- Compiler will remain silent, unless we try to use otherClass. Then it will display an error message. In this situation we should use the full name:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass {
  …
  otherPackage1.otherClass
  …
  otherPackage2.otherClass
  …
}
```

SHORT VERSUS FULL REFERENCES

- Short reference:

```
import java.util.*;
class MyClass extends Date { … }
```

- Full reference:

```
class MyClass extends java.util.Date { … }
```

- Only the public components in imported package are accessible for non-sub-classes in the importing code!

EXAMPLE: PACKAGES 2

- The importing code has access to the public class Balance of the MyPack package and its two public members:

```
import MyPack.*;
class TestBalance {
 public static void main(String args[]) {
     Balance test = new Balance("J. J. Jaspers", 99.88);
     test.show();
 }
}
```

JAVA SOURCE FILE

- Finally, a Java source file consists of:
  1) a single package instruction (optional)
  2) several import statements (optional)
  3) a single public class declaration (required)
  4) several classes private to the package (optional)
- At the minimum, a file contains a single public class declaration.

EXERCISE: ACCESS

1) Create a package emacao. Don't forget to insert your package into a directory of the same name. Insert a class AccessTest into this packge. Define public, default and private data members and methods in your class AccessTest.
2) Define a second class Accessor1 in your package that accesses the different kinds of data members of methods (private, public, default). See what compiler messages you get.
3) Define class Accessor2 outside the package. Again try to access all methods and data members of the class AccessTest. See what compiler messages you get.
4) Where are the differences between Accessor1 and Accessor2 ?

INTERFACE

- Using **interface**, we specify what a class must do, but not how it does this.
- An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.
- An interface is defined with an interface keyword.

INTERFACE FORMAT

General format:

```
access interface name {
  type method-name1(parameter-list);
  type method-name2(parameter-list);
  …
  type var-name1 = value1;
  type var-nameM = valueM;
  …
}
```

INTERFACE COMMENTS

- Two types of access:

  1) public – interface may be used anywhere in a program

  2) default – interface may be used in the current package only
- Interface methods have no bodies – they end with the semicolon after the parameter list. They are essentially abstract methods.
- An interface may include variables, but they must be final, static and initialized with a constant value.
- In a public interface, all members are implicitly public.

INTERFACE IMPLEMENTATION

- A class implements an interface if it provides a complete set of methods defined by this interface.

  1) any number of classes may implement an interface

  2) one class may implement any number of interfaces

- Each class is free to determine the details of its implementation.
- Implementation relation is written with the implements keyword.

IMPLEMENTATION FORMAT

- General format of a class that includes the implements clause:

```
access class name
extends super-class
implements interface1, interface2, …, interfaceN {
  …
}
```

- Access is public or default.

IMPLEMENTATION COMMENTS

- If a class implements several interfaces, they are separated with a comma.
- If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

EXAMPLE: INTERFACE

- Declaration of the Callback interface:

```
interface Callback {
  void callback(int param);
}
```

- Client class implements the Callback interface:

```
class Client implements Callback {
  public void callback(int p) {
      System.out.println("callback called with " + p);
  }
}
```

MORE METHODS IN IMPLEMENTATION

- An implementing class may also declare its own methods:

```
class Client implements Callback {
  public void callback(int p) {
       System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
       System.out.println("Classes that implement ‶ +
       ‶interfaces may also define ″ + ‶other members, too.");
  }
}
```

INTERFACE AS A TYPE

- Variable may be declared with interface as its type:

```
interface MyInterface { … }
…
MyInterface mi;
```

- The variable of an interface type may reference an object of any class that implements this interface.

```
class MyClass implements MyInterface { … }
MyInterface mi = new MyClass();
```

CALL THROUGH INTERFACE VARIABLE

- Using the interface type variable, we can call any method in the interface:

```
interface MyInterface {
  void myMethod(…) ;
  …
}
class MyClass implements MyInterface { … }
…
MyInterface mi = new MyClass();
…
mi.myMethod();
```

- The correct version of the method will be called based on the actual instance of the interface being referred to.

EXAMPLE: CALL THROUGH INTERFACE 2

- TestIface declares the Callback interface variable, initializes it with the new Client object, and calls the callback method through this variable:

```
class TestIface {
 public static void main(String args[]) {
     Callback c = new Client();
     c.callback(42);
 }
}
```

CALL THROUGH INTERFACE VARIABLE 2

- Call through an interface variable is one of the key features of interfaces:
  1) the method to be executed is looked up dynamically at run-time
  2) the calling code can dispatch through an interface without having to know anything about the callee
- Allows classes to be created later than the code that calls methods on them.

EXAMPLE: INTERFACE CALL 1

- Another implementation of the Callback interface:

```
class AnotherClient implements Callback {
 public void callback(int p) {
      System.out.println("Another version of callback");
      System.out.println("p squared is " + (p*p));
 }
}
```

EXAMPLE: INTERFACE CALL 2

- Callback variable c is assigned Client and later AnotherClient objects and the corresponding callback is invoked depending on its value:

```
class TestIface2 {
  public static void main(String args[]) {
       Callback c = new Client();
       c.callback(42);
       AnotherClient ob = new AnotherClient();
       c = ob;
       c.callback(42);
  }
}
```

COMPILE-TIME METHOD BINDING

- Normally, in order for a method to be called from one class to another, both classes must be present at compile time.
- This implies:
  1) a static, non-extensible classing environment
  2) functionality gets pushed higher and higher in the class hierarchy to make them available to more sub-classes

RUN-TIME METHOD BINDING

- Interfaces support dynamic method binding.
- Interface disconnects the method definition from the inheritance hierarchy:
    1) interfaces are in a different hierarchy from classes
    2) it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface

INTERFACE AND ABSTRACT CLASS

- A class that claims to implement an interface but does not implement all its methods must be declared abstract.
- Incomplete class implements the Callback interface but not its callback method, so the class is declared abstract:

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
       System.out.println(a + " " + b);
  }
}
```

EXAMPLE: STACK INTERFACE

- Many ways to implement a stack but one interface:

```
interface IntStack {
      void push(int item);
      int pop();
}
```

- Lets look at two implementations of this interface:

1) FixedStack – a fixed-length version of the integer stack
2) DynStack – a dynamic-length version of the integer stack

EXAMPLE: FIXEDSTACK 1

- A fixed-length stack implements the IntStack interface with two private variables, a constructor and two public methods:

```
class FixedStack implements IntStack {
      private int stck[];
      private int tos;
      FixedStack(int size) {
            stck = new int[size]; tos = -1;
      }
```

EXAMPLE: FIXEDSTACK 2

```
    public void push(int item) {
        if (tos==stck.length-1)
            System.out.println("Stack is full.");
        else stck[++tos] = item;
    }
    public int pop() {
        if (tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else return stck[tos--];
    }
}
```

EXAMPLE: FIXEDSTACK 3

- A testing class creates two stacks:

```
class IFTest {
        public static void main(String args[]) {
                FixedStack mystack1 = new FixedStack(5);
                FixedStack mystack2 = new FixedStack(8);
```

EXAMPLE: FIXEDSTACK 4

- It pushes and them pops off some values from those stacks:

```
        for (int i=0; i<5; i++) mystack1.push(i);
        for (int i=0; i<8; i++) mystack2.push(i);
        System.out.println("Stack in mystack1:");
        for (int i=0; i<5; i++)
                System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for (int i=0; i<8; i++)
                System.out.println(mystack2.pop());
        }
}
```

EXAMPLE: DYNSTACK 1

- Another implementation of an integer stack.
- A dynamic-length stack is first created with an initial length. The stack is doubled in size every time this initial length is exceeded.

```
class DynStack implements IntStack {
      private int stck[];
      private int tos;
      DynStack(int size) {
            stck = new int[size];
            tos = -1;
      }
```

EXAMPLE: DYNSTACK 2

- If stack if full, push creates a new stack with double the size of the old stack:

```
public void push(int item) {
      if (tos==stck.length-1) {
            int temp[] = new int[stck.length * 2];
            for (int i=0; i<stck.length; i++)
                  temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
      }
      else stck[++tos] = item;
}
```

EXAMPLE: DYNSTACK 3

- If the stack is empty, pop returns the zero value:

```
        public int pop() {
                if(tos < 0) {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else return stck[tos--];
        }
}
```

EXAMPLE: DYNSTACK 4

- The testing class creates two dynamic-length stacks:

```
class IFTest2 {
      public static void main(String args[]) {
            DynStack mystack1 = new DynStack(5);
            DynStack mystack2 = new DynStack(8);
```

EXAMPLE: DYNSTACK 5

- It then pushes some numbers onto those stacks, dynamically increasing their size, then pops those numbers off:

```
        for (int i=0; i<12; i++) mystack1.push(i);
        for (int i=0; i<20; i++) mystack2.push(i);
        System.out.println("Stack in mystack1:");
        for (int i=0; i<12; i++)
                System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for (int i=0; i<20; i++)
                System.out.println(mystack2.pop());
        }
}
```

EXAMPLE: TWO STACKS 1

- Testing two stack implementations through an interface variable. First, some numbers are pushed onto both stacks:

```
class IFTest3 {
      public static void main(String args[]) {
            IntStack mystack;
            DynStack ds = new DynStack(5);
            FixedStack fs = new FixedStack(8);
            mystack = ds;
            for (int i=0; i<12; i++) mystack.push(i);
            mystack = fs;
            for (int i=0; i<8; i++) mystack.push(i);
```

```
                mystack = ds;
                System.out.println("Values in dynamic stack:");
                for(int i=0; i<12; i++)
                        System.out.println(mystack.pop());
                mystack = fs;
                System.out.println("Values in fixed stack:");
                for(int i=0; i<8; i++)
                        System.out.println(mystack.pop());
        }
}
```

INTERFACE VARIABLES

- Variables declared in an interface must be constants.
- A technique to import shared constants into multiple classes:
  1) declare an interface with variables initialized to the desired values
  2) include that interface in a class through implementation
- As no methods are included in the interface, the class does not implement
- anything except importing the variables as constants.

EXAMPLE: INTERFACE VARIABLES 1

- An interface with constant values:

```
import java.util.Random;
interface SharedConstants {
      int NO = 0;
      int YES = 1;
      int MAYBE = 2;
      int LATER = 3;
      int SOON = 4;
      int NEVER = 5;
}
```

EXAMPLE: INTERFACE VARIABLES 2

- Question implements SharedConstants, including all its constants.
- Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
      Random rand = new Random();
      int ask() {
            int prob = (int) (100 * rand.nextDouble());
            if (prob < 30) return NO;
            else if (prob < 60) return YES;
            else if (prob < 75) return LATER;
            else if (prob < 98) return SOON;
            else return NEVER;
      }
}
```

EXAMPLE: INTERFACE VARIABLES 3

- AskMe includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {
	static void answer(int result) {
		switch(result) {
			case NO: System.out.println("No"); break;
			case YES: System.out.println("Yes"); break;
			case MAYBE: System.out.println("Maybe"); break;
			case LATER: System.out.println("Later"); break;
			case SOON: System.out.println("Soon"); break;
			case NEVER: System.out.println("Never"); break;
		}
	}
```

EXAMPLE: INTERFACE VARIABLES 4

- The testing function relies on the fact that both ask and answer methods, defined in different classes, rely on the same constants:

```
      public static void main(String args[]) {
            Question q = new Question();
            answer(q.ask());
            answer(q.ask());
            answer(q.ask());
            answer(q.ask());
      }
}
```

INTERFACE INHERITANCE

- One interface may inherit another interface.
- The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
      void myMethod1(…) ;
}
interface MyInterface2 extends MyInterface1 {
      void myMethod2(…) ;
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

INHERITANCE AND IMPLEMENTATION

- When a class implements an interface that inherits another interface, it must provide implementations for all inherited methods:

```
class MyClass implements MyInterface2 {
     void myMethod1(…) { … }
     void myMethod1(…) { … }
     …
}
```

EXAMPLE: INTERFACE INHERITANCE 1

- Consider interfaces A and B.

```
interface A {
      void meth1();
      void meth2();
}
B extends A:
interface B extends A {
      void meth3();
}
```

EXAMPLE: INTERFACE INHERITANCE 2

- MyClass must implement all of A and B methods:

```
class MyClass implements B {
      public void meth1() {
            System.out.println("Implement meth1().");
      }
      public void meth2() {
            System.out.println("Implement meth2().");
      }
      public void meth3() {
            System.out.println("Implement meth3().");
      }
}
```

EXAMPLE: INTERFACE INHERITANCE 3

- Create a new MyClass object, then invoke all interface methods on it:

```
class IFExtend {
      public static void main(String arg[]) {
            MyClass ob = new MyClass();
            ob.meth1();
            ob.meth2();
            ob.meth3();
      }
}
```

EXERCISE: INTERFACE

1) Define two interfaces:

a) an interface CardUse for card use with methods read to read the state and reduceBy with a parameter amount to change the state of the card. The user has to identify himself by a PIN for this operation;

b) an interface CardChange for the administration of the card by authorized people, that need a method reset to reset the card state, a method fill to fill the card with an amount of money and a method changePIN to change the PIN for the card.

2) Define a third interface CardAll that includes all card operation (Use inheritance).

3) Change the interface CardAll into an abstract class that implements the balance of the card and a basic solution for the methods fill and reduceBy leaving the rest of the methods abstract. Choose the correct access specifier to make the balance accessible to subclasses but not to the public; Check this;